1-1-2005

An efficient architecture for discrete wavelet transform and best-basis algorithm for images

Aroutchelvame Mayilavelane
Ryerson University

AN EFFICIENT ARCHITECTURE FOR DISCRETE WAVELET TRANSFORM AND BEST-BASIS ALGORITHM FOR IMAGES

by

Aroutchelvame Mayilavelane
M.Tech, Indian Institute of Technology (IIT), Chennai, India

A thesis

presented to Ryerson University

in partial fulfillment of the

requirement for the degree of

Master of Applied Science

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Aroutchelvame Mayilavelane

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Aroutchelvame Mayilavelane

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Name	Signature	Address	Date

An Efficient Architecture for Discrete Wavelet Transform and Best-Basis Algorithm for Images

Master of Applied Science
Department of Electrical & Computer Engineering
Ryerson University

Abstract

The hardware acceleration of the wavelet transform for real-time systems has become an essential research field. In the first part of the thesis, an efficient architecture that performs both forward and inverse lifting-based discrete wavelet transform is proposed. The proposed architecture reduces the hardware requirement by exploiting the redundancy in the arithmetic operation involved in DWT computation. The proposed architecture does not require any extra memory to store intermediate results. The proposed architecture consists of predict module, update module, address generation module, control unit and a set of registers to establish data communication between predict and update modules. The symmetrical extension of images at the boundary to reduce distorted images has been incorporated in our proposed architecture as mentioned in JPEG2000. The DWT architecture is proposed for both (5,3) wavelet and (9,7) wavelet. Best-basis algorithm that is designed for signal compression and de-noising uses WPT to select the best-basis node for a given additive cost function. In the second part of the thesis, we propose the architecture for best-basis algorithm for images (2D signals) that uses the proposed wavelet architecture to perform WPT decomposition. A new algorithm to implement the natural logarithm function using Maclaurin series is proposed to implement the cost function used for best-basis algorithm. These architectures have been described in VHDL at the RTL level and simulated successfully using ModelSim simulation environment. These architecures are implemented in Virtex II Pro FPGA series of Xilinx.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Kaamran Raahemifar, for everything I have learned from working with him during two years of my graduate study, for his many advices, his enormous patience and for being more a friend than a boss.

I would like to thank all people in the Department of Electrical & Computer Engineering and the System-on-Chip Group in particular, for their help and cordiality. Thanks to the department for providing financial support for my study.

I would like to thank Bahman Zafarifar and Alex Pineiro for discussions on the wavelet transform and natural logarithm implementation. Also, I would like to thank all of my friends and my roommates who helped making this research a nice experience.

I thank Xilinx for providing the FPGA board and necessary softwares for the implementation of my research work.

My parents Mayilavelan & Maheswari, my brothers and cousins and my uncles and aunts have supported me in one way or another. My parents worked hard to ensure that I receive a proper education. My brothers are always ready to extend their helping hands. They are all wonderful and I take this opportunity to thank them for all their support.

Dedicated to my parents for their support and love...

Table of Contents

List of Tables

List of Figures

Glossary

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
AAE	Average Absolute Error
BRAM	Block RAM
CL	Current Level
CORDIC	Coordinate Rotation Digital Computer
DCT	Discrete Cosine Transform
DSP	Digital Signal Processing/Processor
DWT	Discrete Wavelet Transform
DWPT	Discrete Wavelet Packet Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
IOB	Input Output Block
ln	Natual Logarithm
LCD	Liquid Crystal Display
LDB	Local Discriminant Basis
LN	Natural Logarithm
LUT	Look-Up Table
MSE	Mean Square error
MULT 18X18	18-bit * 18-bit Multiplier
PSNR	*Peak Signal-to-Noise Ratio*
RAM	*Read Access memory*
RTL	Register Transfer Level
SNR	Signal-to-Noise Ratio
VHDL	VHSIC Hardware Description Langauge
VHSIC	Very High Speed Integrated Circuits
WPT	Wavelet Packet Transform

CHAPTER 1

INTRODUCTION

1.1 Overview

Interest in wavelet transformations has greatly increased, as their applications have become numerous such as image processing and image compression. Digital images have become more common in the multimedia world. The digital images replace their old analog ancestors. One such application is found in digital cameras. In digital cameras, one expects to inspect the result immediately after taking the picture. To support this behavior, the picture has to be compressed, stored on a flash memory card, decompressed and shown at LCD display in nearly real-time. Features like high-speed previews have to be provided. Furthermore, one could expect to store more number of images on memory stick. Therefore, efficient hardware image compression algorithms with excellent visual properties are necessary.

Practical image data sequences normally contain a substantial amount of redundancy. Redundancy in signals can appear in the form of smoothness of the signal or in other words correlation between the neighboring signal values. The image sequence that embeds redundancy can be presented more compactly if the redundancy is removed

by means of a suitable transform. A popular transform that has been used for years for compression of digital still images and image sequences is the Discrete Cosine Transform (DCT). The DCT transform uses cosine functions of different frequencies for analysis and for decorrelation of data. The DCT provides signal frequency content into fixed, equal bandwidth partitions. By providing only the frequency content of the signal, the DCT is unable to represent non-stationary signal properties in the transform domain. Often, images are non-stationary. To overcome this problem, the JPEG algorithm [1] uses a block based transform in which DCT is applied to image block separately. Because each block is transformed on an individual basis, there are often inefficiencies between blocks. The Discrete Wavelet Transform (DWT) rectifies this problem by providing a representation of a given image in both time and scale domains. Because of this, the DWT has been shown to significantly outperform the DCT in image compression applications, leading to their inclusion in the JPEG 2000 standard [2-3]. The DWT is a reversible transform and can be either a "lossy" or "lossless" process depending on the selection of wavelet. Besides image compression, DWT are also aptly suited for image editing and progressive transmission applications since they provide a multiresolution decomposition of a signal.

The DCT based JPEG algorithm yields good results of compression ratio till 10:1. As the compression ratio increases, the quantization of the coefficients causes blocking effects in the decompressed image. When compression ratio reaches 24:1, it only allows the DC coefficients, which are the average of the pixels of an 8x8 block, to be encoded. Consequently, the input image is approximated by a series of 8x8 blocks of local averages, which is visually very annoying. For DWT followed by Embedded Zero Tree

encoding algorithm, in contrast, compressions of the ratios of 100:1 have been achieved, while still yielding a reconstructed image with an acceptable quality.

The lifting scheme [4-7] has been introduced for efficient computation of the DWT. Its main advantage with respect to the classical filter bank structure lies in its better computational efficiency [24] and in the fact that it enables a new method for filter design. Using the lifting scheme, it is easy to use integer arithmetic without encountering problems due to finite precision or rounding. Applying the inverse transform in the lifting scheme is very easy and, as long as the transform coefficients are not quantized, would always result in a perfect reconstruction of the original picture.

Some of the applications of DWT are digital video compression, telecommunications, signal and image processing and processing of non-stationary signals in areas such as bio-medicine. In those applications, the computational burden on the transform part is very high. To meet this additional burden in real-time applications, the hardware design and implementation of these transforms has itself taken on much importance. In this case, the high-computational parts of the program are designed in hardware and loaded into reconfigurable hardware unit such as FPGA.

The performance of these applications can be increased if the transform provides good spectral and temporal resolutions in arbitrary regions of the time-frequency plane. This flexibility is provided by Wavelet Packet Transform (WPT). It is a generalized representation of DWT which allows the further decomposition of the high-pass output i.e. detailed information.

In WPT analysis, the signal of size $N = 2^n$ can be expanded in 2^n different ways and this number may be large. So, it is important to use an efficient method to find its

best-tree representation, i.e. the best-tree that minimizes certain additive cost function. Coifman and Wickerhauser proposed a classical entropy-based algorithm to find the best-tree and their algorithm is called best-basis algorithm [8]. Best basis algorithm is primarily designed to reduce the storage space needed for a signal and also it can be used for de-noising. This algorithm looks for time-frequency representation of the signal in wavelet packet or trigonometric basis. The entropy-based algorithm involves the computation of logarithm function. The hardware implementation of logarithm function and other elementary function is the performance bottleneck of the best-basis algorithm in real-time systems. The higher the number of bits of precision is required by the logarithm function, the more is the number of clock cycles required by the hardware.

Software routines applying techniques such as polynomial and rational expressions have been used to evaluate the elementary functions such as logarithm function [9]. Even though these techniques compute logarithm function with accurate results, they are often too slow compared to the real-time applications. The hardware implementation of the elementary functions have been developed as an alternative to the software routines, providing high speed solutions implemented in dedicated hardware. The CORDIC based algorithms [10-11] are considered to be one of the best hardware based methods due to their low area requirements. The main drawback in this algorithm is their approximate linear convergence of one radix-r digit per step, resulting in long execution times for small radices and high precision. The latency can be reduced by increasing the radix. However this method leads to an increase in the cost of implementation with increase in radix.

This thesis has been broadly classified into two parts. In the first part of the thesis, we propose an efficient architecture for DWT (WPT), using (5,3) and (9,7) wavelet mentioned in JPEG2000, that implements lifting scheme. The proposed DWT (WPT) architecture requires less hardware area and it does not require extra storage elements such as memory and FIFO to store the intermediate results. In the second part of the thesis, we propose architecture for best-basis algorithm for 2D signal or image. This part of the thesis also includes a proposed algorithm and architecture for implementing natural logarithm using Maclaurin series. The hardware design implements different techniques such as pipelining, parallel operation modules, data reusability and special features of FPGA to maximize its performance.

1.2 Research goal and the contribution of the Author

The computational complexity of DWT and best-basis algorithm is an obstacle for using those in practical applications. Therefore, the hardware implementation of the algorithm is desirable. The main objective of this thesis is to propose an efficient architecture for DWT and best-basis algorithm. The contribution of the author is described in the following:

- Proposing an efficient architecture for DWT or WPT using (5,3) and (9,7) wavelets that requires less hardware area and does not require extra storage devices to store the intermediate results,
- Proposing a novel algorithm and architecture for natural logarithm function using Maclaurin series,

- Proposing an efficient architecture for best-basis algorithm for 2D signal or image which includes detailed architecture description for calculation of cost-function, address generation for best-basis RAM and calculation of best-tree,
- Implementing the design in VHDL,
- Performing simulations in ModelSim to verify functional correctness of the design,
- Implementing the design in Xilinx Virtex II Pro FPGA using Xilinx Implementation tool,
- Comparing the performance of the DWT architecture with some existing popular architectures,
- Performing simulations with large picture size for different precision of filter coefficients and analyzing the error variation for both (5,3) and (9,7) wavelets.
- Performing simulations for best-basis algorithm for different images or 2D signals to verify the architecture.

1.3 Organization of the thesis

The main objective of the thesis is to propose an efficient architecture for DWT or WPT, natural logarithm function and the best-basis algorithm. The thesis is organized as follows:

Chapter 2 starts with the theoretical background, briefly introducing the DWT and the best-basis algorithm. The lifting scheme based DWT computation results in less number of arithmetic operations compared to the filter-based DWT computation. The lifting-based DWT and the best-basis algorithm with detailed example are described in

this chapter. Detailed survey of some existing lifting-based DWT architectures and hardware implementation of elementary functions such as natural logarithm is presented in this chapter.

Chapter 3 proposes an efficient architecture for DWT or WPT that requires less hardware area and does not require extra storage devices to store intermediate results. The DWT architecture for both (5,3) and (9,7) wavelets is described in detail in this chapter.

Chapter 4 proposes architecture for best-basis algorithm for 2D signal or image. This chapter also discusses in detail the architecture for calculating cost-function, generating addresses for best-basis RAM and determining best-tree. The architecture for two cost functions, Threshold function and Shannon function, is also proposed. Also, an algorithm and architecture to implement natural logarithm function using Maclaurin series is proposed in this chapter.

Chapter 5 presents results and discussion for the DWT or WPT architecture and the best-basis algorithm architecture.

Chapter 6 summarizes the important contributions and results in the proposed architectures and lists out the future work related to this research.

CHAPTER 2

BACKGROUND

This chapter starts with a brief introduction of discrete wavelet transform and lifting scheme. It also provides survey of some of the existing DWT architectures. In the next section, some of the existing hardware implementation of elementary functions such as logarithm function is discussed. In the last section, the best-basis algorithm is briefly discussed with an example.

2.1 Discrete Wavelet Transform (DWT)

2.1.1 Introduction

The idea of the DWT is mainly based on the sub-band coding scheme. In the discrete time function, the filters of different cutoff frequencies are used to analyze the signal at different scales. The signal is passed through a series of high pass filters to analyze the high frequencies, and it is passed through a series of low pass filters to analyze the low frequencies. The resolution of the signal, which is a measure of the amount of detail information in the signal, is changed by the filtering operations, and the

scale is changed by upsampling and downsampling (subsampling) operations. Subsampling a signal corresponds to reducing the sampling rate, or removing some of the samples of the signal. Upsampling a signal corresponds to increasing the sampling rate of a signal by adding new samples to the signal. In discrete signals, frequency is expressed in terms of radians. Accordingly, the sampling frequency of the signal is equal to 2π radians in terms of radial frequency [12-15].

The DWT analyzes the signal at different frequency bands with different resolutions by decomposing the signal into a coarse approximation and detail information. The decomposition of the signal into different frequency bands is simply obtained by successive highpass and lowpass filtering of the time domain signal. The original signal x(n) is first passed through a halfband highpass filter h(n) and a lowpass filter g(n) [12-15]. After the filtering, half of the samples can be eliminated according to the Nyquist's rule, since the signal now has a highest frequency of π /2 radians instead of π. The signal can, therefore, be subsampled by 2, simply by discarding every other sample.

The DWT coefficient can be obtained for 2D signals such as image as explained below. The DWT decomposes the image/signal in the space-resolution domain. The one stage of DWT computation produces four sub-images representing an approximated image (LL) and the details along directions (HH, HL and LH) as shown in Fig.2.1. The operations involved are low-pass filtering (g(n)), high-pass filtering (h(n)) and down-sampling by 2 to keep only necessary information [14]. In one-level decomposition, the above operation is performed on the rows of the input, x(n), to generate the high-pass (H) and low-pass (L) outputs. The same operation is performed on the H and L values to

generate "HH and HL" and "LH and LL" output respectively. The LL outputs are fed to the next level of decomposition.

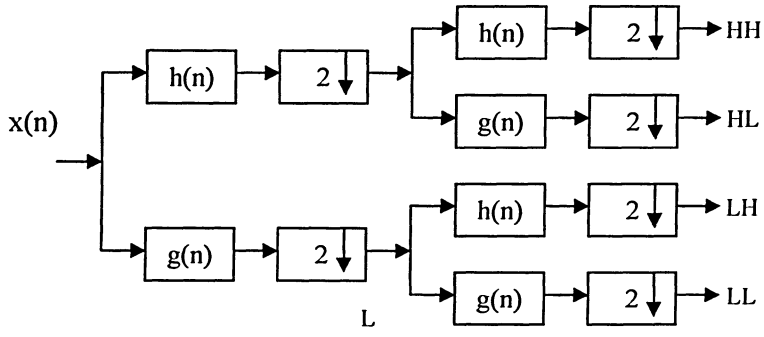


Fig.2.1: *DWT for 2D Signals*

2.1.2 Lifting Scheme

Wavelets discussed in the previous section are referred to as first generation wavelets. Second generation wavelets are much more flexible and used to define wavelet bases for bound intervals. The main difference between first and second generation wavelets is that it does not involve convolution. Lifting scheme can be used to construct second generation wavelet.

The basic idea of the lifting scheme is very simple – try to use the correlation in the data to remove redundancy. The 1D DWT decomposition using lifting scheme consists of three stages: split, predict, update and scaling [4-7] as shown in Fig.2.2. The 2D DWT decomposition can be easily achieved by applying the lifting scheme both row-wise and column-wise.

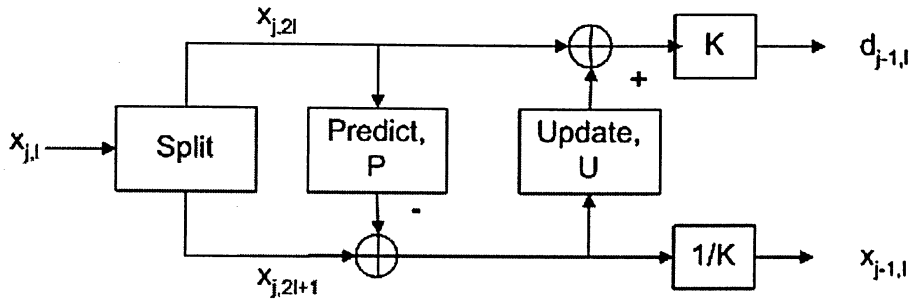


Fig2.2: *Lifting Scheme*

In the first stage (split), the input data is divided into two disjoint sets of samples. There is no restriction on splitting the data but the only thing needed is some procedure to join the two samples to obtain the original data [4-7]. One of the possibilities of the split of the input $x_{j,l}$ is grouping the even indexed samples $x_{j,2l}$ and the odd indexed samples $x_{j,2l+1}$ in different sets.

In the second stage (predict), one set of samples can be used to predict the other set of samples based on the correlation present in the input signal [4-7]. The prediction operator 'P' can be constructed based on the correlation structure of the input data. Practically, it might not be possible to accurately predict the one set of sample based on the other set of samples. Thus, the set of samples to be predicted or detailed co-efficient can be replaced with the difference between itself and the predictor output. If the samples 'x_{2l}', 'x_{2l+1}' and 'x_{2l+2}' considered in the split stage are highly correlated, one can easily device the predictor function using the even samples as variables to predict the odd samples. To predict the odd sample '$x_{j,2l+1}$', the predictor function can be the average of the neighboring sample on the left '$x_{j,2l}$' and the other sample on the right '$x_{j,2l+2}$.' The detail co-efficient can be given by $d_{j-1,l} = x_{j,2l+1} - \frac{1}{2}(x_{j,2l} + x_{j,2l+2})$.

The update stage utilizes the key properties of the coarser signals i.e. they have the same average value of the signal [4-7]. Thus the update stage implements this property by finding the update function 'U' using the detailed coefficients with the sample used for predictor function. In our case, this can be achieved using the equation $x_{j-1,l} = x_{j,2l} + ¼(d_{j-1,l-1}+d_{j-1,l})$.

In the scaling stage, the even samples are multiplied by 1/K and odd samples by K in the case of forward transform. The inverse DWT is obtained by traversing in the reverse direction, changing the factor K to 1/K and factor 1/K to K.

All these values can be computed-in-place i.e. the even index locations can be overwritten with averages (coarse coefficient) and the odd ones with the detailed coefficient.

One can immediately build inverse scheme for it. Again the inverse lifting scheme has three stages: undo update, undo predict and merge.

Given $d_{j-1,l}$ and $x_{j-1,l}$, one can recover the even samples by simply subtracting the update information as $x_{j,2l} = x_{j-1,l} - ¼(d_{j-1,l-1} + d_{j-1,l})$ – undo update.

Given $x_{j,2l}$ and $d_{j-1,l}$, one can recover the odd samples by adding the predict function as shown $x_{j,2l+1} = d_{j-1,l} + ½(x_{j,2l} + x_{j,2l+2})$ – undo predict.

Given both odd and even samples, one has to simply zipper them together to recover the original signal - Merge.

The main advantages [4-7] of the lifting scheme are:

- Lifting scheme is fast and easy to implement in hardware because it replaces the Fourier transform.

- Lifting can be done in-place, therefore no auxiliary memory needed. At every stage, the calculated value replaces the old one.
- The inverse transform can be immediately found by undoing the operations of the forward transform. It can be realized by reversing the order of the arithmetic operation.

2.1.3 Boundary Treatment

Real-world signals do not extend infinitely in time or space, but are limited to a finite interval. So when the signal comes close enough to the edge, the filters need some sample values that are not defined. If zero padding is employed to avoid the boundary discontinuity, the transform results in large coefficients. This will induce artifacts in the image and severe encoding inefficiency. As a solution to this problem, classical signal processing extends the data for the computations near the boundary by either periodic extension or boundary extension.

In periodic extension, the finite signal is extended periodically by putting copies of itself in front of and behind the original signal. After the wavelet transform, the coefficients that lie outside of the interval of defined signal is simply discarded. These discarded coefficients can be recovered easily because they are the same as the retained coefficients. However, unless the first and the last samples have the same value, we introduce unwanted discontinuities at the boundaries of the original signal. These discontinuities will locally enlarge the wavelet coefficients and make compression of the signal more difficult.

An easy solution for handling the finite length signals is to extend them such that they become symmetric and periodic [16]. Symmetric extension consists in extending the signal with the signal samples obtained by a reflection of the signal centered on the first sample for extension to the left, and in extending the signal with the signal samples obtained by a reflection of the signal centered on the last sample for extension to the right.

2.1.4 DWT Architectures

Several architectures have been proposed for hardware implementation of convolution-based DWT [17-21] because the DWT computation is basically the filter convolution as explained in Section 2.1.1. After introduction of second generation wavelets using lifting scheme, it has been used widely in DWT architecture because it requires fewer arithmetic computations and provides faster and efficient filtering of the DWT than that of the convolution-based DWT architecture. Thus, using lifting-based DWT architecture outperforms convolution-based DWT architectures. Also, the line-based method of computing DWT has been proposed to reduce the internal memory requirement from a frame size to a few line buffers with proper memory management [22-23]. This section discusses some of the efficient architectures for 2D DWT using the lifting scheme.

The architecture described in [24] calculates forward and inverse 2D DWT in row-column fashion on a block of data of size NxN. The architecture reads block of data, performs the transform and outputs LH, HL and HH data at each level of decomposition. The LL data is used for the next level of decomposition. The architecture consists of row

module with two row processors and register file, column module with two column processors and register files and two memory modules (one for row processing and the other for column processing). This architecture can be configured to perform 2D DWT for several DWT filterbanks. Each row and column processor comprises of a different configuration of adders, multipliers and shifters in the data path for several DWT filterbanks which was explained in detail in [24]. This architecture needs more intermediate or internal memory to perform the computation. Row processing memory module uses two banks with one read/write access for each bank whereas column processing memory module consists of four banks with one read/write operation for three banks and dual read operation for the last one.

The architecture in [25] describes hardware accelerator for the lifting scheme based DWT. For 2D signals, this architecture performs DWT in row-wise and then in column-wise. This architecture achieves the acceleration using the techniques such as pipelining, data-reusability and parallel operations. This architecture consists of predict module, update module and FIFOs to store intermediate results. The FIFOs are used to store some of the input signals and the output of the predict module and then forward those values to the update module. The filter coefficients for both predict and update modules are supplied from the memory. In order to meet the boundary conditions, different filter coefficients are supplied for the input signals at the boundary compared to the coefficients supplied for non-boundary input signals. Since the architecture uses different coefficients at various points of the input signal, this architecture was unable to exploit the redundancy in the arithmetic operations of the DWT computation.

The architecture described in [26] implements line-based architecture for the 2D inverse wavelet transform. The architecture performing row-wise computation is straightforward. The column-wise computation is started as soon as the row-wise computation is completed for the current line of the 2D signal. Line buffering in the form of FIFOs and additional logic are necessary to perform column-wise DWT computation because the data are fetched in different order for the column-wise computation. This architecture starts the computation before the whole 2D signal is stored in the input memory. However, the proposed architecture in this thesis does not involve line-based transform, therefore this architecture is not considered for comparison.

The architecture in [27] folds the computations of all decomposition levels into the same low-pass and high-pass units to achieve higher hardware utilization. The architecture requires very large quantity of registers with increase in decomposition level. Apart from that, the whole architecture has to be designed separately for various levels of decomposition. The proposed architecture utilizes the same hardware to compute DWT coefficients irrespective of the level of decomposition.

2.2 Best-Basis Algorithm

The 2D DWT as shown in Fig.2.1 decomposes the image into four sub-bands: approximation, vertical detail, horizontal detail and diagonal detail. Iterating this filter structure on all sub-bands creates a so-called atomic decomposition of the image in the form of a full, balanced quad-tree. This decomposition provides a large number of all possible decompositions. Each node of that quad-tree corresponds to a sub-band that is a

projection of the image to a subset of basis functions. This decomposition is called Wavelet Packet or WPT. The octave decomposition obtained from DWT is good for images that have most of their energy in a low frequency part. For some images that contain stronger high frequency elements, WPT is a better choice. A relatively simple and effective optimization algorithm called best-basis algorithm was proposed by Coifman [8]. It trims the atomic decomposition tree to minimize some defined cost function M. This algorithm simplifies optimization of the whole tree to a parent-children comparison of criterion M in a bottom-up direction. The algorithm takes the next four steps:

Step 1: Decompose image into an atomic tree using wavelet packet.

Step 2: Compute cost of each node using one of the cost functions listed in the later part of this section.

Step 3: Starting at the bottom of the tree, repeat step 4 on all the nodes except the last level leaves until the root is reached.

Step 4: Make the following decision:

If $\sum_{k=1}^{4} M(x_{i,kthchild}) < M(x_{ithparent})$ then

(i) preserve subtree below the node $x_{ithparent}$

(ii) Replace the cost of the parent with the total cost of the children nodes as shown below:

$$M(x_{ithparent}) = \sum_{k=1}^{2} M(x_{i,kthchild})$$

else

prune subtree below $x_{ithparent}$

end if

The cost function or information cost is the essential part of best-basis algorithm. Some of the cost functions [28] are listed below:

$$M = \sum_{k \varepsilon Z} \mu(|u(k)|) \tag{2.1}$$

- Number of elements above threshold value:

$$\mu(\omega) = \begin{cases} 0, \text{if} |\omega| \geq t \\ 1, \text{if} |\omega| < t \end{cases} \tag{2.2}$$

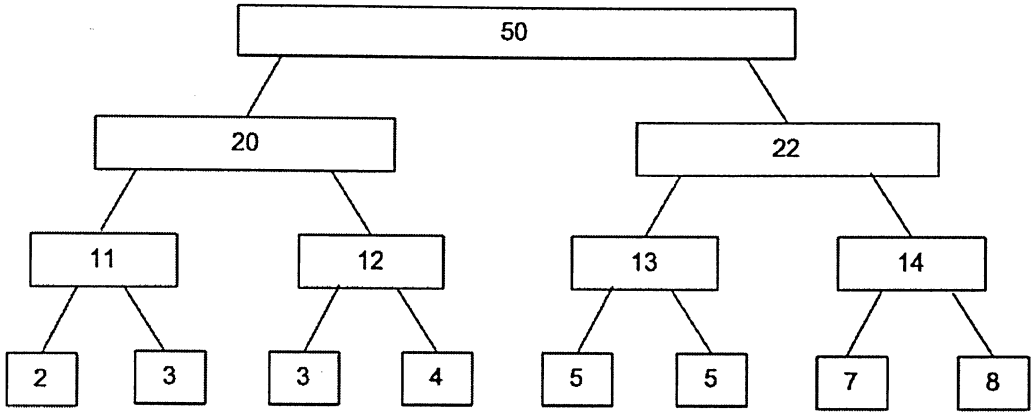
- Logarithm of energy: $\mu(\omega) = \log|\omega|^2$ (2.3)

- Entropy:

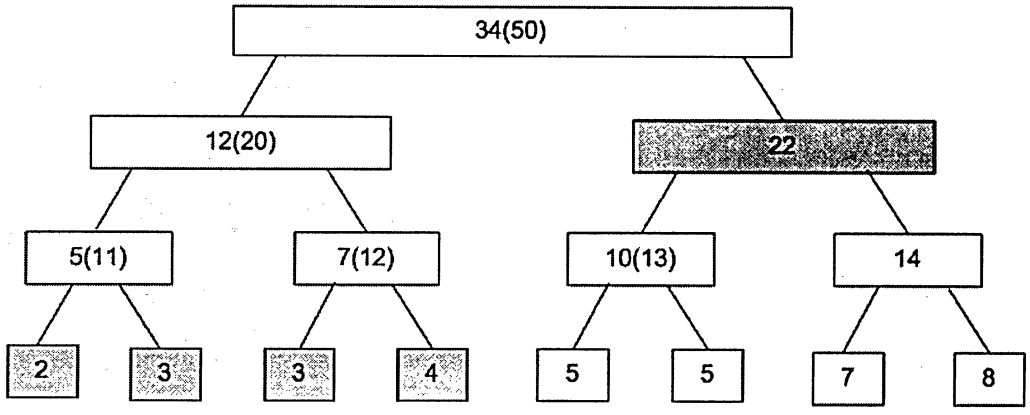
$$M(u) = \sum_{k} |u(k)|^2 \log \frac{1}{|u(k)|^2} \tag{2.4}$$

The best-basis algorithm for 1D signal is explained with an example shown in Fig.2.3. The first step of the algorithm is to decompose the signal into quadtree structure and to compute cost of each node in the tree using one of the cost functions M given in equations (2.1-2.4). Starting from the bottom level of the quadtree, the cost of parent node is compared with the total cost of the corresponding children node. The cost of each pair of adjacent children nodes are compared to that of their parent. This is simply accomplished by comparing the cost function at the parent node with sum of cost function at children nodes. Either the parent node or the total cost of whichever has less cost is selected and the search is continued to the top branch in the tree. If the total cost of the children nodes is less than that of their parent, the children nodes are the better choice. In that case, the cost of the parent node is updated with the cost of the children and the

old parent cost is mentioned inside the parenthesis as shown in Fig.2.3. The nodes whose cost has not been updated with have been selected as best-basis nodes.



First Stage: Compute cost and mark the nodes



Second Stage: Perform Step-4 of the algorithm and select best-basis

Fig.2.3: *Best-basis Selection process*

2.3 Architectures for Elementary Functions

Elementary functions such as logarithm function are important for scientific computing, logarithmic number system processor and DSP and 3D-graphics applications.

Some of other elementary functions are square-root, exponential and trigonometric functions those are widely used in scientific computations [29-31]. Software routines applying techniques such as polynomial and rational expressions have been used to evaluate it. Although these routines provide accurate results, they are often too slow for real-time applications.

The hardware implementation of the elementary functions such as logarithm, exponential has been broadly classified into two groups – non-iterative and iterative methods. The non-iterative method includes direct table look-up, polynomial and rational approximations and table-based methods [32-34] and it is usually suitable for low-precision calculations. The iterative method includes CORDIC, digit-recurrence and on-line algorithms [35-38] and functional iteration methods such as Newton-Raphson and Goldsmidt algorithms [39] and it is usually suitable for both low-precision and high-precision computations.

Direct table look-up is suitable for very–low precision calculations, but the huge memory requirements of such technique make it an inefficient method for even single–precision computations. Another hardware implementation is based on approximating the elementary functions in the form of polynomial approximations [40]. However, the degree of the polynomial to be employed is usually high and a large number of multiplications and additions must be performed resulting in long execution times.

Table-based methods involve both direct table look-up and polynomial approximations method. The use of table look-up allows a low-degree polynomial to be employed and the low-degree polynomial allows a significant reduction in size of the

look-up tables used for computation. Some of the table-based methods are discussed in [32-34].

High-radix CORDIC based and high-radix digit-recurrence algorithms are important methods to implement elementary functions in hardware because of their low area requirements for high-precision computations compared to that of table-based methods. The main drawback in this algorithm is their linear convergence of one radix-r digit per step, resulting in long execution times for small radices and high precision. High-radix digit recurrence methods have been proposed for the computation of several elementary functions [35-38]. However, this method leads to an increase in the cost of implementation with increase in radix.

Functional iteration methods, such as Newton-Raphson and Goldschmidt algorithms [39], are based on multiplication operations and have quadratic convergence which results in low-latency algorithms for high-precision computations at the expense of increased hardware requirements. One of the main drawbacks of using this method is the difficulty in obtaining correct rounded result.

We propose an algorithm and architecture for natural logarithm implementation using Maclaurin series in Chapter 4.

CHAPTER 3

ARCHITECTURE FOR LIFTING-BASED DISCRETE WAVELET TRANFORM

3.1 Introduction

The Discrete Wavelet Transform (DWT) based on lifting scheme was discussed in detail in Section 2.1. The lifting scheme is a fast implementation of the DWT. However, the software implementation of the lifting-based DWT on general-purpose processors is often too slow for real-time applications. General-purpose processors execute the programs in a sequential manner. However, in the hardware domain, the architecture for lifting-based DWT can be implemented such that the parallelism in its implementation can be exploited methodically. Also in the hardware domain, large amount of needed data can be stored in the registers and accessed immediately for future computation.

The lifting-based DWT computation involves considerable amount of parallel operations. The proper architectural design of DWT can buffer the data read from the memory and reduce the large amount of data transfer to and from the input memory. The dual-port internal RAM blocks in Xilinx Virtex II and Virtex II Pro FPGA family can be

exploited to implement the lifting-based DWT architecture. Also, most of the existing DWT architectures use either memory or FIFOs to store the intermediate results obtained. The proposed architecture performs both forward and inverse lifting-based DWT. The architecture does not require any extra memory or FIFOs to store the intermediate results.

This chapter is organized as follows. The arithmetic operations involved in lifting-based DWT for (5,3) and (9,7) wavelets are discussed in Section 3.2. In Section 3.3, the proposed architecture is presented with detailed description of predict module, update module and their integration.

3.2 Arithmetic Operations in Lifting-Based DWT

The lifting scheme provides many advantages such as fewer arithmetic operations, in-place implementation and easy management of boundary extension compared to convolution-based DWT architectures. The arithmetic operations of wavelet (5,3) and (9,7) filters, adopted in JPEG 2000 [41-42], are presented in this section in order to explain the redundancy in the arithmetic operation involved in the calculation of the lifting-based DWT computation. The lifting-based implementation of (5,3) wavelet as mentioned in JPEG2000 is shown in Fig. 3.1. The calculation of two consecutive high-pass and low-pass coefficients for (5,3) wavelet for an input signal $x_{j,i}$ is shown below:

High-pass coefficients, $d_{j-1,i}$:

$$d_{j-1,i} = x_{j,2i+1} + \alpha*(x_{j,2i}) + \alpha*(x_{j,2i+2}), \tag{3.1}$$

$$d_{j-1,i+1} = x_{j,2i+3} + \alpha*(x_{j,2i+2}) + \alpha*(x_{j,2i+4}). \tag{3.2}$$

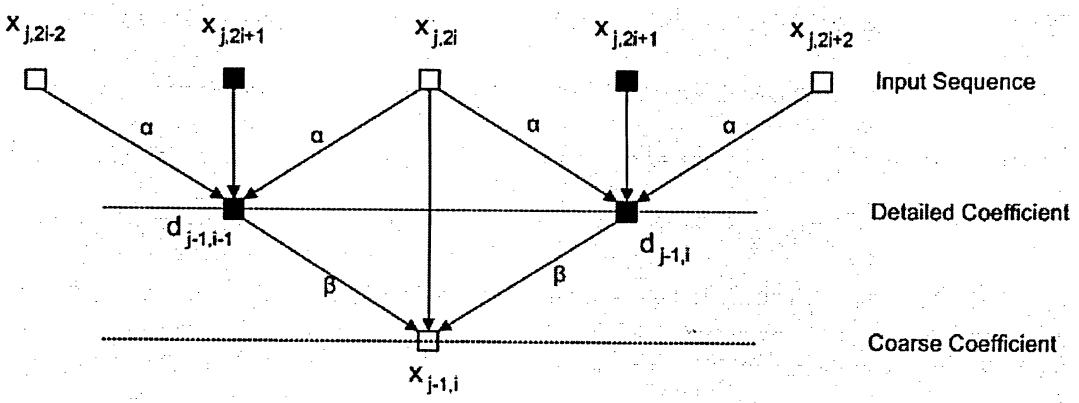


Fig.3.1: *Lifting Scheme for (5,3) wavelet*

Low-pass coefficients, $x_{j-1,i}$:

$$x_{j-1,i} = x_{j,2i} + \beta * (d_{j-1,i-1}) + \beta * (d_{j-1,i}), \tag{3.3}$$

$$x_{j-1,i+1} = x_{j,2i+2} + \beta * (d_{j-1,i}) + \beta * (d_{j-1,i+1}). \tag{3.4}$$

where $\alpha = 0.5$ and $\beta = 0.25$ are the (5,3) filter coefficients [41]. From the equations (3.1) and (3.2), it is found that the product value of α times $x_{j,2i+2}$ calculated at the particular clock cycle is required at the next clock cycle. Similarly from equations (3.3) and (3.4), the product value of β times $d_{j-1,i}$ at the particular clock cycle is required at the next clock cycle. Therefore, in the proposed architecture for the predict module calculation, we perform one multiplication in each cycle for calculating $[\alpha*(x_{j,2i+2})]$ and the other value $[\alpha*(x_{j,2i})]$ can be obtained from previous clock cycle, instead of performing two multiplications in every clock cycle as mentioned in [25]. Also, the proposed architecture needs only one multiplier in the update module. Thus, the proposed architecture utilizes the redundancy of the above mentioned arithmetic operation reducing the number of multipliers required.

The lifting-based implementation of Daubechies 9/7 wavelet as mentioned in JPEG2000 is shown in Fig. 3.2. It involves two scaling steps and four lifting steps (first and second intermediate stages, high-pass stage and low-pass stage) as given below:

$$I_{2i+1} = x_{j,2i+1} + \alpha * (x_{j,2i}) + \alpha * (x_{j,2i+2}), \tag{3.5}$$

$$I_{2i} = x_{j,2i} + \beta * (I_{2i-1}) + \beta * (I_{2i+1}), \tag{3.6}$$

$$d_{j-1,i} = I_{2i-1} + \chi * (I_{2i-2}) + \chi * (I_{2i}), \tag{3.7}$$

$$x_{j-1,i} = I_{2i} + \delta * (d_{j-1,i}) + \delta * (d_{j-1,i+1}), \tag{3.8}$$

$$d'_{j-1,i} = K * (d_{j-1,i}), \tag{3.9}$$

$$x'_{j-1,i} = (1/K) * (x_{j-1,i}), \tag{3.10}$$

where α = -1.586134342, β = -0.052980118, χ = +0.882911075, δ = +0.443506852 are the coefficients for (9,7) wavelet and K = 1.230174104 is the scaling factor [31]. Equations (3.5), (3.6), (3.7) and (3.8) represent the calculations of first intermediate, second intermediate, high-pass and low-pass stages respectively. The calculation of four lifting steps for two consecutive values for 9/7 wavelet is shown below:

First Immediate Stage, I_{2i+1}:

$$I_{2i-1} = x_{j,2i-1} + \alpha * (x_{j,2i-2}) + \alpha * (x_{j,2i}) \tag{3.11}$$

$$I_{2i+1} = x_{j,2i+1} + \alpha * (x_{j,2i}) + \alpha * (x_{j,2i+2}) \tag{3.12}$$

Second Intermediate Stage, I_{2i}:

$$I_{2i-2} = x_{j,2i-2} + \beta * (I_{2i-3}) + \beta * (I_{2i-1}) \tag{3.13}$$

$$I_{2i} = x_{j,2i} + \beta * (I_{2i-1}) + \beta * (I_{2i+1}) \tag{3.14}$$

High-pass Stage, $d_{j-1,i}$:

$$d_{j-1,i} = I_{2i-1} + \chi * (I_{2i-2}) + \chi * (I_{2i}) \tag{3.15}$$

$$d_{j-1,i+1} = I_{2i+1} + \chi * (I_{2i}) + \chi * (I_{2i+2}) \tag{3.16}$$

Low-pass Stage, $x_{j-1,i}$:

$$x_{j-1,i} = I_{2i} + \delta * (d_{j-1,i}) + \delta * (d_{j-1,i+1}) \tag{3.17}$$

$$x_{j-1,i+1} = I_{2i} + \delta * (d_{j-1,i+1}) + \delta * (d_{j-1,i+2}) \tag{3.18}$$

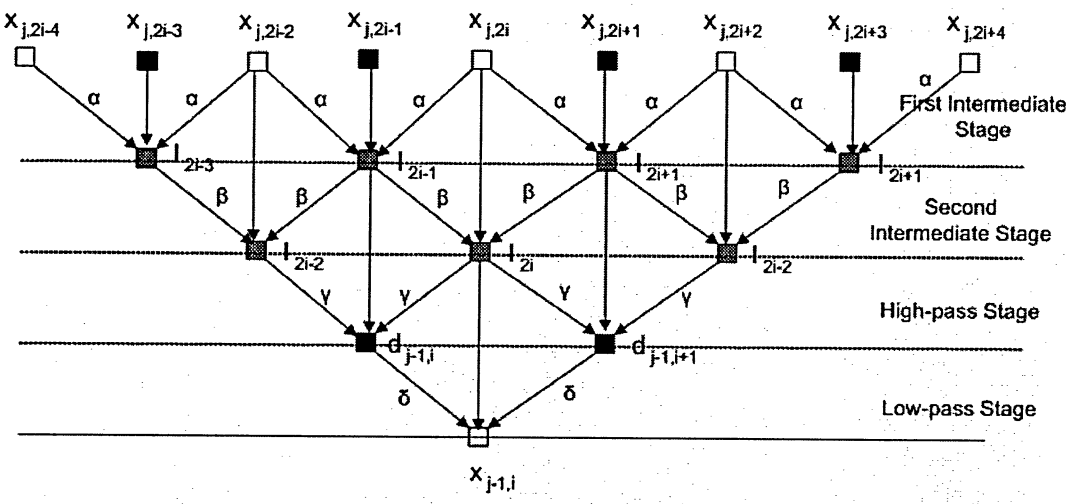


Fig.3.2: *Lifting Scheme for (9,7) wavelet without scaling*

From the equations (3.11) and (3.12), it is found that the product value of α times $x_{j,2i}$ calculated at the particular clock cycle is required at the next clock cycle. Similar redundancy in multiplying operation can be easily observed for the other stages from equations (3.13-3.18). Thus, the proposed architecture performs one multiplying operation in every clock cycle for each stage and the other multiplier output can be obtained from the previous clock cycle.

Similarly, the proposed architecture needs two multipliers each for predict and update modules in the case of (13,7) wavelet. Thus, the proposed architecture utilizes the

redundancy of the above mentioned arithmetic operation reducing the number of multipliers required.

3.3 The Proposed Architecture

In this section, the architecture for (5,3) and (9,7) wavelets are described in detail. This architecture can be used to carry out both forward and inverse discrete wavelet transform.

3.3.1 Predict Module

The predict module performs arithmetic calculation to determine detailed or high-pass coefficients, $d_{j-1,i}$ given in equation (3.1) for (5,3) wavelet. The general-purpose processor performs four memory accesses, two multiplications and two additions to implement predict module. The hardware can be designed to perform all the arithmetic operations in parallel. Again, one multiplication output is obtained from its calculation in the previous cycle. It can be easily seen that, in every clock cycle, the module needs only one new even input sample, $x_{j,2i+2}$, from the memory and uses the other even input sample, $x_{j,2i}$, that is read in the previous cycle. This is achieved by storing the even sample, $x_{j,2i}$, in the register. This reduces memory read to one for all even input samples required for high-pass coefficient calculation. However, the odd input sample, $x_{j,2i+1}$, has to be read from the memory simultaneously. This problem is overcome by using dual-port RAM to store the input samples. The read operations of even and odd samples

occupy both ports of the dual-port RAM but the detailed or high-pass coefficient obtained from the predict module has to be written back to the RAM in the same clock cycle. In order to write the detailed or high-pass coefficient to the RAM, the input dual-port RAM has to be operated twice the frequency of the system clock.

The predict module for (5,3) wavelet is shown in Fig.3.3. Initially, the input register R_1 is loaded with the even sample from the input RAM. In the meantime, the predict filter coefficient 'α' and the corresponding odd sample are made available to calculate the detailed coefficient $d_{j-1,i}$. The second register R_2 stores the output of the multiplier in the current cycle and in the meantime the register R_2 supplies the multiplier output obtained in the previous cycle. Thus we reduce the number of multipliers required for predict operation for (5,3) wavelet to one whereas the number required for the architecture described in [25] is two. Similarly, for (13,7) wavelet, only two multipliers required for predict module instead of four. For (5,3) wavelet, we can use shifters instead of multipliers. As pointed out in Section 2.1.2, the predict module for forward and inverse transform differs in only one arithmetic operation i.e. addition for forward transform and subtraction for inverse transform. The fw_iv signal shown in Fig. 3.3 determines whether the predict module operates forward or inverse transform.

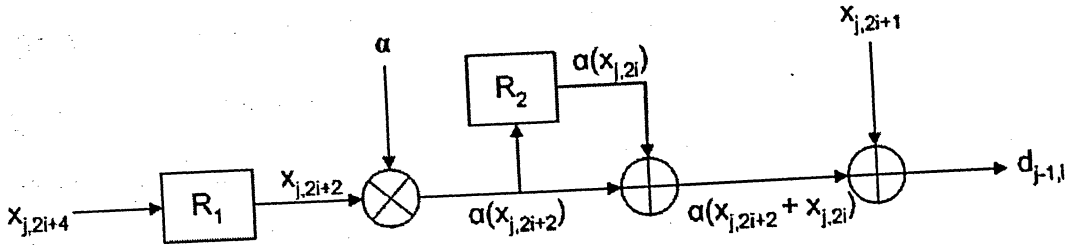


Fig.3.3: *Predict Module*

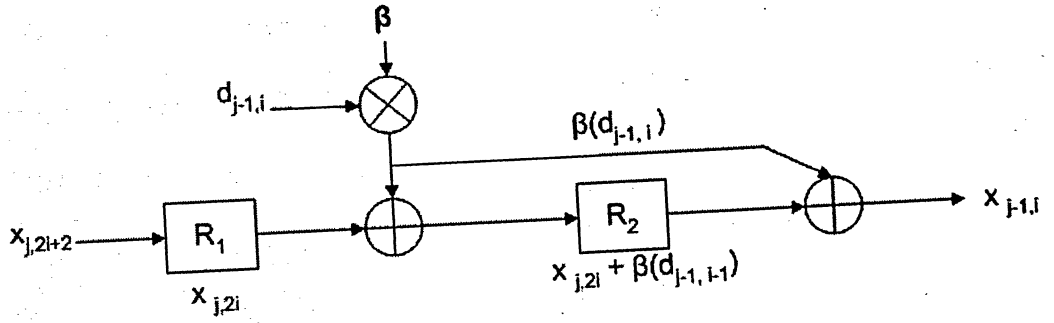


Fig.3.4: *Update Module*

3.3.2 Update Module

The update module calculates the low-pass or coarse coefficients, $x_{j-1,i}$ given in the equation (3.3) for (5,3) wavelet. Implementing update module in the general-purpose processor involves four memory accesses, two multiplications and two additions. The hardware is designed to perform all the arithmetic operations in parallel and only one multiplication is required to perform instead of two. The maximum number of memory accesses that could be carried out for each system clock cycle is four. But the predict module uses three memory accesses to the input RAM. The update module has to use the remaining one memory access to calculate and store coarse or low-pass coefficient in the same system clock cycle. The inputs of the update module are from the output of the

predict module and the even sample that is read from the input RAM for the predict module. Therefore, the output of the predict module and the even sample is fed directly to the update module instead of reading those values from the input memory again. Thus, the remaining one memory access to the input RAM is used to write the coarse or low-pass coefficient in the same system clock cycle.

The structure of the update module for (5,3) wavelet is shown in Fig. 3.4. The input register R_1 is loaded with the even sample. In the next clock cycle, the multiplier is fed with the detailed coefficient and the update coefficient 'β' and the output of the multiplier is fed to both the adders as shown in Fig. 3.4. Similarly, for (13,7) wavelet, the module needs only two multipliers for update module. In this case also, the multipliers can be replaced with shifters for (5,3) wavelet. The update module for forward and inverse transform differs by one arithmetic operation as mentioned in predict module. Thus, the fw_iv signal determines the type of the transform to be performed.

As mentioned in JPEG2000 [41], the signal is symmetrically extended by two signal values on the left side and one signal value on the right side for (5,3) wavelet and four signal values on the left side and three signal values on the right side for (9,7) wavelet to reduce artifacts at the boundary. The boundary treatment problem is solved by passing proper address to the proposed architecture.

3.3.3 Lifting-Based (5,3) Wavelet Architecture

The proposed DWT architecture consists of predict module, update module, control unit to generate proper address and set of registers to establish data

communication between the modules. The proposed architecture calculates and writes one high-pass coefficient and one low-pass coefficient to the memory in each system clock cycle.

The predict and update modules discussed in the sections 3.3.1 and 3.3.2 are interconnected with set of registers for (5,3) wavelet as shown in Fig. 3.5. The general-purpose processor executes predict and update calculation sequentially. whereas, in hardware, both calculations can be carried out simultaneously. But the problem in hardware design now is that the update module needs the outputs of the predict module in forward transform and the predict module needs the output of the update module in the case of inverse transform.

As discussed in the previous section, the predict module uses three memory accesses out of four available memory accesses to the input RAM whereas the update module requires three more memory accesses (two read operations for detailed coefficient and even sample and one write operation for coarse coefficient) to the input RAM. However, the memory accesses required for the update module can be reduced to one as given below:

1. The set of registers have been introduced between the output of the predict module, $d_{j-1,i}$, and the input (detailed coefficient) of the update module. These numbers of registers in between both modules pass on the correct detailed coefficient required at the input of the update module.

2. From equations (3.1-4), it can be seen that the update module uses the same even samples that the predict module uses. However, the even sample read from the input

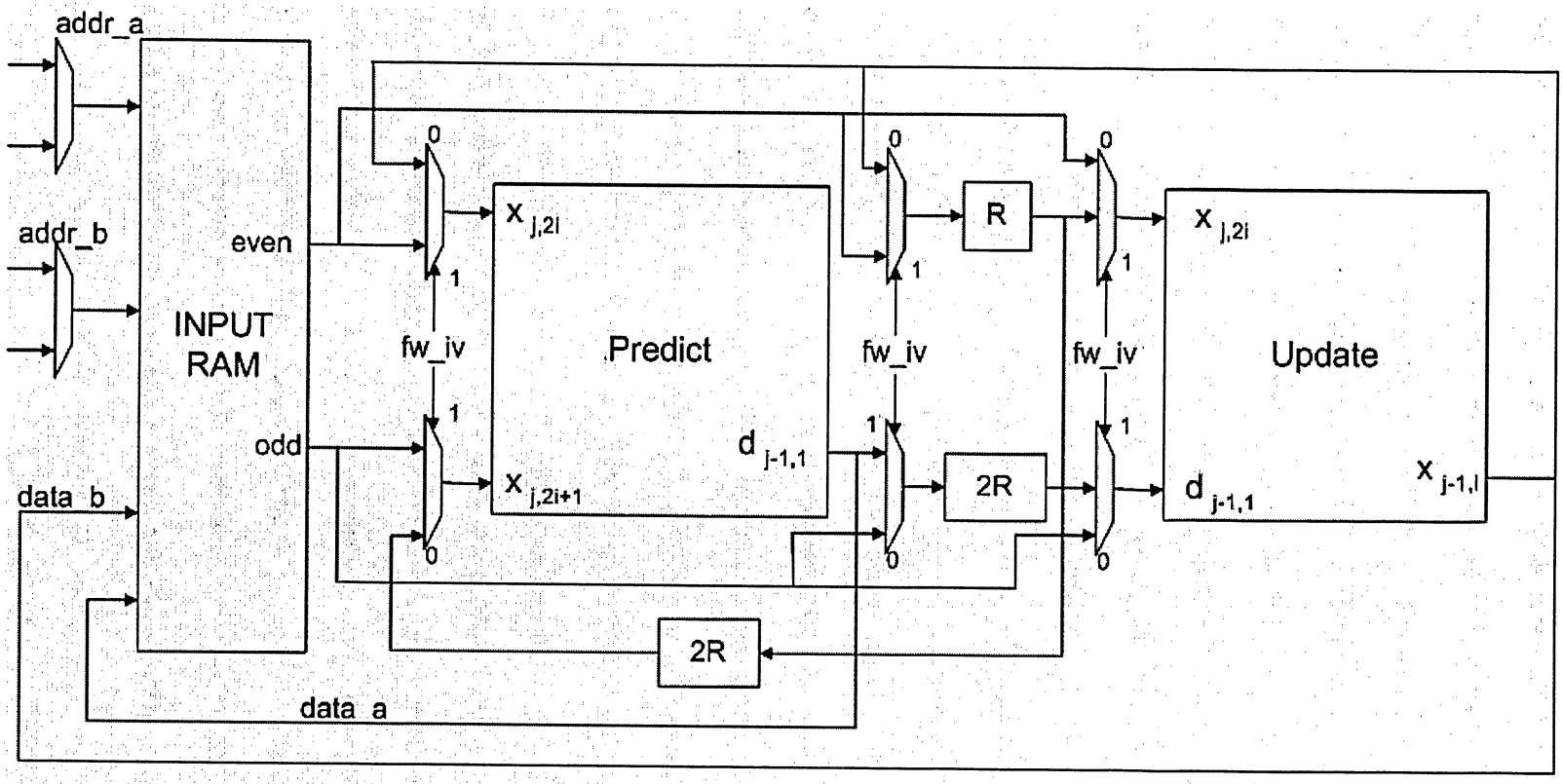


Fig.3.5: *Architecture for (5,3) Wavelet*

RAM is used by both the modules at different clock cycle i.e. the predict module uses it first and then the update module needs it. Thus, placing a few registers before the even sample is fed to the update module solves the synchronization or timing problem and the update module reuses the even sample read from the input RAM.

Thus, two read operations for the update module are avoided by using set of registers in between both modules. Only one memory access available to the input RAM for the update module is used to perform write operation of the coarse coefficient. Now, the proposed architecture is able to calculate one detailed coefficient and one coarse coefficient at every clock cycle. The proposed architecture operates dual-port RAM twice as fast as the system clock frequency to obtain detailed and coarse coefficients at every clock cycle. The first half of the system clock is used to write the detailed and coarse coefficients into the input RAM whereas the second half of the system clock is used to read even and odd samples from the input RAM. The technique discussed till this point is based on the forward transform.

For inverse transform, the two inputs for update module are read from the input RAM directly and the outputs of both predict and update modules are written back to the input RAM. Thus, all the four memory accesses available for one system clock frequency are used. The two inputs required for predict module is fed in a similar method applied to the update module in the forward transform.

The multiplexers placed before the set of registers, the predict module and the update module select the proper signal required by them based on the forward or inverse transform. The above described architecture performs lifting scheme for 1D signal. For the forward 2D transform, it starts with iteration level zero as current level (CL). The

forward 1D transform is first applied to all the rows and then to all the columns. Subsequently, the transform starts with the next level of iteration until the required number of decomposition level is accomplished. For the inverse transform, the inverse 1D transform is applied exactly in the opposite order. Flowcharts in Fig.3.6 illustrate the 2D forward and inverse transform.

3.3.4 Lifting-Based (9,7) Wavelet Architecture

The proposed lifting based architecture for (9,7) wavelet consists of one predict module to perform first intermediate stage, three update module to perform second intermediate, high-pass and low-pass stages, control unit to generate proper addresses and set of registers to establish data communication between modules. This architecture can be used to perform both forward and inverse DWT and DWPT.

The proposed architecture for 9/7 wavelet with four lifting steps is shown in Fig. 3.7. The scaling steps in 9/7 wavelet have not shown in Fig. 3.7. The set of registers is used in between the stages to properly pass the output of one stage to the next stage. Because of these registers, the architecture does not require any extra memory/FIFOs to store the intermediate results. This architecture uses dual-port input RAM which operates twice as fast as the system clock frequency to obtain the high-pass and low-pass coefficients at every clock cycle. The read and write operations to the input RAM and the data transfer between stages are similar to that explained for the (5,3) wavelet architecture. The forward or inverse transform is performed based on the value of fw_iv signal (0 or 1).

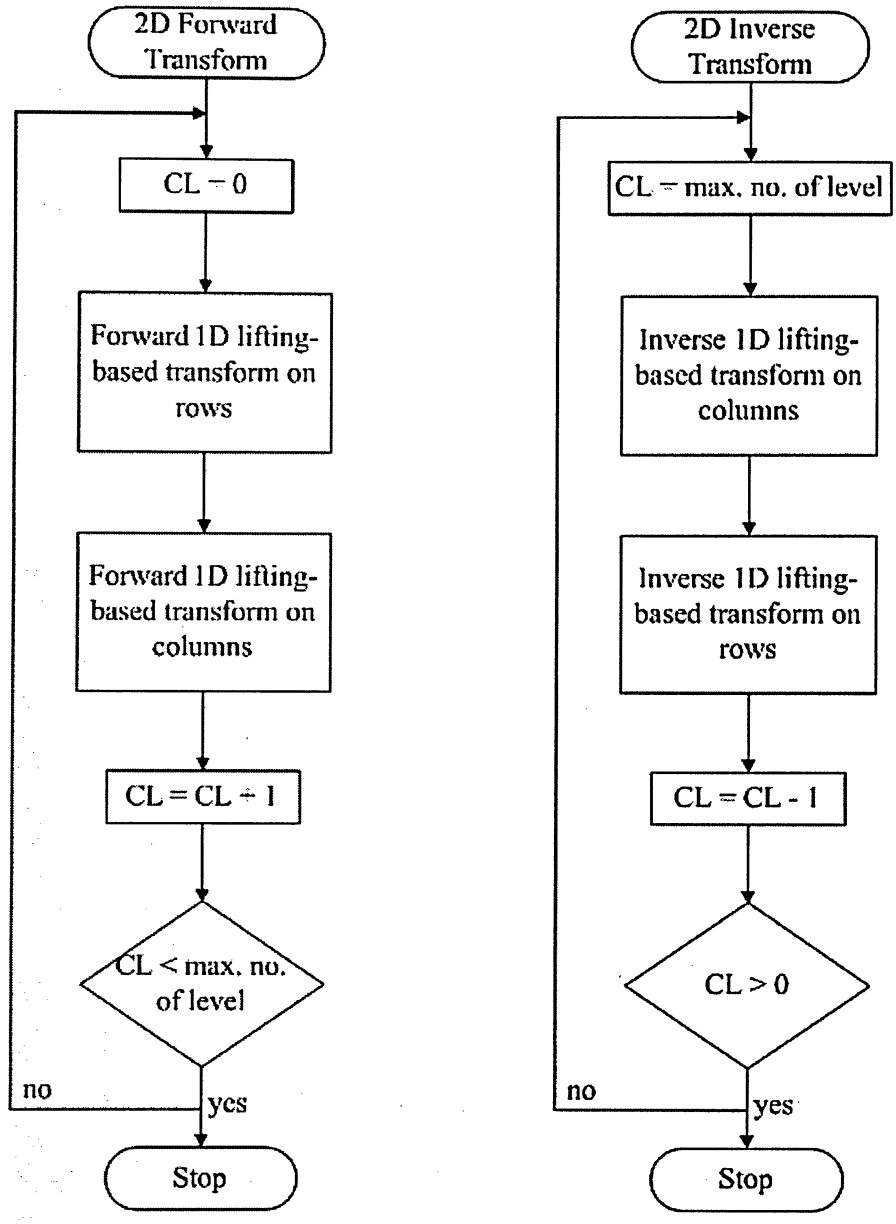


Fig.3.6: *Flowchart of the 2D Forward and Inverse Transform*

The lifting-based (13,7) wavelet architecture has one predict module, one update module, control unit and set of registers for data communication between the modules. However, both predict and update modules are not the same as that discussed in Sections

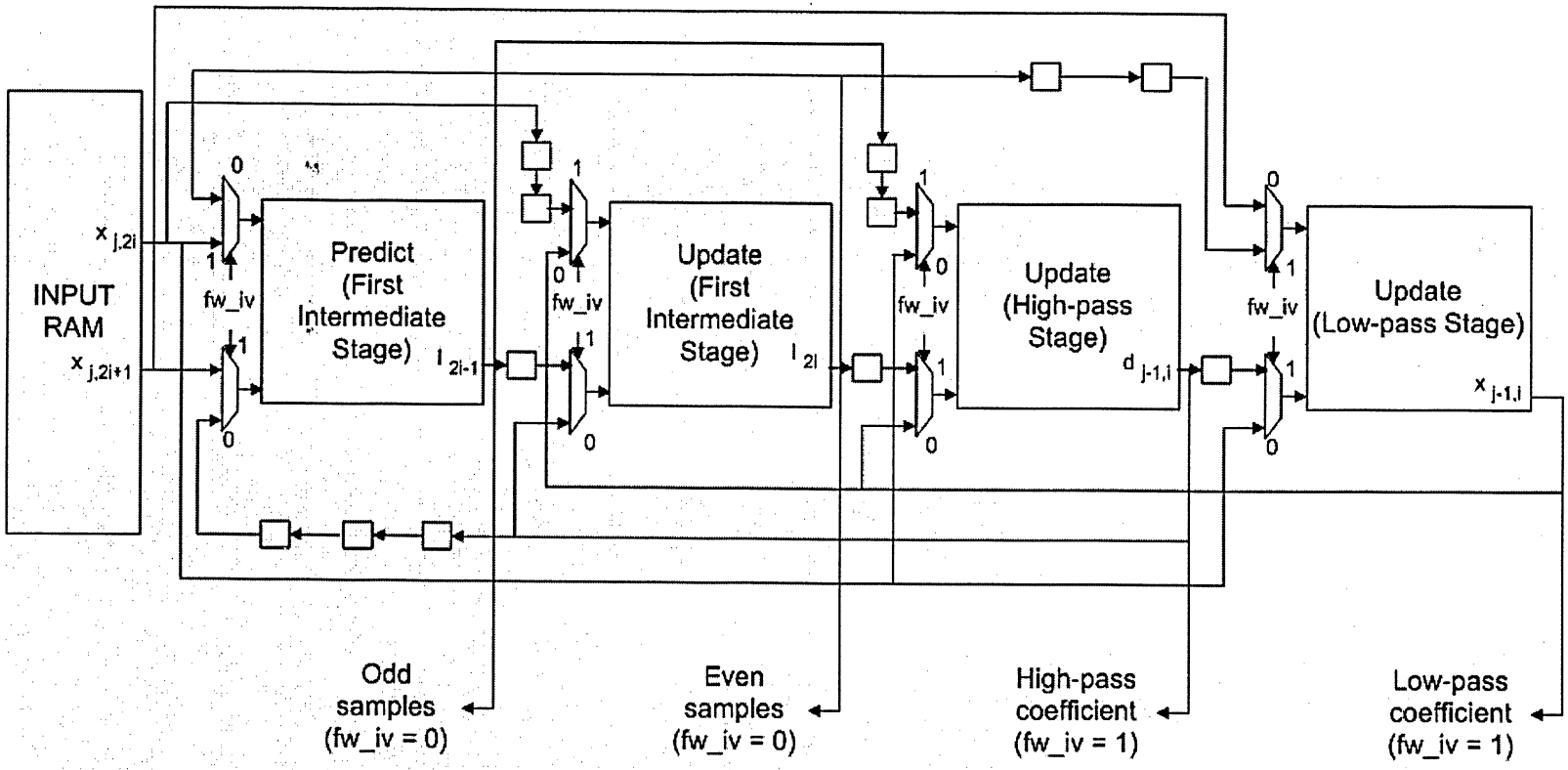


Fig.3.7: *Architecture for (9,7) wavelet*

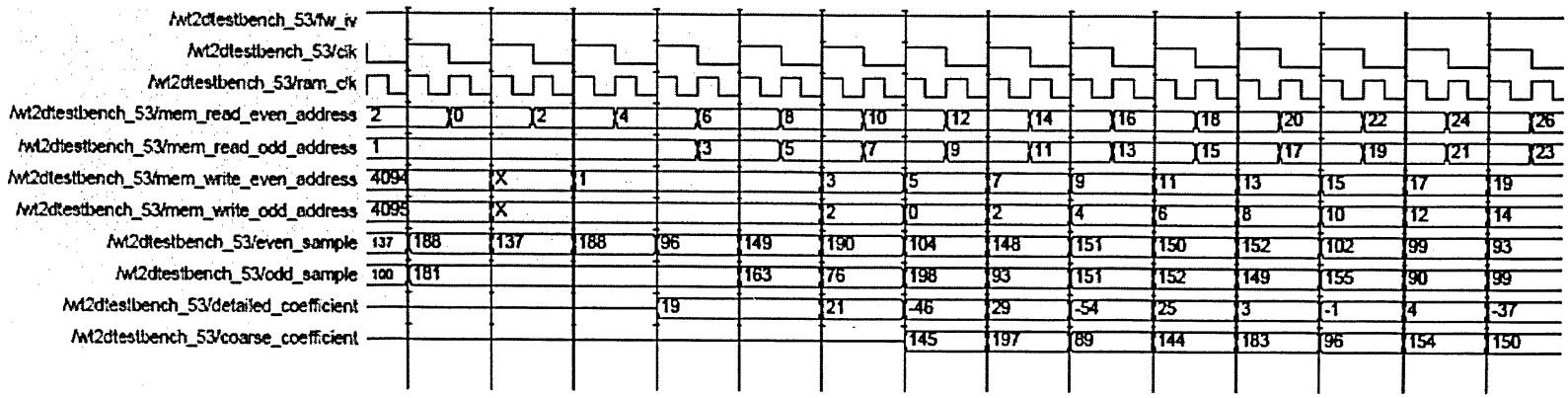


Fig.3.8: *Functional Output of Lifting-based (5,3) wavelet*

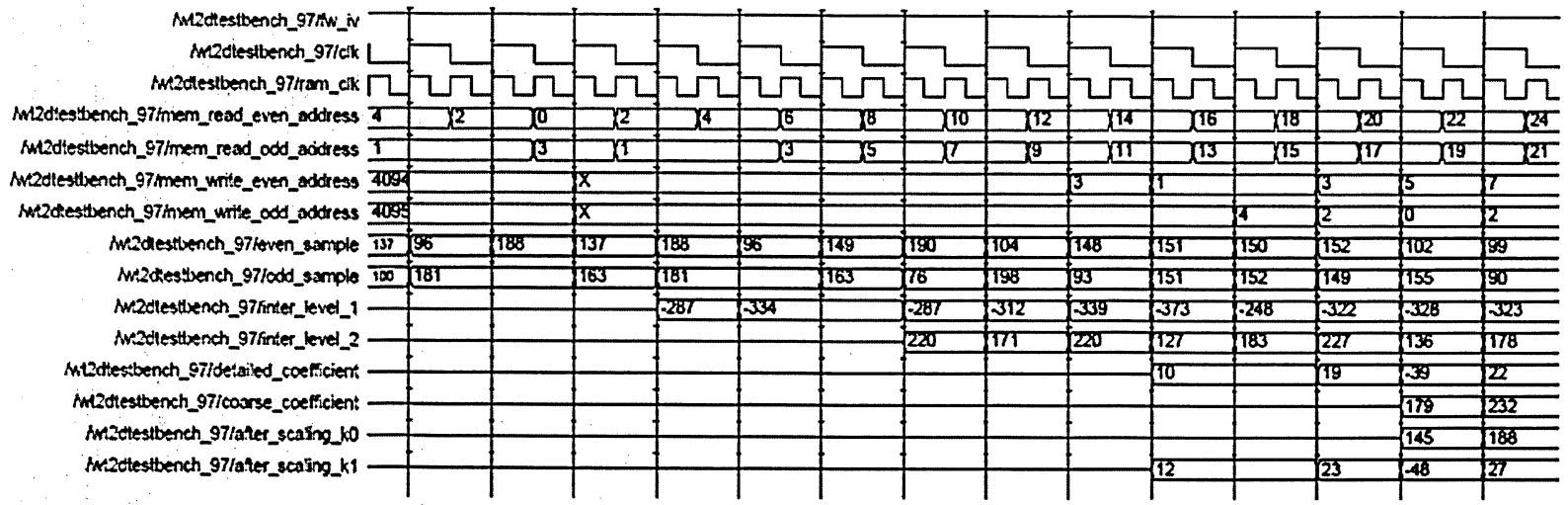


Fig.3.9: *Functional Output of Lifting-based (9,7) wavelet*

3.3.1 and 3.3.2. The predict module and the update module need two multipliers (shifters) to calculate detailed and coarse coefficients respectively. This architecture will look similar to that of (5,3) wavelet.

The proposed architecture for (5,3) wavelet is described in VHDL hardware description language for functional correctness and the waveform obtained from the simulator environment is shown in Fig.3.8. The "fw_iv" signal is set to one to perform forward DWT transform. The input RAM (dual-port) is operated at the frequency (ram_clk) twice that of the system frequency (clk). The even and odd samples are read from the input RAM when the system clock (clk) is low and the output of predict and update modules are written into the input RAM when the "clk" signal is high. The "mem_read_even_address" and "mem_read_odd_address" signals supply proper address to the input RAM to read the even and odd samples respectively when the system clock (clk) is low. The "mem_write_even_address" and "mem_write_odd_address" signals supply proper address to write the coarse (update output) and detailed (predict output) coefficients respectively when the system clock signal is high. In the first two clock cycle, the two consecutive even sample values (188 and 137 read from the even addresses 2 and 0 as shown in the Fig.3.8) are read form the input RAM and one odd sample (181 from the odd address 1) is read in the third clock cycle. The predict module calculates the detailed coefficient using those even and odd samples and writes the detailed coefficient in the input RAM (19 write into the address 1) in the fourth clock cycle. Similarly, the update module calculates the coarse coefficient and writes it in the input RAM in the seventh clock cycle (145 writes in the address 0). Since the whole process is pipelined, the predict and update modules write the detailed and coarse coefficients at every clock

cycle. Similarly, the proposed architecture for (9,7) wavelet is described in VHDL hardware description language for functional correctness and the waveform obtained from the simulator environment is shown in Fig.3.9. The signals shown in the Fig.3.9 are meant for the same functionality described above. But (9,7) wavelet includes four stages of lifting operation i.e. first intermediate, second intermediate, predict and update stages. The outputs of these stages are represented by the signals in Fig.3.9 are inter_level_1, inter_level_2, detailed_coefficient and coarse_coefficient signals respectively.

CHAPTER 4

ARCHITECTURE FOR BEST-BASIS ALGORITHM FOR 2D SIGNALS

4.1 Introduction

Best basis algorithm belongs to a class of entropy-based algorithms for efficient representation and signal compression. Classical entropy-based algorithms for best basis selection were first introduced by Coifman and Wickerhauser [8]. Best basis algorithm is primarily designed to reduce the storage space needed for a signal and also it can be used for de-noising. When the information content of the signal coordinates in the new basis is low it means that the distribution of coefficients is such that the energy of the signal is concentrated in a few coordinates. Another natural application of best basis algorithm is de-noising. This is achieved simply by finding a representation with a few significant terms (less entropy representation) so that one can neglect the coefficients less than a threshold value [51]. If signal to noise ratio is not too low one is able to extract the desired signal from the noisy one by threshold operation hence getting rid of unimportant

coordinates. In this chapter the hardware implementation of best-basis algorithm is discussed in detail.

This chapter is organized as follows. Section 4.2 discusses the algorithm and the architecture for natural logarithm implementation. The architecture for best basis algorithm and the architecture for best tree calculation for 2D signal are discussed in detail in the Section 4.3.

4.2 Algorithm and Architecture for Logarithm Function

4.2.1 Algorithm

The proposed algorithm is based on Maclaurin (or Taylor) series for natural logarithm (ln) function [43] which is given below:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots, -1 \le x \le +1 \qquad (4.1)$$

The main drawback in calculating logarithm function using Maclaurin series is that it involves more arithmetic (multiplication and addition) operations to get more accurate value. If $|x|$ is much smaller than one, the accurate value of $\ln(1+x)$ obtained from equation (4.1) involves lesser arithmetic operations.

The proposed algorithm uses the first two terms in Maclaurin series to obtain ln(1+x) as shown below:

$$\ln(1+x) = x - \frac{x^2}{2}, -1 \le x \le +1 \qquad (4.2)$$

To calculate natural logarithm of 'X' i.e. $\ln(X)$, where $X > 1$, divide X by another number R such that $|x'| = |X/R|$ is close to 1 so that $x = x'-1$ is very small. Thus,

$$\ln(X) = \ln(R) + \ln(x') \tag{4.3}$$

The constant ln(R) is read from the look-up table or memory. Since |x| is much smaller than one, the ln(x′) = ln(1+x) is calculated using the equation (4.2). The value of R is chosen depending upon the number of bits of accuracy needed after the decimal point of the logarithm output. The algorithm for natural logarithm implementation is explained below:

Step 1: Determine the range R_{max}, $2^n \ldots 2^{n+1}$, where n is an integer, such that $2^n < X \leq 2^{n+1}$.

Step 2: To obtain $4l$ bits or l decimal digits accuracy after decimal point of natural logarithm of X, divide the range R_{max} into $m = 2^{l-1}$ equal sub-ranges $R_1, R_2, \ldots, R_n$.

$$\text{i.e. } R_1 \rightarrow 2^n \cdots \left[\frac{2^{n+1} - 2^n}{m}\right] + 2^n$$

$$R_1 \rightarrow 2^n \cdots \left[\frac{2^n}{m} + 2^n\right]$$

$$R_2 \rightarrow \left[2^n + \frac{2^n}{m}\right] \cdots \left[2^n + \frac{2.2^n}{m}\right]$$

$$R_3 \rightarrow \left[2^n + \frac{2.2^n}{m}\right] \cdots \left[2^n + \frac{3.2^n}{m}\right] \ldots.$$

$$R_m \rightarrow \left[2^n + \frac{(m-1)2^n}{m}\right] \cdots 2^{n+1} \tag{4.4}$$

Step 3: Select the range R_x in which X falls.

$$\text{i.e.}\left[2^n + \frac{(x-1)2^n}{m}\right] < X < \left[2^n + \frac{x.2^n}{m}\right] \tag{4.5}$$

Step 4: Find the mid-point of the range, R_X.

$$R_X(\text{mid}) = \frac{1}{2}\left[\left(2^n + \frac{(x-1)2^n}{m}\right) + \left(2^n + \frac{x.2^n}{m}\right)\right] \tag{4.6}$$

Step 5: Normalize X in the range R_X as given below:

Let x′ be normalized X.

$$x' = \frac{X}{R_X(\text{mid})} \tag{4.7}$$

The natural logarithm of X is calculated by

$$\ln(X) = \ln[R_X(\text{mid}).x'] = \ln[R_X(\text{mid})] + \ln(x'). \tag{4.8}$$

The ln(x′) is obtained using equation (4.2) whereas ln[R_x(mid)] is read from look-up table or memory. The proposed algorithm for logarithm implementation is a non-iterative method applying Maclaurin series and the flowchart explaining the proposed algorithm is shown in Fig.4.1. For example, calculate natural logarithm of X = 184 with *l* = 2 decimal digit accuracy after decimal point from the above algorithm.

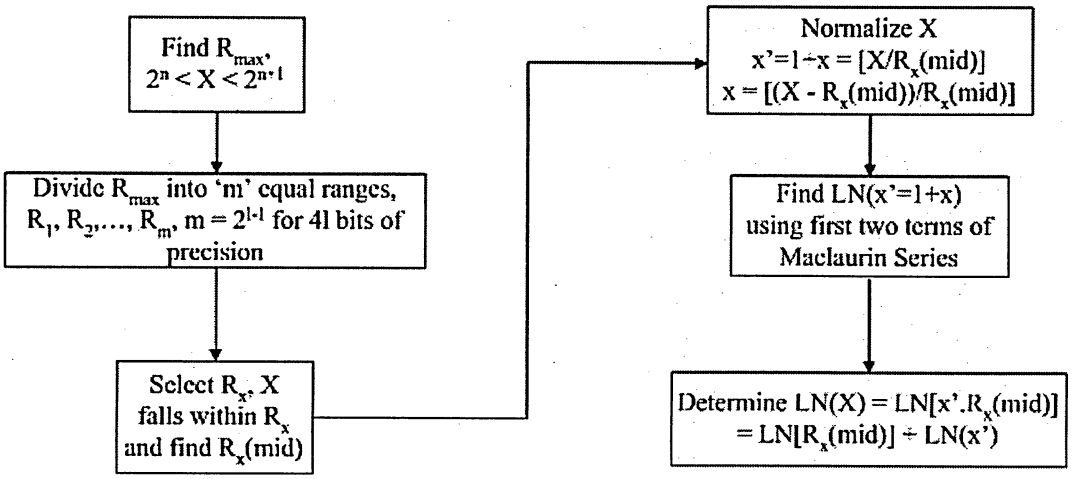


Fig.4.1: *Flowchart of Natural Logarithm implementation using Maclaurin series to find LN(X)*

Step 1: The range R_{max} is (2^7 ... 2^8).i.e. 128 < 184 <256.

Step 2: Number of sub-ranges, m = 2. The first range R_1 is 128 ... 192 and the second range is R_2 is 192 ... 256.

Step 3: The selected range R_x is R_1 because X (=184) falls within the range of R_1 i.e. 128 < X < 192.

Step 4: The mid-point of the selected range R_1 is 160 i.e. R_1(mid) = 160.

Step 5: x = x′ - 1 = (184-160)/160 = 24/160 and ln(184) = ln(160) + ln(1+x)=5.2139. The expected value is 5.2149 and the error is 0.0010. It is evident from the error value that the natural logarithm value of X is accurate for 8 bits or two decimal digits after the decimal point.

The above example clearly explains the computation of natural logarithm value for integer numbers. We present another example to compute natural logarithm for the values of X less than one. To calculate the natural logarithm for X = 0.0512 with 16-bit

of precision i.e. four decimal digit of precision, the proposed algorithm needs a little modification which is explained with the example.

Step 1: Determine the range R_{max} in which X falls. The range R_{max} is $(2^{-4} \ldots 2^{-5})$.i.e. $2^{-4} < 0.0512 < 2^{-5}$.

Step 2: The number of sub-ranges to obtain 16-bit of precision is m = 8 or $2^{(I=3)}$. The ranges are

R_1 - 0.03125 – 0.03515625

R_2 - 0.03515625 – 0.0390625

R_3 - 0.0390625 – 0.04296875

R_4 - 0.04296875 – 0.046875

R_5 - 0.046875 – 0.05078125

R_6 - 0.05078125 – 0.0546875

R_7 - 0.0546875 – 0.05859375

R_8 - 0.05859375 – 0.0625

Step 3: The selected range is R_6.i.e. $0.05078125 < X < 0.0546875$.

Step 4: The mid-point of selected range is $R_6(mid) = 0.052734375$.

Step 5: The value of x is $(X/R_6(mid) - 1) = -0.029096$. The $\ln(1+x) = -0.02952$ obtained using the equation (Y.2). Thus, the value of $\ln(X) = \ln(R_6(mid)) + \ln(1+x) = -2.9720$. The actual value of ln(X) is -2.9720157 and the absolute error is 0.0000157.

The last example presented below explains the computation of natural logarithm for mantisaa part of the floating point representation [44]. The computation of ln(X = 1.5120) with 16-bit or four decimal digit of precision is explained below:

Step 1: The range R_{max} is always in the range from $2^{16}/2^{15}$ to $2^{15}/2^{15}$.i.e. $1 < X < 2$.

Step 2: The number of sub-ranges to obtain 16-bit of precision is m = 8 or $2^{(I=3)}$. The ranges are

R_1 - 1.000 – 1.125

R_2 - 1.125 – 1.250

R_3 - 1.250 – 1.375

R_4 - 1.375 – 1.500

R_5 - 1.500 – 1.625

R_6 - 1.625 – 1.750

R_7 - 1.750 – 1.875

R_8 - 1.875 – 2.000

Step 3: The selected range is R_5 (1.500 – 1.625).

Step 4: The mid-point of the selected range is $R_5(mid) = 1.5625$.

Step 5: The value of x is $(X/R_5(mid) - 1) = -0.03232$. The $\ln(1+x) = -0.03284$ is obtained from equation (Y.2). Thus, the value of $\ln(X) = \ln(R_5(mid)) + \ln(1+x) = 0.4134$. The actual value of ln(X) is 0.413433 and the absolute error is 0.000033.

4.2.2 Architecture

The proposed architecture of natural logarithm implementation consists of three stages: Selection-of-value stage, Normalization stage and Log-unit stage. The architecture involves two multiplication and three addition operations irrespective of the number of

bits of accuracy required. The architecture for natural logarithm with 8-bit or two decimal digit accuracy after the decimal point is shown in Fig.4.1.

Selection-of-value Stage: This stage supplies the value of $R_x(mid)$, inverse of $R_x(mid)$, $\ln[R_x(mid)]$ and R_x for all possible sub-ranges of R_{max} from the look-up table or memory to the next stage. As mentioned in the proposed algorithm, the number of sub-ranges depends on the number of bits of accuracy needed after decimal point. For example, the number of sub-ranges is two for the 8-bit accuracy after decimal point of the natural logarithm function. Also it supplies the number of shifts that the shifters have to carry out in the next two stages. Finally, this stage selects the $R_x(mid)$, inverse of $R_x(mid)$, $\ln[R_x(mid)]$ and R_x corresponding to the range R_x where X falls.

Normalization Stage: It also finds the normalized X i.e. x′. The $x = x' - 1$ is obtained by first subtracting X with $R_x(mid)$ and then multiplying the difference with the inverse of $R_x(mid)$ selected in this stage. Thus this stage involves one subtractor and one multiplier.

Log-unit Stage: This stage calculates $\ln(1+x)$ given in the equation (4.2). This involves one square and one subtraction operations. This stage also includes one more addition operation to get the final natural logarithm value of $\ln(X)$.

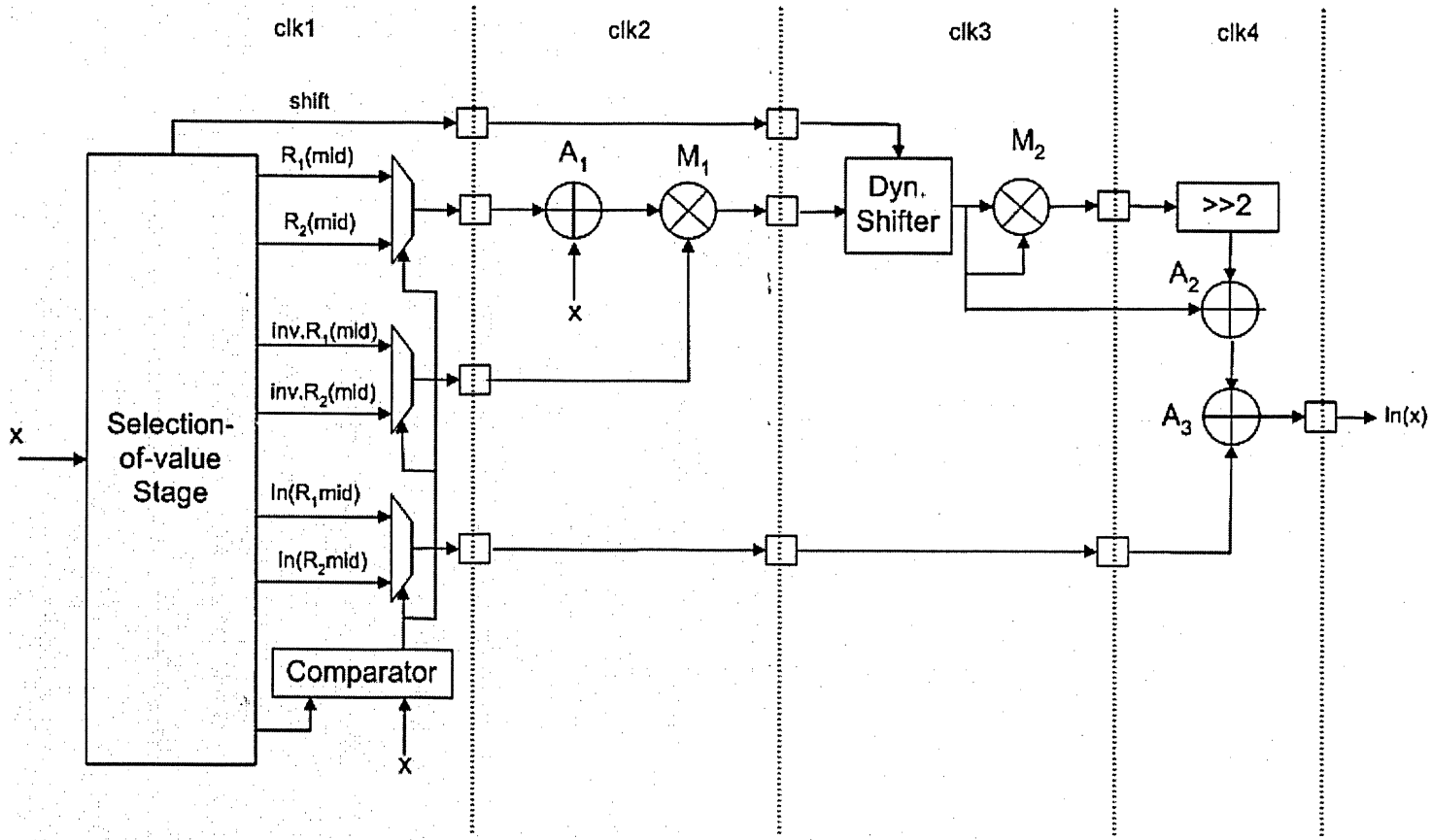


Fig.4.2: *Natural Logarithm Architecture with 8 bits of accuracy*

The natural logarithm architecture with 11 bits of accuracy is described in VHDL hardware description language and simulated for functional correctness as shown in Fig.4.2 with the ModelSim simulator.

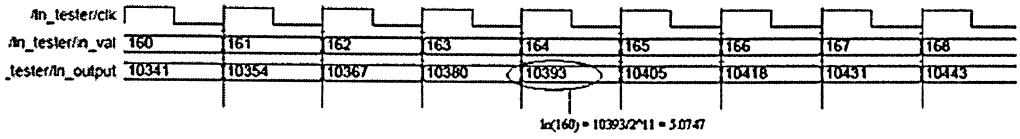


Fig.4.3: *Timing Diagram of Natural Logarithm Architecture with 8 bits of precision*

4.3 Architecture for Best-Basis Algorithm

The best-basis algorithm with detailed example for 2D signal was explained in Section 2.2. In this section, the architecture to implement the best-basis algorithm for 2D signal from two-level wavelet packet decomposition is discussed in detail. The proposed architecture for best-basis algorithm is classified into three stages. The block diagram of the proposed best-basis architecture is shown in Fig.4.3. In the first stage, the WPT decomposition is carried out using the architecture of lifting-based DWT proposed in Chapter 3. The second stage calculates the cost function using the architecture of the logarithm proposed in Section 4.2. The output of the cost function is written into another dual-port RAM called Best-basis RAM. In the final stage, the architecture for best-basis selection, which is discussed later in this section, determines the best-basis node. The proposed best-basis architecture includes the following sub-modules or sub-architectures:

- Wavelet Packet Architecture (discussed in detail in Chapter 3),
- Best-Tree Selector Architectures,

- Dual-port Best-Basis RAM with Address Generator module,
- Cost-Function Architecture – Architectures for two different cost functions are discussed in this section. Those two cost function architectures are Threshold-Function architecture and Shannon-Function architecture.

4.3.1 Best-Tree Selector Architecture

The best-tree selector architecture for 2D signal is shown in Fig.4.4. The input to the architecture is fed from the dual-port RAM, named best-basis RAM. For 2D signal, each parent node is linked to four children node. Thus, the cost of all the four children nodes have to be added before it is compared with the cost of the parent node. The architecture reads the cost of the two children nodes from the best-basis dual-port RAM in the first clock-cycle and the cost of the other two nodes are read in the second clock cycle. In the second clock cycle, the cost of the first two children nodes are added and stored in the register R_1. In the third clock cycle, the cost of the second two children nodes are added and stored in the register R_2. In the fourth clock cycle, the cost of the parent node is read from the best-basis RAM and stored in the register R_4. The values in the registers R_1 and R_2 are added and stored in the register R_3 in the fourth clock cycle. In the fifth clock cycle, the total cost of the children nodes stored in the register R_3 and the cost of the parent node stored in the register R_4 are compared. If the total cost of the children nodes is less than that of the parent node cost, all the four children nodes are

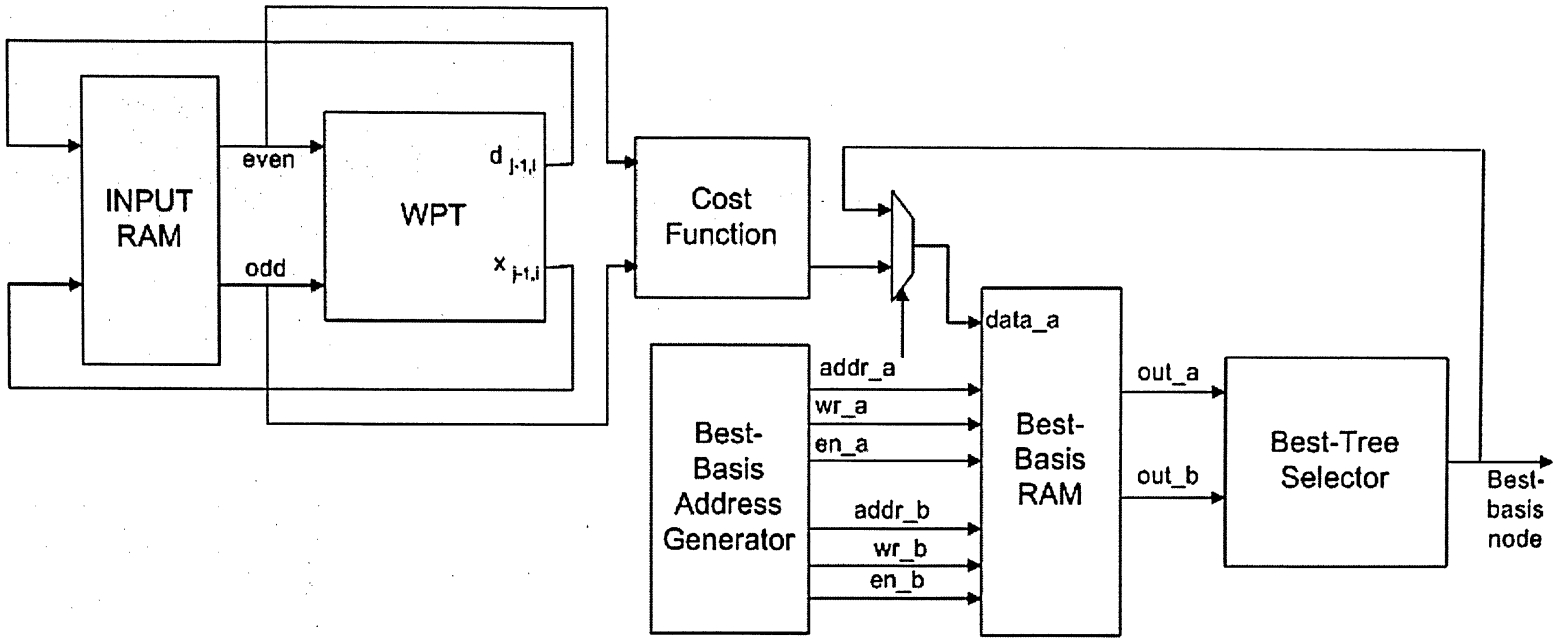


Fig.4.4: *Best-Basis Architecture*

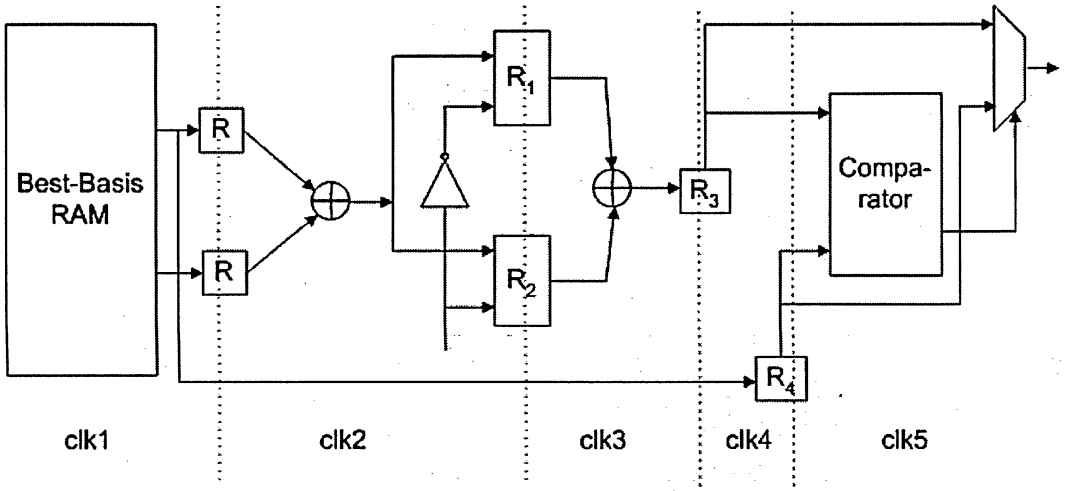


Fig.4.5: *Best-Tree Selector*

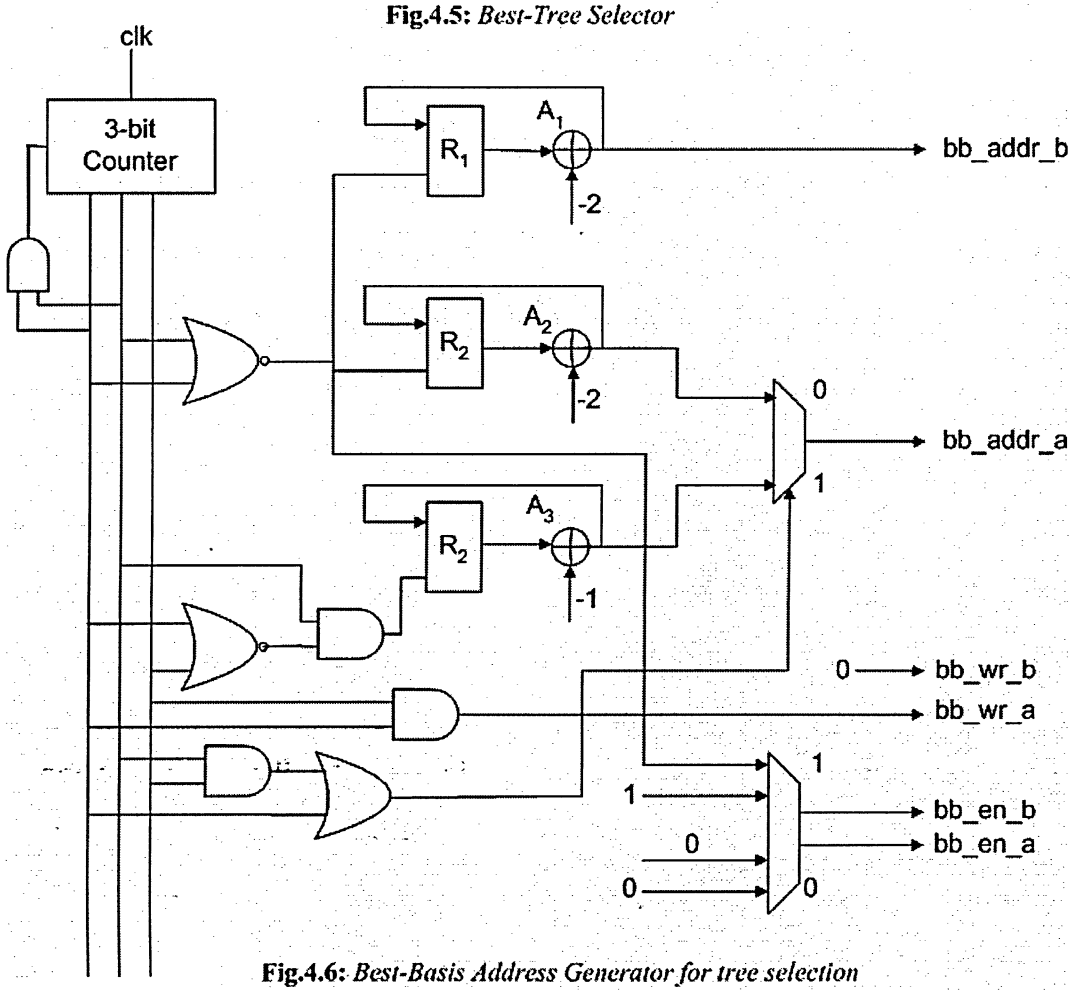
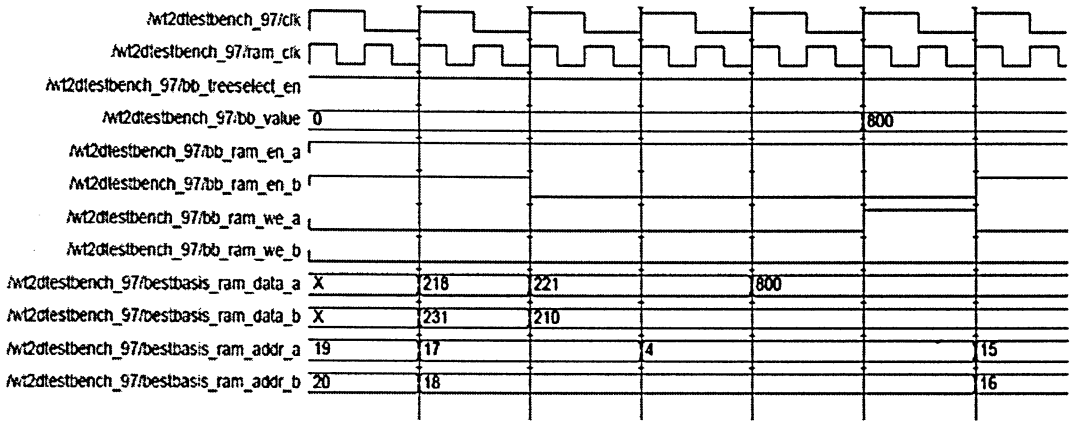


Fig.4.6: *Best-Basis Address Generator for tree selection*

selected and vice versa. In the last clock (sixth) cycle, the parent node cost is updated in the best-basis RAM. The waveform for the best-tree selector obtained from the ModelSim simulator environment is shown in Fig.4.6.



bb_value - Selected parent node cost to be written in the best-basis RAM.
bestbasis_ram_addr_a - represents the signal "bb_addr_a".
bestbasis_ram_addr_b - represents the signal "bb_addr_b".
bestbasis_ram_data_a - The data read from the address "bb_add_a".
bestbasis_ram_data_b - The data read from the address "bb_add_b".
bb_ram_en_a - represents the signal "bb_en_a".
bb_ram_en_a - represents the signal "bb_en_b".

Fig.4.7: *Waveform of Best-Tree Selector*

4.3.2 Address Generator for Best-Tree Selector Architecture

The Address Generator for best-tree selector architecture consists of 3-bit counter, three subtractors and combinational circuits to generate signals for the best-basis RAM at appropriate time intervals. The address generator is shown in Fig.4.5. The signals generated at each counter value are explained in detail in Table 4.1. In the first two clock cycles (count = 0 and 1), the cost of the four children nodes is read from the best-basis RAM using the adders A_1 and A_2. In the third clock cycle (count = 2), the parent-node address is generated from the adder A_3. From now onwards until the count value is reset to zero, the parent address is set to the port-A of the best-basis RAM. In the sixth clock

cycle (count = 5), the write-enable of port-A is set to one and the selected cost value from the best-basis tree selector architecture is written to the parent-node address. The counter is reset to zero in the next clock cycle.

	3-bit counter output	Signals generated
	000, 001	• First pair of child-node addresses is generated when count is zero and the second pair of child-node addresses is generated when count is one. • The enable signal (bb_en_b) for port-B of the best-basis RAM is set one. • The addresses (bb_addr_a and bb_addr_b) for port-A and port-B are set to odd (decr_child_odd_addr) and even (decr_child_even_addr) addresses of the children nodes respectively. • The adders A1 and A2 generate even and odd addresses of the children nodes.
	010	• The total cost of four children-nodes is calculated.
	011, 100	• The address (bb_addr_a) for port-A is set to parent-node address.
	101	• Set the write enable signal of port-A is set to one. The selected cost from the best-tree selector architecture is written to the best-basis RAM during this clock-cycle.
	110	• Reset the counter.

Table 4.1: *Signals generated at each counter value for Best-Tree selection*

4.3.3 Cost-Function Architecture

The best basis algorithm finds a set of wavelet bases that provide the most desirable representation of the data relative to a particular cost function. The cost function may be chosen to fit a particular application. In this section, the architectures for two cost functions namely, threshold function and Shannon function, are proposed.

Threshold-Function Architecture

One of the simplest cost-function of the best-basis algorithm is a threshold function. The threshold function counts the number of wavelet coefficients in a particular wavelet packet node whose absolute value is greater than a threshold value 't' as shown in equation (4.9).

$$\cos t = \sum_{i=0}^{N-1} (|x(i)| > t)?1:0 \qquad (4.9)$$

To calculate the best basis, the tree is traversed from the bottom and each node is assigned with its cost value relative to the threshold function in this case. When the wavelet packet tree is constructed, all the nodes are marked with flags (say, flag is set to one, if it is selected). The flag of the nodes is modified on calculating the best-basis set. Assume P1 and C1 as the cost of the parent node and the sum of the cost of children nodes associated with the parent node respectively. The best-tree architecture performs the following to find the best-basis set:

- If P1 $\leq$ C1, set the flag of the parent node to one as part of the best-basis set and also set all the flags of the nodes in the sub-tree of the parent node to one.
- If P1 > C1, the cost of the parent node P1 is replaced with the sum of the cost of children nodes C1.

The threshold function architecture consists of two comparators and two adders as shown in Fig.4.7. In each clock cycle, the even and odd values fed to the architecture are

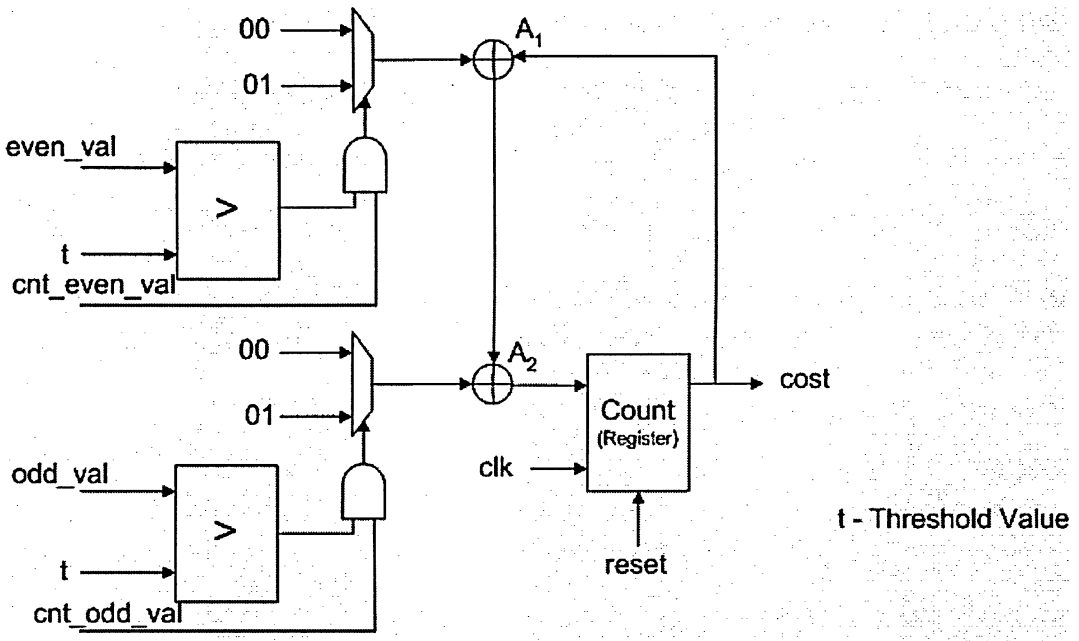


Fig.4.8: *Threshold Function Architecture*

compared with the threshold value. If the even value is greater than the threshold value and the cnt_even_val signal (this signal decides if the valid even value fed to the architecture) is one, the count value is incremented by one using adder A_1. The output of A_1 is then fed to the next adder A_2. If the odd value is greater than the threshold value and the cnt_odd_val signal is set high, the output of A_1 is incremented and stored in the flip-flop. The output of the flip-flop gives the cost of the node.

Shannon-Function Architecture

Another important cost function of the best-basis algorithm is the Shannon entropy function. The Shannon entropy function provides a measure of the economy of the representation of the signal. The Shannon entropy is given as

$$cost = \sum_{i} x^2[i] \ln(x^2[i]) \qquad (4.10)$$

The Shannon-function consists of one natural logarithm unit discussed in Section 4.2, two multipliers and one adder and it is shown in Fig.4.8(a). Since the output of the log-unit is available at the fourth clock cycle, the square value of input is passed through three registers to make it available at the multiplier M_2. The cost of all the coefficients is summed together and stored in the flip-flop. The cost function architecture for Shannon function is shown in Fig.4.8(b) using two blocks of Shannon function architecture. This architecture calculates the total cost of the wavelet packet node.

After best-tree selection, the information about the nodes to be reconstructed is obtained using the flag set for each wavelet packet nodes. The best-basis nodes are actually reconstructed by applying inverse WPT selectively according to the flag of each node. If the flag of the node 'i' is equal to one, the particular node is reconstructed from the four children nodes associated with it. If the flag of the node 'i' is set to zero, its reconstruction from its four children nodes is avoided. Thus the wavelet packet nodes selected from the best-tree selection are constructed. In order to construct the original signal from the best-basis nodes, reconstruct the nodes whose flags are set to zero. This gives the original signal selected from the best-basis nodes.

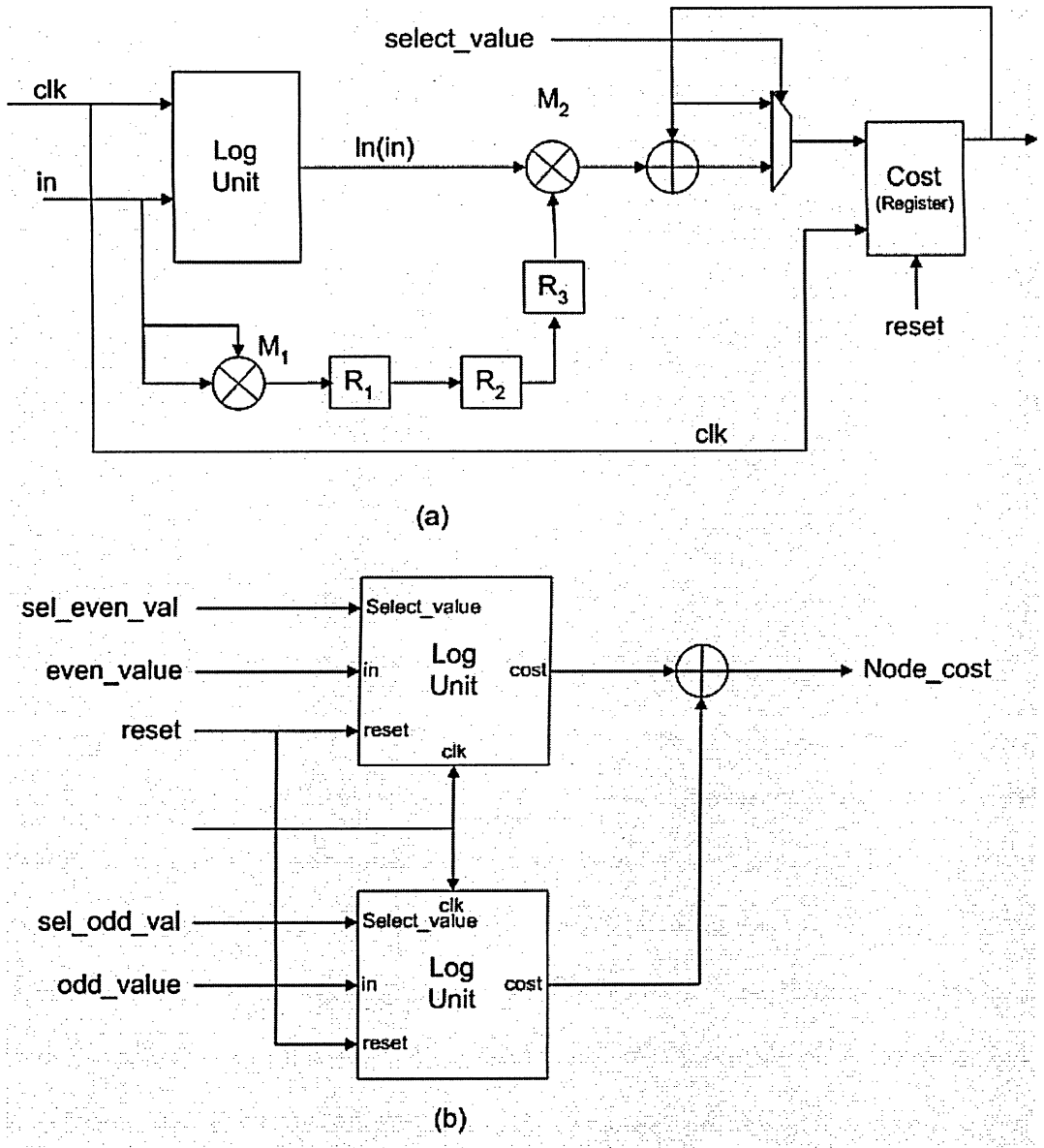


Fig.4.9: *Shannon Function Architecture*

CHAPTER 5

RESULTS AND DISCUSSION

5.1 DWT

The proposed architecture can implement lifting based DWT for each level sequentially. This architecture does not require any external storage devices to store the intermediate results and thus avoids delay caused by memory access. Because a set of registers controlled by global clock is being used, the control unit does not need to take the intermediate results in and out of the extra storage device. The output data rate of this architecture is two (one from predict module and the other from update module) per clock cycle.

The performance analysis is studied in terms of hardware (multipliers and adders) requirement and computation time for (5,3), (9,7) and (13,7) wavelets. Because the set of registers controlled by the clock is employed, the architecture does not require any extra memory/FIFO to store the intermediate results. Table 5.1 provides the comparative evaluation of the proposed architecture with other architectures [24-25] in terms of area and computation time for one level of decomposition of the signal of size NxN.

The proposed architecture needs less number of multipliers compared to other architectures proposed in [24- 25]. The proposed architecture utilizes less number of multipliers compared to the architectures mentioned in [24]. The architecture proposed in [24] has better computation time than the proposed architecture in the case of (5,3) and (9,7) wavelets but it requires same computation time approximately as the proposed architecture for (13,7) wavelet. Furthermore the architecture proposed in [24] requires more hardware area for (5,3) and (13,7) wavelet. The main advantage of the proposed architecture is that it utilizes less number of multipliers compared to other architectures.

Architectures	Multiplier/ Shifter	Adder	Intermediate Memory/FIFO	Computation Time
(5,3) Wavelet				
Andra [24]	4	8	Memory required	≈(N/2)xN
Kuzma[25]	4	4	FIFO required	≈(NxN)
Proposed	2	4	None	≈(NxN)
(9,7) Wavelet				
Andra [24]	4	8	Memory required	≈(N/2)xN
Kuzma[25]	8	8	FIFO required	≈(NxN)
Proposed	4	8	None	≈(NxN)
(13,7) Wavelet				
Andra [24]	8	16	Memory required	≈(NxN)
Kuzma[25]	8	8	FIFO required	≈(NxN)
Proposed	4	8	None	≈(NxN)

Table 5.1: *Comparison of the proposed DWT architecture with existing ones.*

The accuracy of DWT coefficients depends on the precision of both the input data and the filter coefficient. To explore the effects of precision on the proposed design, test images (Lena, Barbara, Zelda and Peppers available at http://cmp.felk.cvut.cz/~fojtik/gallery/gray.htm) of size 512 x 512 are used to verify the architecture. In VHDL simulation, the original image and the reconstructed image after forward and inverse DWT are compared. The simulation results for error calculation per pixel in terms of four different measures are computed and shown in Tables 5.2 – 5.3 with zero bit precision of input values and in Tables 5.4 – 5.7 with 8 bits of precision of input values. The four types of measuring error calculation are discussed in brief below:

- Average absolute error (AAE) per pixel – is the ratio between the sum of the error values in each pixel and the total number of pixels.

$$AAE = \frac{1}{N*N}\sum_{x=1}^{N}\sum_{y=1}^{N}|e(x,y)|, \tag{5.1}$$

  where e(x,y) is the error of the pixel (x,y) between reconstructed image and original image of size N*N.

- Average mean square error (MSE) per pixel [45] – is the ratio between the sum of the square of the error values in each pixel and the total number of pixels.

$$MSE = \frac{1}{N*N}\sum_{x=1}^{N}\sum_{y=1}^{N}\left|e^2(x,y)\right| \tag{5.2}$$

- Average signal-to-noise ratio (SNR) in dB [46] – is the power ratio between the signal and the background noise.

$$SNR(dB) = 10\log_{10}\left(\frac{\sum_{x=1}^{N}\sum_{y=1}^{N} I^2(x,y)}{\sum_{x=1}^{N}\sum_{y=1}^{N}\left|e^2(x,y)\right|}\right), \quad (5.3)$$

where I(x,y) is the pixel value at (x,y) of the image I.

- Average peak signal-to-noise ratio (PSNR) [47] in dB – is the power ratio of the maximum pixel value of the signal and the background noise.

$$PSNR(dB) = 10\log_{10}\left(\frac{MAX(255)^2}{MSE = \frac{1}{N*N}\sum_{x=1}^{N}\sum_{y=1}^{N}\left|e^2(x,y)\right|}\right) \quad (5.4)$$

The average error values per pixel for the test images recovered after one, two and three level of (9,7) wavelet packet decomposition with zero bit precision of input values are shown in Tables 5.2, 5.3 and 5.4 respectively. From these tables, it is evident that the increase in the precision of the filter coefficients of the proposed architecture does not minimize the error after certain precision level. Comparing the error values obtained for the precision of filter coefficient more than or equal to 18, the error values remain the same. It concludes that the error cannot be reduced further with increase in the precision of more than 18 without increasing the precision of the input values. For further reduction in the error, the precision of the input values may have to be increased. For 8 bits precision of input values, the calculated average error value is very less. Therefore, depending upon the accuracy needed the precision of both input values and lifting coefficients have to be increased. But the increase in the precision of the input values increases the size of the input RAM required.

Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	2.226	8.188	33.342	14.934
13	2.159	7.816	33.545	15.136
16	2.154	7.790	33.559	15.150
18	**2.153**	**7.783**	**33.563**	**15.153**
20	**2.153**	**7.783**	**33.563**	**15.153**
(a) For "Lena" Image				
11	2.218	8.169	32.602	14.944
13	2.168	7.902	32.746	15.088
16	2.164	7.877	32.760	15.102
18	**2.164**	**7.877**	**32.760**	**15.102**
20	**2.164**	**7.877**	**32.760**	**15.102**
(b) For "barbara" Image				
11	2.253	8.351	33.151	14.848
13	2.205	8.109	33.279	14.976
16	2.194	8.031	33.321	15.017
18	**2.193**	**8.028**	**33.323**	**15.020**
20	**2.193**	**8.028**	**33.323**	**15.020**
(c) For "Peppers" Image				
11	2.197	8.013	30.942	15.027
13	2.176	7.918	30.994	15.079
16	2.169	7.878	31.016	15.101
18	**2.169**	**7.878**	**31.016**	**15.101**
20	**2.169**	**7.878**	**31.016**	**15.101**
(d) For "Zelda" Image				
11	2.223	8.180	32.509	14.938
13	2.177	7.936	32.641	15.070
16	2.171	7.894	31.061	15.101
18	**2.169**	**7.891**	**32.666**	**15.094**
20	**2.169**	**7.891**	**32.666**	**15.094**
(e) Average Pixel Error values for all the images				

Table 5.2: *Average Error per pixel represented in various terminologies for the images recovered after one level of (9,7) wavelet decomposition with zero bit precision of input data.*

Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	4.371	30.847	27.582	9.173
13	4.285	30.009	27.702	9.293
16	4.272	29.886	27.720	9.311
18	**4.267**	**29.813**	**27.730**	**9.321**
20	**4.267**	**29.813**	**27.730**	**9.321**
(a) For "Lena" Image				
11	4.384	31.019	26.807	9.149
13	4.296	30.167	26.928	9.270
16	4.270	29.845	26.975	9.317
18	**4.269**	**29.829**	**26.977**	**9.319**
20	**4.269**	**29.829**	**26.977**	**9.319**
(b) For "barbara" Image				
11	4.422	31.493	27.386	9.083
13	4.356	30.943	27.463	9.160
16	4.326	30.677	27.496	9.196
18	**4.320**	**30.543**	**27.519**	**9.216**
20	**4.320**	**30.543**	**27.519**	**9.216**
(c) For "Peppers" Image				
11	4.291	29.957	25.215	9.300
13	4.224	29.281	25.314	9.399
16	4.206	29.089	25.343	9.428
18	**4.206**	**29.089**	**25.343**	**9.428**
20	**4.206**	**29.089**	**25.343**	**9.428**
(d) For "Zelda" Image				
11	4.367	30.829	26.748	9.176
13	4.290	30.100	26.852	9.281
16	4.266	29.824	26.892	9.321
18	**4.265**	**29.089**	**26.892**	**9.428**
20	**4.265**	**29.089**	**26.892**	**9.428**
(e) Average Pixel Error values for all the images				

Table 5.3: *Average Error per pixel represented in various terminologies for the images recovered after two level of (9,7) wavelet decomposition with zero bit precision of input data.*

Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	6.855	76.429	23.642	5.233
13	6.746	75.107	23.722	5.318
16	6.718	73.997	23.782	5.373
18	**6.712**	**73.844**	**23.791**	**5.382**
20	**6.712**	**73.844**	**23.791**	**5.382**
(a) For "Lena" Image				
11	6.891	76.984	22.860	5.201
13	6.819	75.643	22.936	5.278
16	6.761	74.676	22.992	5.334
18	**6.760**	**74.695**	**22.991**	**5.332**
20	**6.760**	**74.695**	**22.991**	**5.332**
(b) For "barbara" Image				
11	6.926	78.010	23.447	5.144
13	6.789	75.314	23.600	5.297
16	6.781	75.957	23.612	5.308
18	**6.777**	**74.959**	**23.620**	**5.317**
20	**6.777**	**74.959**	**23.620**	**5.317**
(c) For "Peppers" Image				
11	6.798	75.233	21.216	5.301
13	6.696	73.334	21.372	5.412
16	6.689	73.199	21.335	5.420
18	**6.689**	**73.199**	**21.335**	**5.420**
20	**6.689**	**73.199**	**21.335**	**5.420**
(d) For "Zelda" Image				
11	6.868	76.664	22.791	5.220
13	6.762	74.850	22.908	5.326
16	6.737	74.457	22.930	5.359
18	**6.735**	**74.174**	**22.934**	**5.363**
20	**6.735**	**74.174**	**22.934**	**5.363**
(e) Average Pixel Error values for all the images				

Table 5.4: *Average Error per pixel represented in various terminologies for the images recovered after three level of (9,7) wavelet decomposition with zero bit precision of input data. wavelet decomposition*

Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	1.000	1.000	42.474	24.065
13	0.989	0.989	42.522	24.113
16	0.906	0.906	42.902	24.493
18	**0.766**	**0.766**	**43.634**	**25.225**
20	**0.766**	**0.766**	**43.634**	**25.225**
(a) For "Lena" Image				
11	1.000	1.000	41.724	24.065
13	0.985	0.985	41.789	24.131
16	0.894	0.894	42.211	24.553
18	**0.762**	**0.762**	**42.904**	**25.246**
20	**0.740**	**0.740**	**43.029**	**27.371**
(b) For "barbara" Image				
11	1.000	1.000	42.372	24.069
13	0.980	0.980	42.457	24.154
16	0.897	0.897	42.840	24.536
18	**0.764**	**0.764**	**43.536**	**25.233**
20	**0.743**	**0.743**	**43.658**	**25.354**
(c) For "Peppers" Image				
11	0.999	0.999	39.982	24.068
13	0.974	0.974	40.096	24.181
16	0.876	0.876	40.556	24.641
18	**0.759**	**0.759**	**41.177**	**25.262**
20	**0.743**	**0.743**	**41.269**	**25.354**
(d) For "Zelda" Image				
11	1.000	1.000	41.638	24.067
13	0.982	0.982	41.716	24.145
16	0.893	0.893	42.100	24.557
18	**0.763**	**0.763**	**42.813**	**25.242**
20	**0.748**	**0.748**	**42.898**	**25.826**
(e) Average Pixel Error values for all the images				

Table 5.5: *Average Error per pixel represented in various terminologies for the images recovered after one level of (9,7) wavelet decomposition with eight bits precision of input data.*

Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	1.000	1.000	42.474	24.065
13	0.994	0.994	42.497	24.088
16	0.926	0.926	42.808	24.399
18	**0.795**	**0.795**	**43.470**	**25.061**
20	**0.772**	**0.772**	**43.600**	**25.191**
(a) For "Lena" Image				
11	0.999	0.999	41.723	24.065
13	0.992	0.992	41.760	24.102
16	0.917	0.917	42.101	24.443
18	**0.793**	**0.793**	**42.731**	**25.073**
20	**0.769**	**0.769**	**42.867**	**25.209**
(b) For "barbara" Image				
11	1.000	1.000	42.370	24.067
13	0.987	0.987	42.427	24.124
16	0.921	0.921	42.727	24.424
18	**0.789**	**0.789**	**43.397**	**25.094**
20	**0.771**	**0.771**	**43.497**	**25.192**
(c) For "Peppers" Image				
11	1.000	1.000	39.981	24.067
13	0.983	0.983	40.053	24.138
16	0.899	0.899	40.440	24.525
18	**0.791**	**0.791**	**41.000**	**25.085**
20	**0.770**	**0.770**	**41.118**	**25.203**
(d) For "Zelda" Image				
11	1.000	1.000	41.637	24.066
13	0.989	0.989	41.684	24.113
16	0.916	0.916	42.019	24.448
18	**0.792**	**0.792**	**42.650**	**25.078**
20	**0.771**	**0.771**	**42.771**	**25.199**
(e) Average Pixel Error values for all the images				

Table 5.6: *Average Error per pixel represented in various terminologies for the images recovered after two level of (9,7) wavelet decomposition with eight bits precision of input data.*

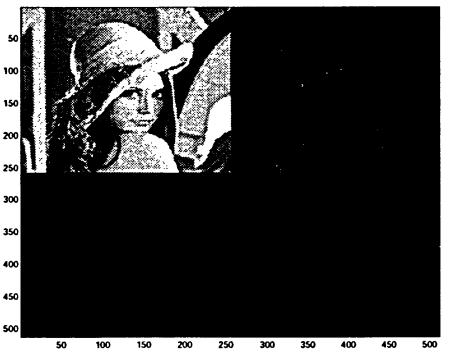
Precision	AAE	MSE	SNR (dB)	PSNR (dB)
11	1.000	1.000	42.474	24.065
13	0.994	0.994	42.499	24.090
16	0.917	0.917	42.848	24.439
18	**0.779**	**0.779**	**43.557**	**25.148**
20	**0.779**	**0.779**	**43.557**	**25.148**
(a) For "Lena" Image				
11	0.999	0.999	41.723	24.065
13	0.991	0.991	41.765	24.106
16	0.906	0.906	42.151	24.492
18	**0.780**	**0.780**	**42.803**	**25.145**
20	**0.760**	**0.760**	**42.913**	**25.255**
(b) For "barbara" Image				
11	1.000	1.000	42.371	24.066
13	0.986	0.986	42.431	24.128
16	0.909	0.909	42.782	24.479
18	**0.778**	**0.778**	**43.461**	**25.158**
20	**0.758**	**0.758**	**43.571**	**25.268**
(c) For "Peppers" Image				
11	1.000	1.000	39.982	24.067
13	0.980	0.980	40.066	24.151
16	0.884	0.884	40.514	24.599
18	**0.774**	**0.774**	**41.092**	**25.177**
20	**0.762**	**0.762**	**41.161**	**25.246**
(d) For "Zelda" Image				
11	1.000	1.000	41.638	24.066
13	0.988	0.988	41.690	24.119
16	0.904	0.904	42.074	24.502
18	**0.778**	**0.778**	**42.728**	**25.157**
20	**0.765**	**0.765**	**42.801**	**25.229**
(e) Average Pixel Error values for all the images				

Table 5.7: *Average Error per pixel represented in various terminologies for the images recovered after three level of (9,7) wavelet decomposition with eight bits precision of input data.*

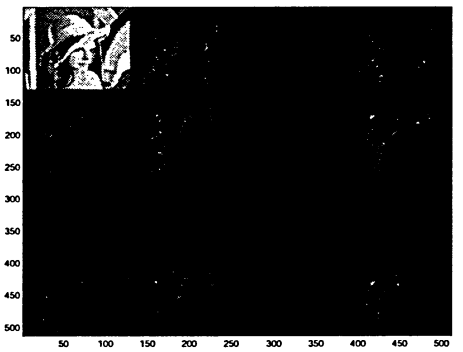
The absolute error value per pixel for the test images after one, two and three level of (5,3) wavelet packet decomposition is zero. Thus, the reconstructed image from the wavelet packet decomposition in (5,3) wavelet is the same as the original image whereas the reconstructed image from (9,7) wavelet is distorted and not same as the original image.

The simulation results for various level of decomposition of 512x512 Lena image for both (5,3) and (9,7) wavelets are shown in Fig.5.1 and Fig.5.2 respectively. From these figures, it is clear that the reconstructed image of Lena looks the same as the original image for (5,3) wavelet whereas, for (9,7) wavelet, the distortion in the reconstructed image is seen clearly as the number of decomposition level is increased.

(a) Original Image



(b) One Level of Decomposition



(c) Two Level of Decomposition

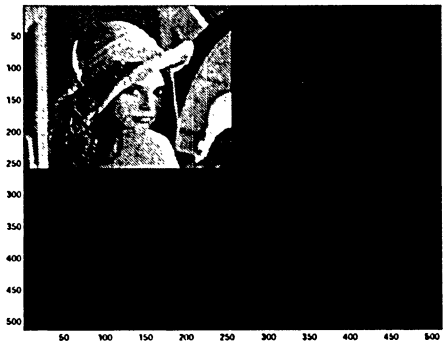
(d) Recovered Image after one level of decomposition

(e) Recovered Image after two level of decomposition

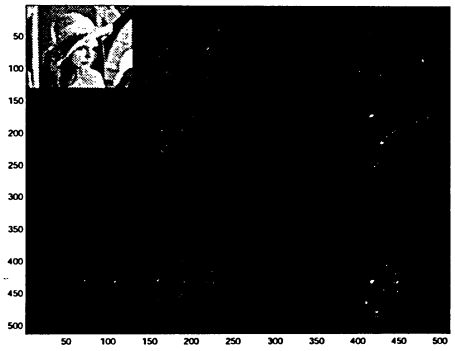
(f) Recovered Image after three level of decomposition

Fig.5.1: *Lena Images of size 512*512 after 2D WPT Decomposition using (5,3) Wavelet.*

(a) Original Image



(b) One Level of Decomposition



(c) Two Level of Decomposition

(d) Recovered Image after one level of decomposition

(e) Recovered Image after two level of decomposition

(f) Recovered Image after three level of decomposition

(g) Recovered Image after four level of decomposition

(h) Recovered Image after five level of decomposition

Fig.5.2: *Lena Images of size 512*512 after 2D WPT Decomposition using (9,7) Wavelet.*

5.2 Best-Basis Algorithm for Images

The proposed best-basis architecture for 2D signals or images includes DWT architecture proposed in Chapter 3 to obtain the wavelet packet decomposition of the image. In the next step, the wavelet packet coefficient is passed through the cost-function architecture to calculate each node cost. The proposed architecture computes the node cost using either Threshold-function architecture or Shannon-function architecture which were discussed in Section 4.3.3. The cost of each node is stored in the dual-port RAM (Best-basis RAM). The cost of children nodes and parent node is read from the best-basis RAM and fed to the best-tree selector architecture. The best-tree selector architecture performs the best-basis algorithm and determines the best-basis node. The architecture for best-basis algorithm for 1D signal was proposed in [48-49]. The architecture in [49] performs convolution-based 1D wavelet packet transform and it does not discuss about the cost-function used to determine the cost of the nodes. The proposed best-basis architecture in this thesis performs lifting-based 2D wavelet packet decomposition and discusses about the cost-function used to determine the node cost.

The proposed architecture to determine the best-basis node from two-level of DWPT decomposition is described in VHDL hardware description language and simulated successfully on ModelSim simulation environment for various images (Lena, Zelda, Bird and Barbara) of size 64x64. The JAVA program implementing best-basis architecture for 2D signal is developed to compare the results. The VHDL program selects the best-basis nodes from the two-level decomposition of the images. The output

of the VHDL program is compared with the best-basis nodes selected from the software program written in JAVA. The output of the software shows only the node-number selected by the algorithm. The numbering of the nodes for two-level DWPT is shown in Fig.5.3. The best-basis algorithm is performed on the wavelet packet obtained from both (5,3) and (9,7) wavelets. For (9,7) wavelet, Figs. 5.4, 5.6, 5.8 and 5.10 show the quad-tree with the cost of the nodes obtained from both Shannon and Threshold functions and the best-basis nodes for the images Lena, Bird, Zelda and Barbara respectively. Figs. 5.5, 5.7, 5.9 and 5.11 show the quad-tree with cost of the nodes obtained from both Shannon and Threshold functions and the nodes selected from the best-basis algorithm for the images Lena, Bird, Zelda and Barbara for (5,3) wavelet respectively. The best-basis nodes selected for each image from both ModelSim simulation and JAVA program are the same.

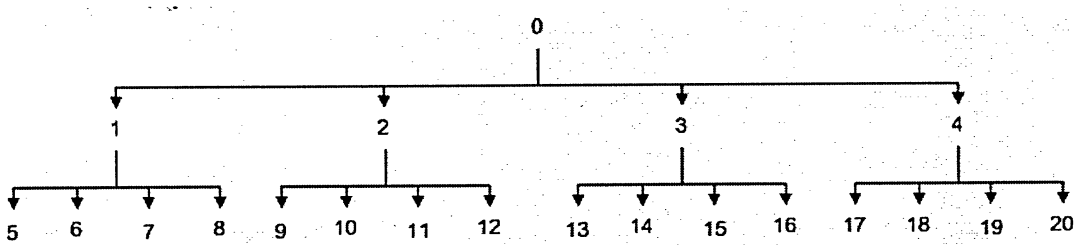
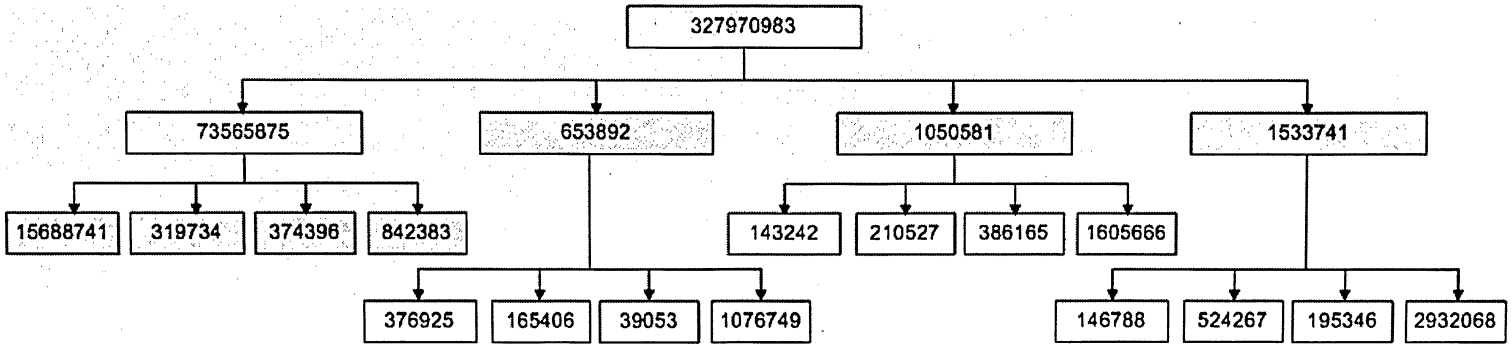
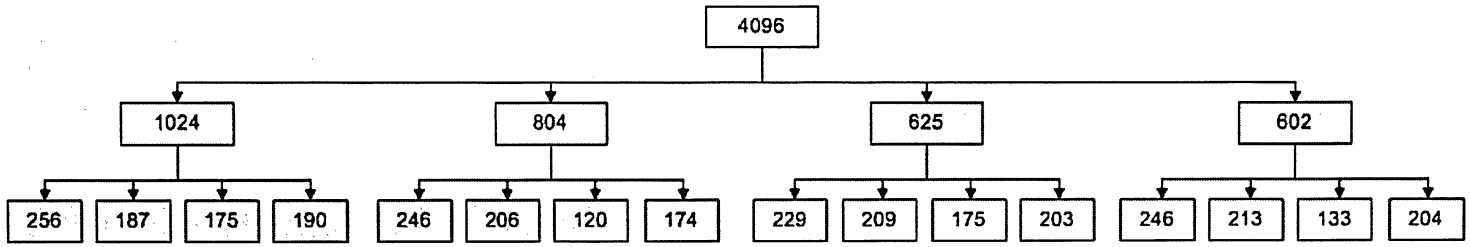


Fig.5.3 *Numbering of Wavelet Packet Node*



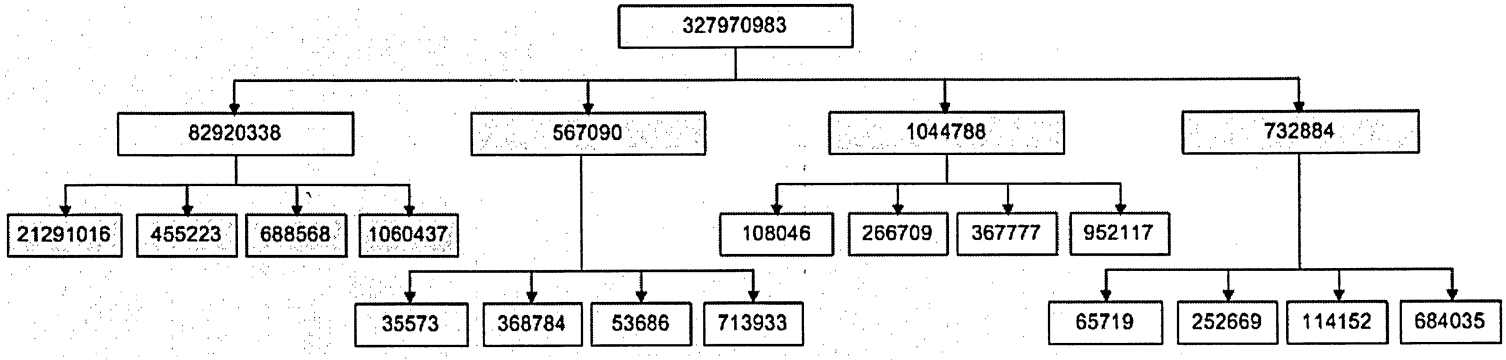
(a) Shannon Cost Function



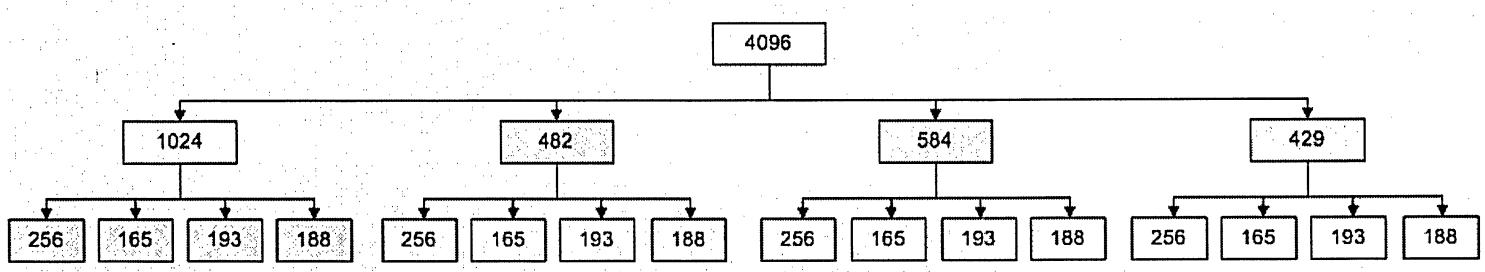
(b) Threshold Cost Function

Wavelet (9,7)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 9, 10, 11, 12, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 9, 10, 11, 12, 3, 4

Fig.5.4: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Lena Image of size 64 x 64 from two-level of (9,7) wavelet decomposition.*



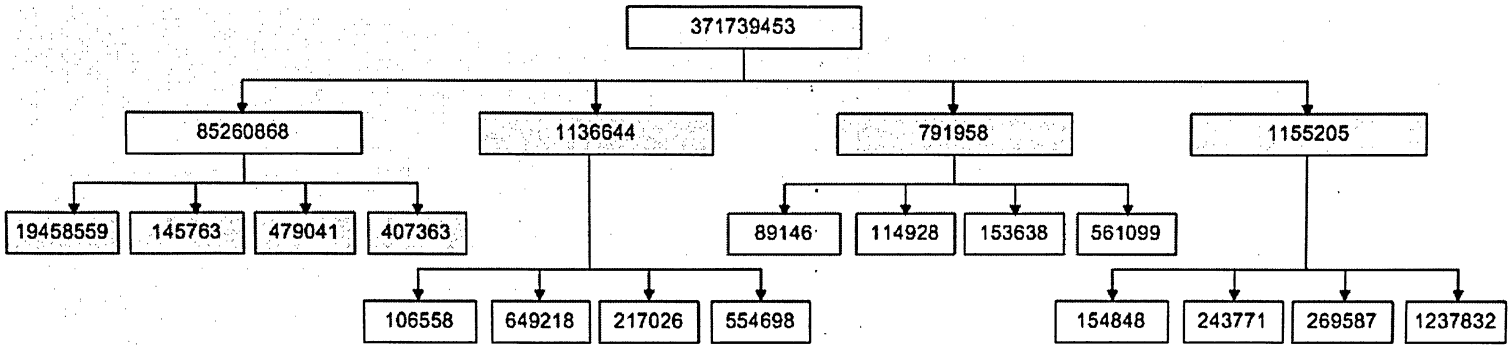
(a) Shannon Cost Function



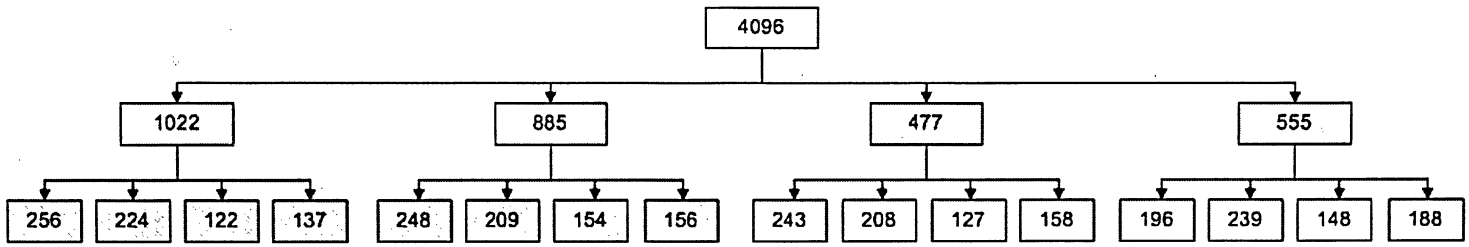
(b) Threshold Cost Function

Wavelet (5,3)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.5: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Lena Image of size 64 x 64 from two-level of (5,3) wavelet decomposition.*



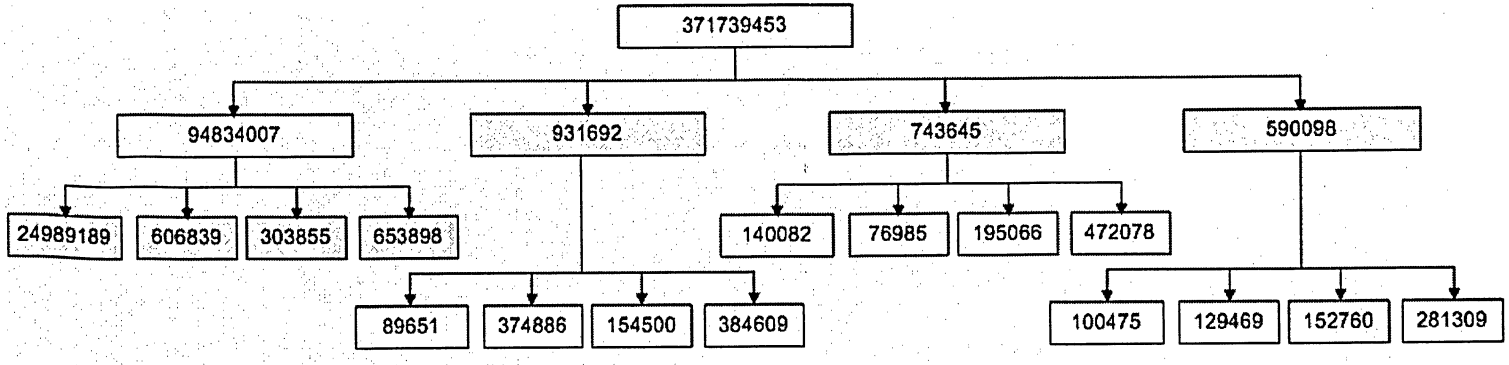
(a) Shannon Cost Function



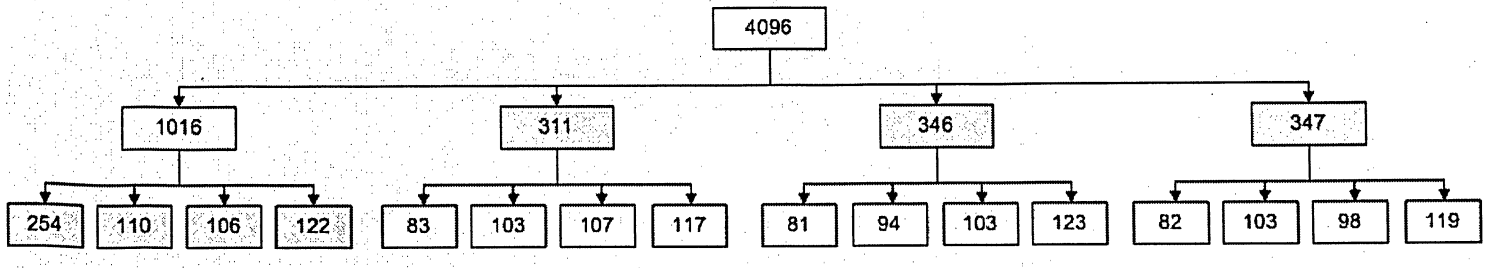
(b) Threshold Cost Function

Wavelet (9,7)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 9, 10, 11, 12, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 9, 10, 11, 12, 3, 4

Fig.5.6: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Bird Image of size 64 x 64 from two-level of (9,7) wavelet decomposition.*



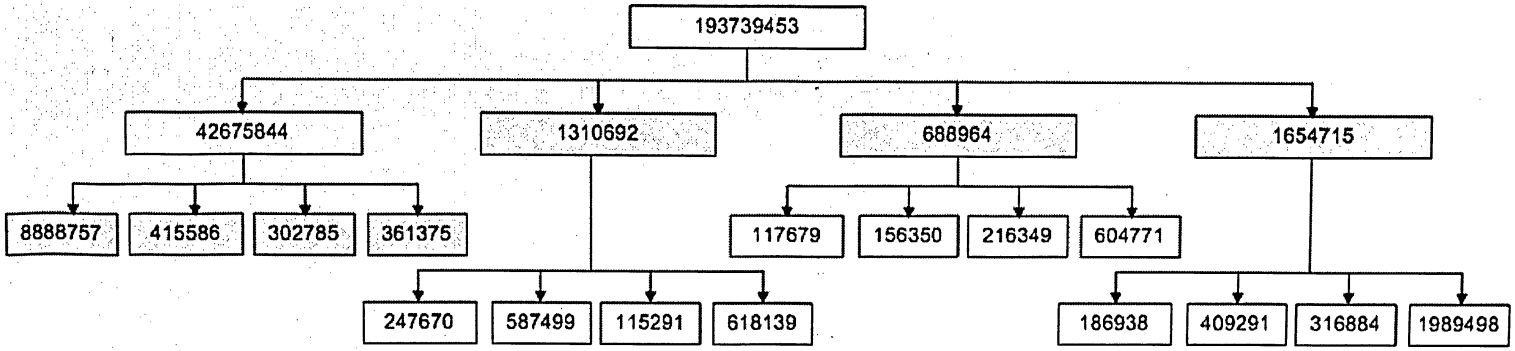
(a) Shannon Cost Function



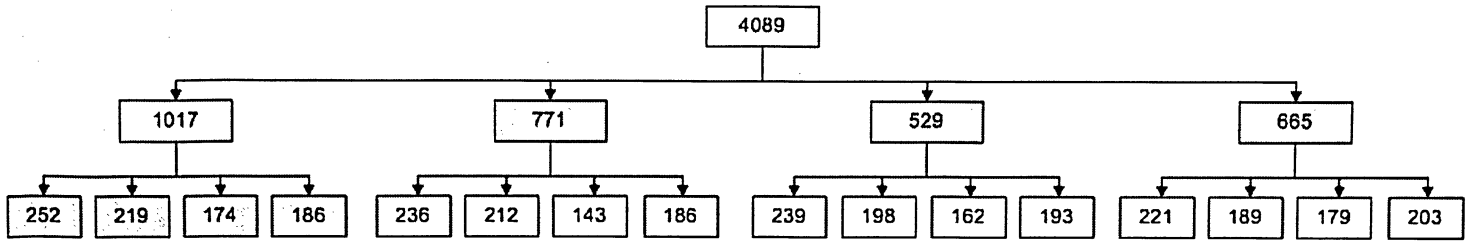
(b) Threshold Cost Function

Wavelet (5,3)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.7: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Bird Image of size 64 x 64 from two-level of (5,3) wavelet decomposition.*



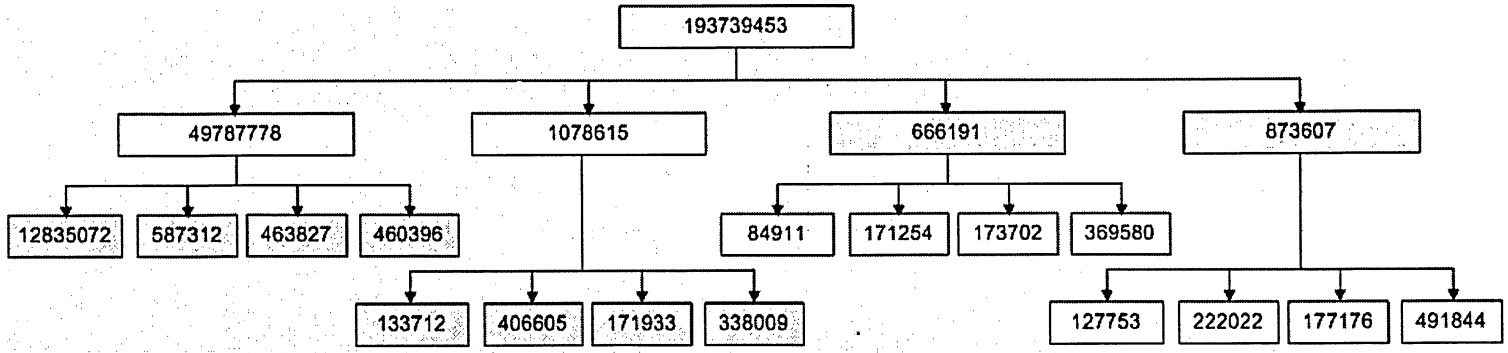
(a) Shannon Cost Function



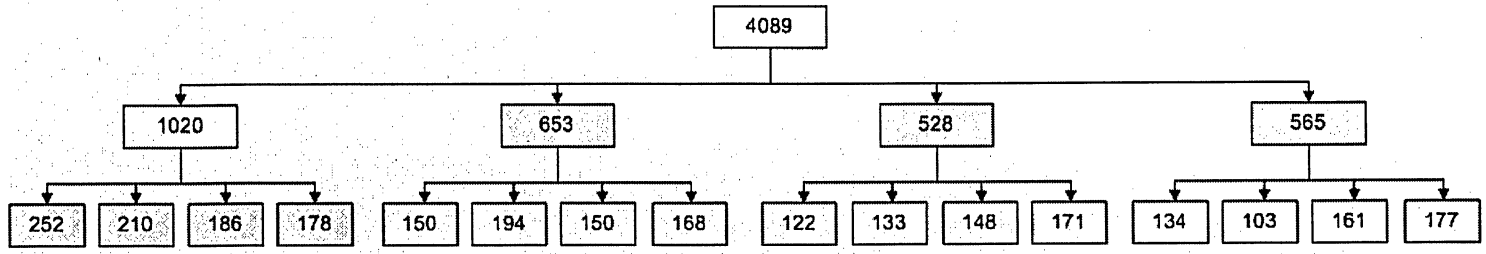
(b) Threshold Cost Function

Wavelet (9,7)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.8: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Zelda Image of size 64 x 64 from two-level of (9,7) wavelet decomposition.*



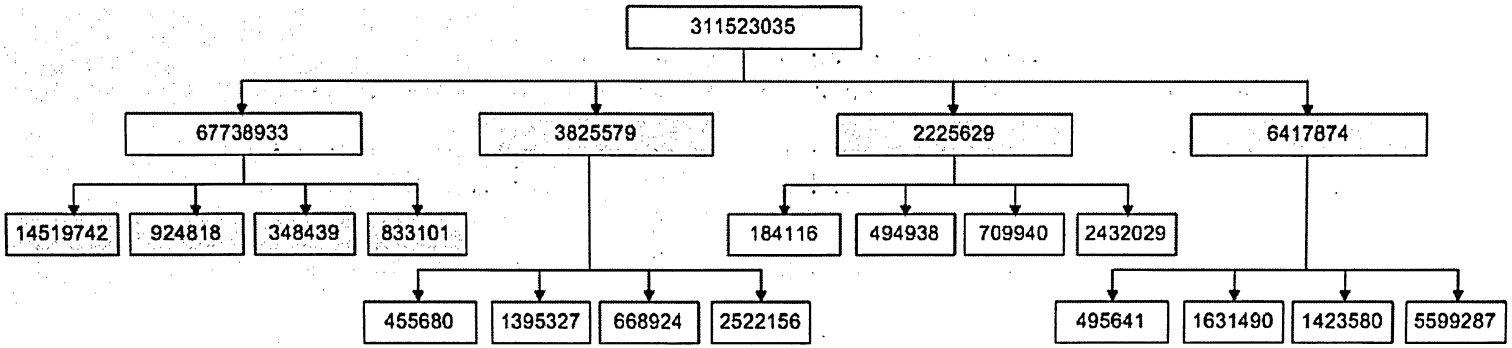
(a) Shannon Cost Function



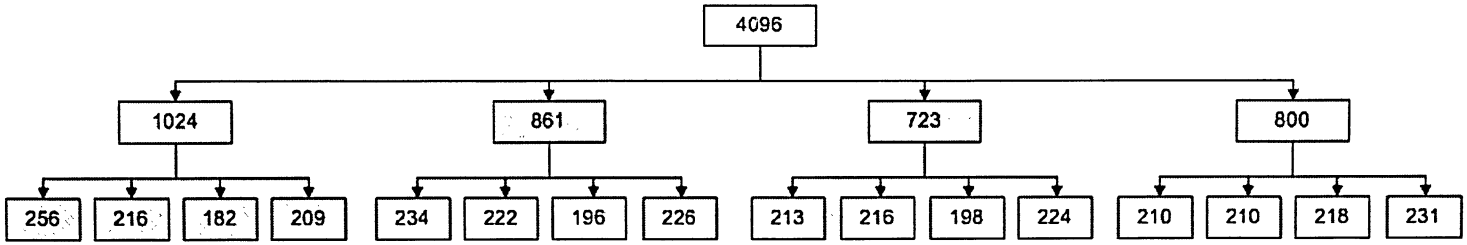
(b) Threshold Cost Function

Wavelet (5,3)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 9, 10, 11, 12, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 9, 10, 11, 12, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.9: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Zelda Image of size 64 x 64 from two-level of (5,3) wavelet decomposition.*



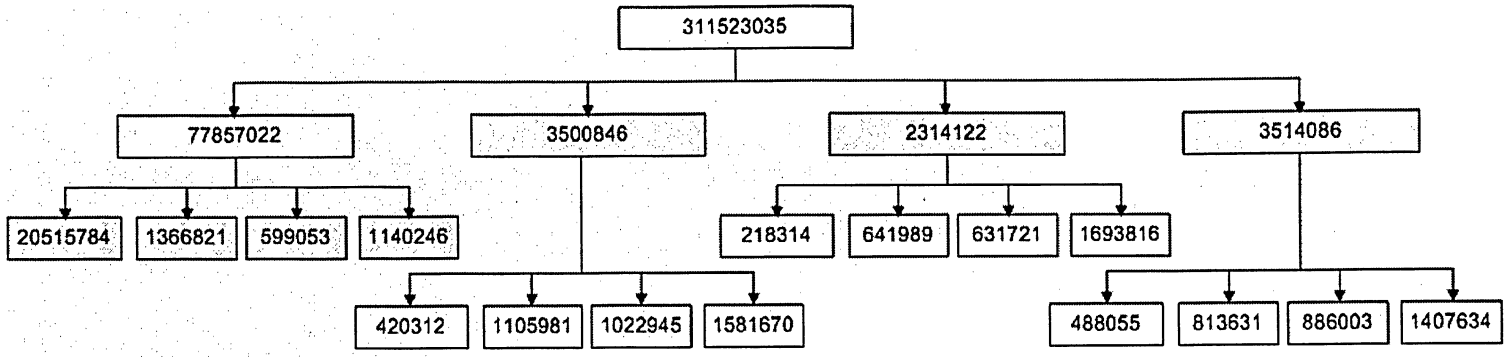
(a) Shannon Cost Function



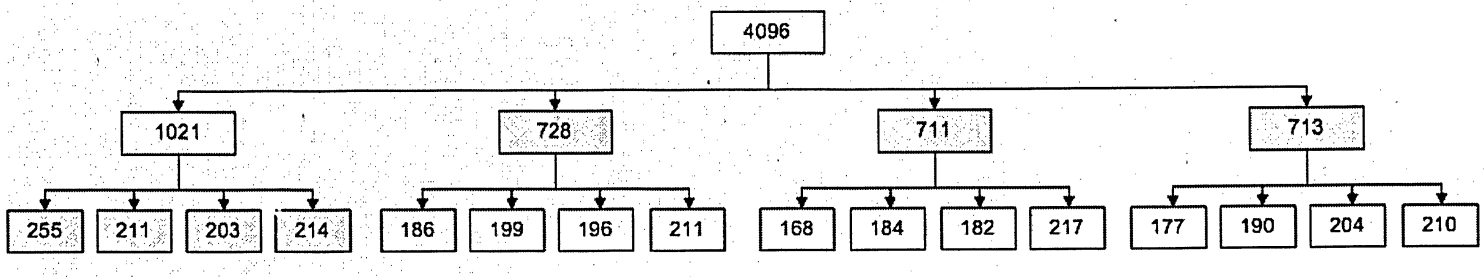
(b) Threshold Cost Function

Wavelet (9,7)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.10: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Barbara Image of size 64 x 64 from two-level of (9,7) wavelet decomposition.*



(a) Shannon Cost Function



(b) Threshold Cost Function

Wavelet (5,3)	Shannon Function	Threshold Function
Hardware	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4
Software	5, 6, 7, 8, 2, 3, 4	5, 6, 7, 8, 2, 3, 4

Fig.5.11: *Best-Basis Node obtained from Shannon & Threshold Cost Function for Barabara Image of size 64 x 64 from two-level of (5,3) wavelet decomposition.*

5.3 FPGA Implementation

The proposed architecture is implemented in an FPGA, namely the Virtex II Pro FPGA series of Xilinx. The design is described in VHDL hardware description language and simulated for functional correctness with the ModelSim simulator. The Xilinx implementation tool is used to implement the design in order to achieve the realistic timing results for performance analysis. The CORE Generator tool of the Xilinx tool is used to generate RAM blocks. The FPGA implementation of the proposed design for a picture size of 64*64 pixels is achieved. However, simulations for all desired picture sizes are carried out for performance analysis in ModelSim simulator. Even though the maximum value of the pixel or input data of the image is 255 i.e. bits, the input data or pixel width is set to 16 bits for better accuracy of the result. The final DWT coefficients are stored in the input RAM in FPGA. These values are read from the input RAM through serial port using the MicroBlaze Soft Processor using the Fast Simplex Link (FSL) channel. The DWT coefficients are read using FSL channel as mentioned in [54]. The data of 32-bit width is read through the serial port and therefore, two consecutive values in the input RAM are read using FSL through serial port at every clock cycle. These values are displayed on Tera Term Pro HTTP terminal as shown in Fig.5.12. Except for the picture RAM size, the rest of the resource usage and timings are almost constant for different configurations of the design.

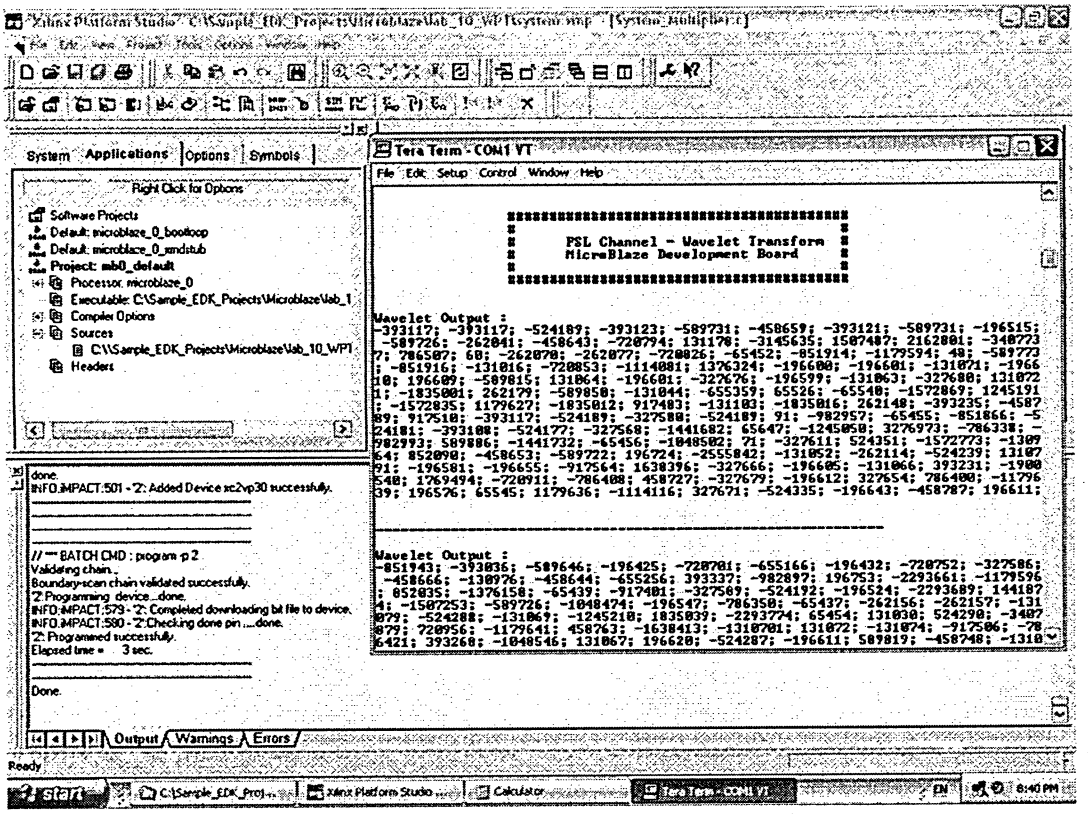


Fig.5.12: *The output read from Xilinx Virtex II Pro FPGA through serial port displayed on Tera Term Pro HTTP Terminal*

Image Dimensions:	64*64
Transform:	2D Lifting-Based DWT
Filter Type:	(9,7)
Target Device:	XC2VP30
Target Package:	FF896
Speed Grade:	-6
Synthesis Tool:	XST, Xilinx Implementation Tool.

Wavelet Transform (9,7):

Number of Slices:	1564	out of 13696	11%
Number of Slice Flip Flops:	919	out of 27392	3%
Number of 4 input LUTs:	2792	out of 27392	10%

Number of Bonded IOBs:	377	out of	556	67%
Number of BRAMs:	30	out of	136	22%
Number of MULT18X18s:	6	out of	136	4%

Natural Logarithm with 11 bits of precision:

Number of Slices:	256	out of	13696	1%
Number of Slice Flip Flops:	369	out of	27392	1%
Number of 4 input LUTs:	343	out of	27392	1%
Number of Bonded IOBs:	315	out of	556	56%
Number of BRAMs:	1	out of	136	0%
Number of MULT18X18s:	2	out of	136	1%

Best-Basis Algorithm for images (Threshold Cost Function):

Number of Slices:	2024	out of	13696	1%
Number of Slice Flip Flops:	1303	out of	27392	1%
Number of 4 input LUTs:	3620	out of	27392	1%
Number of Bonded IOBs:	493	out of	556	56%
Number of BRAMs:	32	out of	136	0%
Number of MULT18X18s:	6	out of	136	1%

As mentioned before, the resource usage in the case of wavelet transform and best-basis algorithm is almost the same for different configurations except for the picture RAM size.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, an efficient architecture for both (5,3) and (9,7) lifting-based DWT and best-basis algorithm for images or 2D signal has been proposed. Also, a new algorithm and architecture for natural logarithm using Maclaurin series has been proposed to implement the Shannon cost function in best-basis algorithm. The proposed DWT architecture exploits the arithmetic redundancy involved in lifting scheme to reduce the number of multipliers required. Because the set of registers are used to route the output of one stage to the next stage of the lifting steps of both (5,3) and (9,7) wavelets, the proposed DWT architecture does not utilize extra memory or FIFOs to store the intermediate results. The proposed DWT architecture is compared with other existing architectures and shown that the proposed architecture utilizes less hardware area. The proposed architecture for best-basis algorithm performs the best-tree selection from the two-level of wavelet packet decomposition of images. The best-basis architecture utilizes two cost functions, Threshold and Shannon cost functions, to compute the cost of the

nodes. The best-tree selector architecture implements the best-basis algorithm to determine the best-basis nodes. The best-basis nodes obtained from the proposed architecture for various images are also presented. The summary of the work and future extension of the work are presented in this chapter.

DWT

The following summarizes the contributions and the achievements of the author by proposing lifting-based DWT architecture:

- Efficient architecture for lifting-based DWT using (5,3) and (9,7) wavelets. The proposed architecture utilized less hardware area by exploiting the arithmetic redundancy in the calculation of the lifting scheme.
- The proposed architecture does not require additional or extra storage devices such as memory, FIFO to store the intermediate results unlike other existing architectures.
- The proposed DWT architecture was compared with already existing architecture for (5,3), (9,7) and (13,7) wavelets based on the requirement of the multipliers, adders and extra storage devices. In comparison, it is evident from the table that the proposed architecture uses minimum number of multipliers with no extra storing devices. The simulation of the proposed architecture was performed for different precision of filter coefficients for both (5,3) and (9,7) wavelets. The result shows that the error value per pixel obtained from different formulae decreases with increase in the precision of the filter coefficients of (9,7) wavelets.

Since the whole image is reversed without any error from (5,3) wavelet, the increase in the precision of filter coefficients does not affect its performance.

Natural Logarithm

- An efficient algorithm and architecture for the implementation of natural logarithm function was proposed in Chapter 4.

Best-Basis Algorithm

- The architecture for best-basis algorithm for 2D signal or image was proposed in Chapter 5. The proposed DWT architecture in Chapter 3 was used to compute the quadtree decomposition of the 2D signal. The proposed architecture used one of the cost function, Threshold function or Shannon function, to arrive the node cost. The proposed logarithm algorithm in Chapter 4 was used to implement the Shannon cost function.
- The architectures for best-tree calculation for 2D signal, address generation for best-basis dual-port RAM and information cost calculation of each node were proposed and discussed in details in Chapter 6.
- The proposed architecture for 2D signal was evaluated based on several images and compared with the software output.

- Implementing all the proposed architectures – The proposed architectures were described in VHDL hardware description language. The design was simulated successfully in ModelSim simulation tool for function correctness and

synthesized using Xilinx tool. In order to obtain the realistic data, the VHDL code was implemented in Xilinx Virtex II Pro FPGA using the Xilinx implementation tool. The FPGA implementation results were discussed in Chapter 5.

6.2 Future Research

The research has the potential for further scope of work. In the following we highlight or propose some ideas that could lead to future research recommendations:

- An algorithm and architecture for natural logarithm function was proposed in the Chapter 4. The extensive study and analysis can be carried out by determining the area and delay estimates of the algorithm in a particular technology as discussed in [50]. The proposed algorithm can be further extended for the other elementary functions such as exponential and trigonometric functions such as sine and cosine functions.
- The idea of the proposed best-basis architecture for images can be further extended to another algorithm called Local Discriminant Basis (LDB) algorithm [52-53]. LDB performs similar steps as best-basis algorithm except that the cost function in LDB determines the discriminant measure as mentioned in [52].

REFERENCES

[1] G.K. Wallace, "The JPEG still picture compression standard," *IEEE Trans. on Consumer Electronics*, Vol.38, No.1, pp.xviii-xxxiv, Feb.1992.

[2] Christopoulos, C., Skodras, A., Ebrahimi, T., "The JPEG2000 Still Image Coding System: An Overview," *IEEE Trans. on Consumer Electronics*, Vol.46, No.4, pp.1103-1127, Nov. 2000.

[3] M. Charrier, D. S. Cruz, and M. Larsson, "JPEG2000, The next millennium compression standard for still images," *Proc. IEEE Int. Conf. Multimedia Computing and Systems (ICMCS)*, Vol.1, pp. 131–132, June 1999.

[4] W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions," *in Proc. SPIE*, Vol. 2569, pp. 68–79, 1995.

[5] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Journal of Appl. and Comput.. Harmonic Analysis*, Vol. 3 pp:186-200, 1996.

[6] W. Sweldens and P. Schroeder, "Building your own wavelets at home," *in Wavelets in Computer Graphics*, pp.15-87. ACM SIGGRAPH Course notes, 1996 (Available at http://cm.bell-labs.com/who/wim/papers/athome.pdf).

[7] W. Sweldens, "Wavelets and the lifting scheme: A 5 minute tour," *Zeitschrift für Angewandte Mathematik und Mechanik*, Vol.76, No.2, pp.41-44, 1996 (English version: Available at http://cm.bell-labs.com/who/wim/papers/iciam95.pdf).

[8] R.R. Coifman and M.V. Wickerhauser, "Entropy-based Algorithms for Best Basis Selection," *IEEE Trans. Information Theory*, Vol.38, No.2, pp.713-719, 1992.

[9] D. M. Lewis, "114 MFLOPS logarithmic number system arithmetic unit for DSP applications," *IEEE Journal of Solid-State Circuits*, Vol.30, pp.1547–1553, Dec 1995.

[10] R. Andraka, "A Survey of CORDIC Algorithms for FPGA based computers," *in Proceedings of the ACM/SIGDA 6th International Symposium of FPGA, ACM Press*, pp.191-200, New York, Feb.1998.

[11] W.B. Ligon, G. Monn, D. Stanzoine, F. Stivers and K.D. Underwood, "Implementation and Analysis of Numerical Components for Reconfigurable Computing," *IEEE Conf. on Aerospace*, Vol.2, pp.325-335, March 1999.

[12] S.G. Mallat, "*A Wavelet Tour of Signal Processing,*" San Diego, Academic Press, 1998.

[13] O. Rioul and M. Vetterli, "Wavelets and Signal Processing," *IEEE Signal Processing*, Vol. 8, No.4, pp. 14-38, Oct. 1991.

[14] W. Sweldens, "Wavelets: What Next?," *IEEE Proceedings*, Vol.84, No.4, pp.680-685, April 1996.

[15] Robi Polikar, "The Wavelet Tutorial," Available at http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html.

[16] W. Cai and M. Adjouadi, "Minimization of Boundary Artifacts on Scalable Image Compression Using Symmetric-Extended Wavelet Trsasnsform," *IEEE Intl. Conf. on Information Technology: Coding and Computing (ITCC)*, Vol.1, pp.598-602, 2004.

[17] K. K. Parhi and T. Nishitani, "VLSI architectures for discrete wavelet transforms," *IEEE Trans. Very Large Scale Integration System*, Vol. 1, pp.191–202, June 1993.

[18] C. Chakrabarti and M. Vishwanath, "Efficient realizations of the discrete and continuous wavelet transforms: from single chip implementations to mappings on SIMD array computers," *IEEE Trans. Signal Processing*, Vol. 43, pp. 759–771, Mar. 1995.

[19] M. Vishwanath, R. M. Owens, and M. J. Irwin, "Discrete wavelet transforms in VLSI," *Intl. Conf. on Application Specific Array Processors*, pp.218-229, Aug.1992.

[20] M. Vishwanath, R. M. Owens, and M. J. Irwin, "VLSI architectures for the discrete wavelet transform," *IEEE Trans. Circuit System II*, Vol. 42, pp.305–316, May 1995.

[21] C. Chakrabarti, M. Vishwanath, and R. M. Owens, "Architectures for wavelet transforms: A survey," *J. VLSI Signal Processing*, Vol.14, pp.171–192, 1996.

[22] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Trans. Image Processing*, Vol. 9, pp.378–389, Mar. 2000.

[23] C. Chrysafis and A. Ortega, "Line based reduced memory, wavelet image compression," *in Proceedings DCC '98 Data Compression Conference*, pp.398-407, Los Alamitos, CA, USA, 1998.

[24] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI architecture for lifting-based forward and inverse wavelet transform," *IEEE Trans. Signal Processing*, Vol.50, pp.966–977, Apr. 2002.

[25] G.Kuzmanov, B. Zafarifar, P. Shrestha, S.Vassiliadis, "Reconfigurable DWT Unit Based on Lifting," *Proc. "13th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC2002)"*, Veldhoven, The Netherlands, pp.325-333, Nov.2002.

[26] G. Dillen, B. Georis, J.D Legat and O. Cantineau, "Combined Line-Based Architecture for the 5-3 and 9-7 Wavelet Transform of JPEG2000," *IEEE Trans. on Circuits and Systems for Video Technology*, Vol.13, No.9, Sept.2003.

[27] P.Y. Chen, "VLSI Implementation for One-Dimensional Multilevel Lifting-Based Wavelet Transform," *IEEE Trans. on Computers*, Vol.53, No.4, April 2004.

[28] B. Banjanin, B. Gergic, P. Planinsic and Z. Cucej, "Entropy-threshold method for best basis selection," *Elsevier Journal Image and Vision Computing*, Vol.19, pp.477-484, 2001.

[29] E. Antelo, T. Lang, and J.D. Bruguera, "High-Radix CORDIC Rotation Based on Selection by Rounding," *Journal of VLSI Signal Processing*, Vol.25, No.2, pp.141–153, 2000.

[30] J.A. Pineiro and J.D. Bruguera, "High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root," *IEEE Transactions on Computers*, Vol.51, No.12, pp.1377–1388, Dec 2002.

[31] J.A. Pineiro, M.D. Ercegovac and J.D. Bruguera, "Algorithm and Architecture for logarithm, exponential and powering computation," *IEEE Tans. on Computers*, Vol.53, No.9, pp.1085-1096, Sept.2004.

[32] D. DasSarma and D.W. Matula, "Faithful Bipartite ROMReciprocal Tables," *IEEE Transactions on Computers*, Vol.47, No.11 pp.1216–1222, November 1998.

[33] M. J. Schulte and J. E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Transactions on Computers*, Vol.48, No.8 pp.842–847, 1999.

[34] N. Takagi, "Powering by a Table Look–up and a Multiplication with Operand Modification," *IEEE Transactions on Computers*, Vol.47, No.11 pp.1216–1222, 1998.

[35] M. D. Ercegovac and T. Lan,. "*Division and Square Root: Digit Recurrence Algorithms and Implementations*," Kluwer Academic Publishers, 1994.

[36] J.A. Pineiro, M. D. Ercegovac and J. D. Bruguera, "Algorithm and Architecture for Logarithm, Exponential and Powering Computation," *IEEE Trans. on Computers*, Vol.53, pp.1085-1096, Sept. 2004.

[37] E. Antelo, T. Lang, and J. D. Bruguera, "Very-high radix CORDIC vectoring with scalings and selection by rounding," *in Proc. 14th Symp. Computer Arithmetic*, pp.204–213, April 1999.

[38] E. Antelo, T. Lang, and J. D. Bruguera, "Very-High Radix Circular CORDIC: Vectoring and Unified Rotation/Vectoring," *IEEE Trans. on Computers*, Vol.49, No.7, July 2000.

[39] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, and G. Wei, "Improving Goldschmidt Division, Square Root and Square Root Reciprocal," *IEEE Transactions on Computers*, Vol.49, No.7 pp.759–763, 2000.

[40] J.-M. Muller, "*Elementary Functions. Algorithms and Implementation*," Birkhauser, 1997.

[41] ISO/IEC. International Standard, 15444-1: 2000(E), "*JPEG2000 Image Coding System – Part I Core coding system.*"

[42] M. Charrier, D. S. Cruz, and M. Larsson, "JPEG2000, The next millennium compression standard for still images," *Proc. IEEE Int. Conf. Multimedia Computing and Systems (ICMCS)*, Vol.1, pp.131–132, June 1999.

[43] Erwin Kreyszig, "*Advanced Engineering Mathematics*," 8th Edition, A Wiley Publication, 2001.

[44] IEEE, New York. "*ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*," 1985.

[45] G.J. Battaglia, "Mean Square Error," *AMP Journal of Technology*, Vol.5, pp.31-36, June, 1996.

[46] J.G. Proakis, D.G. Manolakis, "*Digital Signal Processing – Principles, Algorithms and Applications*," 3rd Edition, Prentice-Hall, 1996.

[47] A. Saffor, K.H. Ng, A.R. Bin Ramli, D. Dowsett, "A Comparison of JPEG and Wavelet Compression Applied to Computed Tomography Brain, Chest, and Abdomen Images," *The Internet Journal of Medical Simulation and Technology*, Vol.1 No.1, 2002. (Available at http://www.ispub.com/ostia/index.php?xmlFilePath=journals/ijmst/vol1n1/tomo.xml)

[48] A. Uhl, "Wavelet Packet best basis selection on moderate parallel MIMD architectures," *Parallel Computing (Elsevier)*, pp.149-158, 1996.

[49] M.A. Trenas, J. Lopez, M. Sanchez, F. Arguello and E.L. Zapata, "Architecture for Wavelet Packet Transform with Best Tree searching," *IEEE Intl. Conf. on Application Specific Systems, Architectures and Processors*, pp.289-298, July 2000.

[50] J.A. Piniero, "*Algorithms and Architectures for Elementary Function Computation*," PhD Dissertation, University of Santiago de Compostela, 2003.

[51] R.R. Coifman and M.V. Wickerhauser, "Adapted Waveform "Denoising" for Medical Signals and Images," *IEEE Engineering in Medicine & Biology Magazine*, Vol.14, No.5, pp.578-586, 1995.

[52] N. Saito and R.R. Coifman, "Local Discriminant Bases and Their Applications," *J. Mathematical Imaging and Vision*, Vol.5, No.4, pp.337-358, 1995, Invited paper.

[53] N. Saito, "*Local Feature Extraction and Its Applications Using a Library of Bases*," Ph.D Thesis, Dept. of Mathematics, Yale University, New Haven, CT 06520 USA, Dec. 1994.

[54] Hans-Peter Rosinger, "*Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel*," May 12, 2004 (Available at http://direct.xilinx.com/bvdocs/appnotes/xapp529.pdf).

APPENDIX A

VHDL Code of the modules involved in Lifting-based DWT and Best-Basis Algorithm

VHDL model of Predict Module:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use work.wavelet.all;

entity predict is
        generic (N: integer := 2);
        port (
                lam_in: in data_type;
                gam_in: in data_type;
                pred_coeff: in coeff_type;
                lam_en: in std_logic;
                clk: in std_logic;
                fw_iv: in std_logic;
                gam_out: out data_type
        );
end predict;

architecture predict_arch of predict is
        signal mult_outs: mult_out_vector(0 to N-1);
        signal lam_stages: data_type;
        signal gamma_cal: data_type;

        component multiplier
                generic (A_length, B_length: integer);
                port (
                        A: in std_logic_vector(A_length-1 downto 0);
                        B: in std_logic_vector(B_length-1 downto 0);
                        O: out std_logic_vector(A_length+B_length-1 downto 0)
                );
        end component;
begin

        setLambdaValues: process(clk, lam_en)
        begin
                if(clk'event and clk = '1') then
                        if(lam_en = '1') then
                                lam_stages <= lam_in;
                        end if;
                end if;
```

```
        end process;

mul: multiplier generic map(data_length, coeff_length) port map(lam_stages, pred_coeff, mult_outs(0));

setOtherMultiplierOutputs: process(clk)
begin
        if(clk'event and clk = '1') then
                mult_outs(N-1) <= mult_outs(0);
        end if;
end process;

findGammaOut: process(mult_outs, gam_in, fw_iv)
        variable sum: std_logic_vector(lifting_adder_output_width-1 downto 0);
        variable sum_scaled: data_type := (others => '0');
begin
        sum := (others => '0');
        for i in 0 to (N - 1) loop
                sum := sum + sxt(mult_outs(i), sum'length);
        end loop;
sum_scaled := sum(data_length+predict_coeff_scaling_in_bits-1 downto predict_coeff_scaling_in_bits);
        if(fw_iv = '1') then
                gamma_cal <= gam_in - sum_scaled;
        else
                gamma_cal <= gam_in + sum_scaled;
        end if;
        sum_scaled_out <= sum_scaled;
end process;

setGammaOutput: process(clk)
begin
        if(clk'event and clk = '1') then
                gam_out <= gamma_cal;
        end if;
end process;
end predict_arch;
```

VHDL model of Update Module:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use work.wavelet.all;

entity update is
        generic(NTilde: integer := 2);
        port (
                lam_in: in data_type;
                gam_in: in data_type;
                upd_coeff: in coeff_type;
                clk: in std_logic;
                lam_en: in std_logic;
                next_lam: in std_logic;
                fw_iv: in std_logic;
                lam_out: out data_type
        );
```

```
end update;

architecture update_arch of update is
        signal mult_outs: mult_type;
        signal mult_out_scaled: data_type;
        signal lambdas: data_vector(0 to Ntilde - 1);
        signal lam_outs: data_vector(0 to Ntilde - 1);

        component multiplier
                generic (A_length, B_length: integer);
                port (
                        A: in std_logic_vector(A_length-1 downto 0);
                        B: in std_logic_vector(B_length-1 downto 0);
                        O: out std_logic_vector(A_length+B_length-1 downto 0)
                );
        end component;

begin
        assign_lam_ins: process(clk, lam_en)
        begin
                if(clk'event and clk = '1') then
                        if(lam_en = '1') then
                                for i in 0 to (NTilde-2) loop
                                        lambdas(i) <= lam_outs(i+1);
                                end loop;
                                lambdas(NTilde-1) <= lam_in;
                        end if;
                end if;
        end process assign_lam_ins;

mul: multiplier generic map(data_length, coeff_length) port map(gam_in, upd_coeff(0), mult_outs(0));
mult_out_scaled(0) <= mult_outs(0)(data_length+update_coeff_scaling_in_bits-1 downto
update_coeff_scaling_in_bits);

mul_out <= mult_out_scaled(0);

process(mult_out_scaled, lambdas, fw_iv)
begin
        for i in 0 to (NTilde/2-1) loop
                if(fw_iv = '1') then
                        lam_outs(i) <= lambdas(i) + mult_out_scaled(i);
                        lam_outs(NTilde/2+i) <= lambdas(NTilde/2+i) + mult_out_scaled(i);
                else
                        lam_outs(i) <= lambdas(i) - mult_out_scaled(i);
                        lam_outs(NTilde/2+i) <= lambdas(NTilde/2+i) - mult_out_scaled(i);
                end if;
        end loop;
end process;

assign_lam_out: process(clk)
begin
        if(clk'event and clk = '1') then
                lam_out <= lam_outs(0);
        end if;
end process assign_lam_out;
```

```
end update_arch;
```

VHDL model of Threshold Cost Function Module:

```
library ieee;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.wavelet.all;

entity ThresholdFunction is
        port (
                clk: in std_logic;
                count_even_value: in std_logic;
                count_odd_value: in std_logic;
                reset_count: in std_logic;
                in_even_value: in std_logic_vector(data_length-1 downto 0);
                in_odd_value: in std_logic_vector(data_length-1 downto 0);
                bestbasis: out std_logic_vector(15 downto 0)
        );
end ThresholdFunction;

architecture costfunction of ThresholdFunction is
begin

best_basis: process(clk)
        variable count: std_logic_vector(15 downto 0) := (others => '0');
begin
        if(clk'event and clk = '1') then
                if(reset_count = '1') then
                        count := (others => '0');
                end if;
                if(count_even_value = '1') then
                        if(not (in_even_value < min_val_on_positive_number and in_even_value >
                                min_val_on_negative_number)) then
                                        count := count + "001";
                        end if;
                end if;

                if(count_odd_value = '1') then
                        if(not (in_odd_value < min_val_on_positive_number and in_odd_value >
                                min_val_on_negative_number)) then
                                count := count + "001";
                        end if;
                end if;
                        bestbasis <= count;
                end if;
        end process best_basis;
end costfunction;
```

VHDL model of Shannon Cost Function Module:

```
library ieee;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.wavelet.all;
```

```
entity ShannonEqn is
	port (
		clk: in std_logic;
		reset_count: in std_logic;
		consider_value: in std_logic;
		in_value: in std_logic_vector(15 downto 0);
		out_bbcost: out std_logic_vector(31 downto 0)
	);
end ShannonEqn;

architecture equation of ShannonEqn is
	component LN_unit port (
		clk: in std_logic;
		input: in std_logic_vector(15 downto 0);
		LN_output: out std_logic_vector(15 downto 0)
	);
	end component;
	for all: LN_unit use entity work.NaturalLog(LN_arch);

	component multiplier
		generic (A_length, B_length: integer);
		port (
			A: in std_logic_vector(A_length-1 downto 0);
			B: in std_logic_vector(B_length-1 downto 0);
			O: out std_logic_vector(A_length+B_length-1 downto 0)
		);
	end component;

	signal abs_in_value: std_logic_vector(15 downto 0) := (others => '0');
	signal LN_from_LogUnit: std_logic_vector(15 downto 0) := (others => '0');
	signal mult_output: std_logic_vector(17 downto 0) := (others => '0');
	signal delay_mult_output_1: std_logic_vector(16 downto 0) := (others => '0');
	signal delay_mult_output_2: std_logic_vector(16 downto 0) := (others => '0');
	signal delay_mult_output_3: std_logic_vector(16 downto 0) := (others => '0');
	signal x_square: std_logic_vector(16 downto 0) := (others => '0');
	signal cost_of_coeff_x: std_logic_vector(32 downto 0) := (others => '0');

begin
	process(in_value)
	begin
		if(in_value(15) = '1') then
			abs_in_value <= not (in_value) + 1;
		else
			abs_in_value <= in_value;
		end if;
	end process;

	mul_1: multiplier generic map(16, 16) port map(abs_in_value, abs_in_value, mult_output);

	LN: LN_unit
	port map (
		clk => clk,
		input => abs_in_value,
		LN_output => LN_from_LogUnit
	);
```

```
        Delay_multiplieroutput_unit:
        process(clk)
        begin
                if(clk'event and clk = '1') then
                        delay_mult_output_1 <= mult_output(15 downto 0);
                        delay_mult_output_2 <= delay_mult_output_1;
                        delay_mult_output_3 <= delay_mult_output_2;
                        x_square <= delay_mult_output_3;
                end if;
        end process Delay_multiplieroutput_unit;

        mul_2: multiplier generic map(16, 16) port map(x_square, LN_from_LogUnit, cost_of_coeff_x);

        process(clk, reset_count, consider_value, cost_of_coeff_x)

                variable indiv_cost: std_logic_vector(20 downto 0) := (others => '0');
                variable node_cost: std_logic_vector(31 downto 0) := (others => '0');
                variable prev_node_cost: std_logic_vector(31 downto 0) := (others => '0');
                variable count: integer := 0;
        begin
                indiv_cost := cost_of_coeff_x(31 downto 11);
                if(clk'event and clk = '1') then
                        prev_node_cost := node_cost;
                end if;
                if(reset_count = '1') then
                        node_cost := (others => '0');
                elsif(consider_value = '1') then
                        node_cost := prev_node_cost + indiv_cost;
                        count := count + 1;
                end if;
                out_bbcost <= node_cost;
        end process;
end equation;
-----------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use work.wavelet.all;

entity ShannonFunction is
        port (
                clk: in std_logic;
                is_even_value: in std_logic;
                is_odd_value: in std_logic;
                reset_cost: in std_logic;
                in_even_value: in std_logic_vector(15 downto 0);
                in_odd_value: in std_logic_vector(15 downto 0);
                out_bbcost_node: out std_logic_vector(31 downto 0)
        );
end ShannonFunction;

architecture costfunction of ShannonFunction is

        component ShannonEqn
```

```
        port (
                clk: in std_logic;
                reset_count: in std_logic;
                consider_value: in std_logic;
                in_value: in std_logic_vector(15 downto 0);
                out_bbcost: out std_logic_vector(31 downto 0)
        );
    end component;

    signal out_even_bbcost: std_logic_vector(31 downto 0) := (others => '0');
    signal out_odd_bbcost: std_logic_vector(31 downto 0) := (others => '0');

begin

    BBCF_1: ShannonEqn
    port map (
            clk => clk,
            reset_count => reset_cost,
            consider_value => is_even_value,
            in_value => in_even_value,
            out_bbcost => out_even_bbcost
    );

    BBCF_2: ShannonEqn
    port map (
            clk => clk,
            reset_count => reset_cost,
            consider_value => is_odd_value,
            in_value => in_odd_value,
            out_bbcost => out_odd_bbcost
    );

    process(clk, out_odd_bbcost, out_even_bbcost)
    begin
            out_bbcost_node <= out_odd_bbcost + out_even_bbcost;
    end process;
end costfunction;
```

VHDL model of Best-Tree Selector Module:

```
library ieee;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use work.wavelet.all;

entity BestTreeSelector is
        port (
                clk: in std_logic;
                bb_enable: in std_logic;
                set_flag_in_TreeSelector: in std_logic;
                in_cost_of_child1: in std_logic_vector(15 downto 0);
                in_cost_of_child2: in std_logic_vector(15 downto 0);
                in_cost_of_parent: in std_logic_vector(15 downto 0);
                parent_node: in std_logic_vector(10 downto 0);
                flag_level_0: out std_logic;
                flag_level_1: out std_logic_vector(3 downto 0);
```

```
                total_child_cost: out std_logic_vector(15 downto 0);
                out_selected_cost: out std_logic_vector(15 downto 0)
        );
end BestTreeSelector;

architecture BestTree_arch of BestTreeSelector is
        signal cost_child1: std_logic_vector(15 downto 0) := (others => '0');
        signal cost_child2: std_logic_vector(15 downto 0) := (others => '0');
        signal added_cost1: std_logic_vector(15 downto 0) := (others => '0');
        signal added_cost2: std_logic_vector(15 downto 0) := (others => '0');
        signal totalchildcost: std_logic_vector(15 downto 0) := (others => '0');
        signal selected_cost: std_logic_vector(15 downto 0) := (others => '0');
        signal level_0: std_logic := '0';
        signal level_1: std_logic_vector(3 downto 0) := "0000";
begin
        process(clk, bb_enable)
        begin
                if(clk'event and clk = '1') then
                        if(bb_enable = '1') then
                                cost_child1 <= in_cost_of_child1;
                                cost_child2 <= in_cost_of_child2;
                        end if;
                end if;
        end process;
        sumChildCost: process(clk, bb_enable, cost_child1, cost_child2)
                variable bln_toggle: boolean := false;
                variable add_out: std_logic_vector(15 downto 0) := (others => '0');
        begin
                if(bb_enable = '1') then
                        add_out := cost_child1 + cost_child2;
                        totalchildcost <= added_cost1 + added_cost2;
                        total_child_cost <= added_cost1 + added_cost2;
                        bln_toggle := not bln_toggle;
                end if;
                if(clk'event and clk = '1') then
                        if(bb_enable = '1') then
                                if(bln_toggle) then
                                        added_cost1 <= add_out;
                                else
                                        added_cost2 <= add_out;
                                end if;
                        end if;
                end if;
        end process;
        setSelectedCost: process(clk, bb_enable, node_0, node_1, node_2, node_3, node_4,
        set_flag_in_TreeSelector)
        begin
                if(bb_enable = '1') then
                        if(clk'event and clk = '1') then
                                out_selected_cost <= selected_cost;
                                flag_level_0 <= node_0;
                                flag_level_1 <= node_4 & node_3 & node_2 & node_1;
                        end if;
                end if;
        end process;
end BestTree_arch;
```