University of Alberta

EFFICIENT ARCHITECTURES FOR 1-D AND 2-D LIFTING-BASED WAVELET TRANSFORMS

by

Hongyu Liao ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2005

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **EFFICIENT ARCHITECTURES FOR 1-D AND 2-D LIFTING-BASED WAVELET TRANSFORMS** submitted by **Hongyu Liao** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Mrinal K. Mandal, Supervisor

Dr. Bruce F. Cockburn, Co-Supervisor

Dr. Duncan G. Elliott

Dr. José Nelson Amaral

Date:

Abstract

The lifting scheme reduces the computational complexity of the discrete wavelet transform (DWT) by factoring the wavelet filters into cascades of simple lifting steps that process the input samples in pairs. We developed four compact and efficient hardware architectures for implementing lifting-based DWTs, namely, 1-D and 2-D versions of what we call recursive and dual-scan architectures. The 1-D recursive architecture exploits interdependencies among the wavelet coefficients by interleaving, on alternate clock cycles using the same datapath hardware, the calculation of higher-order coefficients along with that of the first-stage coefficients. The resulting hardware utilization exceeds 90% in the typical case of a 5-stage 1-D DWT operating on 1024 samples. The 1-D dual-scan architecture increases the datapath hardware utilization to 100% by processing two independent data streams together using shared functional blocks. The recursive and dual-scan architectures can be readily extended to the 2-D case. The 2-D recursive architecture is roughly 25% faster than conventional implementations, and it requires a buffer that stores only a few rows of the data array instead of a fixed fraction (typically 25% or more) of the entire array. The 2-D dual-scan architecture processes the column and row transforms simultaneously, and eliminates the memory buffer for the row transform coefficients.

Acknowledgements

I would like to acknowledge my supervisors, Dr. Bruce Cockburn and Dr. Mrinal Manal, for their patience and support on helping me to finish this thesis.

I would also like to thank the rest of my thesis committee members: Dr. Duncan Elliott and Dr. José Amaral. Their valuable feedback helped me to improve the thesis in many ways.

Table of Contents

List of Figures

List of Tables

List of Acronyms

9/7	Daubechies 9/7 wavelet [6]
D-4	Daubechies 4-tap wavelet [6]
Daub-4	Daubechies 4-tap wavelet [6]
DCT	Discrete Cosine Transform
DSA	Dual Scan Architecture
DSP	Digital Signal Processing/Processor
DSP-RAM	A DSP architecture proposed in [51]
DWT	Discrete Wavelet Transform
FIFO	First In, First Out
FIR	Finite Impulse Response
FWT	Fast Wavelet Transform
HH	Image subband with high frequency components horizontally and vertically
HL	Image subband with high frequency components horizontally and low frequency components vertically
IDWT	Inverse Discrete Wavelet Transform
JPEG	Joint Picture Experts Group
LH	Image subband with low frequency components horizontally and high frequency components vertically
LL	Image subband with low frequency components horizontally and vertically
PE	Processing Element
PSNR	Peak Signal-to-Noise Ratio
RA	Recursive Architecture
SIMD	Single Instruction stream, Multiple Data stream
SNR	Signal-to-Noise Ratio

Chapter 1

Introduction

Wavelets have been the subject of a great deal of research recently due, in large part, to promising applications in signal processing and data compression [26]. Compared to conventional Fourier analysis, signal analysis using the wavelet transform is more effective when analyzing physical situations where the signal contains discontinuities and sharp spikes. Many promising wavelet applications have been found in such areas as communications, controls, turbulence, human vision, radar, and earthquake prediction, etc. The wavelet transform has also been shown to be an excellent tool in data compression applications. For the same perceived visual quality, the 2-D wavelet transform outperforms the discrete cosine transform (DCT) in image compression [23]. Significantly, the 2-D biorthogonal discrete wavelet transform (DWT) has been adopted in the recently established JPEG-2000 still image compression standard [37].

1.1 Motivation

There are several trends that are currently motivating the search for improved signal processing using wavelets. The growth of Internet applications that require the transmission of sound, images, and video over limited bandwidth communication channels has been a major motivation for research on digital signal processing and compression. A more recent factor has been the rise in the popularity of digital photography, both still image and video. Such applications require techniques that reduce the volume of data while preserving the apparent quality. Different data compression methods offer different trade-offs between the compression efficiency and the perceived quality. Another research priority that is especially important in portable devices is to find improved data compression methods that require less digital hardware to implement and less power to operate.

One solution to the problem of maximizing the perceived image quality while minimizing the required data size is provided by the JPEG-2000 standard [37]. JPEG-2000 is a wavelet-based image compression standard that provides both lossy and lossless compression. Compared to the traditional JPEG standard [38], JPEG-2000 excels in applications that require greater quality and/or lower bit rates. Although there are some digital cameras on the market that support JPEG-2000 format image viewing, no camera offers on-board JPEG-2000 compression so far. One of the reasons for this appears to be finding a cost-effective solution that may be acceptable to potential customers. The computational burden of processing the JPEG-2000 algorithm in real-time is apparently beyond the capacity of the present signal processors in digital cameras. Recently, Analog Devices developed a JPEG-2000 chip for real-time applications [39]. However, cost and efficiency are still major considerations for the manufacturers of such chips. Since the DWT is the most computationally-intensive algorithm in JPEG-2000, a more efficient DWT core would likely improve the performance as well as reduce the cost of a JPEG-2000 processor. In addition, low power consumption and compact size are also critical priorities in designs for handheld devices.

Another important application of the wavelet transform is noise filtering or denoising. Wavelet denoising outperforms traditional filtering techniques in terms of effectiveness, flexibility and simplicity [24]. The most straightforward method of wavelet denoising is as follows: first, calculate the DWT of the input signal; then, discard the DWT coefficients that are less than certain thresholds at each given resolution scale; finally, reconstruct the input signal by calculating the inverse DWT (IDWT) of the remaining DWT coefficients [31]. Wavelet denoising has been successfully implemented in many real-time processing applications, such as, speech, radar signals, electrocardiogram-type (ECG-type) signal, and images [28][31]. Since the computation of the forward and inverse DWT typically consumes most of the CPU time during wavelet denoising, incorporating efficient DWT architectures in the above applications will significantly improve the real-time performance.

In order to satisfy the demand for real–time signal or image processing applications, improving the hardware implementations of the discrete wavelet transform has become very important. Many DWT architectures have been proposed in the last decade; most of them are based on Mallat's tree algorithm [3]. The lifting scheme is a relatively new, efficient algorithm for calculating the DWT and constructing wavelet bases. A few hardware implementations based on the lifting scheme have been proposed in the last few years [32][33]. However, these lifting-based architectures are not optimized for applications that read only one input sample at a time. They typically process pairs of samples. Since many digital systems have only one data bus, it is necessary to develop a lifting-based architecture that is efficient for a single input at a time applications.

In this thesis, we propose two kinds of lifting-based architectures, which we call the dual-scan and recursive architectures, to further improve the efficiency of the hardware implementation by exploiting the decimation structure of the lifting scheme algorithm.

The dual-scan architectures process two independent signals (e.g. two rows of an image) simultaneously to achieve 100% hardware utilization. The recursive architectures interleave the computations of all stages of the DWT into a shared datapath to achieve higher hardware utilization rates. The recursive architectures significantly reduce the memory requirements and off-chip memory access time, so they are not only faster but also smaller in hardware size and consume less power. Therefore, the recursive architectures should be especially well suited for hand-held devices.

1.2 Overview of the Wavelet Transform

Wavelets were first introduced in the early 1980s by J. Morlet as a mathematical tool for the analysis of seismic signals [1]. In the mid eighties, Mallat and Meyer introduced multiresolution analysis and the fast wavelet transform [3]. Based on their research, Daubechies achieved a breakthrough in wavelet research by constructing a set of compactly supported orthonormal wavelet basis functions [2]. Daubechies' wavelets are probably the most popular wavelet bases being used today.

Wavelets are a set of functions (or "building blocks") that satisfy certain mathematical requirements and can be used to represent other functions or signals. The most important property of the wavelets is scalability, which means that a wavelet function can be dilated to approximate the low frequency components of a signal, as well as be translated (shifted) to localize the time or space information of a signal. This is analogous to viewing a scene through a zoom lens: you can zoom out to see a bigger but vaguer and less detailed picture, or zoom in to reveal the details of an object in a more localized area. Hence, one could argued that "scale function" might have been more an appropriate name for wavelets.

Contrary to Fourier analysis, which uses sine or cosine basis functions that stretch out infinitely in time and are highly localized in frequency, the wavelet functions are finite and localized in both time and frequency. In other words, the sizes of the building blocks ('frequency') of a wavelet transform are finite, and the translation ranges are limited ('localized'). Because of this property, the wavelets can not only represent the low and high frequency components of a signal, they can also represent the time or location information of the signal. Hence, wavelets tend to be more efficient in situations where the signal contains discontinuities and sharp spikes.

The DWT underlies the wavelet analysis of digital signals. In the DWT, wavelets are translated by integers, and usually dilated (scaled) by powers of two. This particular kind of DWT is called the dyadic DWT [3]. The most widely adopted algorithm for calculating the DWT is Mallat's tree algorithm, or the fast wavelet transform (FWT), which uses filter bank techniques to reduce the DWT computational complexity to $O(n)$, where n is the number of signal samples (*i.e.* the signal length).

The lifting scheme, developed by Sweldens in 1996, was first used as a method to implement a reversible integer DWT [14]. Soon it was found that the lifting scheme could also be used as a new approach to construct biorthogonal wavelet bases [17]. The wavelet bases constructed using the lifting scheme are called second-generation wavelets to distinguish them from the classical wavelets. The second-generation wavelets are no longer created as the translation and dilation of one wavelet function; they can instead be constructed entirely in the spatial domain. The lifting scheme can be used to construct wavelets for grids of arbitrary dimensions and with irregular sampling intervals [25]. Later, Daubechies and Sweldens showed that any wavelet can be factored into lifting steps. By factoring the existing wavelets into lifting steps. the computational complexity can be reduced by up to 50% [20]. Due to the greater efficiency of the new algorithm, the lifting-based 9/7 and 5/3 wavelet filters have also been adopted in the recent JPEG-2000 standard [37].

1.3 Outline of the Thesis

The remainder of this thesis is organized as follows:

In Chapter 2, the wavelet transform and related concepts are introduced. The lifting scheme is also described in this chapter, with an emphasis on the factorization of wavelet filters. The lifting scheme and the factorization algorithm are the basis of our research.

Chapter 3 reviews the existing wavelet architectures that have been described in international journals and conferences. The idea of recursive architectures was inspired by some of the previous implementations of Mallat's algorithm.

The proposed 1-D architectures are described in Chapter 4. In this chapter, we first introduce building blocks for the proposed architectures, and then describe the 1-D dual-scan architecture and the 1-D recursive architecture. We present the Daubechies-4 and 9/7 wavelet architectures as examples for implementing symmetric and asymmetric wavelet filters. In Chapter 5, we describe the 2-D dual-scan architecture and the 2-D recursive architecture. We also show how these architectures can improve the hardware utilization and reduce the required memory size.

In Chapter 6, we discuss hardware implementation issues associated with the proposed architectures. All of the proposed architectures were implemented and verified in simulation using VHDL models. We synthesized the recursive architectures using the

Xilinx ISE logic synthesis environment [44] to determine the actual sizes of the proposed architectures in a field-programmable gate array (FPGA). To estimate the size of these architectures in semi-custom integrated circuit designs, we compiled the same VHDL models using the Synopsys Design Analyzer software tool [45], and placed and routed them with the Cadence Design Framework II tools [46] provided by Canadian Microelectronics Corporation (Kingston, ON). Evaluations of the proposed architectures are also provided in this chapter.

Finally, we conclude our thesis in Chapter 7 by summarizing the contributions, and proposing possible future extensions of our research.

Chapter 2

Review of Wavelet Theory

Wavelet theory is a relatively recent development in applied mathematics, and it has been evolving very fast since it was formally introduced less than two decades ago [1]. As a matter of fact, new wavelets and new concepts are emerging at such a rate that the very meaning of "wavelet analysis" keeps changing to incorporate new ideas [14]. In this chapter, we will briefly introduce the formal definitions of key concepts, such as, wavelets, the discrete wavelet transform, and the lifting scheme. First, we review the relevant theory of wavelets and the wavelet transform based on classical multiresolution analysis [3].

2.1 Wavelets

Informally, a wave is an oscillating function of time or space, such as a sinusoid. The fundamental idea behind wave analysis is to decompose a signal f into a weighted sum or linear combination of wave functions Ψ_i. Thus a function f is to be expressed as:

$$f = \sum_i a_i \Psi_i . \tag{2.1}$$

In order to accurately approximate the signal f using a relatively small number of coefficients a_i, it is important to select an appropriate family of functions Ψ_i. Fourier analysis is the traditional wave analysis technique in which signals are represented with weighted sums of sinusoids. A drawback of Fourier analysis is that the individual sinusoids extend infinitely along the input dimensions. It is difficult, therefore, to accurately represent an impulse signal (a "spike") using Fourier analysis because the corresponding Fourier series would require an infinite number of terms. In addition, localization of the spike in the continuous domain would be unclear from the spectrum in the frequency domain.

Compared to the wave functions used in Fourier analysis, wavelets are oscillating waves that are limited in both time (or space) and frequency (or scale). By "limited" we mean that the amplitude envelopes of the oscillating wavelet basis functions becomes varnishingly small outside a finite range of time and frequency. This is still not a precise definition of a wavelet. More precisely, all wavelets have the following properties [14].

1. Wavelets are building blocks for representing general functions. If the wavelet set is denoted by $\Psi_{j,k}(t)$, for indices j,k = 1,2...., a linear expansion of a time domain function $f(t)$ would be $f(t)=\sum_k\sum_j a_{j,k}\Psi_{j,k}(t)$ for some set of wavelet coefficients $a_{j,k}$.

2. Wavelets as well as most signals of interest are localized in both space and frequency. This means most of the energy of typical signals tends to be well represented by only a few expansion coefficients, $a_{j,k}$, of the signal's wavelet decomposition. Hence wavelets are often able to compactly represent localized features in a signal.

3. Wavelets are supported by fast transform algorithms. More precisely, the computational complexity of calculating the wavelet coefficients of a signal is O(n) or O($n$log$n$) in the length n of the signal.

2.2 Multiresolution Analysis and the Wavelet Transform

A systematic way of constructing a wavelet basis is provided by multiresolution analysis [3]. First, we introduce some concepts used in our discussions.

2.2.1 Conventions

- $\mathbb{R}$ is the set of real numbers;
- $\mathbb{Z}$ is the set of integers;
- $\mathbb{R}^n$ is the Euclidean vector space:

$$\mathbb{R}^n = \left\{ \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} : y_1, \cdots, y_n \in \mathbb{R} \right\}$$

- $L^2(\mathbb{R})$ denotes Hilbert function space, which is the vector space of square integrable functions in $\mathbb{R}$, and $L^2(\mathbb{R})$ is defined as:

$$L^2(\mathbb{R}) = \left\{ f(x) \in \mathbb{R} \mid \int_{-\infty}^{\infty} f^2(x)dx < \infty \right\}$$

- $\langle f, g \rangle$ is the inner product of functions *f* and *g*.

In Hilbert function space, the inner product of two functions *f* and *g* is defined as:

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(x)g(x)dx .$$

In a Euclidean vector space, the inner product $\langle \underline{x}, \underline{y} \rangle$ of two equal length vectors $\underline{x}, \underline{y}$ can be expressed as:

$$\langle \underline{x}, \underline{y} \rangle = \left\langle \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \right\rangle = \sum_{i=1}^{n} x_i y_i .$$

- $\|f\|$ is the norm of f, and is defined by $\|f\| = \sqrt{\langle f, f \rangle}$.
- $A \oplus B$ is the direct sum of two sets of integers, A and B, and where $A \oplus B = \{a + b : a \in A \text{ and } b \in B\}$.
- $\boldsymbol{a} \perp \boldsymbol{b}$ denotes that vectors a and b are orthogonal. Orthogonalty is equivalent to the following condition:

$$\boldsymbol{a} \perp \boldsymbol{b} \Leftrightarrow \langle \boldsymbol{a}, \boldsymbol{b} \rangle = 0.$$

- $\delta_{i,j}$ is the *Kronecker delta* function, defined by $\delta_{i,j} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$.

2.2.2 Multiresolution Analysis

In multiresolution analysis [1] we decompose space $L^2(\mathbb{R})$ into a sequence of nested subspaces V_j as follows:

$$\cdots \subset V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \subset \cdots \subset L^2(\mathbb{R})$$

The relationship between nested subspaces V_j and V_{j+1} is given by:

$$V_j \subset V_{j+1}, \forall j \in \mathbb{Z},$$

with their union closure

$$\bigcup_{-\infty}^{+\infty} V_j = L^2(\mathbb{R}),$$

and their intersection is zero signal, denoted by $\{0\}$, which means as j goes to negative infinity, V_j shrinks down to zero.

$$\bigcap_{-\infty}^{+\infty} V_j = \{0\}.$$

If we define a function $f(x)$ that belongs to one subspace V_j, and the size of subspace V_{j+1} is twice as large as V_j, then $f(x)$ has the following properties:

1. $$f(t) \in V_j \Leftrightarrow f(2t) \in V_{j+1} \qquad (2.2)$$

2. $$f(t) \in V_j \Leftrightarrow f(t-2^{-j}k) \in V_j, \quad k \in \mathbb{Z} \qquad (2.3)$$

We call the property in Eq. 2.2 *dilation* or *scaling* from one time resolution to the next, and we call the property in Eq. 2.3 *translation* or *shifting* in time.

If we can find a function $\varphi(t) \in L^2(\mathbb{R})$ such that the collection $\{\varphi(t-k) : k \in \mathbb{Z}\}$ forms a Riesz basis* for subspace V_0, then we call this function a *scaling function* or *father function.* For the other subspaces V_j, $j \neq 0$, we can define a set of functions from the scaling function $\varphi(t)$ by scaling and translation as follows:

$$\varphi_{j,k}(t) = 2^{j/2}\varphi(2^j t - k). \qquad (2.4)$$

If we denote the orthogonal complement of V_j in V_{j+1} by W_j, then by definition V_{j+1} can be decomposed as:

$$V_{j+1} = V_j \oplus W_j, \text{ where } W_j \perp V_j. \qquad (2.5)$$

Hence, $L^2(\mathbb{R})$ can also be expressed as:

$$L^2(\mathbb{R}) = \bigcup_{j=-\infty}^{+\infty} V_j = V_0 \oplus W_0 \oplus W_1 \oplus \cdots = \bigoplus_{j=-\infty}^{+\infty} W_j. \qquad (2.6)$$

This means that V_j is a "coarse-resolution" representation of V_{j+1}, and W_j represents the "high-resolution" difference between V_{j+1} and V_j. Therefore, the W_j can also be called "detail subspaces".

---

* $\{f(t-k) \mid k \in \mathbb{Z}\}, f(t) \in L^2(\mathbb{R})$ forms a Riesz basis if the following condition is satisfied: if $0 < A \le B < \infty$, then $A\|\{c_k\}\|^2 \le \|\sum_{k\in\mathbb{Z}} c_k f(t-k)\|^2 \le B\|\{c_k\}\|^2, \forall\{c_k\} \in L^2(\mathbb{R})$.

If we can find a function $\psi(t) \in W_0$ such that

$$\psi(t) \in W_0 \Leftrightarrow \psi(t-k) \in W_0,$$

and the collection $\psi(t-k), k \in \mathbb{Z}$, form an orthonormal* basis for W_0, then $\psi(t)$ is a *wavelet function* or *mother function.* For the other subspaces W_j, $j \neq 0$, we can also define:

$$\psi_{j,k}(t) = 2^{j/2}\psi(2^j t - k). \tag{2.7}$$

Because we require $W_j \perp V_j$ in Eq. 2.5, the corresponding wavelet functions and scaling functions are orthogonal:

$$\langle \varphi_{j,k}(t), \psi_{j,l}(t) \rangle = \int \varphi_{j,k}(t) \cdot \psi_{j,l}(t)dt = 0, \quad j,k,l \in \mathbb{Z}.$$

2.2.3 The Discrete Wavelet Transform (DWT)

From Eq. 2.6, we have

$$L^2(\mathbb{R}) = V_{j_0} \oplus W_{j_0} \oplus W_{j_0+1} \oplus \cdots, \text{ where } j_0 \in \mathbb{Z}.$$

Therefore, for a specific resolution level j_0, any function $f(t) \in L^2(\mathbb{R})$ can be represented as a series involving the projections of itself onto the orthonormal subspaces V and W:

$$f(t) = \sum_{k=-\infty}^{\infty} c_{j_0}(k) \cdot \varphi_{j_0,k}(t) + \sum_{k=-\infty}^{\infty} \sum_{j=j_0}^{\infty} d_j(k) \cdot \psi_{j,k}(t). \tag{2.8}$$

Replacing $\varphi_{j_0,k}(t)$ and $\psi_{j,k}(t)$ using Eqs. 2.4 and 2.7, we have

$$f_{j_0}(t) = \sum_{k=-\infty}^{\infty} c_{j_0}(k) 2^{j_0/2} \varphi(2^{j_0} t - k) + \sum_{k=-\infty}^{\infty} \sum_{j=j_0}^{\infty} d_j(k) 2^{j/2} \psi(2^j t - k). \tag{2.9}$$

* Vector set $\{v_k \mid k \in \mathbb{Z}\}$ is orthonormal, if $\langle v_k, v_j \rangle = \delta(k-j)$.

Consequently, function $f(t)$ is now expanded into a new form of series, called a *wavelet series*. The coefficients $c_j(k)$ and $d_j(k)$ are called the *discrete wavelet transform* (DWT) of the signal $f(t)$. The $c_j(k)$ represent the *low frequency components* of the signal $f(t)$, while the $d_j(k)$ represent the *high frequency components*. If the wavelet system is orthogonal (as we assumed before), then the DWT coefficients can be calculated as the following inner products [26]:

$$c_j(k) = \langle f(t), \varphi_{j,k}(t) \rangle = \int f(t) \cdot \varphi_{j,k}(t) dt$$

and

$$d_j(k) = \langle f(t), \psi_{j,k}(t) \rangle = \int f(t) \cdot \psi_{j,k}(t) dt .$$

2.2.4 The Fast Wavelet Transform (FWT)

Since

$$V_0 \subset V_1 \text{ and } W_0 \subset V_1,$$

$\varphi(t)$ and $\psi(t)$ can be expressed in terms of a weighted sum of shifted versions of $\varphi(2t)$:

$$\varphi(t) = \sqrt{2} \sum_{k=-\infty}^{\infty} h(k) \cdot \varphi(2t-k), \quad k \in \mathbb{Z}$$

$$\psi(t) = \sqrt{2} \sum_{k=-\infty}^{\infty} g(k) \cdot \varphi(2t-k), \quad k \in \mathbb{Z},$$

where the coefficients $h(k)$ and $g(k)$ are sequences of real or complex numbers that uniquely define the scaling function $\varphi(t)$ and the wavelet function $\psi(t)$, respectively. The factor $\sqrt{2}$ maintains the norm of the scaling function unchanged.

Since $V_{j+1} = V_j \oplus W_j$, we can also rewrite Eq. 2.8 for any resolution level j as

$$f_j(t) = \sum_{l=-\infty}^{\infty} \lambda_{j,l} \cdot \varphi_{j,l}(t) + \sum_{l=-\infty}^{\infty} \gamma_{j,l}(l) \cdot \psi_{j,l}(t) \qquad (2.10)$$

with the l^{th} transform coefficients $\lambda_{j,l}$ and $\gamma_{j,l}$ at stage j defined by:

$$\lambda_{j,l} = \sqrt{2} \sum_{k=-\infty}^{\infty} h_{k-2l} \cdot \lambda_{j+1,k} \qquad (2.11)$$

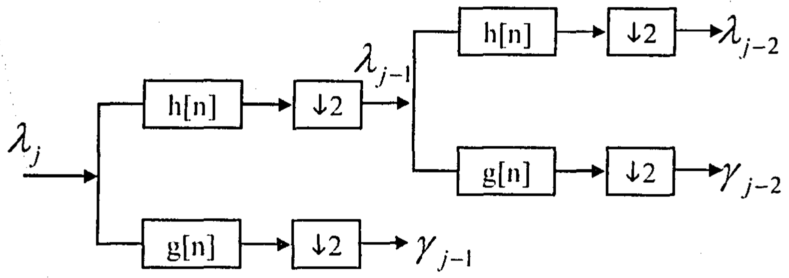
$$\gamma_{j,l} = \sqrt{2} \sum_{k=-\infty}^{\infty} g_{k-2l} \cdot \lambda_{j+1,k} \,. \qquad (2.12)$$

The vectors of $\{h_k\}$ and $\{g_k\}$ can be considered to be the filter coefficients of a pair of low-pass and high-pass digital filters, respectively. If these filters are finite impulse response (FIR) filters, the computational complexity of the coefficients $\lambda_{j,l}$ and $\gamma_{j,l}$ is $O(n)$, that is, linearly proportional to the signal length n. Compared to the DWT defined in Eq. 2.10, the computational cost is significantly reduced. Hence, the transform defined in Eqs. 2.11 and 2.12 is called the *fast wavelet transform* (FWT).

In the FWT algorithm, starting with the scaling coefficients at a given resolution (stage), the wavelet coefficients can be calculated recursively using Eqs. 2.11 and 2.12. These two equations can, therefore, be implemented efficiently using the tree-shaped filter bank structure shown in Figure 1(a). At each resolution level, the scaling coefficients $\{\lambda_j\}$ of the previous level are first filtered by the two half-band filters, h[n] and g[n], and then the outputs of the filters are decimated by a factor of two, yielding $\{\lambda_{j-1}\}$ and $\{\gamma_{j-1}\}$. The decimated low-pass filter output signal $\{\lambda_{j-1}\}$ can be further decomposed as shown in the figure. Because of the tree-like structure of the filter bank, this algorithm is also known as Mallat's *tree algorithm* or the *pyramid algorithm*.

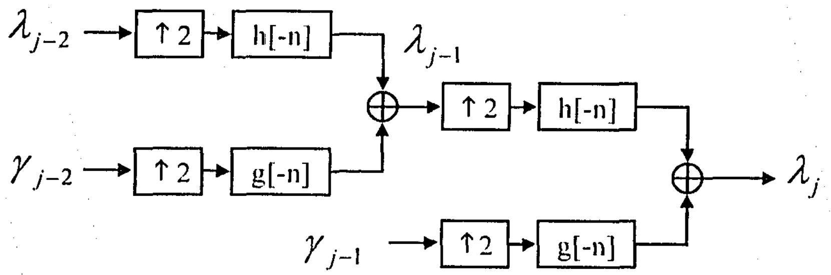
The inverse transform can be obtained by reversing the operations: the original signal $\{\lambda_{j,l}\}$ at stage j can be reconstructed recursively from the lower resolution coefficients using the following equation:

$$\lambda_{j,l} = \sum_{k=-\infty}^{\infty} \lambda_{j-1,k} \cdot h_{l-2k} + \sum_{l'=-\infty}^{\infty} \varphi_{j-1,l'} \cdot g_{l-2l'} \qquad (2.13)$$



↓2 = Down-sampling by a factor of two

(a)



↑2 = Up-sampling by a factor of two

(b)

Figure 1. Structure of the Pyramid Algorithm.
a) Two-stage signal decomposition using analysis filters. b) Two-stage signal reconstruction using synthesis filters, where h[n] and g[n] are the low-pass and high-pass filter coefficients, respectively.

In Eq. 2.13, note that the filter coefficients $\{h_k\}$ and $\{g_k\}$ are identical to those used in Eq. 2.11-12, but they appear in the reverse order. The structure of the corresponding inverse DWT pyramid algorithm is shown in Figure 1(b). The DWT coefficients at each transform level are first up-sampled (interpolated) by a factor of two, and then filtered by a pair of synthesis filters g[-n] and h[-n], where "-n" denotes the time reversal of the filter coefficients. The outputs of both filters are summed for further composition at the next level.

2.2.5 Orthogonal Wavelets

If $\begin{cases} W_j \perp V_j \\ \langle \varphi_{j,k}(t), \varphi_{j,l}(t) \rangle = \delta_{k,l} \\ \langle \psi_{j,k}(t), \psi_{j',l}(t) \rangle = \delta_{k,l}\delta_{j,j'} \end{cases}$, $j, k, l \in \mathbb{Z}$, then $\varphi_{j,k}(t)$ and $\psi_{j,k}(t)$ are orthonormal. Wavelets that satisfy such conditions are called *orthogonal wavelets*. The DWT coefficients $\{\lambda\}$ and $\{\gamma\}$ of such wavelets can be calculated by the inner products of the scaling and wavelet functions as:

$$\begin{cases} \lambda_{j,l} = \langle f, \varphi_{j,l} \rangle \\ \gamma_{j,l} = \langle f, \psi_{j,l} \rangle \end{cases}.$$

According to Parseval's theorem, the energy of the signal $f(t)$ is equal to the sum of the energy of their wavelet coefficients:

$$\|f\|^2 = \sum_l \lambda_{j,l}^2 + \sum_l \gamma_{j,l}^2 . \qquad (2.14)$$

Therefore, a decomposition in the orthonormal wavelet basis is considered to be stable because a slight change in $f(t)$ will only cause slight changes in $\lambda_{j,l}$ and $\gamma_{j,l}$.

2.2.6 Biorthogonal Wavelets

The conditions for constructing an orthonormal basis can be relaxed to a two-bases system, with primal and dual bases as follows:

$$\text{primal: } V_j,\ W_j,\ \varphi_{j,k}(t),\ \psi_{j,k}(t)$$

$$\text{dual: } \tilde{V}_j, \tilde{W}_j, \tilde{\varphi}_{j,k}(t), \tilde{\psi}_{j,k}(t).$$

Bases that satisfy the conditions

$$\begin{cases} \tilde{V}_j \perp W_j \\ V_j \perp \tilde{W}_j \\ \langle \tilde{\varphi}_{j,k}(t), \varphi_{j,l}(t) \rangle = \delta_{k,l} \\ \langle \tilde{\psi}_{j,k}(t), \psi_{j',l}(t) \rangle = \delta_{k,l}\delta_{j,j'} \end{cases}, j,k,l,j',k',l' \in \mathbb{Z}.$$

are called *biorthogonal bases*. The wavelet coefficients λ and γ are calculated by the following equation set:

$$\begin{cases} \lambda_{j,l} = \langle f, \tilde{\varphi}_{j,l} \rangle \\ \gamma_{j,l} = \langle f, \tilde{\psi}_{j,l} \rangle \end{cases}, j,l \in \mathbb{Z}.$$

The reverse transform can still be calculated using Eq. 2.13. Hence, for the biorthogonal wavelets, the tree algorithm can be implemented by using the *dual filter pair* $(\tilde{h}, \tilde{g})$ for decomposing the input signal and *the primal filter pair* (h, g) for reconstruction.

For the biorthogonal wavelets, Parseval's theorem does not hold, but the energy of the wavelet coefficients is still limited:

$$A\|f\|^2 \le \sum_l \lambda_{j,l}^2 + \sum_l \gamma_{j,l}^2 \le B\|f\|^2,$$

where $0 < A \le B < \infty$. Biorthogonal wavelets have some special features, one of which is that it is possible to synthesize biorthogonal wavelets and scaling functions which are symmetric or antisymmetric and compactly supported[◊]. This makes it possible to use the folding technique for boundary treatment, as will be described in this chapter.

2.2.7 The Two-Dimensional DWT

To calculate a higher dimensional DWT, there are two major approaches: one is the separable algorithm, and the other one uses real (non-separable) multi-dimensional wavelets [7]. The separable algorithm involves calculating the 1-D DWT for each dimension independently. The structure of the common separable 2-D DWT algorithm is shown in Figure 2, where G and H represent the lowpass and highpass subband filters, respectively. We first calculate the 1-D DWT horizontally on the rows of the input image, and then calculate the 1-D DWT vertically on the columns of coefficients from the horizontal DWT results. Consequently, the image is decomposed into four *subbands*, usually denoted by *LL, LH, HL,* and *HH.* The *LL* subband can then be further decomposed recursively using the same algorithm.

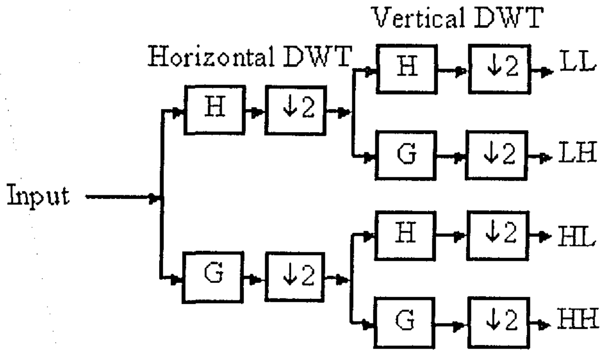


Figure 2. Block Diagram of the 2-D Separable DWT

[◊] For a function $f(t)$ with "finite energy" ($\int_{-\infty}^{\infty} |f(t)|^2 < \infty$), if $f(t)=0$ when $t< T_1$ and $t> T_2$, then the interval $[T_1, T_2]$ is called the *support* of f, and we say f has *compact support*.

Intuitively, we expect that the separable 2-D DWT would be efficient for approximating images with important details that are aligned in the vertical and horizontal directions, but may not be ideal for efficiently representing detail in other directions. The non-separable 2-D DWTs use real-valued 2-D wavelet bases, which can preserve the directional information in the original image. Several methods of constructing the non-separable wavelet bases have been proposed [7][18][21].

Since the computational complexity of the separable algorithm is much lower, it is widely used in most applications. Due to its popularity, we investigated efficient implementations of this algorithm in our proposed architectures.

2.3 The Lifting Scheme

While new wavelets with desirable properties are in demand, constructing them in the traditional ways requires significant computational effort. The lifting scheme provides a simple and efficient new method for constructing and processing wavelets [11]. Unlike the traditional approaches, the lifting scheme does not rely on the frequency domain but instead constructs wavelets purely in the spatial domain. Thus, the lifting scheme is more flexible for building wavelets for specific applications.

The wavelets constructed by the lifting scheme are called *second-generation wavelets*; wavelets restricted to the translations and dilations of one mother wavelet function are now referred to as *first-generation wavelets* or *classical wavelets*. The second-generation wavelets are more general since all of the classical wavelets can be generated by the lifting scheme. The decomposition of any classical wavelet filter into lifting steps can be easily obtained via the Euclidean algorithm [20].

2.3.1 Introduction to the Lifting Scheme

As described in the first section, the wavelet transform of a signal is a multi-resolution representation of that signal using wavelets as the basis functions. At each level, the low-pass part of the signal is decomposed recursively into a high-pass and a low-pass part at the next lower resolution.

The *lifting scheme* is an efficient implementation of these filtering operations at each level when computing a discrete wavelet transform. Suppose we have the original samples $\{\lambda_{0,k}\}$ of a signal $f(t)$. We want to decompose, or decorrelate, this signal into the low resolution part $\{\lambda_{j,k}\}$ and the high resolution part $\{\gamma_{j,k}\}$, where the index $j \in (-1,-2,-3,\ldots)$ identifies the decomposition stage or level. The lifting process consists of three steps: *split*, *predict*, and *update*, as shown in Figure 3.

We begin with a trivial wavelet, called the "Lazy wavelet", which just splits the signal into even and odd parts. This step can be expressed as:

$$\begin{cases} \lambda'_{j,k} = \lambda_{j,2k+1} \\ \gamma_{j,k} = \lambda_{j,2k} \end{cases}, \forall k \in \mathbb{Z}, \tag{2.14}$$

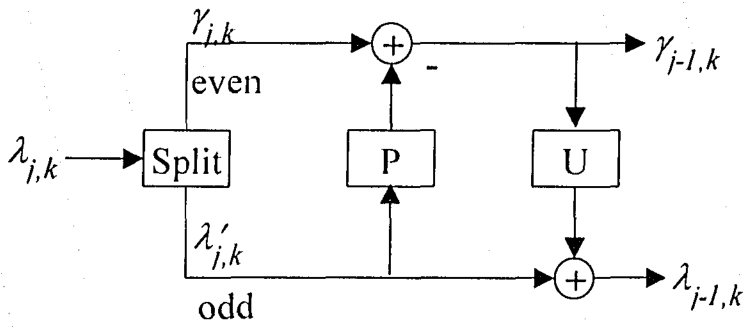


Figure 3. The Split, Predict, and Update Steps in the Lifting Scheme.

Then we predict the even samples $\{\gamma_{j,k} \mid k \in \mathbb{Z}\}$ from the odd samples $\{\lambda_{j,k} \mid k \in \mathbb{Z}\}$, based on correlation present in the original data. so that:

$$\gamma_{j-1,k} = \gamma_{j,k} - P(\lambda'_{j,k}) \tag{2.15}$$

where P is the *predict operator*. This is called the *real decorrelating step*.

Finally, we construct an update operator and update $\{\lambda'_{j,k} \mid k \in \mathbb{Z}\}$ as follows:

$$\lambda_{j-1,k} = \lambda'_{j,k} + U(\gamma_{j-1,k}) \tag{2.16}$$

These steps can be iterated over decreasing values of the index j creating a *multi-level transform* or *multi-resolution decomposition.*

The inverse transform can be performed by simply reversing the steps and interchanging the operations + and -. The algorithm for the corresponding inverse transform can be expressed as:

$$\begin{cases} \lambda'_{j+1,k} = \lambda_{j,k} - U(\gamma_{j,k}) \\ \gamma_{j+1,k} = \gamma_{j,k} + P(\lambda'_{j,k}) \qquad , -n \le j \le n, \\ \{\lambda_{j+1,k}\} = Join(\lambda'_{j+1,k}, \gamma_{j+1,k}) \end{cases} \tag{2.17}$$

where n defines the range of the decomposition stage, and the *Join* function refers to the interleaved composition of λ (as odd part) and γ (as even part) into a new signal.

2.3.2 Factoring Wavelet Filters into Lifting Steps

It is shown in [20] that *any* polyphase matrix representing a wavelet transform with finite filters can be factored into a finite product of upper and lower triangular 2×2 matrices, and a diagonal normalization matrix. Each of these matrices corresponds to a *lifting step*. Factoring wavelets into lifting steps can significantly improve the wavelet processing speed if the proper filters are developed. To factor an existing wavelet, the high-pass and low-pass wavelet transform filters should first be separated into even and odd parts. Consider the high-pass wavelet filter $g(z)$ and low-pass wavelet filter $h(z)$ as follows:

$$g(z) = \sum_{i=0}^{J-1} g_i(z) \tag{2.18}$$

$$h(z) = \sum_{i=0}^{J-1} h_i(z), \tag{2.19}$$

where J is the filter length. We can split the high-pass and low-pass filters into even and odd parts as follows:

$$g(z) = g_e(z^2) + z^{-1} g_o(z^2)$$
$$h(z) = h_e(z^2) + z^{-1} h_o(z^2)$$

The filters can also be expressed as an *analysis polyphase matrix* as follows:

$$P(z) = \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \tag{2.20}$$

Using the Euclidean algorithm [20], which recursively finds the greatest common divisors of the even and odd parts of the original filters, each of the filter polynomials in the matrix can be expressed in the form $f(z) = c(z) \cdot q(z) + b(z)$. Therefore, the forward transform polyphase matrix $\tilde{P}(z)$ can be factored into lifting steps as follows:

$$\tilde{P}(z) = \prod_{i=1}^{m} \begin{bmatrix} 1 & 0 \\ -s_i(z^{-1}) & 1 \end{bmatrix} \begin{bmatrix} 1 & -t_i(z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1/K & 0 \\ 0 & K \end{bmatrix}, m \leq K. \tag{2.21}$$

where $s_i(z)$ and $t_i(z)$ are Laurent polynomials* corresponding to the update and prediction steps, respectively, and K is a non-zero constant. The inverse DWT is described by the following equation:

$$P(z) = \prod_{i=1}^{m} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \qquad (2.22)$$

As an example, the low-pass and high-pass filters corresponding to the Daubechies 4-tap wavelet can be expressed as [20]:

$$h(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3}$$
$$g(z) = -h_3 z^2 + h_2 z^1 - h_1 + h_0 z^{-1},$$

where

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}, h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}, h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}, h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

After separating the filters into even and odd parts, and arranging them in the form of Eq. 2.20, we obtain the polyphase matrix:

$$P(z) = \tilde{P}(z) = \begin{bmatrix} h_0 + h_2 z^{-1} & -h_3 z - h_1 \\ h_1 + h_3 z^{-1} & h_2 z + h_0 \end{bmatrix}$$

The even portions of the filters (polynomials in the first row) can be expressed as:

$$\begin{cases} h_0 + h_2 z^{-1} = \left(h_1 + h_3 z^{-1}\right) s(z) + h_e^{new}(z) \\ -h_3 z - h_1 = \left(h_2 z + h_0\right) s(z) + g_e^{new}(z) \end{cases}$$

One of the common solutions of the above equations is $s(z) = -\sqrt{3}$, so we have

$$\tilde{P}(z) = \begin{bmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1+\sqrt{3}}{\sqrt{2}} & \frac{-1+\sqrt{3}}{\sqrt{2}} z \\ h_1 + h_3 z^{-1} & h_2 z + h_0 \end{bmatrix}$$

* A Laurent polynomial $P(x)$ in the n variables $x = (x_1, x_2, \ldots, x_n)$ is given by $P(x) = \sum_{(k_1,\ldots,k_n)\in A} P_{k_1\ldots k_n} x_1^{k_1} \ldots x_n^{k_n}$, where $P_{k_1\ldots k_n} \in \mathbb{C}$ and A, the support of $P(x)$, is a finite subset of integer group $\mathbb{Z}^n$.

Now, the odd portions of the filters can be expressed as:

$$\begin{cases} h_1 + h_3 z^{-1} = \dfrac{1+\sqrt{3}}{\sqrt{2}} t(z) + h_o^{new}(z) \\ h_2 z + h_0 = \dfrac{-1+\sqrt{3}}{\sqrt{2}} z t(z) + g_o^{new}(z) \end{cases}$$

We can find a common solution $t(z) = \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4} z^{-1}$. Repeating the above steps, we can further factor the analysis polyphase matrix as:

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

Similarly, one Daubechies 9/7 analysis wavelet filter with symmetric coefficients [6]:

$$\tilde{h}_e(z) = h_4(z^2 + z^{-2}) + h_2(z + z^{-1}) + h_0 \text{ and } \tilde{h}_o(z) = h_3(z^2 + z^{-1}) + h_1(z+1)$$

can be factored as [20]:

$$\tilde{P}(z) = \begin{bmatrix} 1 & \alpha(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta(1+z) & 1 \end{bmatrix} \begin{bmatrix} 1 & \gamma(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta(1+z) & 1 \end{bmatrix} \begin{bmatrix} \zeta & 0 \\ 0 & \zeta^{-1} \end{bmatrix}$$

where $\alpha=-1.58613$, $\beta=-0.05268$, $\gamma=0.88291$, $\delta=0.44351$, and $\varsigma=1.14960$.

Factoring wavelet transforms into lifting steps can significantly reduce the computational cost. Asymptotically, for long filters, the cost of computing the lifting algorithm is roughly one half of the cost of the standard algorithm using the original filters [20]. For the 9/7 wavelet filter example given above, the computational cost for each wavelet coefficient is five multiplications and four additions. Compared to the traditional Mallat's tree algorithm, which costs 14 multiplications and additions, we have a speedup of 56%. Consider a general wavelet filter which is not symmetric, and whose low-pass and high-pass filters have $|h| = 2N$ and $|g| = 2M$ taps, respectively. The cost of the standard algorithm is $4(N+M)+2$. Assuming $|h_e| = N$, $|h_o| = N-1$, $|g_e| = M$, and $|g_o| = M-1$, we can factor the (h_e, h_o) pair into N steps with the degree of each quotient set equal to one (*i.e.*, $|q_i| = 1$ for $1 \le i \le N$). The (g_e, g_o) pair can be factored into $M-N+1$ lifting steps [20]. So the total cost of the lifting algorithm is: 2 (scaling) + $4\times N$ + $2\times(M-N+1)$ = $2(M+N+2)$ operations. So we have a speedup of $\frac{N+M-1}{N+M+1} \approx 100\%$ as the length of the wavelet filter increases. Table 1 lists some examples of such comparisons [20].

Table 1. Computational cost of the lifting versus the standard algorithm for calculating a pair of wavelet coefficients

Wavelet	**Standard (ops)**	**Lifting (ops)**	**Speedup**
Haar	3	3	0%
D4	14	9	56%
D6	22	14	57%
9/7	23	14	64%
B_spline	17	10	70%
$\|h\|=2N$, $\|g\|=2M$	$4(N+M)+2$	$2(N+M+2)$	≈ 100%

2.4 Comparison of the Classical DWT and the Lifting Scheme

We can construct both orthogonal and biorthogonal wavelets using the classical algorithms [2][6], but we can only build biorthogonal wavelets using the lifting scheme [17]. However, the relatively easy construction of the lifting wavelets makes them attractive for many applications [11]. In addition, constructing wavelets on an interval (boundary wavelets) using the lifting scheme requires much less effort and mathematical training than doing it in the classical way [9][11]. The comparison of classical wavelets with wavelets constructed using the lifting scheme (second generation wavelets) is summarized in Table 2.

Although all classical wavelet filters can be factored into lifting steps, the derived lifting wavelets are no longer the same ones. One obvious reason is that all orthogonal and biorthogonal wavelet filters will be factored into orthogonal lifting steps. We did an experiment to demonstrate the difference. We calculated the one-stage D-4 DWT coefficients of an input series $(x_i, i = 1,2,\ldots,100)$ using both the standard algorithm and the lifting algorithm. The results are shown in Figure 4. Note that the coefficients generated by these two kinds of wavelets are similar but not identical. The lifting algorithm generates larger high frequency components and smaller low frequency components. This can also be seen by comparing the subimages of an image (Lena 512×512) decomposed by the two wavelets (as shown in Figure 5) and their energies (in Table 3).

Table 2. Comparison of the classic wavelet and the second-generation wavelet

	Classical Wavelets	Second-generation Wavelets
Orthogonality	Orthogonal/Biorthogonal	Biorthogonal
Computational Complexity	High	Low
Construction of Wavelets on an Interval	Difficult	Relatively Easy (By polynomial subinterpolation)

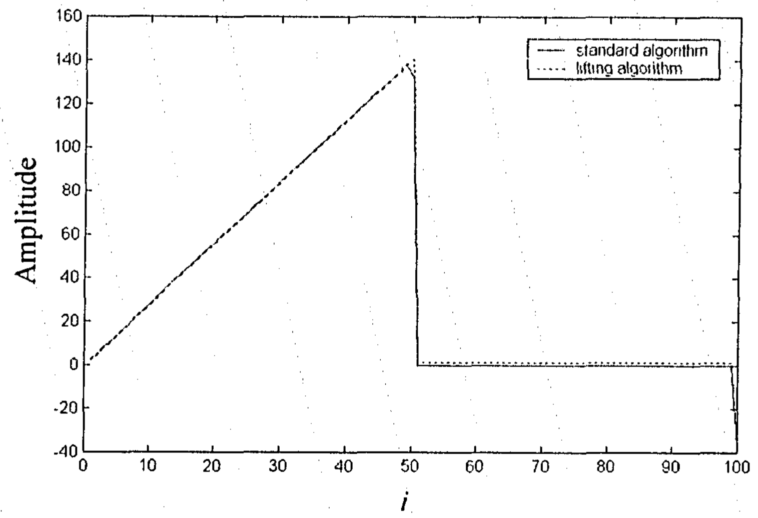
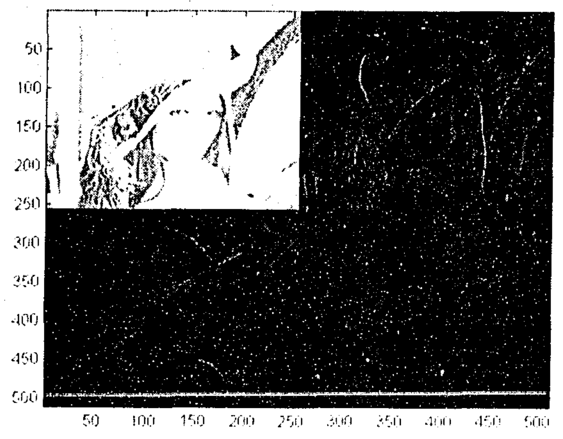


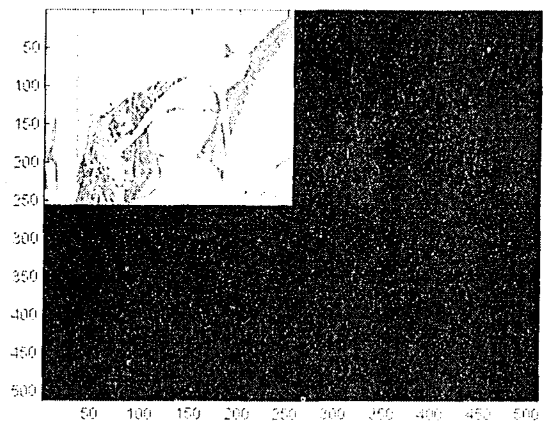
Figure 4. Comparison of the DWT Coefficients Computed by Classic Wavelets and Lifting Scheme

Table 3 Energy of the subimages generated by the standard and lifting D-4 DWT

	Standard D4				Lifted D4			
Stage	LL x10^9	LH X10^9	HL x10^9	HH X10^9	LL x10^9	LH x10^9	HL x10^9	HH x10^9
1	839.3	4.6270	2.406	1.807	29.04	4.6039	3.768	2.469
2	2366.6	4.6094	4.904	3.632	43.73	4.5556	6.924	4.420
3	5593.6	4.5771	9.382	7.165	71.13	4.4786	12.624	8.121
4	9936.5	4.5173	18.494	13.805	123.57	4.3579	23.084	14.902



(a) Subimages generated using the standard Daub-4 wavelet



(b) Subimages generated using the Daub-4 wavelet after being factored into lifting steps

Figure 5. Comparison of Subimages Generated by First (classical) and Second Generation Wavelets.

2.5 Boundary Treatment

Filter bank algorithms usually assume that the signal lengths are infinite. However, signals in the real world do not extend infinitely in time or space but are limited to finite intervals. Generally speaking, simply padding or extending the finite original signal with zeros is not appropriate since this would lead to more wavelet coefficients than the original samples. If one truncates the transformed function (which is equivalent of padding zeros to both ends of the signal) to yield the same number of wavelet coefficients as samples, the resulting transform will badly distort the signal (especially on boundaries) and thus render perfect reconstruction impossible using an inverse transform. What we need is some kind of safe signal extension, or to take into account the finiteness of the signal by changing the transform (using boundary wavelets) near the boundaries of the signal. Because boundary wavelet filters require changes in the filter structures, using signal extension is more appropriate and easier for hardware implementations. In this section, we will review signal extension methods for the DWT [52].

2.5.1 Classical Extension Methods

In classical signal processing applications, it is common practice to extend the data for the computations near the finite signal border by replicating the signal using one of the following three methods:

Zero Padding Extension

The simplest solution is the zero padding extension of a signal: extending the signal by simply padding zeros at both ends of the signal. However, the number of DWT coefficients generated by this solution is usually larger than the number of the signal samples, hence distortion is inevitably introduced in the reconstructed signal. The zero padding extension can only be implemented in limited situations, such as when the distortions at the intervals (boundary) of a signal can be ignored.

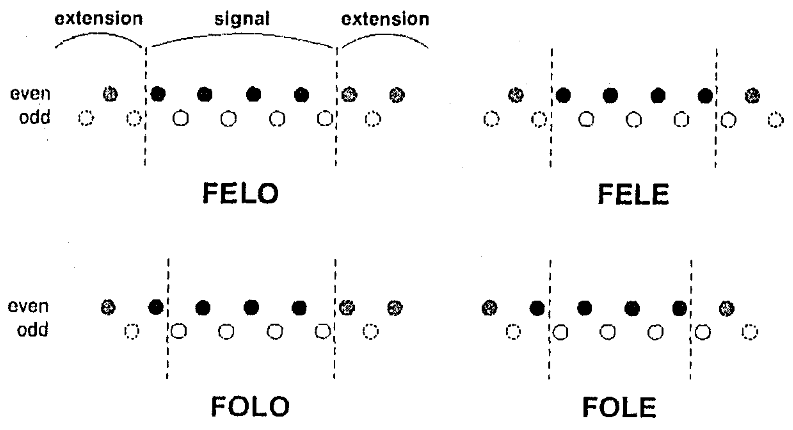
Periodic Extension

A second extension method is the periodic extension of the signal: considering the signal as one period of a periodic signal. After the wavelet transform we can simply discard the coefficients outside of the interval in which the original signal was defined. Since the computation of a wavelet transform is the convolution of the original signal with a FIR filter, the transformed coefficients also represent a periodic signal with the same period as that of the original signal, and thus the original signal can be recovered without any problems. However, unless the first and the last samples have the same value, we will have introduced unwanted discontinuities at the boundaries of the original signal. These discontinuities will generate larger wavelet coefficients in the higher frequency region and thus make compression of the signal less efficient.

Symmetric Extension

For biorthogonal wavelet transforms, a better solution is the symmetric extension of the signal: we extend the finite signal by mirroring it around its endpoints, which makes the discontinuities disappear. But the higher order derivatives may still be discontinuous at the endpoints. Symmetric extension can be considered to be the same as periodic extension applied to a concatenation of the original signal and a mirrored copy of the original signal. Hence after filtering, one has to retain twice as many coefficients. Fortunately, one can discard half of them if the filters are symmetric, yielding the same number of coefficients as the original signal length. The calculation of the DWT using symmetric extension is also called *wavelet folding* because it is equivalent to folding the wavelet filter symmetrically at the edges of the signal intervals.

While the periodic extension of a discrete signal is straightforward and unique, there are four possible ways to extend a discrete signal symmetrically. Figure 6 shows the four resulting possible symmetric signal extensions. However, to ensure to get a perfect reconstruction, one has to apply the correct extension in every step of the decomposition and reconstruction.



FELO First even, last odd; **FELE** First even, last even;
FOLO First odd, last odd; **FOLE** First odd, last even.

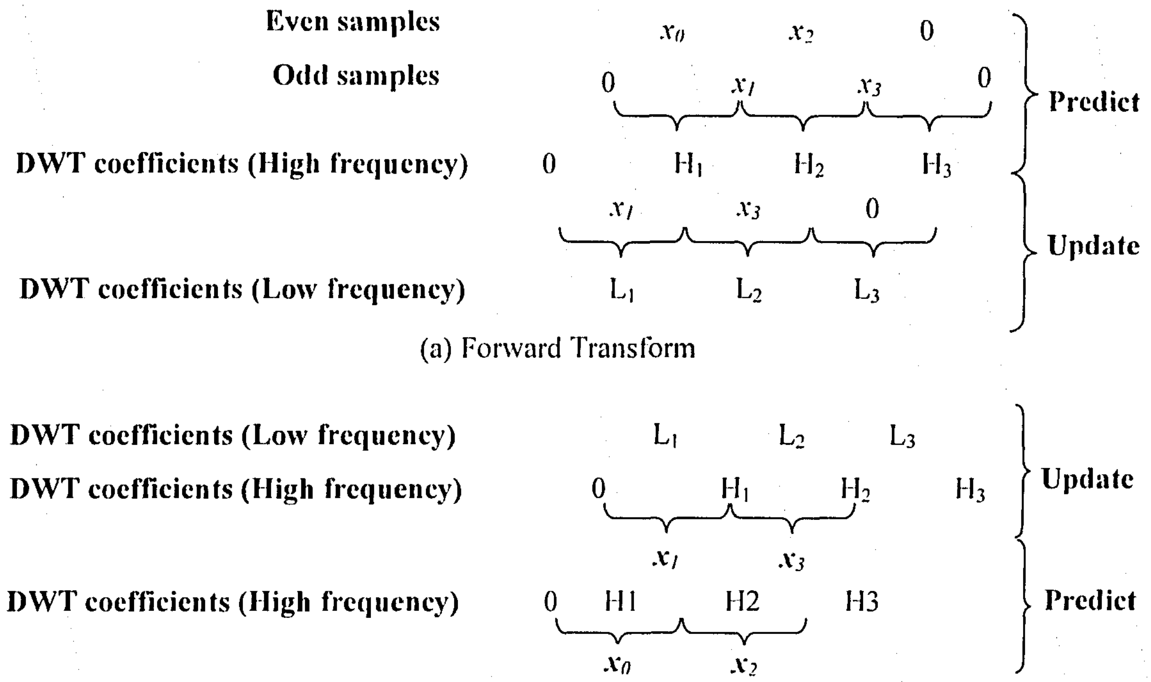
Figure 6 The Four Possible Cases of Symmetric Signal Extension Using Lifting.

2.5.2 Signal Extension with Lifting

A very nice property of second-generation wavelets is that symmetric extension is always possible. We are no longer limited to symmetric (biorthogonal) wavelets. Even if one has a lifting decomposition for a non-symmetric wavelet filter bank (e.g. orthogonal Daubechies wavelets), the implementation using lifting steps can use symmetric extension.

The simplest zero padding extension is also applicable for many second-generation wavelets. As introduced above, factoring classic wavelet transforms into lifting steps is actually replacing the longer wavelet FIR filters with a series of shorter FIRs, which can be two- or three-tap filters in most cases. The factored shorter FIRs are obtained by repeatedly applying the Euclidean algorithm, which finds the greatest common divisor of the even and odd parts (which can be expressed as polynomials $x_e(z^2)$ and $x_o(z^2)$, respectively) of the original longer FIR. Also, it will always be true that $|x_e(z^2)|-|x_o(z^2)|<=1$ (the difference of the degrees of the two filters), so we can always factor a classic wavelet into short FIRs not longer than three taps. But it is well known that signal extension is not necessary for two-tap wavelets, like the Haar wavelet. Thus, we can deduce that zero padding can be used in calculating the lifting schemes as long as each factored step is not longer than three taps.

As shown in Figure 7, when a three-tap predict FIR filter (assuming it is causal) convolves with a four-sample signal (x_i, i=0,1,2,3), the final result after decimation is H_1, H_2 and H_3. So the odd samples of the original signal and the padded zeros are updated to H_1, H_2, and H_3 while the even samples remain unchanged. Similarly, the update filter updates the even samples to L_1, L_2, and L_3. Since the inverse wavelet transform is just the inverse of the forward transform, as introduced above, the even sample x_0 can be restored by multiplying the update FIR filter with the padded zero, DWT coefficients H_1 and L_1. Further, the odd sample x_1 can be restored from 0, H_1, and the sample x_0, which is restored in the previous step. Similarly, samples x_2 and x_3 can be reconstructed using the coefficients H_1, H_2 and L_2. We can observe that the DWT coefficients H_3 and L_3 are redundant, and we can also reach the similar result for the non-causal filters (or one is causal and the other is non-causal) generated by the factoring process. Thus zero padding can be implemented while maintaining perfect reconstruction (PR) of the original signal. Zero padding avoids the extra storage required by other signal extensions and the complicated wavelets on the interval, and hence tends to result in a simpler hardware implementation.



Even samples
Odd samples
DWT coefficients (High frequency)
DWT coefficients (Low frequency)
Predict
Update
(a) Forward Transform
DWT coefficients (Low frequency)
DWT coefficients (High frequency)
DWT coefficients (High frequency)
Update
Predict
(a) Inverse Transform

Figure 7. Zero Padding Extension for Short FIR Filters

Chapter 3

Review of Existing Wavelet Architectures

With the widespread acceptance of the wavelet transform in more and more applications, especially its important role in the next generation image and video processing techniques, efficient hardware implementations of the transform are attracting more and more attention. The DWT algorithms described in the previous chapter can be conveniently implemented on modern computers or microprocessors using a high-level programming language, such as C or BASIC. However, the computational complexity of calculating the DWT algorithms, especially the multi-dimensional transforms, precludes most general-purpose microprocessors from real-time applications.

Digital Signal Processors (DSPs) tend to be much more efficient in demanding DWT and IDWT computations because their structures are optimized for convolution computations, which are essentially the basic computations for the fast DWTs. As a matter of fact, the DSPs have been widely applied to speed up the DWT calculation in many applications [28][29][30]. To improve the efficiency of DSP implementations, it is critical to shorten the access time for the intermediate DWT coefficients, which are processed recursively in the DWT algorithms. This is even more important for the two-dimensional, or higher dimensional applications. Excessive or poorly scheduled data transfer between a DSP and its memory, especially external off-chip memory, will greatly slow down the computation speed. Therefore, the best DSP implementation may not be as straightforward as a high-level computer implementation, if maximum efficiency is to be achieved.

We developed a 1-D DWT implementation on DSP-RAM [29]. DSP-RAM is a single instruction, multiple data (SIMD) processor with an array of simplified DSP processors tightly coupled with SRAM local memories [51]. Our implementation fully exploits the parallel processing and linear array architecture of DSP-RAM, and is capable of efficiently computing 1-D DWT pyramid algorithms.

Although conventional DSP processors are faster and more efficient in the DWT computation, they are not necessarily suitable for applications requiring low power consumption, low cost, and compact size. In such cases, utilizing dedicated circuits specially built for processing the DWT may be more appropriate. Because the data paths of the dedicated circuits are optimized for the DWT computation, and their control paths are simpler than the instruction execution logic in DSP processors, they can normally not only process the DWT more efficiently in terms of hardware utilization and power consumption but much faster (provided they operate at the same system clock frequency). Much previous work has described attempts to design efficient DWT architectures based on Mallat's tree algorithm. Recently, a few architectures based on the lifting DWT have been published. Since the lifting steps can be implemented with ladder type data flow structures, the natural lifting architectures are different from the direct FIR implementations. In the following sections, we will first review some typical classical architectures based on Mallat's algorithm, and then introduce the lifting architectures that have been published in recent years.

3.1 Classical Architectures

An early DWT architecture was Knowles' multiplexer-based architecture published in 1990 [4], as shown in Figure 8. The rectangles on the top of the figure represent register arrays. This architecture uses these serial-in-parallel-out register queues to store the input signal samples {s(i)} and the low-pass filter outputs of each DWT stage. A multiplexer (mux) selects the outputs of the queues (register arrays), and feeds them to the high-pass (labeled G) and low-pass (labeled H) DWT filter pair. The outputs ({ct_m}, where m denotes the DWT stage) of the low-pass DWT filter are dispatched to the queues by a demultiplexer (demux) unit. The high-pass DWT filter products {c_m} are sent to the output directly. The advantage of this architecture is that the data flow is regular, hence, it is efficient and relatively easy to implement in hardware.

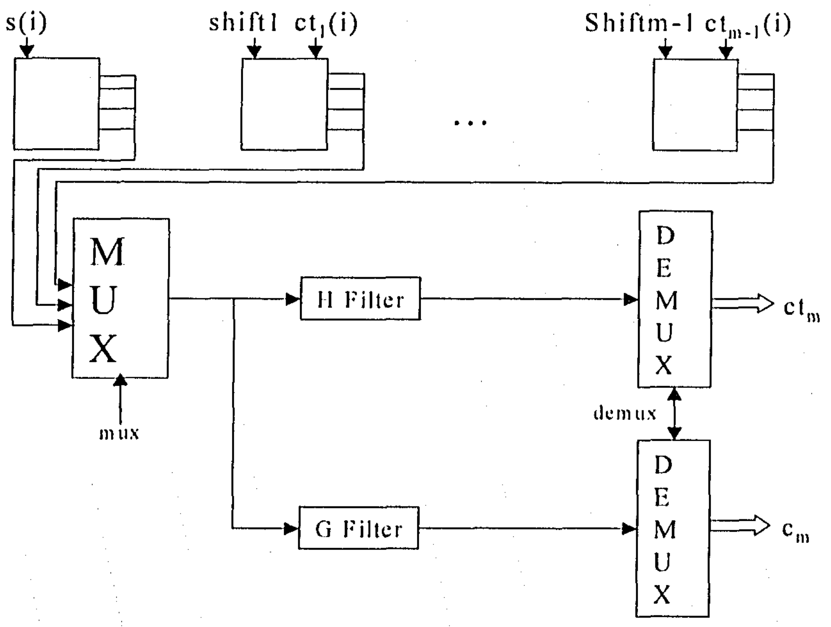


Figure 8. Knowles' Mux Based DWT Architecture [4]

Not long after Knowles proposed the mux-based DWT architecture. he and Lewis proposed possibly the first 2-D DWT architecture [5]. They analyzed the characteristics of the Daub-4 wavelet, and found that the multiplication computations in the Daub-4 DWT can be replaced by shift and add operations. Hence, the total number of transistors required to implement the convolver is only about one eighth of the number of a conventional convolver design. The exploitation of the Daub-4 DWT filter coefficients in this way is certainly a clever invention, but implementing the same technique on the other DWT filters is not feasible.

For processing the edges of an image, they proposed a data scan method, which snakes through the data, and reverses the scan direction on alternate lines. In this way, the input data stream is continuous as consecutive samples are only one pixel apart. Since the filter they proposed is not phase linear, the wavelets must be reversed at line changes. To reverse the wavelet, the signs of the coefficients in the normal reconstruction DWT filters are altered across the edges of the image. This solution for the boundary treatment is innovative and efficient for the multiplierless architecture. However, reversing the wavelet on each line means a extra set of DWT filters for other architectures.

Many more DWT architectures have been proposed since 1992. Two types of representative wavelet architectures, namely the folded architecture and the digit-serial architecture, were proposed in [8]. This folded architecture is shown as Figure 9. The delay array on the left side of Figure 9 is a serial-in-parallel-out register queue; the register array on the right side is a FIFO register array with output ports at each register. The outputs of the delay array are sent to the high-pass (i.e. G) and low-pass (i.e. H)

DWT filters at every other clock cycle. The high-pass filter outputs are shifted out as the final DWT coefficients; the low-pass filter outputs are shifted to the register array. The switch network selects input data for the DWT filters from the input delay array and the register array.

The folded architecture of the synthesis filter is similar to that of the analysis filter. The major difference is that the structure of the input array of the synthesis filter is also a register array, which converts the input data to the format (or sequence) required for the DWT synthesis computation.

The hardware utilization of the folded architecture is relatively high, but the interconnection network and the control circuit are complex. It is more suitable for applications that are sensitive to processing latency, and require the computation output as early as possible.

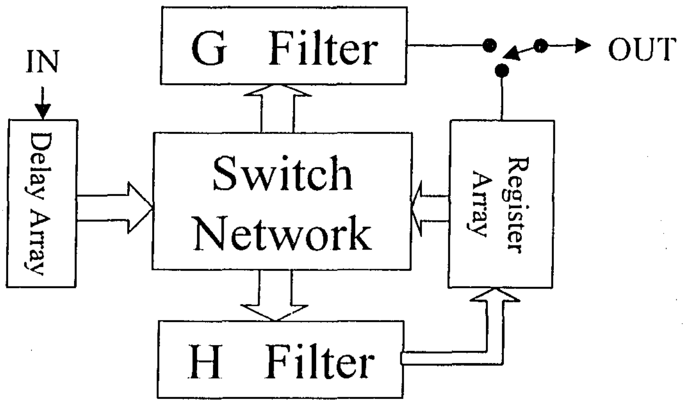


Figure 9. The 1-D Folded Analysis Wavelet Architecture Proposed in [8]

The digit-serial architecture can further improve the hardware utilization efficiency and reduce the interconnection overhead in the folded architecture. The digit-serial circuit only processes part of the word-length of each sample at each clock cycle. The number of bits processed per clock cycle is called the digit size. If the digit size is 1, then the digit-serial architecture becomes a bit-serial filter; if the digit size is the same as the word length, it becomes a bit-parallel filter. A diagram of a three-level digit-serial analysis wavelet filter is shown in Figure 10. In this example, the digit size for the first DWT stage is half a word-length. The second DWT stage filter is one quarter of a word-length; the third stage is one-eighth of a word-length, and so on. In such way, the process time of each sample doubles for every higher DWT stage. Therefore, the process time for each DWT stage remains the same.

By implementing filters of different digit sizes for different levels of the wavelet analysis or synthesis, the digit-serial architecture can achieve complete hardware utilization and requires simpler routing. The drawback of this architecture is the increase of the system latency, and the constraint on the word length selection. The word length must be multiple of 8 or 16 for a 3-level or 4-level DWT, respectively.

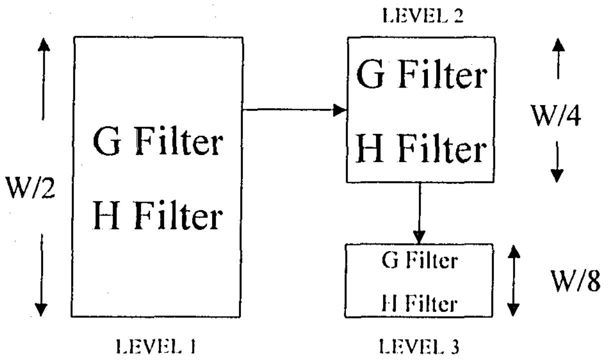


Figure 10. The Diagram of the 1-D Digit-Serial Architecture [8]

The proposed systolic architecture in [16], as shown in Figure 11, is an improvement over the digit-serial architecture described above. One feature of the systolic architecture is that it computes both the high-pass and the low-pass frequency coefficients in the same FIR filter to achieve high hardware utilization efficiency. The systolic architecture consists of a filter unit, storage units and a control unit. The filter unit is a FIR filter reconfigured alternatively at different times as either the high-pass or low-pass filters by changing the coefficients at each tap. The control unit directs the data flow between the storage units and the filter, and changes the coefficients in the filter accordingly. The design of the control unit is similar to that of the folded architecture [8] introduced before. The single filter structure of the systolic architecture improves the efficiency and reduces the hardware complexity, but it also increases the processing latency.

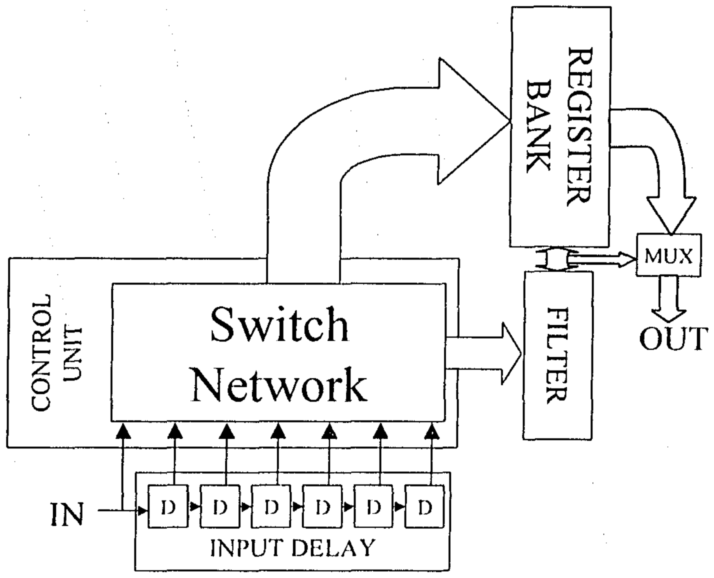


Figure 11. A Systolic Wavelet Architecture [16].

To reduce the latency, a parallel filter architecture implementing a modified recursive pyramid algorithm (MPRA) is proposed in [10]. In the MPRA schedule, the lower DWT stages (octaves) are performed before the higher octaves in order to avoid possible clashes. If the first output of any octave is scheduled such that there is no conflict with any of the lower octave outputs, then it is guaranteed that there will be no conflict with all of the outputs of that octave. The scheduling is critical for reusing the same filters for different octaves.

The proposed parallel filter architecture for a 1-D DWT is shown in Figure 12. The architecture contains two L-tap parallel filters to compute the low-pass and the high-pass outputs, and a storage unit of size LJ to store the input samples that are required to compute the J octave outputs.

The parallel filter architecture for a 2-D non-separable DWT implementing the MRPA algorithm is shown in Figure 13. The 2-D parallel filter architecture contains two programmable parallel filters. Each filter consists of L^2 programmable multipliers and $(L^2 - 1)$ adders to sum up the products. One filter computes the outputs of two coefficient bands (LL and LH, or HL and HH). The outputs of the lowest band (LL) of each DWT stage (octave) are stored in one of the shift register arrays. The sizes of the register arrays are shown in Figure 13, where N is the width of the input image.

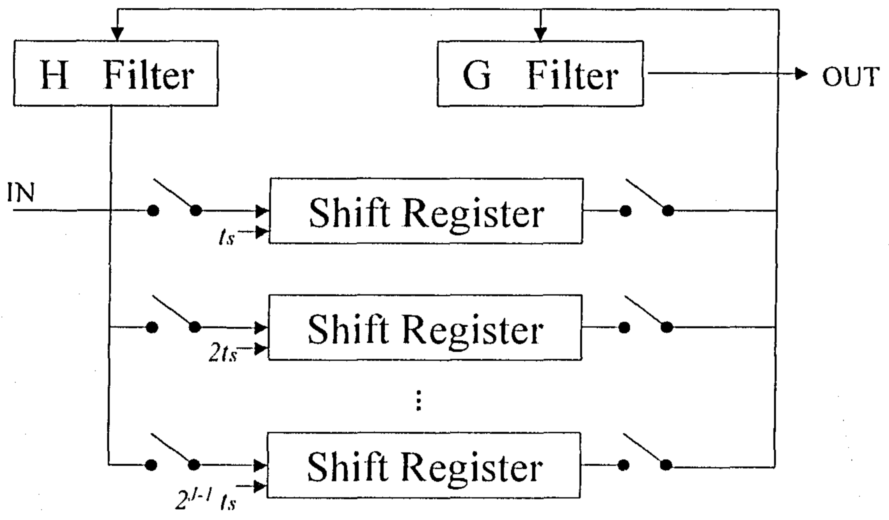


Figure 12. The Parallel Filter Architecture for 1-D DWT [10].

The first octave is computed at every other cycle, and the higher octave computations are interspersed between the first octave computations. Since the filter size is $L \times L$, the delay in the parallel filter is different for each octave. Consequently, the time when the first output of each octave being computed is different: the higher the octave, the longer the computation delay.

In [32], two 1-D DWT architectures are proposed that use the polyphase decomposition technique and the folding technique, respectively. The polyphase decomposition technique exploits the decimation of the fast DWT algorithm by separating the filter coefficients into even-order and odd-order parts. In the even clock cycles, the input data are fed to the odd part of the filter; in the odd clock cycles, the input data are fed to the even part. The outputs of the even and odd parts are summed to produce the output, as shown in Figure 14. Compared to the direct implementation of the DWT algorithm, the polyphase decomposition filter can reduce the processing time by a half.

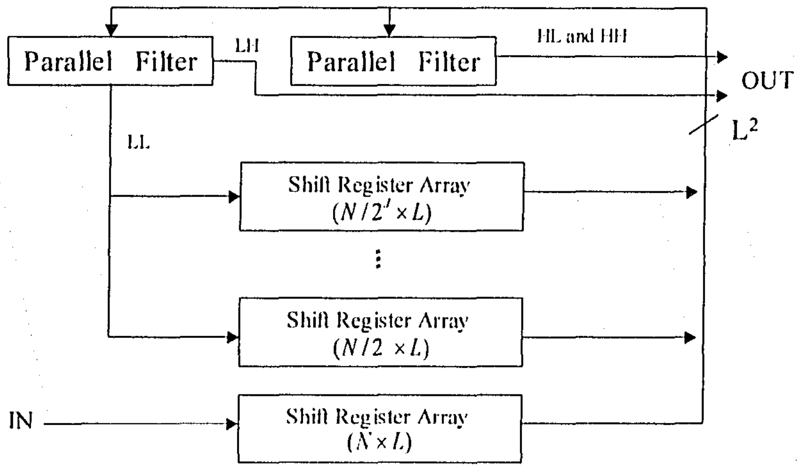

Parallel Filter
LH
Parallel Filter
HL and HH
OUT
L^2
LL
Shift Register Array $(N/2^J \times L)$
Shift Register Array $(N/2 \times L)$
IN
Shift Register Array $(N \times L)$

Figure 13. Block Diagram of the 2-D Parallel Filter Architecture [10].

The filter shown in Figure 15 employs the coefficient folding technique. Each set of multipliers in the architecture is shared by two coefficients. The switches route data through the datapath. Because most of the components of the filter are shared, the coefficient folding architecture reduces the hardware cost of the datapath by approximately one half.

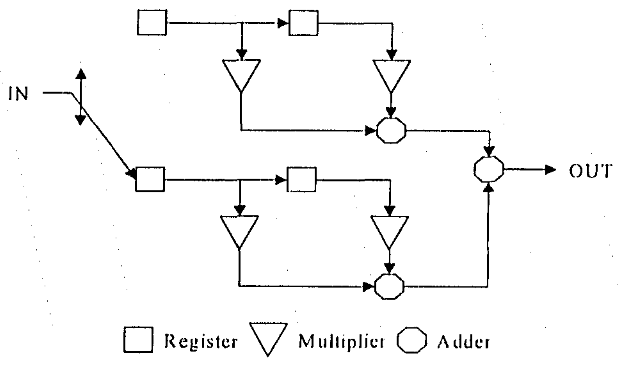


Figure 14. The Decimation Filter Employing the Polyphase Decimation Technique [32].

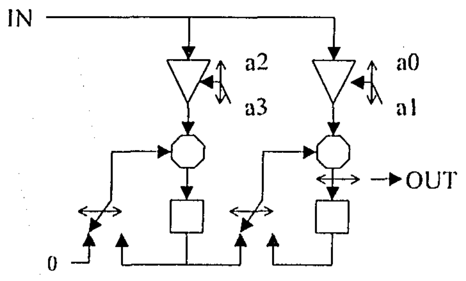


Figure 15. Decomposition Filter Employing the Coefficient Folding Technique [32].

One advantage of the polyphase decomposition technique is speed, and a feature of the coefficient folding technique is high hardware utilization rate. Combining these two techniques can produce a fast and efficient DWT architecture. The proposed 2-D architecture consists of a transform module, a RAM module, and a multiplexer. The 2-D transform module employs two folded filters for computing the row transforms (high-pass and low-pass), and four parallel filters for computing the column transforms (decomposing the row transform results into LL, LH, HL, and HH bands). The outputs of the first stage are stored in the RAM bank and are further decomposed after the computation for the first stage is done. The advantages of the proposed architecture are near 100% hardware utilization, fast computation time, regular data flow, and low control complexity.

3.2 Existing Architectures Based on the Lifting Algorithm

One of the earliest hardware implementations of the lifting algorithm is the parallel architecture proposed by Jiang *et al.* [22]. The proposed parallel architecture, namely *Split-and-Merge,* adopts a new *Boundary Postprocessing* technique, which ensures that the boundary samples are transformed correctly. The basic idea of the technique is to model the DWT as a finite machine that updates each raw input sample progressively into a wavelet coefficient. The new boundary processing technique reduces the otherwise significant communication overhead that normally hampers the efficiency of parallel systems. As a result, the proposed parallel architecture requires data to be communicated only once between neighboring processors for any arbitrary level of wavelet decomposition.

The computation procedure of the parallel architecture is as follows. The first step is the *split* operation: the input data are separated into two sets and sent to two processors. The second step is the *merge* operation: the data are analyzed/synthesized, and the processed states and results are stored in the registers. The state information from the neighboring processor is then combined together with its own corresponding state information to complete the whole DWT computation.

Since only one inter-processor communication is necessary to exchange boundary state information to compute the DWT, the proposed parallel algorithm improves the efficiency of the lifting scheme implementation on a parallel processing network. The drawback of Jiang's parallel architecture is the same as the other parallel processing implementations: large size and complex communication network.

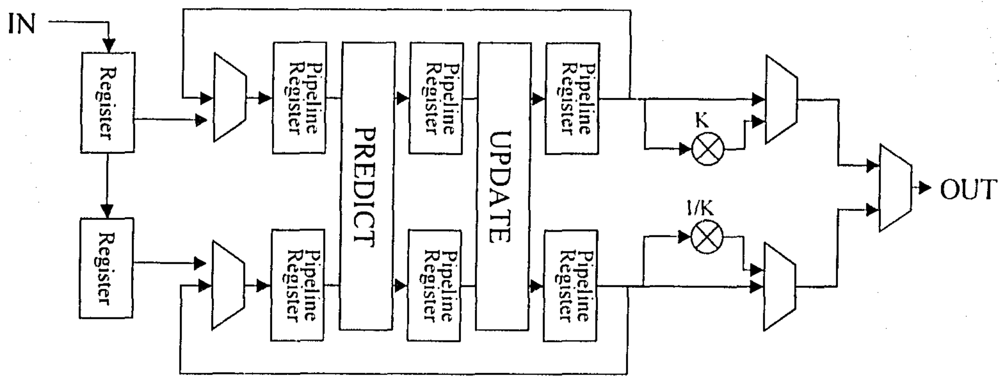


Figure 16. 1-D Folded Architecture [33].

In 2001, Lian *et al.* proposed a folded architecture to improve the hardware utilization [33] for 5/3 and 9/7 filters. The folded architecture exploited the symmetry and decimation of the wavelet filters by using just half of the hardware necessary for the standard lifting filter. The diagram of the proposed folded architecture is shown in Figure 16. It is similar to its classical filter counterpart [32] in the way that both reuse the common processing units in the datapath to improve the efficiency of hardware utilization. Although the folded architecture appears to be able to achieve 100% hardware utilization rate, the actual utilization may be much less that. For example, the scaling multipliers are used only once at each stage, so its utilization rate is 50% or less for filters that have more than two lifting steps. Since the multipliers are the largest components in the 1-D filters, the utilization of the scaling multipliers should not be ignored. Another possible drawback of the folded architecture is that it might consume more power than the standard architecture due to the more complicated interconnection and more frequent signal switching of the DWT filter coefficients, etc.

Andra *et al.* proposed an architecture that computes one level of the separable 2-D DWT at a time [34]. The architecture contains two row processors to compute along the rows and two column processors to compute along the columns, as shown in Figure 17. Each row or column processor is constructed according to the basic computation architecture illustrated in Figure 18. The basic architecture is composed of adders, a multiplier, and a shifter. The shifter is used to carry out the scaling step.

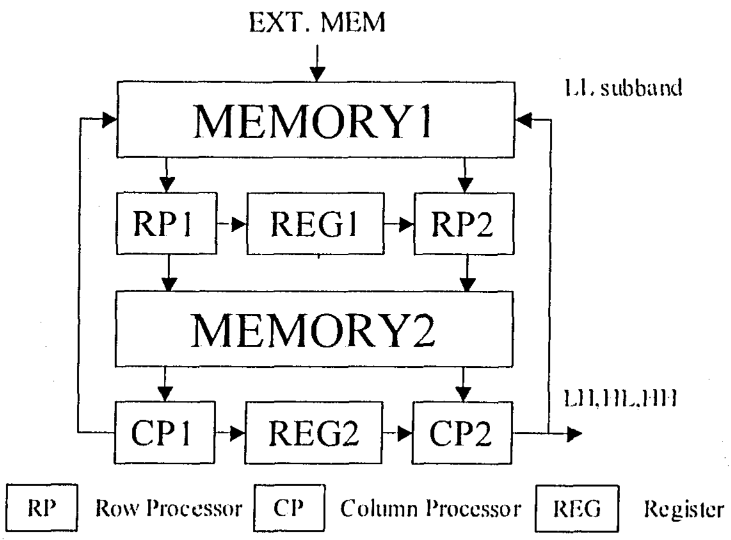


Figure 17. Block Diagram of the 2-D Architecture in [34].

To compute the 2-D DWT, the architecture inputs a block of size $N \times N$ from external memory, and writes to MEMORY1. The row processors RP1 and RP2 read in the data from MEMORY1, compute the horizontal DWT, and write the results to MEMORY2. When there are enough data for processing in MEMORY2, the column processors start calculating the vertical DWT. The decomposed subbands LH, HL, and

HH are sent to the output, and the LL subband is fed back to MEMORY1 for the next level of decomposition.

The major drawback of this architecture is that the dataflow is irregular for filters with 2*M* lifting steps and filters with 4*M* lifting steps. For the 2*M* filters, the LL subband is generated at CP1; for the 4*M* filters, the LL subband is generated at CP2. Since the architecture is designed in such a way that the LL subband can only be written to MEMORY1 through CP2, the LL subband has to be sent to CP2 through MEMORY2 first. Hence, there is significant latency for the 2*M* filters.

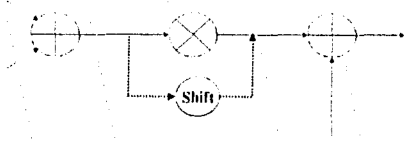


Figure 18. Basic Architecture of Each Processor.

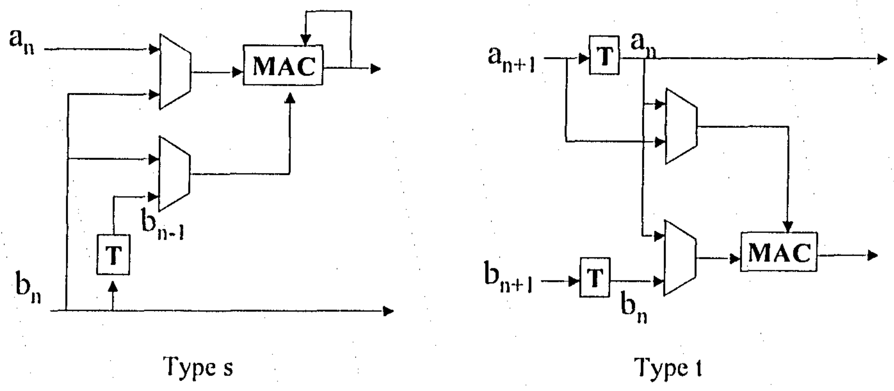


Figure 19. Basic Circuits for the Parallel Architecture Proposed in [35].

The parallel architecture proposed by Arguello *et al.* [35] is a configurable architecture that is capable of computing a wide range of wavelet packet transforms, with variations in the type of filters, the number of stages, and the form of the tree-structured filter bank. The architecture is a folded filter which requires that the input data be stored previously in a memory. It is not designed to provide minimum latency between the input and output. To achieve a configurable filter, the proposed architecture employs basic *s* and *t* circuits (where *s* and *t* are FIR filters (polynomials) expressed as in Equation (2.22)) as building blocks, which are shown in Figure 19. They can perform the basic operations listed in Table 4. The advantage of this architecture is flexiblility, and the modular design is easy to implement in 1-D designs. The drawback of the architecture is that it usually takes more cycles to compute the DWT transforms than the standard lifting algorithm [20].

Table 4. Basic operations that are carried out by the proposed architecture to compute the lifting steps

Type *s*	**Type *t***
$a_n + Kb_n$	$b_n + Pa_n$
$a_n + Lb_{n-1}$	$b_n + Qa_{n+1}$
$ACC + Lb_{n-1}$	$ACC + Qa_{n+1}$
$a_n + M(b_n + b_{n-1})$	$b_n + R(a_n + a_{n+1})$
Na_n	$1/N \bullet b_n$

In this chapter, we presented some representative DWT architectures ranging from the classical filters that implement Mallat's algorithm, to the recently proposed lifting architectures. Each of these architectures has its own advantages. Some are efficient in hardware utilization, some feature fast computation, and some require less silicon area to implement. However, there is still room for improvement and innovation to design even more efficient architectures. In the following chapters, we will propose efficient architectures based on the lifting algorithm.

Chapter 4

Proposed 1-D Architectures*

To implement the lifting algorithm described in Chapter 2, the input signal has to be first separated into even and odd samples. Each pair of input samples (one even and one odd) is then processed according to the specific analysis polyphase matrix. For many applications, the data can be read no faster than one input sample per clock cycle, so sample pairs are usually processed at every other clock cycle. Hence, this is a limitation on the speed and efficiency of a direct implementation of the lifting scheme. To overcome this bottleneck, the proposed recursive architectures exploit the available idle cycles and re-use the same hardware to recursively interleave the DWT stages. The dual-scan architectures thus gain efficiency by keeping the datapath hardware busy with two different streams of data.

4.1 The 1-D Recursive Architecture

Because of the down-sampling resulting from the splitting step at each stage in the lifting-based DWT, the number of low frequency coefficients is always half the number of input samples from the preceding stage. Further, because only the low frequency DWT coefficients are decomposed in the dyadic DWT, the total number of the samples to be processed for an L-stage 1-D DWT is:

$$N(1+1/2+1/4+\cdots+1/2^{L-1}) = N(2-1/2^{L-1}) < 2N, \tag{4.1}$$

* The architectures described in this chapter were published in references [36][47].

where N is the number of the input samples. For a finite-length input signal, the number of input samples is always greater than the total number of intermediate low frequency coefficients to be processed at the second and higher stages. Accordingly, there are time slots available to interleave the calculation of the higher stage DWT coefficients while the first-stage coefficients are being calculated.

4.1.1 Design Details

The *recursive architecture* (RA) is a general scheme that can be used to implement any wavelet filter that is decomposable into lifting steps [36]. As 1-D examples will describe RA implementations of the Daub-4 and 9/7 wavelet filters. In the next chapter, we will show how the RA can be extended to 2-D wavelet filters. Theoretically, the RA can be extended to even higher dimensions in a similar way.

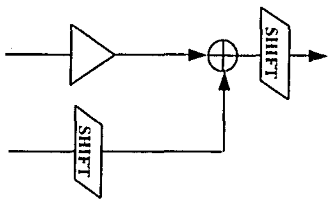


Figure 20. MAC for Asymmetric Wavelet Filters

The RA is a modular scheme made up of basic circuits such as delay units, pipeline registers, multiplier-accumulators (MACs), and multipliers. Since the factored Laurent polynomials $s_i(z)$ and $t_i(z)$ for symmetric (biorthogonal) wavelet filters are themselves symmetric, and those for asymmetric filters are normally asymmetric, we can use two kinds of MACs to minimize the computational cost. The MAC for asymmetric filters (shown in Figure 20) consists of a multiplier, an adder, and two shifters. The symmetric MAC (shown in Figure 21) has one more adder than the asymmetric MAC. The shifters are used to scale the partial results so that accuracy can be better preserved.

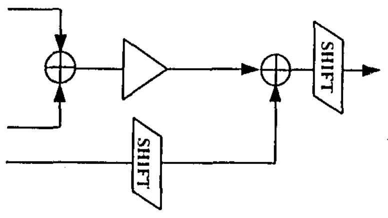


Figure 21. MAC for Symmetric Wavelet Filters

Different kinds of lifting-based DWT architectures can be constructed by combining the four basic lifting step circuits, shown in Figure 22. For the sake of simplicity, the shifters are omitted in Figure 22 and the figures hereafter. The general construction has the following steps:

Step 1: Decompose the given wavelet filter into lifting steps [20].

Step 2: Construct the corresponding cascade of lifting step circuits. Replace each delay unit in each circuit with an array of delay units. The number of delay units in the array is the same as the number of wavelet stages.

Step 3: At the beginning of the cascade construct an array of delay units that will be used to split the inputs for all wavelet stages into even and odd samples. These delay units are also used to temporarily delay the samples so that they can be input into the lifting step cascade at the right time slot. Two multiplexer switches are used to select one even input and one odd input to be passed from the delay units to the first lifting step.

Step 4: Construct a data flow table that expresses how all of the switches are set and how the delay units are enabled in each time slot. There is latency as the initial inputs for the first wavelet stage propagate down through the cascade. A free time slot must then be selected to fix the time when the inputs for the second wavelet stage will be sent into the cascade. All higher order stages must also be scheduled into free time slots in the data flow table.

Step 5: Design the control sequencer to implement the data flow table.

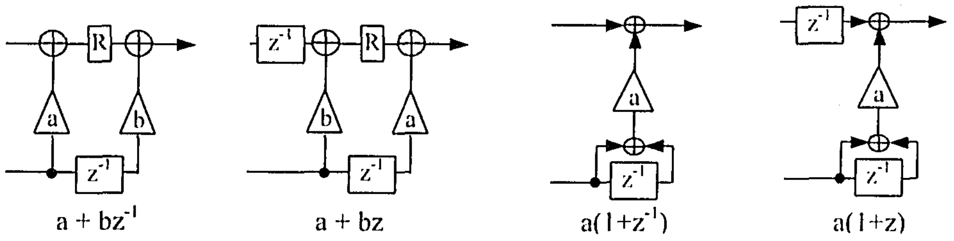
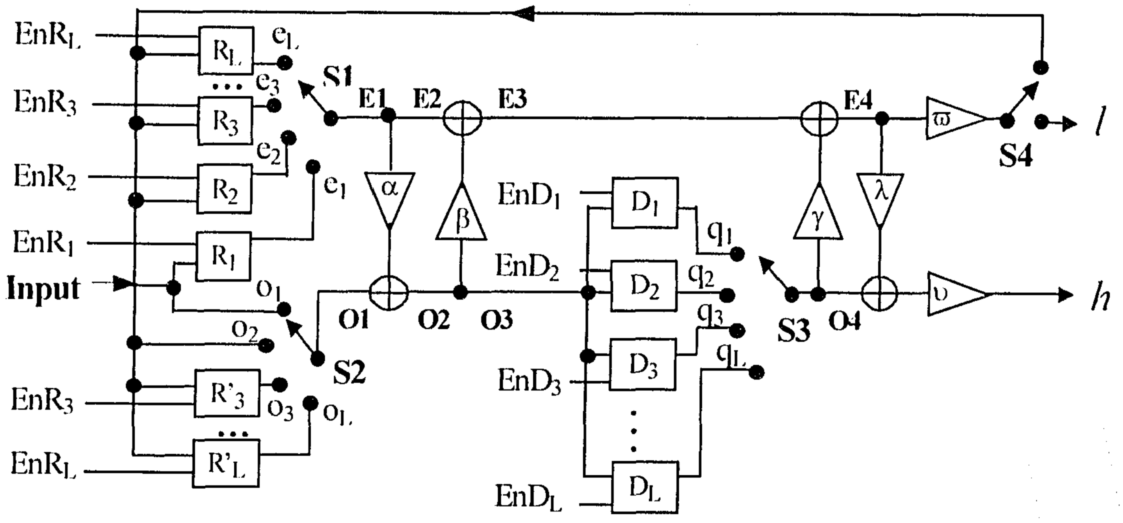


Figure 22. Circuits for the Basic Lifting Steps

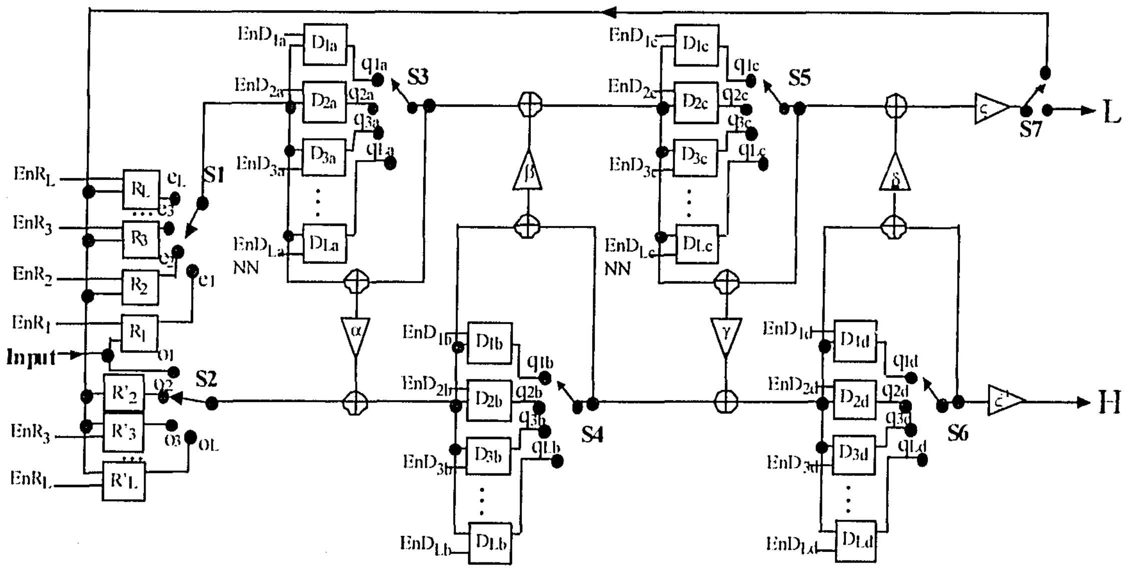
The RA in Figure 23 calculates three stages of the Daub-4 DWT, while the RA in Figure 24 calculates the three-stage 9/7 DWT. Because the control sequence of the RAs for all wavelets is similar, we will discuss the operation of the RA for only the Daub-4 DWT in more detail.

In Figure 23, the input registers R_i (i=1,2,..., L) and R'_i (i=3,..., L) hold the input values for the ith DWT stage. Thus the first stage coefficients can be calculated at every other clock cycle and the data for the other stages can be fed into the lifting step pipeline during the intervening cycles. Using $x_{i,j}$ to denote the jth coefficient of the ith stage, the DWT coefficients can be calculated in the order shown in Figure 25.



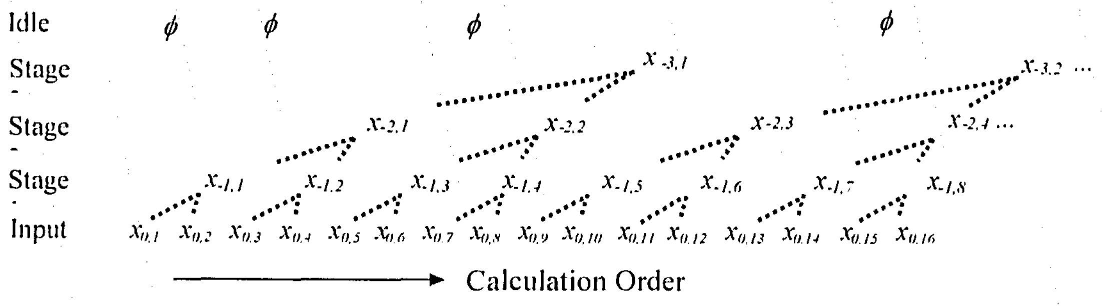
$\alpha=-\sqrt{3}$, $\beta=\sqrt{3}/4$, $\gamma=(\sqrt{3}-2)/4$, $\lambda=1$, $\upsilon=(\sqrt{3}+1)/\sqrt{2}$, $\varpi=\upsilon^{-1}$.

Figure 23. 1-D Recursive Architecture for the Daub-4 DWT. "R" represents registers, and "D" represents delay units. "S$_i$" represents control signals for the data flow.



$\alpha=-1.58613$, $\beta=-0.05268$, $\gamma=0.88291$, $\delta=0.44351$ $\varsigma=1.14960$

Figure 24. 1-D Recursive Architecture for the 9/7 DWT.


Idle
Stage
Stage
Stage
Input
ϕ ϕ ϕ ϕ
$x_{-3,1}$ $x_{-3,2}$ ...
$x_{-2,1}$ $x_{-2,2}$ $x_{-2,3}$ $x_{-2,4}$ ...
$x_{-1,1}$ $x_{-1,2}$ $x_{-1,3}$ $x_{-1,4}$ $x_{-1,5}$ $x_{-1,6}$ $x_{-1,7}$ $x_{-1,8}$
$x_{0,1}$ $x_{0,2}$ $x_{0,3}$ $x_{0,4}$ $x_{0,5}$ $x_{0,6}$ $x_{0,7}$ $x_{0,8}$ $x_{0,9}$ $x_{0,10}$ $x_{0,11}$ $x_{0,12}$ $x_{0,13}$ $x_{0,14}$ $x_{0,15}$ $x_{0,16}$
Calculation Order

ϕ denotes an idle clock cycle where no coefficient is calculated

Figure 25. 1-D DWT Coefficient Computation Order

The input registers also synchronize the even and odd samples of each stage. Since the first two stages can be immediately processed when the odd samples are ready, no input register is needed for the odd samples for these two stages. Register D_i is a delay unit for the i^{th} stage. After splitting the input data into even and odd parts, the Daub-4 DWT is calculated step-by-step as shown in Table 5. In Table 5, E_n and O_n are the outputs of each lifting step; $e_{-i,j}$ and $o_{-i,j}$ denote the even and odd intermediate results of each lifting step. Since the architecture is pipelined by each MAC unit, the outputs of each lifting step are synchronized. As an example, the calculations of the first pair of DWT coefficients are listed bellow:

E1: $x_{0,1} = x_{0,1}$
O1: $x_{0,2} = x_{0,2}$
E2: $e_{-1,1} = x_{0,1}$
O2: $o_{-1,1} = \alpha x_{0,1} + x_{0,2}$
E3: $e_{-1,1} = \beta o_{-1,1} + e_{-1,1}$
O3: $o_{-1,1} = o_{-1,1}$
E4: $e_{-1,1} = z^{-1} \gamma o_{-1,1} + e_{-1,1}$
O4: $o_{-1,1} = z^{-1} o_{-1,1}$
Low frequency DWT coefficient l: $l_{-1,1} = \omega e_{-1,1}$
High frequency DWT coefficient h: $h_{-1,1} = \upsilon(\lambda e_{-1,1} + o_{-1,1})$.

Table 5. Data Flow for the Three-Stage 1-D Recursive Architecture

$x_{i,j}$ is input signal, i and j denote the stage and the sequence, respectively; $e_{-i,j}$ and $o_{-i,j}$ are even and odd intermediate results of each lifting step: $l_{-i,j}$ and $h_{-i,j}$ are low and high frequency DWT coefficients

Clk	Input	E_1 ; O_1	E_2 ; O_2	E_3 ; O_3	E_4 ; O_4	l ; h	Stage
1	$x_{0,1}$						
2	$x_{0,2}$	$x_{0,1}$; $x_{0,2}$					
3	$x_{0,3}$		$e_{-1,1}$; $o_{-1,1}$				
4	$x_{0,4}$	$x_{0,3}$; $x_{0,4}$		$e_{-1,1}$; $o_{-1,1}$			
5	$x_{0,5}$		$e_{-1,2}$; $o_{-1,2}$		$e_{-1,1}$; $o_{-1,1}$		
6	$x_{0,6}$	$x_{0,5}$; $x_{0,6}$		$e_{-1,2}$; $o_{-1,2}$		$l_{-1,1}$; $h_{-1,1}$	1
7	$x_{0,7}$		$e_{-1,3}$; $o_{-1,3}$		$e_{-1,2}$; $o_{-1,2}$		
8	$x_{0,8}$	$x_{0,7}$; $x_{0,8}$		$e_{-1,3}$; $o_{-1,3}$		$l_{-1,2}$; $h_{-1,2}$	1
9	$x_{0,9}$	$l_{-1,1}$; $l_{-1,2}$	$e_{-1,4}$; $o_{-1,4}$		$e_{-1,3}$; $o_{-1,3}$		
10	$x_{0,10}$	$x_{0,9}$; $x_{0,10}$	$e_{-2,1}$; $o_{-2,1}$	$e_{-1,4}$; $o_{-1,4}$		$l_{-1,3}$; $h_{-1,3}$	1
11	$x_{0,11}$		$e_{-1,5}$; $o_{-1,5}$	$e_{-1,1}$; $e_{-1,2}$	$e_{-1,4}$; $o_{-1,4}$		
12	$x_{0,12}$	$x_{0,11}$; $x_{0,12}$		$e_{-1,5}$; $o_{-1,5}$	$e_{-1,1}$; $e_{-1,2}$	$l_{-1,4}$; $h_{-1,4}$	1
13	$x_{0,13}$	$l_{-1,3}$; $l_{-1,4}$	$e_{-1,6}$; $o_{-1,6}$		$e_{-1,5}$; $o_{-1,5}$	$l_{-2,1}$; $h_{-2,1}$	2
14	$x_{0,14}$	$x_{0,13}$; $x_{0,14}$	$e_{-2,2}$; $o_{-2,2}$	$e_{-1,6}$; $o_{-1,6}$		$l_{-1,5}$; $h_{-1,5}$	1
15	$x_{0,15}$		$e_{-1,7}$; $o_{-1,7}$	$e_{-2,2}$; $o_{-2,2}$	$e_{-1,6}$; $o_{-1,6}$		
16	$X_{0,16}$	$x_{0,15}$; $x_{0,16}$		$e_{-1,7}$; $o_{-1,7}$	$e_{-2,2}$; $o_{-2,2}$	$l_{-1,6}$; $h_{-1,6}$	1
17	$x_{0,17}$	$l_{-1,5}$; $l_{-1,6}$	$e_{-1,8}$; $o_{-1,8}$		$e_{-1,7}$; $o_{-1,7}$	$l_{-2,2}$; $h_{-2,2}$	2
18	$x_{0,18}$	$x_{0,17}$; $x_{0,18}$	$e_{-2,3}$; $o_{-2,3}$	$e_{-1,8}$; $o_{-1,8}$		$l_{-1,7}$; $h_{-1,7}$	1
19	$x_{0,19}$	$l_{-2,1}$; $l_{-2,2}$	$e_{-1,9}$; $o_{-1,9}$	$e_{-2,3}$; $o_{-2,3}$	$e_{-1,8}$; $o_{-1,8}$		
20	$x_{0,20}$	$x_{0,19}$; $x_{0,20}$	$e_{-3,1}$; $o_{-3,1}$	$e_{-1,9}$; $o_{-1,9}$	$e_{-2,3}$; $o_{-2,3}$	$l_{-1,8}$; $h_{-1,8}$	1
21	$x_{0,21}$	$l_{-1,7}$; $h_{-1,8}$	$e_{-1,10}$; $o_{-1,10}$	$e_{-3,1}$; $o_{-3,1}$	$e_{-1,9}$; $o_{-1,9}$	$l_{-2,3}$; $h_{-2,3}$	2
22	$x_{0,22}$	$x_{0,21}$; $x_{0,22}$	$e_{-2,4}$; $o_{-2,4}$	$e_{-1,10}$; $o_{-1,10}$	$e_{-3,1}$; $o_{-3,1}$	$l_{-1,9}$; $h_{-1,9}$	1
23	$x_{0,22}$	$x_{0,21}$; $x_{0,22}$	$e_{-1,11}$; $o_{-1,11}$	$e_{-2,4}$; $o_{-2,4}$	$e_{-1,10}$; $o_{-1,10}$	$l_{-3,1}$; $h_{-3,1}$	3

Therefore, the DWT coefficients of the first stage are generated five clock cycles after the first input sample is received. The first low frequency DWT coefficient $l_{-1,1}$ is also stored in register R_2. After the second low frequency DWT coefficient $l_{-1,2}$ is ready, $l_{-1,1}$ and $l_{-1,2}$ are further processed in the idle cycles as shown in Table 5.

The control signals for the switches in a RA can also be deduced from the corresponding data flow table (which is Table 5 in this case). The timing for the register enable signals is shown in Table 6. Switches S1, S2 and S3 steer the data flows at each stage. The timing of the switch control signals is shown in Table 7. Output switch S4 feeds back the low frequency DWT coefficients (except for the last stage) to be further decomposed. The switching timing for S4 is the same as for S1. The problem of efficiently scheduling the operations in the lifting step pipelining is similar to the problem encountered in compiler-supported instruction scheduling of loops that is called *Software Pipelining* [53]. The overlap of the computation of lifting step operations for multiple samples is analogous to the overlap of instructions from multiple iterations of a loop. The pipeline schedules that we used, however, were optimized for the data dependencies in the DWT.

Table 6. Enable Signals for the Input Registers (k is the sample index) of the 1-D RA Implementing the D4 DWT

Time, T_{en} (in clock cycles)	**Enable Signals**		
$2k$	EnR_1	-	EnD_1 **
$4k+4$	EnR_2	-	EnD_2 **
$8k+9$	EnR_3	EnR'_3 *	EnD_3 **
$2^Lk+3\times2^{L-2}+2^{L-1}-1$	EnR_L	EnR'_L *	EnD_L **

* The actual times are: $T_{en} + 2^{L-1}$.

** The actual times are: $T_{en} + 2^{L-1}$ + *Latency from S2 to S3.*

Table 7. Input Switch Control Timing for the 1-D RA Implementing D4 DWT

Time, T_s	**Switch Positions**		
(in clock cycles)	S1	S2	S3 *
$2k+1$	e_1	o_1	q_1
$4k+6$	e_2	o_2	q_2
$8k+16$	e_3	o_3	q_3
$2^Lk+3\times2^{L-2}+2^L+$ *Latency*	e_L	o_L	q_L

* The actual times are: $T_s + 2$.

The design of the controller is relatively simple, due to the regularity of the control signals for the RA, as shown in Table 6 and Table 7. All control signals are generated by counters and flip-flops controlled by a four-state finite state machine. The counters generate periodic signals for the longer period (7>4 clock cycles) control signals, and the flip-flops produce local delays. The simplified state transition diagram of the controller is shown in Figure 26. When the RA receives input samples. it starts to decompose (analyze) the data. The controller keeps track of the number of processed stages. After all stages have been processed, the controller asserts a 'Done' single, and returns to the waiting state. If externally-generated start and stop signals are provided, the long counter for keeping track of the number of input samples is unnecessary. Compared to other direct implementations of lifting-based DWTs, the overhead for the RA controller is very small. The controller normally occupies less than 10% of the total silicon area of the 1-D RA.

The remaining elements of the RA include registers and switches (tri-state buffers). Since the area of the switches is small compared to the size of the whole architecture, the cost of the data storing registers in the datapath tends to dominate. For implementing an L-stage DWT, the RA uses $(L-1)(M+1)$ more registers than a conventional lifting-based architecture, where M is the number of delay registers. Considering that a conventional architecture needs an extra memory bank to store at least $N/2$ intermediate DWT coefficients, the RA architecture is more area-efficient in most applications, where $(L-1)(M+1) \ll N/2$. The power consumption of the RA is also likely to be lower than that of a conventional architecture because the RA eliminates the memory read/write operations and because all data routing is local. By avoiding the fetching of data from memories and the driving of long wires, the power dissipated by the RA switches is minimized. Further discussion of implementation details is beyond the scope of this thesis project, but our preliminary analysis reveals the potential of the RA architecture in small-size and low power designs.

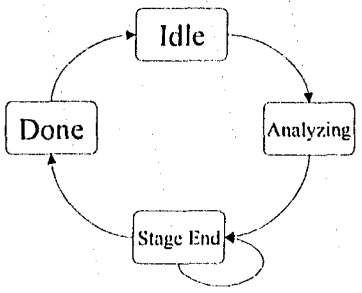


Figure 26. State Transition Diagram of the RA Controller

4.1.2 Evaluation

Since the pipeline delay for calculating an L-stage DWT is $L \times T_d$ (where T_d is the latency from input to output) and the sampling-interval for each stage computation increases by two cycles for each additional stage (shown in Figure 25), the clock cycle count T_P for processing an N-sample DWT can be expressed as:

$$T_P = N + (L \times T_d) + (1 + 2 + \ldots + 2^{L-2}) = N + L \times T_d + 2^{L-1} - 1.$$

The *hardware utilization* can be defined as the ratio of the actual computation time to the total processing time, with time expressed in numbers of clock cycles. At each section of the pipeline structure, the actual clock cycle count T_C is the number of sample pairs to be processed.

$$T_C = (N + N(1 - 2^{1-L}))/2.$$

Note that $N(1-2^{1-L})$ is the number of samples being processed at the second or higher stages. The busy time T_B of the corresponding section can be expressed as:

$$T_B = T_P - T_d = N + (L - 1) \times T_d + 2^{L-1} - 1.$$

Consequently, the hardware utilization U of the L-stage RA is:

$$U = T_C / T_B \times 100\% = \frac{N + N(1 - 2^{1-L})}{2(N + 2^{L-1} + (L-1) \times T_d - 1)} \times 100\% \qquad (4.1)$$

Because U is a continuous concave function of variable L when $L \geq 1$, the maximum hardware utilization can be achieved when $\partial U/\partial L$=0. Ignoring the delay T_d, $\partial U/\partial L$=0 can be expressed as:

$$\frac{\partial U}{\partial L} = \frac{N(L-1)2^{-L} - L2^{L-1}}{2(N + 2^{L-1} - 1)^2} = 0.$$

The above equation is true when $L = 2^{-1}(\log_2 N + \log_2(1 - 1/L) + 1)$. Assuming $L > 1$ and $N >> \sqrt{N}$, the utilization reaches a maximum of about 90% when $L = 0.5\log_2 N$, and gradually reduces to around 50% when L=1 or $\log_2 N$. For a 5-stage DWT operating on 1024 input samples, the utilization approaches 92%. When the number L of decomposition stages increases, the processing time increases significantly and the utilization drops accordingly. As mentioned above, the delay of 2^L was due to the increasing separation (2^L clock cycles) of the input values to each stage. If we decrease the sampling grid for each stage as soon as all previous stages have finished, we can speed up the computation. With a little bit additional of controller overhead, the processing time in clock cycles of an L-stage DWT can be reduced to:

$$N + (L \times T_d).$$

When $N \rightarrow \infty$, the hardware utilization of the 1-D RA approaches 100%. Compared to the conventional implementations of the lifting algorithm, the proposed architectures can achieve a speed-up of up to almost 100%, as shown in Table 8.

Table 8. Computation Time and Hardware Utilization for 1-D Architectures

N: Number of input samples. T_d, T_{delay}: Circuit delay. L: Number of DWT stages

Architecture	Computation Time (clock cycles)	Hardware Utilization
RA	$N+T_dL$	50% - 90%
Direct implementation	$2N(1-1/2^L)+T_{delay}L$	50%
Folded [33]	$2N(1-1/2^L)+T_{delay}L$	≈100%

4.2 The 1-D Dual-scan Architecture

To achieve higher hardware utilization for special cases, we also propose the *dual-scan architecture* (DSA), which interleaves the processing of two independent signals simultaneously to increase the hardware utilization. The 1-D DSA is shown in Figure 27. It consists of a processing element (PE), input and output switches, and two memory units. The PE is a conventional direct hardware implementation of the lifting scheme constructed from the basic building block circuits. The input switches SW are connected to the two input signals when processing the first stage, and are connected to the memory when processing the other stages. Switch SW0 separates the low frequency coefficients of the two input signals. Because the architecture generates one low frequency coefficient at each clock cycle, SW0 is controlled by the system clock. The output switch SW1 is connected to the output only at the final stage. The size of each of memory unit is $M/2$, where M is the maximum number of input samples.

The PE for the 9/7 DWT is shown in Figure 28. Note that the DSA PE structure is almost identical to that of a two-stage DA architecture. S1 and S2 are the input switches that select the input data source. The other switches in the circuit are synchronized to the input switches to select the path for each input channel.

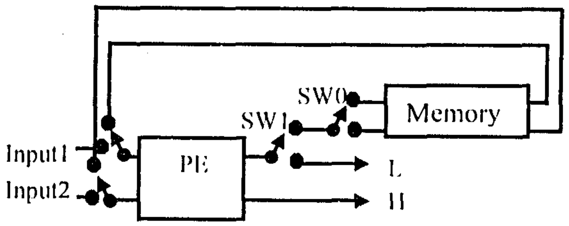


Figure 27. 1-D Dual-scan Architecture

The 1-D DSA calculates the DWT as the input samples are being shifted in, and stores the low frequency coefficients in the internal memory. When all input samples have been processed, the stored coefficients are retrieved to start computing the next stage.

As the 1-D DSA performs useful calculations in every clock cycle, the hardware utilization for the PE is 100%. The processing time for the L-stage DWT of two N-sample signals is $N + L \times T_d$. Compared to conventional implementations for computing two separate signals, the 1-D DSA requires only half the hardware. Hence, given an even number of equal-length signals to process, the speedup of the 1-D DSA is 100%.

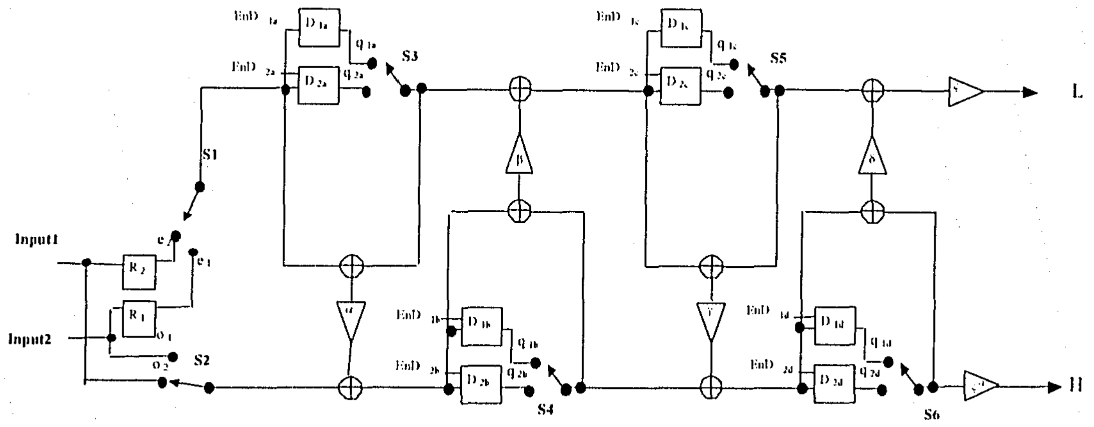


Figure 28. The DSA PE Circuit for the 9/7 DWT.

Chapter 5

Proposed 2-D Architectures

A conventional implementation of a separable 2-D lifting-based DWT is illustrated in Figure 29, where separate row and column processors each use a 1-D lifting architecture. The row processor calculates the DWT of each row of the input image, and the resulting decomposed high and low frequency components are stored in memory bank I. Since this bank normally stores all the horizontal DWT coefficients, its size is N^2 for an $N \times N$ image. When the row DWT is completed, the column processor starts calculating the vertical DWT on the coefficients from the horizontally decomposed image. The LH, HL, and HH subbands are final results and can be shifted out; the LL subband is stored in memory bank II for further decomposition. The size of memory bank II is thus at least $N^2/4$. Such a straightforward implementation of the 2-D DWT is both time and memory-intensive. To increase the computation speed, we propose a 2-D RA and a 2-D DSA for the separable 2-D lifting-based DWT.

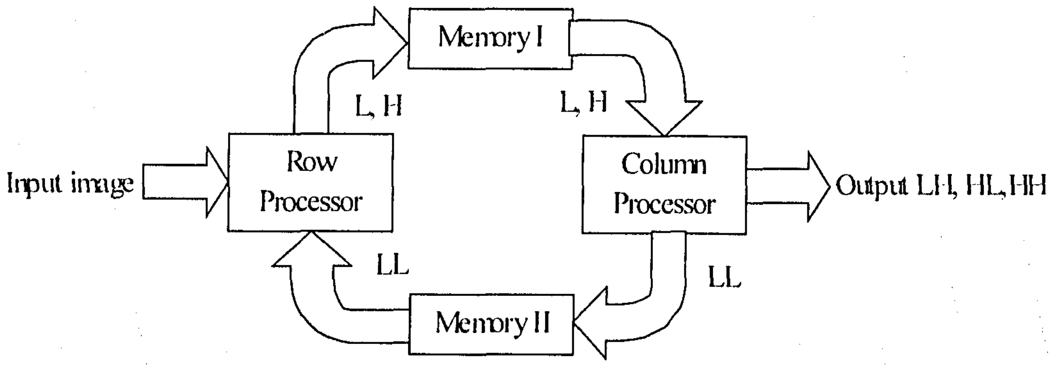


Figure 29. Conventional 2-D Lifting Architecture.

5.1 The 2-D Recursive Architecture

The basic strategy of the 2-D recursive architecture is the same as that of its 1-D counterpart: the calculations of all DWT stages are interleaved to increase the hardware utilization. Within each DWT stage, we use the processing sequence illustrated in Figure 30. The image is scanned into the row processor in a raster format, and the first horizontal DWT calculation is immediately started. The resulting high and low frequency DWT coefficients of the odd lines are collected and pushed into two FIFO registers or two memory banks. The separate storage of the high and low frequency components produces a more regular data flow and reduces the required output switch operations, which in turn consumes less power. The DWT coefficients of the even lines are also rearranged into the same sequence, and are directly sent to the column processor together with the outputs of the FIFO. The column processor starts calculating the vertical DWT in a zigzag format after one row's delay.

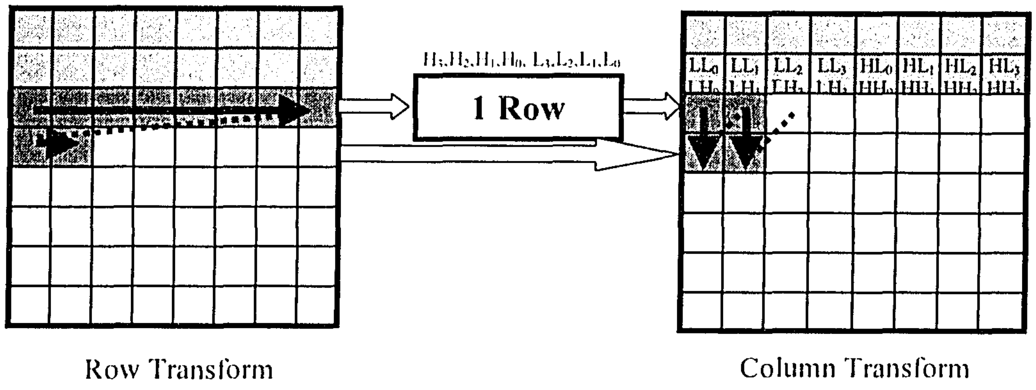


Figure 30. Calculation Sequence of the 2-D RA

A simplified schematic for the 2-D RA is shown in Figure 31. Note that the row DWT is similar to that of the 1-D DWT, so the datapath of the row processor is the same as for the 1-D RA. The column processor is implemented by replacing the delay registers and input circuit of the 1-D RA with delay FIFOs and the circuitry shown in Figure 31.

The interaction between the row and column processor goes as follows: When the row processor is processing the even lines (assuming that it starts with the 0^{th} row), the high and low frequency DWT coefficients are shifted into their corresponding FIFOs. When the row processor is processing the odd lines, the low frequency DWT coefficients of the current lines as well as the previous lines of coefficients stored in the FIFOs are sent to the column processor. Register D_l is used if the low frequency coefficients are generated before their high frequency counterparts. At the same time, the high frequency DWT coefficients of the current lines are shifted into their corresponding FIFOs, and the outputs of these FIFOs are shifted into the FIFOs corresponding to the low frequency. The computations are arranged in such a way that the processing of the DWT coefficients for the first and the other stages can be easily interleaved in neighboring clock cycles. Once the processing of the low frequency components is done, the outputs of both FIFOs are sent to the column processor. The function of the exchange blocks, shown as boxes labelled with an X in Figure 31, is to redirect the data flows between the FIFOs and the input of the column processor. As shown in Figure 32, the exchange block has two input channels, two output channels, and a control signal. When the control signal SW=0, the data from input channel 1 flows to output channel 1, and the data from input channel 2 flows to output channel 2; when SW=1, one data stream flows from input channel 2 to output channel 1, and the other data stream flows from input channel 1 to output channel 2. At the low frequency output of the column processor, a switch selects the *LL* subband and sends it back to the row processor for further decomposition.

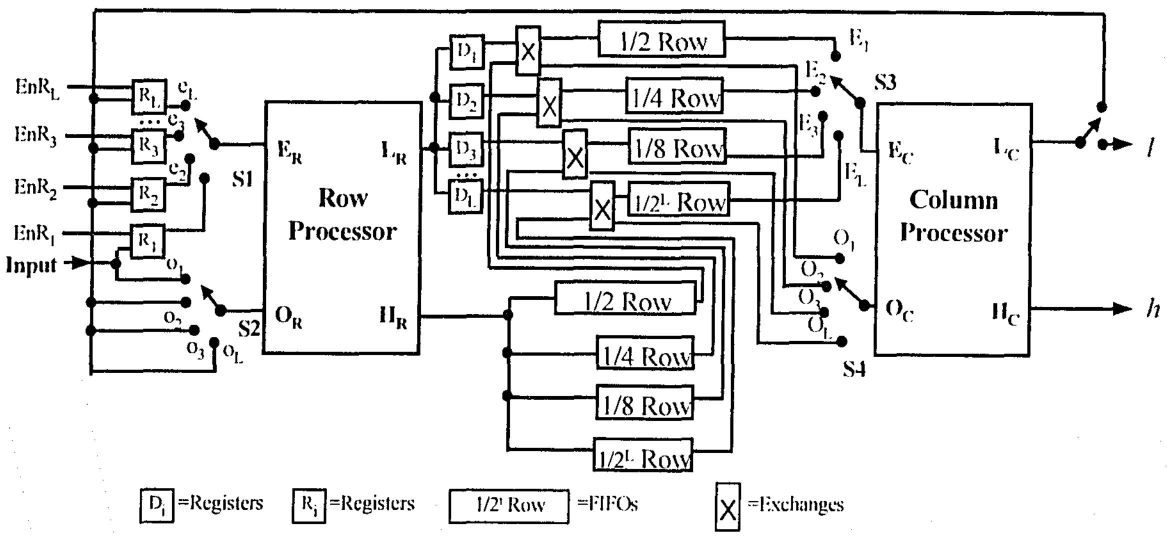


Figure 31. The 2-D Recursive Architecture

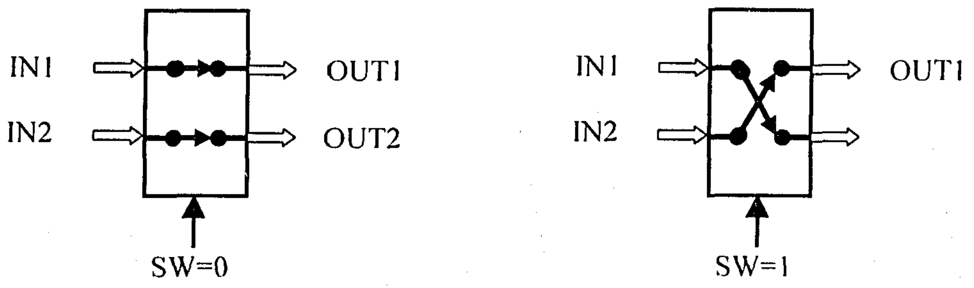


Figure 32. Exchange Operations

As an example, a portion of the data flow for computing an 8×8 sample 2-D Daub-4 DWT is shown in Table 10. As described before, the first pair ($e_{-1,1,1}$ and $o_{-1,1,1}$) of the first stage row transform coefficients are generated at the sixth clock cycle. They are immediately shifted into the high and low frequency FIFOs, respectively. The consecutive DWT coefficients of the same row are in turn pushed into their corresponding FIFOs in the consequent clock cycles until the end of the row (the 12th clock cycle in this case). When the first pair of the row transform coefficients of the second row is ready, the low frequency coefficient ($e_{-1,1,2}$) is sent to the odd input of the column processor, and the high frequency coefficient ($o_{-1,1,2}$) is pushed into the corresponding FIFO. The first low frequency coefficient of the first row ($e_{-1,1,1}$) is also popped out of the FIFO and sent to the even input of the column processor; its high frequency counterpart ($o_{-1,1,1}$) is pushed to the low frequency FIFO. After 4 clock cycles, the column processor generates the first pair of 2-D DWT coefficients, of which the low frequency one ($ll_{-1,1,1}$) is temporarily stored in register R_2. The row processor starts further decomposing the low frequency DWT coefficients after the second low frequency coefficient ($ll_{-1,2,1}$) is generated (at the 21st clock cycle in Table 10).

At the end of the row transform of the second row (at the 20th clock cycle in this case), both FIFOs for the first stage contain only the high frequency row transform coefficients of the first two rows, and start sending these coefficients to the column processor after 1 clock cycle. As shown in Table 10, the calculation of the different stages of the 2-D DWT is continuous and periodic. Thus the control signals for the data flow are easy to generate by relatively simple logic circuits.

Similar to the 1-D RA case, the control signals for the 2-D RA are deduced from the data flow as shown in Table 10. The timing for the switch signals of the 2-D RA for the lifting-based Daub-4 DWT are shown in Table 9, and the enable signals are fixed delay versions of these switch signals. Also, similar to the delay reduction method used in the 1-D RA, the delay time of the 2-D DWT can be minimized. The timing of the control signals for other wavelets is similar, and can be achieved by adjusting the delays in Table 9.

Table 9. Switch Control Timing for the 2-D RA Implementing Daub-4 DWT

Time, T_s (in clock cycles)	Switch Positions		Time, T_s (in clock cycles)	Switch Positions	
	S1	S2		S3	S4
$2k+1$	e_1	o_1	$2(l+1)N+2k+6$	E_1	O_1
$2(l+1)N+4k+9$	e_2	o_2	$4(l+1)N+4k+14$	E_2	O_2
$4(l+1)N+8k+17$	e_3	o_3	$8(l+1)N+8k+22$	E_3	O_3
$2^{L-1}(l+1)N$ $+2^L k+1+2T_dL$	e_L	o_L	$2^L(l+1)N$ $+2^L k+3+2T_dL$	E_L	O_L

Since the high-frequency components are processed one row after the low-frequency components, as shown in Figure 31 and Table 10, the processing delay of the column transform for each stage is roughly one row. Also, because all of the stages are interleaved, the total processing time for an L-stage 2-D DWT is:

$$N \times N + N + 2 \times L \times T_d + 2^{L-1} - 1.$$

Similar to the 1-D implementation, a hardware utilization of about 90% can be achieved when L is close to $\log_2 N$.

Table 10. Data Flow for the Three-Stage 2-D Recursive Architecture
$x_{i,j,k}$ is input signal. i, j and k denote the stage, the row and column sequences, respectively; $e_{-i,j,k}$ and $o_{-i,j,k}$ are even and odd intermediate results of each lifting step; $l_{-i,j,k}$ and $h_{-i,j,k}$ are low and high frequency DWT coefficients.

Clk	Input	Row Processor		FIFOs for Stage 1		Column Processor		Stage
		E_R : O_R	L_R : H_R	High Frequency	Low Frequency	E_C ; O_C	Output	
1	$x_{0,1,1}$							
2	$x_{0,2,1}$	$x_{0,1,1}$, $x_{0,2,1}$						
3	$x_{0,3,1}$							
4	$x_{0,4,1}$	$x_{0,3,1}$; $x_{0,4,1}$						
5	$x_{0,5,1}$							
6	$x_{0,6,1}$	$x_{0,5,1}$; $x_{0,6,1}$	$e_{-1,1,1}$; $o_{-1,1,1}$	$e_{-1,1,1}$	$o_{-1,1,1}$			
7	$x_{0,7,1}$							
8	$x_{0,8,1}$	$x_{0,7,1}$; $x_{0,8,1}$	$e_{-1,2,1}$; $o_{-1,2,1}$	$e_{-1,1,1}$; $e_{-1,2,1}$	$o_{-1,1,1}$; $o_{-1,2,1}$			
9	$x_{0,1,2}$							
10	$x_{0,2,2}$	$x_{0,1,2}$; $x_{0,2,2}$	$e_{-1,3,1}$; $o_{-1,3,1}$	$e_{-1,1,1}$; $e_{-1,2,1}$;$e_{-1,3,1}$	$o_{-1,1,1}$; $o_{-1,2,1}$;$o_{-1,3,1}$			
11	$x_{0,3,2}$							
12	$x_{0,4,2}$	$x_{0,3,2}$; $x_{0,4,2}$	$e_{-1,4,1}$; $o_{-1,4,1}$	$e_{-1,1,1}$; $e_{-1,2,1}$;$e_{-1,3,1}$;$e_{-1,4,1}$	$o_{-1,1,1}$; $o_{-1,2,1}$;$o_{-1,3,1}$;$o_{-1,4,1}$			
13	$x_{0,5,2}$							
14	$x_{0,6,2}$	$x_{0,5,2}$; $x_{0,6,2}$	$e_{-1,1,2}$; $o_{-1,1,2}$	$e_{-1,2,1}$;$e_{-1,3,1}$;$e_{-1,4,1}$;$o_{-1,1,1}$	$o_{-1,2,1}$;$o_{-1,3,1}$;$o_{-1,4,1}$;$o_{-1,1,2}$	$e_{-1,1,1}$; $e_{-1,1,2}$		
15	$x_{0,7,2}$							
16	$x_{0,8,2}$	$x_{0,7,2}$; $x_{0,8,2}$	$e_{-1,2,2}$; $o_{-1,2,2}$	$e_{-1,3,1}$;$e_{-1,4,1}$;$o_{-1,1,1}$;$o_{-1,2,1}$	$o_{-1,3,1}$;$o_{-1,4,1}$;$o_{-1,1,2}$;$o_{-1,2,2}$	$e_{-1,2,1}$; $e_{-1,2,2}$		
17	$x_{0,1,3}$							
18	$x_{0,2,3}$	$x_{0,1,3}$; $x_{0,2,3}$	$e_{-1,3,2}$; $o_{-1,3,2}$	$e_{-1,4,1}$;$o_{-1,1,1}$;$o_{-1,2,1}$;$o_{-1,3,1}$	$o_{-1,4,1}$;$o_{-1,1,2}$;$o_{-1,2,2}$;$o_{-1,3,2}$	$e_{-1,3,1}$; $e_{-1,3,2}$	$ll_{-1,1,1}$; $lh_{-1,1,1}$	1
19	$x_{0,3,3}$							
20	$x_{0,4,3}$	$x_{0,3,3}$; $x_{0,4,3}$	$e_{-1,4,2}$; $o_{-1,4,2}$	$o_{-1,1,1}$;$o_{-1,2,1}$;$o_{-1,3,1}$;$o_{-1,4,1}$	$o_{-1,1,2}$;$o_{-1,2,2}$;$o_{-1,3,2}$;$o_{-1,4,2}$	$e_{-1,4,1}$; $e_{-1,4,2}$	$ll_{-1,2,1}$; $lh_{-1,2,1}$	1
21	$x_{0,5,3}$	$ll_{-1,1,1}$ $ll_{-1,2,1}$						
22	$x_{0,6,3}$	$x_{0,5,3}$; $x_{0,6,3}$	$e_{-1,1,3}$; $o_{-1,1,3}$	$o_{-1,2,1}$;$o_{-1,3,1}$;$o_{-1,4,1}$;$e_{-1,1,3}$	$o_{-1,2,2}$;$o_{-1,3,2}$;$o_{-1,4,2}$;$o_{-1,1,3}$	$o_{-1,1,1}$; $o_{-1,1,2}$	$ll_{-1,3,1}$; $lh_{-1,3,1}$	1
23	$x_{0,7,3}$							
24	$x_{0,8,3}$	$x_{0,7,3}$; $x_{0,8,3}$	$e_{-1,2,3}$; $o_{-1,2,3}$	$o_{-1,3,1}$;$o_{-1,4,1}$;$e_{-1,1,3}$;$e_{-1,2,3}$	$o_{-1,3,2}$;$o_{-1,4,2}$;$o_{-1,1,3}$;$o_{-1,2,3}$	$o_{-1,2,1}$; $o_{-1,2,2}$	$ll_{-1,4,1}$; $lh_{-1,4,1}$	1
25	$x_{0,1,4}$	$ll_{-1,3,1}$ $ll_{-1,4,1}$	$e_{-2,1,1}$; $o_{-2,1,1}$					
26	$x_{0,2,4}$	$x_{0,1,4}$; $x_{0,2,4}$	$e_{-1,3,3}$; $o_{-1,3,3}$	$o_{-1,4,1}$;$e_{-1,1,3}$;$e_{-1,2,3}$;$e_{-1,3,3}$	$o_{-1,4,2}$;$o_{-1,1,3}$;$o_{-1,2,3}$;$o_{-1,3,3}$	$o_{-1,3,1}$; $o_{-1,3,2}$	$hl_{-1,1,1}$; $hh_{-1,1,1}$	1
27	$x_{0,3,4}$							
28	$x_{0,4,4}$	$x_{0,3,4}$; $x_{0,4,4}$	$e_{-1,4,3}$; $o_{-1,4,3}$	$e_{-1,1,3}$;$e_{-1,2,3}$;$e_{-1,3,3}$;$e_{-1,4,3}$	$o_{-1,1,3}$;$o_{-1,2,3}$;$o_{-1,3,3}$;$o_{-1,4,3}$	$o_{-1,4,1}$; $o_{-1,4,2}$	$hl_{-1,2,1}$, $hh_{-1,2,1}$	1
29	$x_{0,5,4}$		$e_{-2,2,1}$ $o_{-2,2,1}$					
30	$x_{0,6,4}$	$x_{0,5,4}$; $x_{0,6,4}$	$e_{-1,1,4}$; $o_{-1,1,4}$	$e_{-1,2,3}$;$e_{-1,3,3}$;$e_{-1,4,3}$;$o_{-1,1,3}$	$o_{-1,2,3}$;$o_{-1,3,3}$;$o_{-1,4,3}$;$o_{-1,1,4}$	$e_{-1,1,3}$; $e_{-1,1,4}$	$hl_{-1,3,1}$; $hh_{-1,3,1}$	1
31	$x_{0,7,4}$							
32	$x_{0,8,4}$	$x_{0,7,4}$; $x_{0,8,4}$	$e_{-1,2,4}$; $o_{-1,2,4}$	$e_{-1,3,3}$;$e_{-1,4,3}$;$o_{-1,1,3}$;$o_{-1,2,3}$	$o_{-1,3,3}$;$o_{-1,4,3}$;$o_{-1,1,4}$;$o_{-1,2,4}$	$e_{-1,2,3}$; $e_{-1,2,4}$	$hl_{-1,4,1}$; $hh_{-1,4,1}$	1
33	$x_{0,1,5}$							
34	$x_{0,2,5}$	$x_{0,1,5}$; $x_{0,2,5}$	$e_{-1,3,4}$; $o_{-1,3,4}$	$e_{-1,4,3}$;$o_{-1,1,3}$;$o_{-1,2,3}$;$o_{-1,3,3}$	$o_{-1,4,3}$;$o_{-1,1,4}$;$o_{-1,2,4}$;$o_{-1,3,4}$	$e_{-1,3,3}$; $e_{-1,3,4}$	$ll_{-1,1,2}$; $lh_{-1,1,2}$	1
35	$x_{0,3,5}$							
36	$x_{0,4,5}$	$x_{0,3,5}$; $x_{0,4,5}$	$e_{-1,4,4}$; $o_{-1,4,4}$	$o_{-1,1,3}$;$o_{-1,2,3}$;$o_{-1,3,3}$;$o_{-1,4,3}$	$o_{-1,1,4}$;$o_{-1,2,4}$;$o_{-1,3,4}$;$o_{-1,4,4}$	$e_{-1,4,3}$; $e_{-1,4,4}$	$ll_{-1,2,2}$ $lh_{-1,2,2}$	1
37	$x_{0,5,5}$	$ll_{-1,1,2}$ $ll_{-1,2,2}$						
38	$x_{0,6,5}$	$x_{0,5,5}$; $x_{0,6,5}$	$e_{-1,1,5}$; $o_{-1,1,5}$	$o_{-1,2,3}$;$o_{-1,3,3}$;$o_{-1,4,3}$;$e_{-1,1,5}$	$o_{-1,2,4}$;$o_{-1,3,4}$;$o_{-1,4,4}$;$o_{-1,1,5}$	$o_{-1,1,3}$; $o_{-1,1,4}$	$ll_{-1,3,2}$; $lh_{-1,3,2}$	1
39	$x_{0,7,5}$							
40	$x_{0,8,5}$	$x_{0,7,5}$; $x_{0,8,5}$	$e_{-1,2,5}$; $o_{-1,2,5}$	$o_{-1,3,3}$;$o_{-1,4,3}$;$e_{-1,1,5}$;$e_{-1,2,5}$	$o_{-1,3,4}$;$o_{-1,4,4}$;$o_{-1,1,5}$;$o_{-1,2,5}$	$o_{-1,2,3}$; $o_{-1,2,4}$	$ll_{-1,4,2}$; $lh_{-1,4,2}$	1
41	$x_{0,1,6}$	$ll_{-1,3,2}$ $ll_{-1,4,2}$	$e_{-2,1,2}$; $o_{-2,1,2}$			$e_{-2,1,1}$; $e_{-2,1,2}$		
42	$x_{0,2,6}$	$x_{0,1,6}$; $x_{0,2,6}$	$e_{-1,3,5}$; $o_{-1,3,5}$	$o_{-1,4,3}$;$e_{-1,1,5}$;$e_{-1,2,5}$;$e_{-1,3,5}$	$o_{-1,4,4}$;$o_{-1,1,5}$;$o_{-1,2,5}$;$o_{-1,3,5}$	$o_{-1,3,3}$; $o_{-1,3,4}$	$hl_{-1,1,2}$; $hh_{-1,1,2}$	1
43	$x_{0,3,6}$							
44	$x_{0,4,6}$	$x_{0,3,6}$; $x_{0,4,6}$	$e_{-1,4,5}$; $o_{-1,4,5}$	$e_{-1,1,5}$;$e_{-1,2,5}$;$e_{-1,3,5}$;$e_{-1,4,5}$	$o_{-1,1,5}$;$o_{-1,2,5}$;$o_{-1,3,5}$;$o_{-1,4,5}$	$o_{-1,4,3}$; $o_{-1,4,4}$	$hl_{-1,2,2}$ $hh_{-1,2,2}$	1
45	$x_{0,5,6}$		$e_{-2,2,2}$; $o_{-2,2,2}$			$e_{-2,2,1}$; $e_{-2,2,2}$	$ll_{-2,1,1}$, $lh_{-2,1,1}$	2
46	$x_{0,6,6}$	$x_{0,5,6}$; $x_{0,6,6}$	$e_{-1,1,6}$; $o_{-1,1,6}$	$e_{-1,2,5}$;$e_{-1,3,5}$;$e_{-1,4,5}$;$o_{-1,1,5}$	$o_{-1,2,5}$;$o_{-1,3,5}$;$o_{-1,4,5}$;$o_{-1,1,6}$	$e_{-1,1,5}$; $e_{-1,1,6}$	$hl_{-1,3,2}$; $hh_{-1,3,2}$	1
47	$x_{0,7,6}$							
48	$x_{0,8,6}$	$x_{0,7,6}$; $x_{0,8,6}$	$e_{-1,2,6}$; $o_{-1,2,6}$	$e_{-1,3,5}$;$e_{-1,4,5}$;$o_{-1,1,5}$;$o_{-1,2,5}$	$o_{-1,3,5}$;$o_{-1,4,5}$;$o_{-1,1,6}$;$o_{-1,2,6}$	$e_{-1,2,5}$; $e_{-1,2,6}$	$hl_{-1,4,2}$; $hh_{-1,4,2}$	1
49	$x_{0,1,7}$					$o_{-2,1,1}$; $o_{-2,1,2}$	$ll_{-2,2,1}$ $lh_{-2,2,1}$	2
50	$x_{0,2,7}$	$x_{0,1,7}$; $x_{0,2,7}$	$e_{-1,3,6}$; $o_{-1,3,6}$	$e_{-1,4,5}$;$o_{-1,1,5}$;$o_{-1,2,5}$; $o_{-1,3,5}$	$o_{-1,4,5}$;$o_{-1,1,6}$;$o_{-1,2,6}$;$o_{-1,3,6}$	$e_{-1,3,5}$; $e_{-1,3,6}$	$ll_{-1,1,3}$; $lh_{-1,1,3}$	1
51	$x_{0,3,7}$							
52	$x_{0,4,7}$	$x_{0,3,7}$; $x_{0,4,7}$	$e_{-1,4,6}$; $o_{-1,4,6}$	$o_{-1,1,5}$;$o_{-1,2,5}$; $o_{-1,3,5}$;$o_{-1,4,5}$	$o_{-1,1,6}$;$o_{-1,2,6}$;$o_{-1,3,6}$;$o_{-1,4,6}$	$e_{-1,4,5}$; $e_{-1,4,6}$	$ll_{-1,2,3}$; $lh_{-1,2,3}$	1
53	$x_{0,5,7}$	$ll_{-1,1,3}$ $ll_{-1,2,3}$					$hl_{-2,1,1}$; $hh_{-2,1,1}$	2

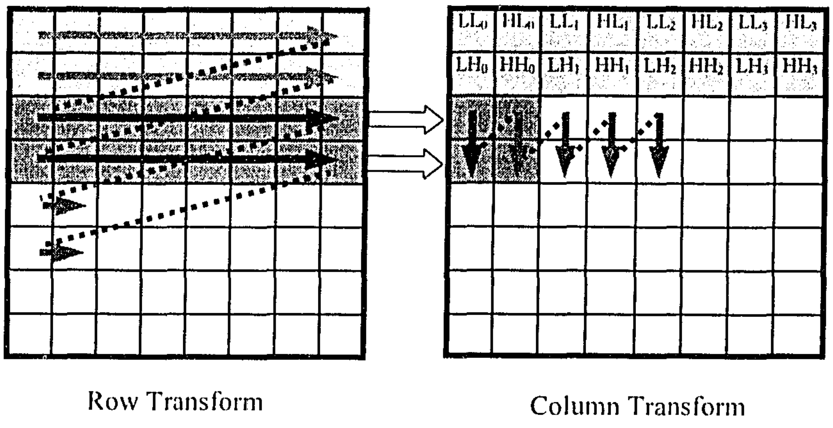


Figure 33. Scan Sequence of the 2-D Dual-scan Architecture

5.2 The 2-D Dual-scan Architecture

In a conventional 2-D DWT algorithm, the vertical DWT is carried out only after the horizontal DWT is finished. This delay between the row and column computations limits the processing speed. The 2-D DSA shortens the delay by adopting a new scan sequence. In applications that can read two pixels per clock cycle from a data buffer, the scan sequence of the 2-D DSA shown in Figure 33 can be used. The row processor scans along two consecutive rows simultaneously, while the column processor also horizontally scans in the row DWT coefficients. In this way, the column processor can start its computation as soon as the first pair of row DWT coefficients is ready. With this improvement, the row and column processors can start computing the same stage of the DWT within only a few clock cycles of each other.

The structure of the 2-D DSA is shown in Figure 34. The registers are used to separately hold the even and odd pixels of each row, and to interleave the input pairs of each two consecutive rows. The control timing of the 2-D DSA is shown in Table 11, where the delay of the row and column processor is assumed to be 1 clock cycle. The row processor of the 2-D DSA is identical to the direct implementation of the 1-D DWT. The column processor is obtained by replacing the 1-pixel delay units in the row processor with 1-row delay units. The low frequency output switch of the column processor directs the *LL* subband of each stage DWT to the memory bank. The *LL* subimage stored in the memory will be returned to the DSA input for further decomposition after the current DWT stage is finished.

Table 11. Data Flow for the 2-D Dual-scan Architecture

$x_{i,j}$ is input signal, i, j the row and column sequences, respectively; $e_{-i,j}$ and $o_{-i,j}$ are even and odd intermediate results of each lifting step; $l_{-i,j}$ and $h_{-i,j}$ are low and high frequency DWT coefficients

Clk	Input	Row Processor		Column Processor	
		E_R ; O_R	L_R ; H_R	E_C ; O_C	L_C ; H_C
1	$x_{1,1}$ $x_{1,2}$				
2	$x_{2,1}$ $x_{2,2}$	$x_{1,1}$; $x_{2,1}$			
3	$x_{3,1}$ $x_{3,2}$	$x_{1,2}$; $x_{2,2}$	$e_{1,1}$; $o_{1,1}$		
4	$x_{4,1}$ $x_{4,2}$	$x_{3,1}$; $x_{4,1}$	$e_{1,2}$; $o_{1,2}$	$e_{1,1}$; $e_{1,2}$	
5		$x_{3,2}$; $x_{4,2}$	$e_{2,1}$; $o_{2,1}$	$o_{1,1}$; $o_{1,2}$	$ll_{1,1}$; $lh_{1,1}$
6			$e_{2,2}$; $o_{2,2}$	$e_{2,1}$; $e_{2,2}$	$hl_{1,1}$; $hh_{1,1}$
7				$o_{2,1}$; $o_{2,2}$	$ll_{2,1}$; $lh_{2,1}$
8					$hl_{2,1}$; $hh_{2,1}$

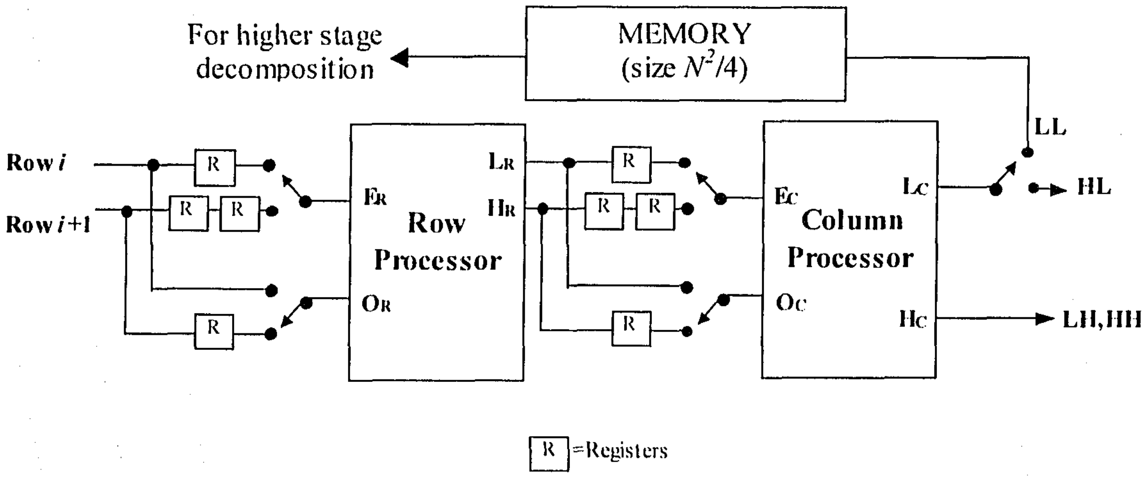


Figure 34. 2-D Dual-scan Architecture

The processing time for each stage is $1/2 \times N \times N + 2 \times T_d$. Because only a quarter of the coefficients are further decomposed, the total processing time for an L-stage 2-D DWT is:

$$2/3 \times N \times N \times (1 - 1/4^L) + 2 \times T_d \times L$$

Compared to a conventional implementation, the DSA uses roughly half of the time to compute the 2-D DWT, and the size of the memory for storing the row transform coefficients is reduced to M rows, where M is the number of delay units in a 1-D filter. The comparisons of the processing time and memory size are shown in Table 12 and Table 13, respectively. In Table 12, the timing for the RA is based on one input pixel per clock cycle, while the others are based on two input pixels per cycle.

Table 12. Computation Time and Hardware Utilization for 2-D Architectures

NxN: Size of the input image. T_d, T_{delay}: Circuit delay. L: Number of DWT stages

Architecture (9/7 DWT)	Computation Time (clock cycles)	Hardware utilization
RA	$N \times N + N + 2 \times L \times T_d + 2^{L-1} - 1$	50% - 70%
DS	$2/3 \times N \times N \times (1-1/4^L) + 2 \times T_d \times L$	≈100%
Direct implementation	$4/3 \times N \times N \times (1-1/4^L) + 2 \times T_d \times L$	50%
ACT[34]	$N \times N \times 4/3 \times (1-1/4^L) + T_d \times L$	≈ 50%

Table 13. Comparison of Memory Size for 2-D Architectures

NxN: Size of the input image. T_d, T_{delay}: Circuit delay. L: Number of DWT stages

Architecture	Memory Size
RA for 9/7 wavelet	$4N$
RA for D4 wavelet	$10N$
DSA for 9/7 wavelet	$N \times N/4 + 4N$
Direct implementation	$5/4 \times N \times N$
ACT[34] for 9/7 wavelet	$\cong N \times N/4$ (external memory)

Chapter 6

Implementation

We implemented the proposed architectures as behavioural-level VHDL models and confirmed their correctness in simulation. In this chapter, we will describe four implementation examples of our proposed architectures. All of our VHDL models are fixed-point designs primarily for the fast verification of our proposed architectures. Nevertheless, the VHDL models of the proposed architectures are still ready to be used in SOPC (System On Programmable Chip) designs or ASIC (Application-Specific Integrated Circuits) designs as DWT engines for processing 8-bit input signals. The fixed-point designs can be easily modified to more precise floating-point designs by replacing the fixed-point arithmetic units with floating-point units and changing the width of registers/files to 32-bit or 64-bit wide.

The rest of the chapter is organized as follows: in the first section, we introduce the word length selection in the fixed-point DWT hardware implementations; in the second section, we describe the 1-D designs: 1-D 9/7 RA and 1-D D-4 DSA; in the third section, we describe the 2-D designs: 2-D 9/7 RA and 2-D D-4 DSA; in the last section, we present the evaluation of our designs.

6.1 Word Length Selection

When the signal is converted to digital form, the precision is limited by the number of bits available. The finite word length of the hardware used for digital processing determines the available precision and dynamic range. The selection of the word length should satisfy both the dynamic range of the DWT coefficients and the distortion tolerance of the reconstructed signal. The dynamic range of the signal processing computation determines the number of bits required to represent the integer part of the DWT coefficients, and it increases with the number of decomposition stages. The number of bits reserved for the integer part should be sufficient to prevent calculation overflow. To estimate the maximum internal growth of the lifting algorithm, we first deduce the equivalent high-pass and low-pass FIR filter polynomials by reversing the process of factoring the DWT filters as follows:

1. For any lifting scheme expressed in polyphase matrix, we have

$$P(z) = \prod_{i=1}^{m} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} = \begin{bmatrix} \sum\limits_{i=0}^{J-1} h_{2i}(z) & \sum\limits_{k=0}^{G-1} g_{2k}(z) \\ \sum\limits_{i=0}^{J-1} h_{2i+1}(z) & \sum\limits_{k=0}^{G-1} g_{2k+1}(z) \end{bmatrix}, \tag{5.1}$$

where G and J are the lengths of the high-pass and the low-pass filter, respectively.

2. For any input signal series $\left[x_{2n}\, x_{2n+1}\right]$, the 2×2 matrix in Eq 5.1 is equivalent to two filters as follows.

$$\begin{cases} g(z) = \sum\limits_{i=0}^{G-1} g_i(z) \\ h(z) = \sum\limits_{i=0}^{J-1} h_i(z) \end{cases} \tag{5.2}$$

The maximum input signal internal growth of these filters is [41]:

$$\begin{cases} \sum_{i=0}^{J-1} |g_i(z)| \\ \sum_{i=0}^{J-1} |h_i(z)| \end{cases}$$

As an example, for the D-4 filters expressed as:

$$\begin{cases} h(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} \\ g(z) = -h_3 z^2 + h_2 z^1 - h_1 + h_0 z^{-1} \end{cases},$$

where $h_0 = \dfrac{1+\sqrt{3}}{4\sqrt{2}}, h_1 = \dfrac{3+\sqrt{3}}{4\sqrt{2}}, h_2 = \dfrac{3-\sqrt{3}}{4\sqrt{2}}, h_3 = \dfrac{1-\sqrt{3}}{4\sqrt{2}}$,

their maximum internal growth for each stage of DWT calculation is:

$$\left|\frac{1+\sqrt{3}}{4\sqrt{2}}\right| + \left|\frac{3+\sqrt{3}}{4\sqrt{2}}\right| + \left|\frac{3-\sqrt{3}}{4\sqrt{2}}\right| + \left|\frac{1-\sqrt{3}}{4\sqrt{2}}\right| \approx 1.673.$$

The number of bits N needs to represent the internal growth for the 4-stage one-dimensional D-4 DWT is $N \geq \log_2(4 \times 1.673) \approx 2.7$. Hence, we need 3 more bits than the input signal word length to represent the integer part of the D-4 DWT coefficients.

Another possible cause of signal distortion is multiplier product round-off error. For an M-bit by N-bit signed multiplication, the result is $N+M$-1 bits wide. The round-off noise is introduced when we truncate the multiplication result. Determining the optimal number of bits to be reserved for the multiplication result is not as straightforward as estimating the internal signal growth. Designing fixed-point digital filters with minimum roundoff noise is described in [42]. In our research, we used a simpler method to estimate the number of bits need to represent the decimal part of the DWT coefficients.

Since the roundoff noise is the only noise source in the DWT calculation, we can estimate the acceptable roundoff noise level from the required signal noise ratio (SNR) or peak SNR (PSNR) of the DWT caculation. From the SNR and the estimated input signal variance, we can deduce the noise variance as follows.

$$\because \qquad SNR = 10\log_{10}\frac{\sigma_S^2}{\sigma_N^2},$$

where σ_S^2 is the variance of the input signal, and σ_N^2 is the variance of noise

$$\therefore \qquad \sigma_N^2 = \frac{\sigma_S^2}{10^{SNR/10}}$$

Since the variance of the roundoff noise of a fixed-point digital filter with N multiplications is [42]

$$\sigma_N^2 = N\frac{2^{-2B}}{12},$$

$$B = -\frac{1}{2}\log_2\frac{12\sigma_N^2}{N} = -\frac{1}{2}\log_2\frac{12\frac{\sigma_S^2}{10^{SNR/10}}}{N}$$

where B is the number of bits representing the decimal part of the output signal. For different input signals and DWT filters, B varies siganificantly. As a rule of thumb, we may start with $\sigma_S^2 = 3000$ and SNR=50 to calculate the number B for image processing applications. It should be noted that N is the number of multiplications for each DWT filter times the number of DWT stages.

Using the calculations described above, we estimate that the maximum internal growth of the 8-bit integer input signal is 3 bits and the number of bits for the decimal part is 5 bits. In reality, the negative and positive filter coefficients will cancel out part of the internal signal growth, and the number of bits assigned for the decimal part should be sufficient to represent the minimum filter coefficient. Therefore, we used 16 bits (11-bit integer and 5-bit decimal) to represent the DWT coefficients in our designs.

6.2 Implementations of the 1-D Architectures

6.2.1 1-D 9/7 Recursive Architecture

The 1-D 9/7 RA architecture consists of a datapath, switching control signal generators, and a controller. The datapath of the1-D 9/7 RA design contains a processing element and an input switch network. The state diagram of the datapath is shown in Figure 24. The switching signal generators generate all the signals for directing the dataflow of each DWT stage to its assigned registers, as described in Chapter 4.

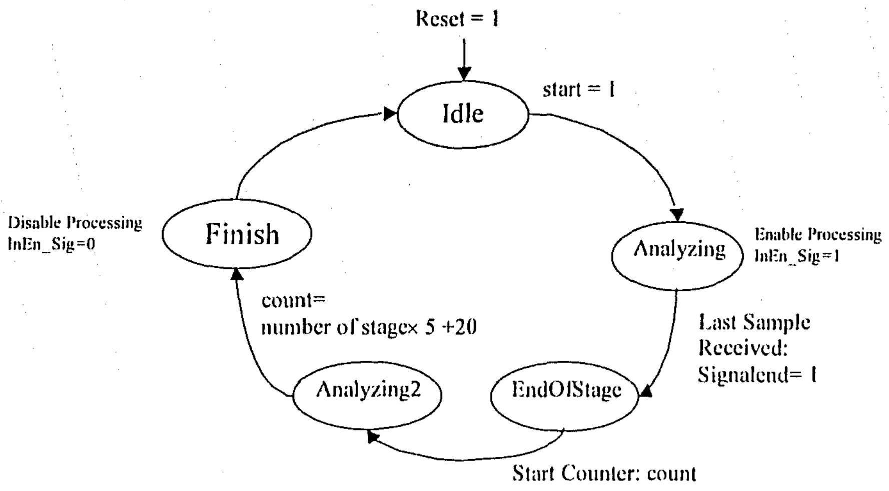


Figure 35. State Transform Diagram of the 1-D 9-7 RA's Controller

The controller is a 5-state controller as shown in Figure 35. The controller generates an enable processing signal to start the DWT computation when it asserts a start signal. which is synchronized with the first input sample. When an input end signal synchronized with the last input sample is asserted, the controller starts a counter that keeps track of the clock cycles required to complete the DWT calculation.

After the last input sample enters the processing element, the time slots assigned to the first stage DWT, as shown in Figure 25, are now available to other stages of DWT computation. Therefore, the processing of all the higher stages of the DWT can be moved to the time slots assigned for their previous stages, and their calculation frequencies can also be doubled. The same process is repeated after each lower stage DWT is finished. In this way, the computation time of the multiple-stage DWT is significantly shortened. Since the pipeline length of the 9/7 processing element is five, and one zero is appended to the last sample of each stage to flush the pipeline, the calculation of the L-stage 9/7 DWT can be completed in $L\times(5+1)$ system clock cycles after the last input sample is received.

When the counter counts up to the preset value ($L\times$ (5+1)), the controller deasserts the computation enable bit, and returns to the standby (Idle) state, as illustrated in Figure 35 .

6.2.2 1-D Daub-4 Dual-Scan Architecture

The VHDL model of the 1-D DSA is a simplified design for calculating three-stage Daub-4 DWT. The design is not refined to process random number of stages of the DWT, but rather for verifying the concept and feasibility of the DSA. The architecture consists of a datapath, as shown in Figure 27, and a controller. The datapath includes a processing element (PE), as shown in Figure 36, a memory bank, and some switches directing the dataflow. The memory bank is configured as two first-in-first-out (FIFO) registers. Since the signal length is reduced by half for each higher-stage DWT computation, we clock the FIFOs at different frequencies to change the virtual length of the FIFOs corresponding to the system clock: if we double the clock of a FIFO, it will take only half of time for a signal to travel through the FIFO. In this way we keep the design simple; however, it needs a higher than system clock frequency to shorten the virtual length of a FIFO to less than half of its physical length. An alternative design is for each FIFO to have multiple output ports, which are located at the end of the FIFO, and also at the ½, ¼, …, length of the FIFO. The clock frequency of this design, which would be half of the PE clock frequency, could be maintained constant for all of the DWT stages.

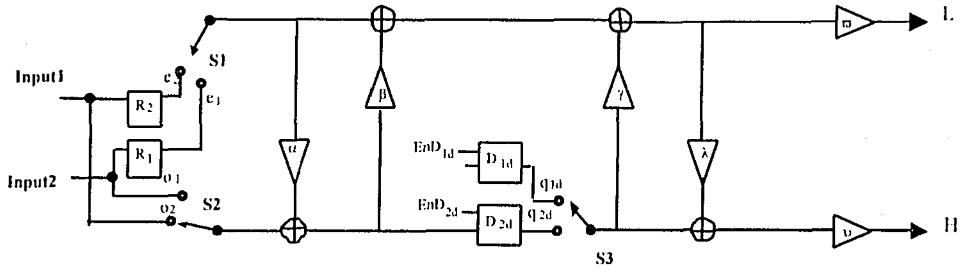


Figure 36. Datapath of 1-D Daub-4 DSA

The state transfer diagram of the controller is shown in Figure 37. The controller enables the DSA processor when it detects the first input sample, and starts a counter to sequence the calculation of the DWT. After a number of clock cycles, which equals the pipeline length of the PE, have elapsed, the controller starts to toggle between the 'ToFIFO1' and the 'ToFIFO2' states. At the 'ToFIFO1' state, the controller directs the high-frequency DWT coefficients of the first signal flow to FIFO1; at the 'ToFIFO2' state, it directs the high-frequency products of the other signal flow to FIFO2. The controller transfers to the 'FinalStg' State after the calculation of the first DWT stage is complete. At this stage, the controller doubles the frequency of the FIFO clock to reduce the FIFOs' virtual lengths. After the last samples of the second stage have entered the FIFOs, the controller detours the high-frequency product of the PE to the output. The controller returns to the 'Idle' state after the processing of all stages has finished.

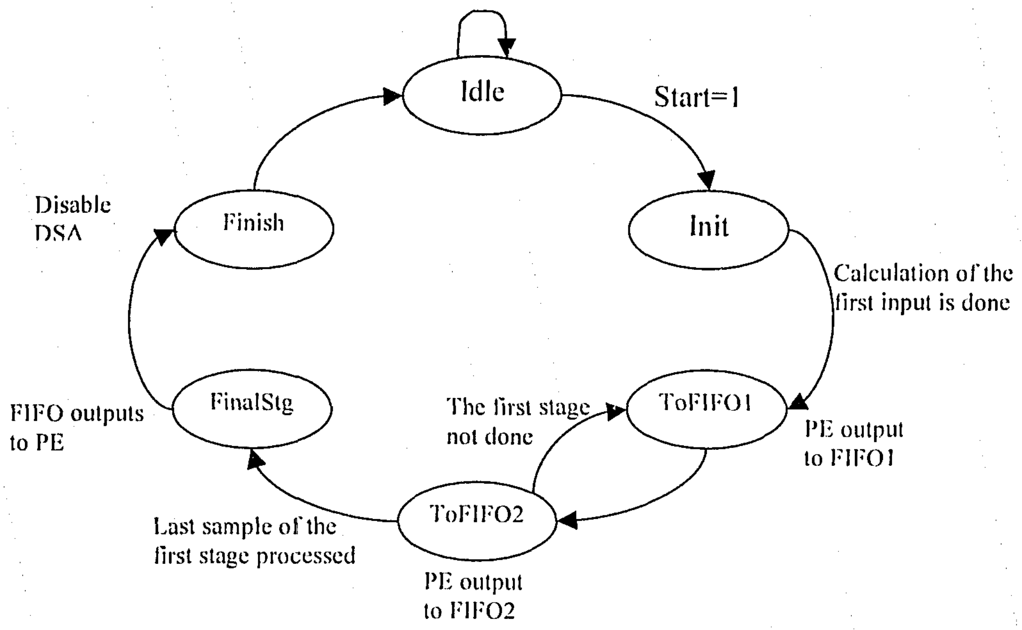


Figure 37. State Transfer Diagram of 1-D DSA's Controller

6.3 Implementations of the 2-D Architectures

6.2.1 2-D Daub-4 Recursive Architecture

As shown in Figure 31, the 2-D Recursive Architecture consists of a Row PE, a Column PE, a controller, registers and memories for temporarily storing the intermediate calculation results, and switches for directing the dataflows of the different

DWT decomposition stages. The timing of the switch signals for the 2-D Daub-4 RA is shown in Table 9.

The controller sequences the calculation of the multi-stage DWT, and governs the row and column transitions. The state transfer diagram of the controller is shown in Figure 38. The controller enables processing when it receives the first input sample. At the end of the processing of each row, the controller performs a zero-padding boundary treatment by reseting all of the registers in the Row Processor. After all of the DWT stages have been calculated, the controller disables further processing.

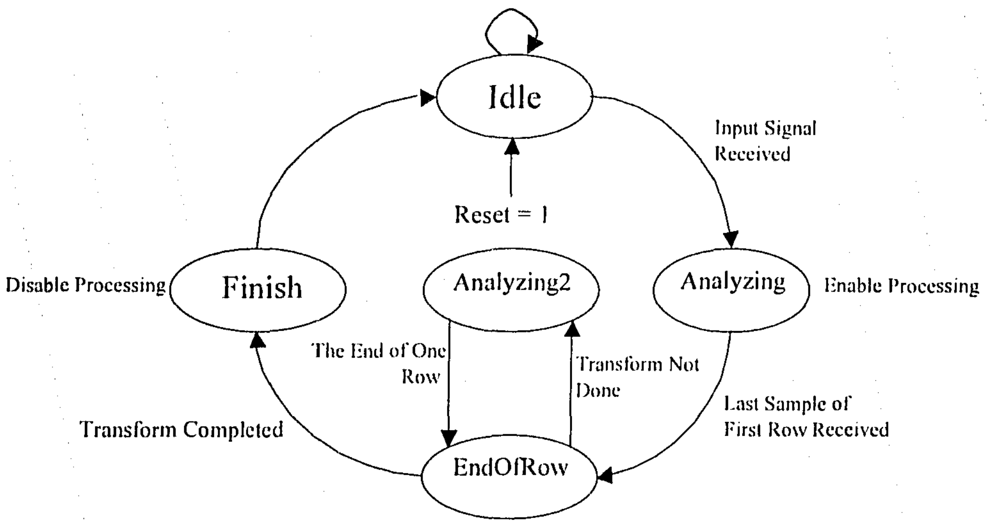


Figure 38. State Transfer Diagram of the 2-D D-4 RA's Controller

6.2.1 2-D Daub-4 Dual-Scan Architecture

The structure of the 2-D DSA is shown in Figure 34. In the 2-D DSA, the memory bank stores the low frequency sub-image (LL) of each decomposition stage. Since the calculations of 2-D DWT in the DSA is identical to all stages, except that the input image sizes are different, we only implemented a DSA for calculating one stage of the

2-D Daub-4 DWT. Expanding the one-stage design to a multiple-stage design is straightforward: add a switch at the low-frequency output of the column processor for directing the decomposed subimage LL to a memory, and a pair of switches for selecting the input source of the row processor between the external signal and the memory.

In the 2-D Daub-4 DSA design, we use row counters and column counters to generate periodical signals for resetting the internal registers (zero-padding). The controller is a simple four-state Moore machine determining the start and stop of the row processor and the column processor, as shown in Figure 39.

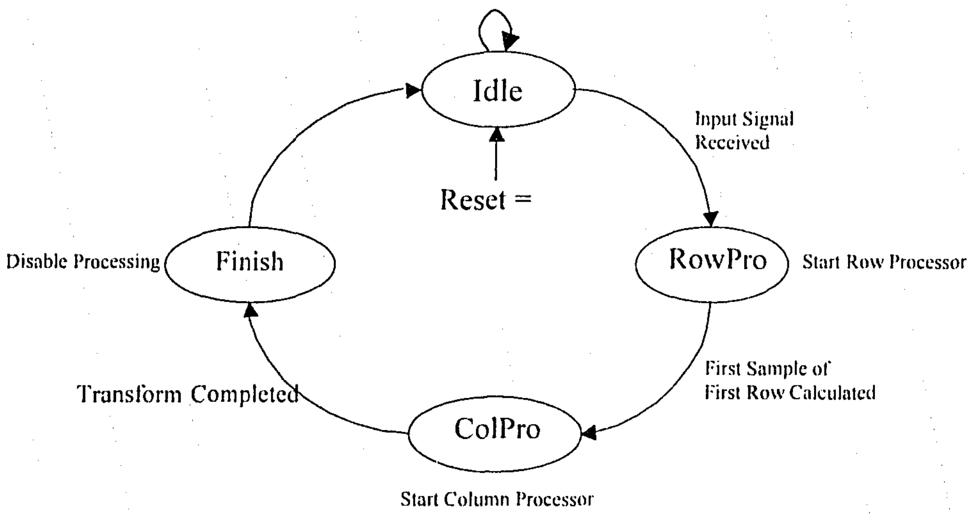


Figure 39. State Transfer Diagram of 2-D DSA

6.4 Evaluation

In order to verify the correctness of our designs, we used our 2-D architectures to calculate the 3-stage forward DWT of the two gray level images shown in Figure 40. The decomposed images of the test images calculated by our 2-D Daub-4 recursive architecture are shown in Figure 41. We compared these decomposed images to the images produced using Matlab (programs can be found in Appendix A), which are considered zero distortion in our evaluation. The histogram of the wavelet coefficient error in three-stage DWT of the Lena image using the 2-D Daub-4 RA is ploted in Figure 42; the histograms of errors in other test images are similar. From Figure 42, we see that the accuracy of our 2-D RA architecture is within ±0.3. Since PSNR is a better criteria in evaluating the quality of reconstructed images, we also calculated the PSNR values of our 2-D architectures in decomposing the selected images. The PSNR and SNR values of the decomposed images calculated by our architectures are listed in Table 14. As discussed in the first section of this chapter, increasing the word length can further improve the performance of the designs.

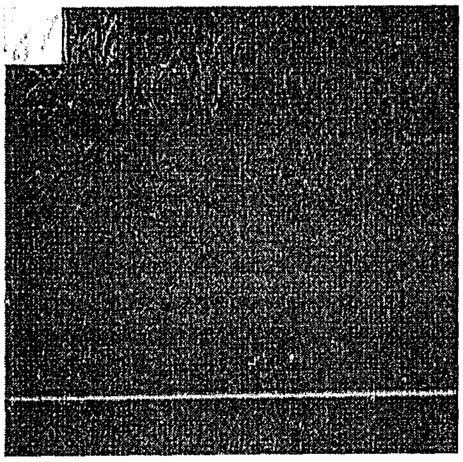
To evaluate the physical sizes of the proposed architecture implemented as silicon layouts, we synthesised our designs with FPGA and ASIC design tools. The proposed architectures were synthesised and implemented for Xilinx's Virtex II FPGA XC2V250, which is a high-performance medium-size FPGA. The 1-D RA implementing the 3-stage 9/7 lifting-based DWT uses 409 logic slices out of the 1536 slices available in the FPGA. The 2-D RA implementing the 3-stage Daub-4 DWT uses 879 logic slices, and can compute the DWT of 8-bit gray level images of sizes up to 6000×6000 at 50 MHz using the built-in RAM blocks and multipliers in the FPGA. To estimate the corresponding silicon areas for ASIC designs, we used Synopsys' Design Compiler [45] to synthesize the above architectures with TSMC's 0.18-μm standard cell library aiming for 50 MHz operation. Since the MAC unit is the critical element in the designs, higher operation frequency can be achieved by implementing faster multipliers or by pipelining the MAC units and minimizing the routing distance of each section of the pipeline. The synthesized designs were then placed and routed using the Silicon Ensemble tool, and the final layouts were generated by using Cadence Design Framework II [46]. The core size of the 1-D RA implementing the 3-stage 9/7 DWT is about 0.177 mm^2 (90% of which is the datapath, 10% is the controller, and the rest is memory). The core size of the 2-D RA that calculates the 3-stage Daub-4 DWT of a 256×256 image is about 2.25 mm^2 (about 15% of which is the datapath, 5% is the controller, and the rest is memory). The core area could be reduced by reimplementing the delay units as register files instead of separate flip-flops, and the performance of the proposed architectures could be further improved by optimizing the circuit designs.

Table 14. SNR/PSNR Values for 3-stage forward DWT

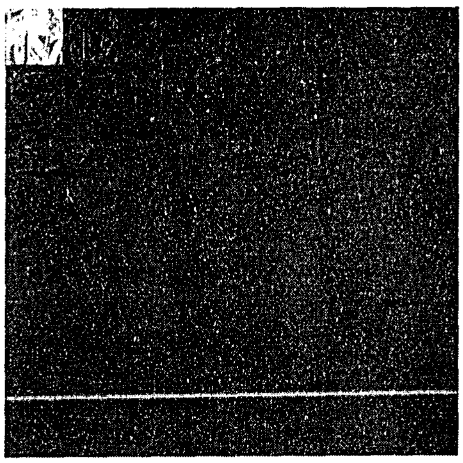
	Lena		Barbara	
	SNR	PSNR	SNR	PSNR
Daub-4	69.6529	75.32	69.2755	75.18
9/7	69.1437	74.85	68.7880	74.73

(a) Lena (b) Barbara

Figure 40. Test Images



(a) Decomposed image of Lena



(b) Decomposed image of Barbara

Figure 41. Decomposed Images of Test Images

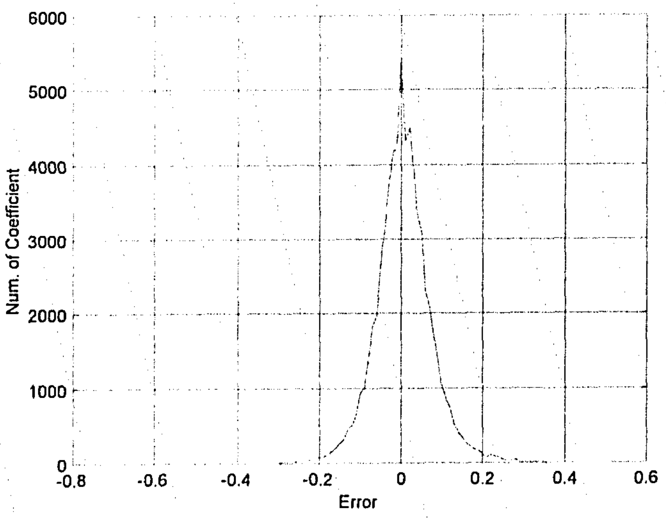


Figure 42. Histogram of the Error of 3-Stage Decomposition of Lena

Chapter 7

Conclusions and Future Work

We proposed a recursive architecture and a dual-scan architecture for computing the DWT based on the lifting scheme. In the previous chapters, we described the procedures for implementing 1-D and 2-D versions of the RA and DSA for calculating any lifting-based wavelet transforms. We also illustrated the details of the hardware design by describing implementations of the 1-D 9/7 RA, the 1-D Daub-4 DSA, the 2-D Daub-4 RA, and the 2-D Daub-4 DSA as examples.

Compared to previous implementations of the lifting-based DWT, the proposed architectures have higher, and hence more efficient, hardware utilization and shorter computation time. In addition, since the recursive architectures can continuously compute the DWT coefficients as soon as the input data become available, the memory size required for storing the intermediate results is minimized. Hence, the sizes and power consumptions of both the 1-D and 2-D recursive architectures are reduced compared to other implementations. In addition, since the designs are modular, they can be easily extended to implement the separable multi-dimensional DWT by cascading multiple basic 1-D DWT processors.

However, there are limitations in our architecture designs: each specific datapath can only calculate one type of lifting-based wavelet. Although it should be relatively straightforward to design new architectures following our procedure, and many image applications only use a few types of DWT, our architectures are not yet capable of providing the convenience and the generality of any lifting schemes. Theoretically, constructing general synthesis procedures for the RA and DSA architectures should be feasible, but that would be beyond the scope of our thesis research. The proposal of such general architectures is one possible direction for future work.

Another direction for future work could be to extend our architectures to implement the lifting scheme for multiwavelet applications. Multiwavelet analysis has been found promising in applications, such as image compression and denoising [48][49]. Recent research has revealed that multiwavelets can also be constructed using the lifting scheme and any compactly supported multiwavelet can be factored into lifting steps. Hence, it seems likely that our architectures could be modified to implement multiwavelets.

References

[1] A. Grossmann and J. Molet, "Decomposition of Hardy functions into square integrable wavelets of constant shape", *SIAM J. Math. Ana.*, vol. 15, pp. 723-736, 1984.

[2] I. Daubechies, "Orthonormal bases of compactly supported wavelets", *Comm. Pure Appl. Math.*, vol. 41, pp. 909–996, 1988.

[3] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans.on Pattern Anal. Machine Intell.*, vol. 11, pp. 674-693, Jul. 1989.

[4] G. Knowles, "VLSI Architecture for the Discrete Wavelet Transform", *IEE Electronics Letters*, v 26 n 15, pp. 1184-1185, Jul 19 1990.

[5] A. Lewis and G. Knowles, "VLSI Architecture for 2-D Daubechies Wavelet Transform without Multipliers", *Electronics Letters*, vol. 27, no. 2, pp. 171-173, Jan.1991

[6] I. Daubechies, "Ten Lectures on Wavelets", *CBMS-NSF Regional Conf. Series in Appl. Math.*, Society for Industrial and Applied Mathematics, vol. 61, pp. 279, Philadelphia, PA, 1992.

[7] J. Kova¡cević and M. Vetterli. "Nonseparable multidimensional perfect reconstruction filter banks and wavelet bases for R^{n}", *IEEE Trans. Inform. Theory*, vol. 38, no. 2, pp. 533–555, Mar. 1992.

[8] K. Parhi and T. Nishitani, "VLSI Architectures for Discrete Wavelet Transforms", *IEEE Trans. on VLSI Systems*, vol. 1, no. 2, pp. 191-202, Jun. 1993

[9] A. Cohen, I. Daubechies, and P. Vial, "Wavelets On the Interval and Fast Wavelet Transforms", *Applied and Computational Harmonic Analysis*, vol. 1, pp. 54-81, Dec 1993.

[10] C. Chakrabarti and M. Vishwanath, "Efficient Realizations of the Discrete and Continuous Wavelet Transforms: From Single Chip Implementation to Mappings on SIMD Array Computers", *IEEE Trans. on Signal Processing*, vol. 43, no. 3, pp. 759-769, Mar. 1995

[11] W. Sweldens and P. Schroder, "Building Your Own Wavelet at Home", Report 1995:5, Industrial Mathematics Initiative, Department of Mathematics, University of South Carolina, 1995.

[12] M. Carnicer, W. Dahmen, and J.M. Pena, "Local decomposition of refinable spaces", *Appl. Comput. Harm. Anal.*, vol. 3, pp. 127-153, 1996.

[13] C. M. Brislawn, "Classification of nonexpansive symmetric extension transforms for multirate filter banks," *Appl. Comput. Harmon. Anal.*, vol. 3, pp. 337–357, 1996.

[14] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets", *Appl. Comput. Harmon. Anal.*, vol. 3, no. 2, pp. 186–200, 1996.

[15] W. Sweldens, "Wavelets: What next?", *Proc. of the IEEE*, vol. 84, no. 4, pp. 680-685, 1996.

[16] A. Grzeszczak, M.K. Mandal, S. Panchanathan, "VLSI implementation of discrete wavelet transform", *IEEE Trans. on VLSI systems*, Part II, vol. 47, no. 12, pp. 1492-1502, 1996.

[17] W. Sweldens, "The lifting scheme: A construction of second generation wavelets", *SIAM J. Math. Anal.*, vol. 29, no. 2, pp. 511–546, 1997.

[18] G. Uytterhoeven and A. Bultheel, "The Red-Black wavelet transform", *Proceedings of the IEEE Benelux Signal Processing Symposium*, Leuven, Belgium, pp. 191–194, Mar. 1998.

[19] W. Sweldens, "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Construction", *J. Fourier Anal. Appl.*, vol. 4, pp. 247-269, 1998.

[20] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps", *J. Fourier Anal. Appl.*, vol. 4, no. 3, pp. 245–267, 1998.

[21] L. He, Y. Zeng, "A Fast Algorithm for Two or More Dimensional Nonseparable Wavelets", *IEEE Conference on Signal Processing Proceedings*, vol.1, pp. 284 - 287, 12-16 Oct. 1998

[22] W. Jiang and A. Ortega, "Parallel Architecture for the Discrete Wavelet Transform based on the Lifting Factorization," *SPIE Conference on Parallel and Distributed Methods for Image processing III,* Denver, Colorado, Jul. 1999.

[23] X. Xiong, *et al.*, "A Comparative Study of DCT- and Wavelet-Based Image Coding", *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 9, no. 5, AUG. 1999.

[24] D. Gunawan, "Denoising Images Using Wavelet Transform", *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Pages:83 – 85, Aug. 1999

[25] J. Kovacevic and W. Sweldens, Interpolating filter banks in arbitrary dimensions, US Patent No. 6,018,753, Jan. 2000.

[26] Y. Sheng, "Wavelet Transform", *The Transforms and Applications Handbook, Second Edition,* CRC Press LLC, 2000.

[27] M. Marcellin, *et al.*, "An overview of JPEG-2000", *Pro. of IEEE Data Compression Conf.,* Snowbird, Utah, USA, pp. 534-541, 2000.

[28] C. Taswell, "The what, how and why of shrinkage wavelet denoising", *Computing in Science and Engineering,* vol. 2, pp. 12-19, May 2000.

[29] H. L. Liao, B. F. Cockburn, and M. K. Mandal, "Efficient Implementations of the Wavelet Transform on the Parallel DSP-RAM Architecture," *Proc. of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1189-1192, Toronto, Canada, 2001.

[30] S. Gnavi, B. Penna, M. Grangetto, E. Magli, G. Olmo, "Wavelet kernels on a DSP: a comparison between lifting and filter banks for image coding," *Applied Signal Processing* "Special Issue on Implementation of DSP and Communication Systems", Vol. 2002, No. 9, pp. 981-989, Sept. 2002.

[31] C. Dolabdjiana, *et al.*, "Classical low-pass filter and wavelet-based denoising technique implemented on a DSP: a comparison study", *Eur. Phys. J. AP 20*, pp. 135-140, Nov. 2002.

[32] P. Wu and L. Chen, "An Efficient Architecture for Two-Dimensional Discrete Wavelet Transform", *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 11, no. 4, pp. 536-544, Apr. 2001.

[33] C. Lian, *et al.*, "Lifting Based Discrete Wavelet Transform Architecture for JPEG2000," *IEEE International Symposium on Circuits and Systems* (ISCAS 2001), Sydney, Australia, pp. 445-448, May 2001.

[34] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform," *IEEE Trans. on Signal Processing*, vol. 50, no. 40, pp. 966-977, April 2002.

[35] F. Arguello, *et al.*, "Architecture for Wavelet Packet Transform Based on Lifting Steps", *J. Parallel Computing*, vol. 28, no. 7-8, pp. 1023-1037, August 2002.

[36] H. Y. Liao, M. K. Mandal, and B. F. Cockburn, "Efficient Implementation of the Lifting-based Discrete Wavelet Transform," *IEE Electronics Letters*, vol. 38, no. 18, pp. 1010-1012, Aug. 29, 2002.

[37] *JPEG2000 Part II Final Committee Draft*, ISO/IEC JTC 1/SC 29/WG 1, http://www.jpeg.org.

[38] *JPEG Part I*, ISO/IEC IS 10918-1 | ITU-T Recommendation T.81, http://www.jpeg.org.

[39] *DV202 JPEG2000 Video Codec*, http://www.analog.com.

[40] A. Alimohammad, S. J. Dillen and B. F. Cockburn, "DSP-RAM: A SIMD Processor-in-Memory for Signal Processing," to be submitted to *IEEE Transactions on Parallel and Distributed Systems*.

[41] A. Antoniou, *Digital Filters – Analysis, Design, and Applications*, McGraw- Hill Inc, New York, March 1994.

[42] C. T. Mullis, and R. A. Roberts, "Synthesis of Minimum Roundoff Noise FixedPoint Digital Filters", *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS*, Vol. CAS-23, No. 9, Sep. 1976.

[43] D. B. Williams, V. K. Madisetti, *Digital Signal Processing Handbook*, CRC Press LLC, 2000.

[44] Xilinx ISE Foundation, Xilinx, Inc., Product Version: ISE 3.1*i*, 2002.

[45] Design Analyzer, Synopsys, Inc., Product Version: 2000.05 –2, 2000.

[46] Design Framework II, Cadence Design Systems, Inc., Product Version 1.10.

[47] H. L. Liao, M. K. Mandal , and B. F. Cockburn, "Efficient Architectures for 1-D and 2-D Lifting-Based Wavelet Transforms," *IEEE Trans. on Signal Processing*, Vol. 52, No. 5, pp. 1315-1326, May 2004.

[48] G. Davis, V. Strela, and R. Turcajova, "Multiwavelet construction via the lifting scheme," *Wavelet Analysis and Multiresolution Methods*, T.-X. He, Ed., Marcel Dekker, Inc., New York, 2000.

[49] S. Goh, Q. Jiang, and T. Xia, "Construction of biorthogonal multiwavelets using the lifting scheme," *Appl. Comput. Harmonic Anal.*, vol. 9, no. 3, pp. 336-352, Nov., 2000.

[50] A. N. Netravali and B. G. Haskell, *Digital Pictures: Representation, Compression, and Standards* (2nd Edition), Plenum Press, New York, 1995.

[51] Z. Wang, B. F. Cockburn, D. G. Elliott, and W. Krzymien, "DSP-RAM: A Logic-Enhanced Memory Architecture for Communication Signal Processing," *1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada, pp. 475-478, Aug. 22-24, 1999.

[52] C. M. Brislawn, "Classification of nonexpansive symmetric extension transforms for multirate filter banks," *Appl. Comput. Harmon. Anal.*, 3:337–357, 1996.

[53] V. H. Allan *et al*, "Software Pipelining," *ACM Computing Surveys*, vol. 27, no. 3, September 1995.

APPENDIX A: MATLAB PROGRAMS

1. Daub-4 Lifting Algorithm

```
function Matrix=d4liftcore3(Matrix,length,Inv)

K=(3^0.5+1)/2^0.5*2^14;
A=-3^0.5*2^14;
B=3^0.5/4*2^14;
C=(3^0.5-2)/4*2^14;
Even=Matrix(:,1);
Odd=Matrix(:,2);

% step 1
if(Inv==0)
  Even=Even;
  Odd=Odd+Even*A;
else
  Even=Even*(1/K);
  Odd=Odd*K;
end

% step 2
if(Inv==0)   Odd=Odd;
  for i=1:length/2
    if(i-1)<=0
      Delay=0;
    else
      Delay=Odd(i-1);
    end
    Even(i)=Even(i)+B*Odd(i)+Delay*C;
  end
else
  Even=Even;
  for i=1:length/2
    if(i+1)>length/2
      Next=0;
    else
      Next=Even(i+1);
    end
    Odd(i)=Odd(i)-Next;
  end
end

% step 3
if Inv==0
  Even=Even;
  for i=1:length/2
    if(i+1)>length/2
      Next=0;
    else
      Next=Even(i+1);
    end
    Odd(i)=Odd(i)+Next;
  end
else
  Odd=Odd;
  for i=1:length/2
    if(i-1)<=0
      Delay=0;
    else
      Delay=Odd(i-1);
    end
    Even(i)=Even(i)-(Odd(i)*B+Delay*C);
  end
end

% step 4
if(Inv==0)
  Even=Even*K;
  Odd=Odd*(1/K);
else
  Even=Even;
  Odd=Odd-Even*A;
end

Matrix=[Even Odd];
```

2. 9/7 Lifting Algorithm

```
function Matrix=lift97core(Matrix,length,Inv)
a=-1.586134342;
b=-0.05298011854;
r=0.8829110762;
d=0.4435068522;
e=1.149604398;

Even=Matrix(:,1); Odd=Matrix(:,2);
% step 1
if (Inv==0)
  Even=Even;
  for i=1:length/2
    if (i-1)<=0
      Delay=0;
    else
      Delay=Even(i-1);
          end
    Odd(i)=Odd(i)+a*(Even(i)+Delay);
  end
else
  Even=Even*(1/e);
  Odd=Odd*e;
end

% step 2
if (Inv==0)
  for i=1:length/2
    if (i+1)>length/2
      Next=0;
    else
      Next=Odd(i+1);
    end
    Even(i)=Even(i)+b*(Odd(i)+Next);
  end
  Odd=Odd;
else
  for i=1:length/2
    if (i+1)>length/2
      Next=0;
    else
      Next=Odd(i+1);
    end
    Even(i)=Even(i)-d*(Odd(i)+Next);
  end
  Odd=Odd;
end

% step 3
if Inv==0
  Even=Even;
  for i=1:length/2
    if (i-1)<=0
      Delay=0;
    else
      Delay=Even(i-1);
    end
    Odd(i)=Odd(i)+r*(Even(i)+Delay);
  end
else
  Even=Even;
  for i=1:length/2
    if (i-1)<=0
      Delay=0;
    else
      Delay=Even(i-1);
    end
    Odd(i)=Odd(i)-r*(Even(i)+Delay);
  end
end

% step 4
if (Inv==0)
  for i=1:length/2
    if (i+1)>length/2
      Next=0;
    else
      Next=Odd(i+1);
    end
    Even(i)=Even(i)+d*(Odd(i)+Next);
  end
  Odd=Odd;
else
  for i=1:length/2
    if (i+1)>length/2
      Next=0;
    else
      Next=Odd(i+1);
    end
    Even(i)=Even(i)-b*(Odd(i)+Next);
  end
  Odd=Odd;
end

% step 5
if (Inv==0)
  Even=Even*e;
  Odd=Odd*(1/e);
else
  Even=Even;   for i=1:length/2
    if (i-1)<=0
      Delay=0;
    else
      Delay=Even(i-1);
    end
    Odd(i)=Odd(i)-a*(Even(i)+Delay);
  end
end

Matrix=[Even Odd];
```

APPENDIX B: VHDL PROGRAMS

1. 1-D 9/7 Recursive Architecture

```
-- a recursive lift arch. for multi-stage 1d 97 DWT
-- Hongyu Liao Aug20/02
--
-- Last update:
LIBRARY ieee;
library comp;
library io_util;
library std;

use ieee.std_logic_1164.all;
use comp.liftcomp.all;
use std.textio.all;
use io_util.tb_utilities.all;
use io_util.io_utils.all;

ENTITY rs97spdup IS
GENERIC (width: positive:= 16; fifo_len:positive:=1; SigLen:positive:=64;stage:positive:=3);
PORT(
        InSignal    : IN        std_logic_vector(width-1 downto 0);
        clock,reset,start,signalend: in std_logic;
        dwt_coeff_L, dwt_coeff_H: inOUT         std_logic_vector(width-1 downto 0);
        Done: inout std_logic);
END rs97spdup;

ARCHITECTURE mix OF rs97spdup IS
component rs97_pe
        GENERIC(width : positive; fifo_len:positive;stage:positive;rc:integer);
        PORT(
                E,O         : IN        std_logic_vector(width-1 downto 0);
                clock,clear,InEn: in std_logic;
                Set0:std_logic_vector(3 downto 0);
                step1en,step2en,step3en,step4en,step1sw,step2sw,step3sw,step4sw:std_logic_vector(stage-1 downto 0);
                dwt_coeff_L, dwt_coeff_H: OUT           std_logic_vector(width-1 downto 0));
END component;

component RS97_1dCtrl
        generic(stage: positive);
        PORT(
                clk         : IN        STD_LOGIC;
                reset       : IN        STD_LOGIC;
                start,signalend         : IN        STD_LOGIC;
                clear:inout std_logic;
                InEn: OUT               STD_LOGIC);
END component;

component rs97_in_sw
        GENERIC(width : positive; stage:positive);
        PORT(
                InSignal1,Insignal2   : IN        std_logic_vector(width-1 downto 0);
                clock,clear: in std_logic;
                even_en,odd_en,sw:std_logic_vector(stage-1 downto 0);
                Even,Odd: OUT       std_logic_vector(width-1 downto 0));
END component;

constant rc:integer:=1;
constant input_lshift:natural:=6;
constant scaler:natural:=14;

signal zeropad:std_logic_vector(input_lshift-1 downto 0):=(others=>'0');
signal clr,InputEn,NInEn,LoutSw_Sig:std_logic;
signal EvenInputEnOrg,OddInputEnOrg, InputSwOrg:std_logic_vector(stage-1 downto 0);
signal EvenInputEn,OddInputEn, InputSw:std_logic_vector(stage-1 downto 0);
```

```
signal
Step1EnOrg,Step2EnOrg,Step3EnOrg,Step4EnOrg,Step1SwOrg,Step2SwOrg,Step3SwOrg,Step4SwOrg:std_logic_vector(stage-1
downto 0);
signal Step1En,Step2En,Step3En,Step4En,Step1Sw,Step2Sw,Step3Sw,Step4Sw:std_logic_vector(stage-1 downto 0);
signal Even,Odd:std_logic_vector(width-1 downto 0);
signal Set0:std_logic_vector(3 downto 0);
signal clkx4Org,clkx4dl,S2enClk,clkx8Org,clkx8Dl,S3enOrg,S3enClk :std_logic;
signal input_sc,L_S2up,LCoeff:std_logic_vector(width-1 downto 0);
signal S1Mac1Set0,S2Mac1Set0,S3Mac1Set0,S1Mac3Set0,S2Mac3Set0,S3Mac3Set0:std_logic;
signal finish: std_logic;
signal S1end,S1endDL,S1endDL2,S2end,S2endDL,S3end,S1Last,S2Last,S3Last:std_logic;
signal Analyzing: std_logic_vector(stage-1 downto 0);
signal PEenable:std_logic;

for input_sw: rs97_in_sw use entity work.rs97_in_sw;
for pe: rs97_pe use entity work.rs97_pe;
for Ctrl: RS97_1dCtrl use entity work.rs97_1dctrl;
for S1sw1,s2clkdiv,s3clkdiv:t_ff use entity comp.t_ff;
for S1en2,S1en3,S1en4,S2EnRDelay,S2sw1:d_ff use entity comp.d_ff;
for S2en2,S2en3,S2en4,S3clkdivdl,S3clk,S3en2,S3en3,S3en4,Loutput:d_ff use entity comp.d_ff;
for S1mux3,S1mux5,S2en1odd,S2mux3,S2mux5,S3en1odd,S3sw1,S3mux3,S3mux5:bit_delay use entity comp.bit_delay;
for S1Set0_1,S1Set0_3,S2Set0_1,S2Set0_3,S3Set0_1,S3Set0_3:bit_delay use entity comp.bit_delay;
for S1Set0dl,S2Set0dl,S3Set0dl,S1enddelay,S1enddelay2,s2enddelay :bit_delay use entity comp.bit_delay;
for s1setend,s2setend,s3setend :t_ff use entity comp.t_ff;
BEGIN
        -- scale the input signal
        input_sc<=InSignal(width-input_lshift-1 downto 0) & zeropad;

        -------------------------------------------
        ---- input switches for DWT Processor ----
        -------------------------------------------
        input_sw: rs97_in_sw GENERIC map(width=>width, stage=>stage)
                          PORT map(
                                  InSignal1=>input_sc,Insignal2=>L_S2up,
                                  clock=>clock,clear=>clr,
                                  even_en=>EvenInputEn,odd_en=>OddInputEn,sw=>InputSw,
                                  Even=>Even,Odd=>Odd);

        -------------------------
        ---- DWT processor ------
        -------------------------
        pe: rs97_pe GENERIC map(width=>width, fifo_len=>fifo_len,stage=>stage,rc=>rc)
                          PORT map(
                                  E=>Even,O=>Odd,  clock=>clock,clear=>clr,InEn=>InputEn,Set0=>Set0,
                                  step1en=>Step1En,step2en=>Step2En,step3en=>Step3En,step4en=>Step4En,
                                  step1sw=>Step1Sw,step2sw=>Step2Sw,step3sw=>Step3Sw,step4sw=>Step4Sw,
                                  dwt_coeff_L=>LCoeff, dwt_coeff_H=>dwt_Coeff_H);

        -------------------------
        --- Controller ---------
        -------------------------
        Ctrl: RS97_1dCtrl  generic map(stage=>stage)
                          PORT map(
                                  clk=>clock, reset=>reset, start=>start,
                                  signalend=>signalend, clear=>clr, InEn=>InputEn);

        ------------------------------------------------
        -- Low frequency coefficients output switches --
        ------------------------------------------------
        LoutSw1: L_S2up<= LCoeff when LoutSw_Sig='0' else
                          (others=>'0');
        LoutSw2: dwt_Coeff_L<= LCoeff when LoutSw_Sig='1' else
                          (others=>'0');

        -------------------------------------------
        --- enable and switch control signals ----
        -------------------------------------------
        NInEn<=not InputEn;
```

```
------------------------------------------------------------------
-- enable and switch control signals for the stage 1 data flow --
------------------------------------------------------------------
EvenInputEnOrg(0)<=InputEn;
OddInputEnOrg(0)<='0';
S1sw1: T_ff port map(clk=>clock,NotEn=>NInEn ,q=>InputSwOrg(0)); -- 2i+1
S1mux1: Step1SwOrg(0)<=InputSwOrg(0); -- 2i+1
S1en2: D_FF port map(d=>Step1SwOrg(0),clk=>clock,clr=>NInEn ,q=>Step2EnOrg(0)); -- 2i+2
S1mux3:bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn ,d=>Step2EnOrg(0),q=>Step2SwOrg(0)); -- 2i+4
S1en3:D_FF port map(d=>Step2SwOrg(0),clk=>clock,clr=>NInEn ,q=>Step3EnOrg(0)); -- 2i+5
S1mux4: Step3SwOrg(0)<=Step3EnOrg(0);
S1en4:D_FF port map(d=>Step3SwOrg(0),clk=>clock,clr=>NInEn ,q=>Step4EnOrg(0)); -- 2i+6
S1mux5:   bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn ,d=>Step4EnOrg(0),q=>Step4SwOrg(0)); -- 2i+8

S1Set0_1:bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn ,d=>signalend,q=>S1Mac1Set0); -- signalend+2
S1Set0_3:bit_delay generic map(del=>4)
                    port map(clk=>clock,clr=>NInEn,d=>S1Mac1Set0,q=>S1Mac3Set0); -- signalend+6

-- speedup
S1Set0dl:bit_delay generic map(del=>3)
                    port map(clk=>clock,clr=>NInEn ,d=>S1Mac3Set0,q=>S1Last); -- signalend+9
s1setend:T_FF port map(clk=>S1Last,NotEn=>NInEn ,q=>S1end);
s1enddelay:bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn ,d=>S1end,q=>S1endDL); --signalend+11
s1enddelay2:bit_delay generic map(del=>3)
                    port map(clk=>clock,clr=>NInEn ,d=>S1endDL,q=>S1endDL2); --signalend+14


Analyzing(0)<=(not S1end) and InputEn;
EvenInputEn(0)<=EvenInputEnOrg(0) and Analyzing(0);
OddInputEn(0)<=OddInputEnOrg(0) and Analyzing(0);
InputSw(0)<=InputSwOrg(0) and Analyzing(0);
Step1Sw(0)<=Step1SwOrg(0) and Analyzing(0);
Step2En(0)<=Step2EnOrg(0) and Analyzing(0);
Step2Sw(0)<=Step2SwOrg(0) and Analyzing(0);
Step3En(0)<=Step3EnOrg(0) and Analyzing(0);
Step3Sw(0)<=Step3SwOrg(0) and Analyzing(0);
Step4En(0)<=Step4EnOrg(0) and Analyzing(0);
Step4Sw(0)<=Step4SwOrg(0) and Analyzing(0);

--------------------------------------------------------------------------
-- enable and control signals for stage 2 data flow, Delay for Stage1=9 --
--------------------------------------------------------------------------
-- first, generate 4xClk signal for stage 2 control signals
s2clkdiv:T_FF port map(clk=>Step4SwOrg(0),NotEn=>NInEn ,q=>clkx4Org); --4i+8~9
-- then generate a clock signal with duty=25%, f=4xClk
S2EnRDelay:d_ff port map(d=>clkx4Org,clk=>clock,clr=>NInEn,q=>clkx4DL); --4i+9~10
S2enClk<=clkx4DL and clkx4Org; --4i+9

EvenInputEnOrg(1)<=S2enClk;
S2en1odd:bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn,d=>S2enClk,q=>OddInputEnOrg(1)); -- 4i+11
S2sw1:D_FF port map(d=>OddInputEnOrg(1),clk=>clock,clr=>NInEn ,q=>InputSwOrg(1)); -- 4i+12 (mod(12/2)=0)
S2mux1:Step1SwOrg(1)<=InputSwOrg(1);
S2en2: D_FF port map(d=>Step1SwOrg(1),clk=>clock,clr=>NInEn ,q=>Step2EnOrg(1)); -- 4i+13
S2mux3:bit_delay generic map(del=>4)
                    port map(clk=>clock,clr=>NInEn ,d=>Step2EnOrg(1),q=>Step2SwOrg(1)); -- 4i+17
S2en3:D_FF port map(d=>Step2SwOrg(1),clk=>clock,clr=>NInEn ,q=>Step3EnOrg(1)); -- 4i+18
S2mux4:Step3SwOrg(1)<=Step3EnOrg(1);
S2en4:D_FF port map(d=>Step3SwOrg(1),clk=>clock,clr=>NInEn ,q=>Step4EnOrg(1)); -- 4i+19
S2mux5:bit_delay generic map(del=>4)
                    port map(clk=>clock,clr=>NInEn ,d=>Step4EnOrg(1),q=>Step4SwOrg(1)); -- 4i+23

S2Set0_1:bit_delay generic map(del=>2)
                    port map(clk=>clock,clr=>NInEn ,d=>S1Last,q=>S2Mac1Set0); --signalend+11
S2Set0_3:bit_delay generic map(del=>4)
                    port map(clk=>clock,clr=>NInEn ,d=>S2Mac1Set0,q=>S2Mac3Set0); --signalend+15
```

```
            -- speedup
            S2Set0dl:bit_delay generic map(del=>3)
                                    port map(clk=>clock,clr=>NInEn ,d=>S2Mac3Set0,q=>S2Last); -- signalend+18
            s2setend:T_FF port map(clk=>S2Last,NotEn=>NInEn ,q=>S2end);
            s2enddelay:bit_delay generic map(del=>2)
                                    port map(clk=>clock,clr=>NInEn ,d=>S2end,q=>S2endDL); --signalend+20

            Analyzing(1)<=InputEn and S1endDL and (not S2end);
            EvenInputEn(1)<=(EvenInputEnOrg(1) and Analyzing(0));
            OddInputEn(1)<=(OddInputEnOrg(1) and Analyzing(0));
            InputSw(1)<=(InputSwOrg(1) and (not S1endDL) and InputEn) or ((not InputSwOrg(0)) and Analyzing(1));
            Step1Sw(1)<=(Step1SwOrg(1) and (not S1endDL) and InputEn) or ((not Step1SwOrg(0)) and Analyzing(1)).
            Step2En(1)<=(Step2EnOrg(1) and (not S1endDL) and InputEn) or ((not Step2EnOrg(0)) and Analyzing(1));
            Step2Sw(1)<=(Step2SwOrg(1) and (not S1endDL) and InputEn) or ((not Step2SwOrg(0)) and Analyzing(1));
            Step3En(1)<=(Step3EnOrg(1) and (not S1endDL) and InputEn) or ((not Step3EnOrg(0)) and Analyzing(1));
            Step3Sw(1)<=(Step3SwOrg(1) and (not S1endDL) and InputEn) or ((not Step3SwOrg(0)) and Analyzing(1));
            Step4En(1)<=(Step4EnOrg(1) and (not S1endDL) and InputEn) or ((not Step4EnOrg(0)) and Analyzing(1));
            Step4Sw(1)<=(Step4SwOrg(1) and (not S1endDL) and InputEn) or ((not Step4SwOrg(0)) and Analyzing(1));

            ----------------------------------------------
            -- enable and control signals for stage 3 --
            ----------------------------------------------
            -- generate 8xclk signal
            s3clkdiv:T_FF port map(clk=>Step4SwOrg(1),NotEn=>NInEn ,q=>clkx8Org); -- 8i+23~26
            -- duty=12.5% clock
            S3clkdivdl:d_ff port map(d=>clkx8Org,clk=>clock,clr=>NInEn,q=>clkx8dl); --8i+26~29
            S3enOrg<= (not clkx8dl) and clkx8Org; --8i+23
            S3clk:d_ff port map(d=>S3enOrg,clk=>clock,clr=>NInEn,q=>S3enClk); --8i+24

            EvenInputEnOrg(2)<=S3enClk;
            S3en1odd:bit_delay generic map(del=>4)
                                    port map(clk=>clock,clr=>NInEn,d=>S3enClk,q=>OddInputEnOrg(2)); -- 8i+28
            S3sw1:bit_delay generic map(del=>2)
                                    port map(clk=>clock,clr=>NInEn ,d=>OddInputEnOrg(2),q=>InputSwOrg(2)); -- 8i+30
(mod((30-12)/4!=0)
            S3mux1:Step1SwOrg(2)<=InputSwOrg(2);
            S3en2: D_FF port map(d=>Step1SwOrg(2),clk=>clock,clr=>NInEn ,q=>Step2EnOrg(2)); -- 8i+31
            S3mux3:bit_delay generic map(del=>8)
                                    port map(clk=>clock,clr=>NInEn ,d=>Step2EnOrg(2),q=>Step2SwOrg(2)); -- 8i+39
            S3en3:D_FF port map(d=>Step2SwOrg(2),clk=>clock,clr=>NInEn ,q=>Step3EnOrg(2)); -- 8i+40
            S3mux4:Step3SwOrg(2)<=Step3EnOrg(2);
            S3en4:D_FF port map(d=>Step3SwOrg(2),clk=>clock,clr=>NInEn ,q=>Step4EnOrg(2)); -- 8i+41
            S3mux5:bit_delay generic map(del=>8)
                                    port map(clk=>clock,clr=>NInEn ,d=>Step4EnOrg(2),q=>Step4SwOrg(2)); -- 8i+49

            S3Set0_1:bit_delay generic map(del=>2)
                                    port map(clk=>clock,clr=>NInEn ,d=>S2Last,q=>S3Mac1Set0); --signalend+20
            S3Set0_3:bit_delay generic map(del=>4)
                                    port map(clk=>clock,clr=>NInEn ,d=>S3Mac1Set0,q=>S3Mac3Set0); --signalend+24

            -- speedup
            S3Set0dl:bit_delay generic map(del=>3)
                                    port map(clk=>clock,clr=>NInEn ,d=>S3Mac3Set0,q=>S3Last); -- signalend+
            s3setend:T_FF port map(clk=>S3Last,NotEn=>NInEn ,q=>S3end);

            Analyzing(2)<=InputEn and S1endDL2 and (not S3end);
            EvenInputEn(2)<=(EvenInputEnOrg(2) and (not S1endDL2) and InputEn) or (Step3SwOrg(1) and Analyzing(2));
            OddInputEn(2)<=(OddInputEnOrg(2) and (not S1endDL2) and InputEn) or (Step1SwOrg(1) and Analyzing(2));
            InputSw(2)<=(InputSwOrg(2) and (not S1endDL) and InputEn) or (((Step2SwOrg(1) and (not S2endDL)) or
(InputSwOrg(0) and S2endDL)) and Analyzing(2));
            Step1Sw(2)<=(Step1SwOrg(2) and (not S1endDL) and InputEn) or (((Step2SwOrg(1) and (not S2endDL)) or
(Step1SwOrg(0) and S2endDL)) and Analyzing(2));
            Step2En(2)<=(Step2EnOrg(2) and (not S1endDL) and InputEn) or (((Step3SwOrg(1) and (not S2endDL)) or
(Step2EnOrg(0) and S2endDL)) and Analyzing(2));
            Step2Sw(2)<=(Step2SwOrg(2) and (not S1endDL) and InputEn) or (((Step3EnOrg(1) and (not S2endDL)) or
(Step2SwOrg(0) and S2endDL)) and Analyzing(2));
            Step3En(2)<=(Step3EnOrg(2) and (not S1endDL) and InputEn) or (((Step4SwOrg(1) and (not S2endDL)) or
(Step3EnOrg(0) and S2endDL)) and Analyzing(2));
            Step3Sw(2)<=(Step3SwOrg(2) and (not S1endDL) and InputEn) or (((Step4EnOrg(1) and (not S2endDL)) or
(Step3SwOrg(0) and S2endDL)) and Analyzing(2));
          Step4En(2)<=(Step4EnOrg(2) and (not S1endDL) and InputEn) or (((Step1SwOrg(1) and (not S2endDL)) or
(Step4EnOrg(0) and S2endDL)) and Analyzing(2));
          Step4Sw(2)<=(Step4SwOrg(2) and (not S1endDL) and InputEn) or (((Step1SwOrg(1) and (not S2endDL)) or
(Step4SwOrg(0) and S2endDL)) and Analyzing(2));
          Set0(0)<=S1Mac1Set0 or S2Mac1Set0 or S3Mac1Set0;
          Set0(2)<=S1Mac3Set0 or S2Mac3Set0 or S3Mac3Set0;

          Loutput: D_FF port map(d=>Step4SwOrg(2),clk=>clock,clr=>NInEn ,q=>LoutSw_Sig); -- 8i+50
          Step1En<=InputSw;

          --PEenable<=InputEn and (Analyzing(0) or Analyzing(1) or Analyzing(2));

          -----------------
          --finish signal--
          -----------------
          -- it is done 8 cycles after the last LoutSw_Sig pulse
          process(clock,LoutSw_Sig)
          variable finishcnt: natural:=0;
          begin
                    if InputEn='1' then
                              if rising_edge(LoutSw_Sig) then
                                        finishcnt:=finishcnt+1;
                              end if;
                    else
                              finishcnt:=0;
                    end if;

                    if finishcnt>SigLen/(2**stage) then
                              finish<='1';
                    else
                              finish<='0';
                    end if;
          end process;


          -----------------------------------------
          ----- Writing results to data files -----
          -----------------------------------------
          output: process(clock)
          file testoutputS3L:text open write_mode is "S3outputL.txt";
          file testoutputS3h:text open write_mode is "S3outputH.txt";
          file testoutputS2h:text open write_mode is "S2outputH.txt";
          file testoutputS1h:text open write_mode is "S1outputH.txt";
          file testoutputL: text open write_mode is "outputL.txt";
          variable OutLS1,OutHS1,OutLS2,OutHS2,OutLS3,OutHS3,OutL: Line;
          begin

          if rising_edge(clock) and finish='0' then
                    -- 1H
                    if Step4Sw(0)='1' then
                              write(OutHS1,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS1h,OutHS1);
                    elsif Step4Sw(1)='1' then
                              write(OutHS2,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS2h,OutHS2);
                    -- 3H
                    elsif Step4Sw(2)='1' then
                              write(OutHS3,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS3h,OutHS3);
                    end if;
                    -- 3L
                    if LoutSw_Sig='1' then
                              write(OutLS3,slv_to_bv(LCoeff),right,8,decimal,false);
                                        writeline(testoutputS3L,OutLS3);
                    elsif OddInputEn(2 downto 1)/="00" or EvenInputEn(2 downto 1)/="00" then
                              write(OutL,slv_to_bv(LCoeff),right,8,decimal,false);
                                        writeline(testoutputL,OutL);
                    end if;
          end if;
```

```
		end process output;
END mix;



-- a controller for 1-D Recursive 97 DWT architecture
--created Aug13/02
-- Last update
library ieee;

use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;


ENTITY RS97_1dCtrl IS
	generic(stage: positive:=3);
	PORT(
		clk		: IN		STD_LOGIC;
		reset		: IN		STD_LOGIC;
		start, signalend: IN	STD_LOGIC;
		clear:inout std_logic;
		InEn: OUT		STD_LOGIC);
END RS97_1dCtrl;

ARCHITECTURE beh OF RS97_1dCtrl IS
	TYPE STATE_TYPE IS (idle,analyzing,endofstage,analyzing2,Finish);
	SIGNAL state: STATE_TYPE;
	signal InEn_sig:std_logic;
BEGIN
	PROCESS (clk)
	variable Count,cntset0r,cntset0c,LStage,CntLastRow: natural:=0;
	variable counting: std_logic;
	BEGIN
		-- Number of stage
		Lstage:=stage;
		IF reset = '1' THEN
			state <=idle;
		ELSIF clk'EVENT AND clk = '1' THEN
			CASE state IS
				WHEN idle =>
					-- reset everything
					counting:='0';

					IF start='1' THEN
						state <= analyzing;
					END IF;
				WHEN analyzing =>
					if signalend='1' then
						state<=endofstage;
					end if;

				when endofstage =>
					counting:='1';
					if count=Lstage*5+20 then
						state <= Finish;
					else
						state<=analyzing2;
					END IF;

				when analyzing2 =>
					if count=10 then
						state<=endofstage;
					end if;

				when Finish =>
					state<=Idle;
			end case;

			if counting='1' then
```

```
                                        count:=count+1;
                                end if;
                        END IF;
                END PROCESS;

                WITH state SELECT
                        InEn_sig <= '0' WHEN        Idle,
                                    '1'     WHEN    others;
                WITH state SELECT
                        clear     <= '1' WHEN       Idle,
                                    '0'     WHEN    others;

                InEn<=InEn_sig;
END beh;


-- a process element for implementing 9/7 DWT
-- Hongyu Liao Jul12/2001
-- the module should be cleared before each data frame
-- Last update: Aug8/02

LIBRARY ieee;
library comp;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use comp.liftcomp.all;

ENTITY rs97_pe IS
        GENERIC(width : positive:= 16; fifo_len:positive:=1;stage:positive:=3;rc:integer:=1);
        PORT(
                E,O           : IN          std_logic_vector(width-1 downto 0);
                clock,clear,InEn: in std_logic;
                Set0: std_logic_vector(3 downto 0);
                step1en,step2en,step3en,step4en,step1sw,step2sw,step3sw,step4sw:std_logic_vector(stage-1 downto 0);
                dwt_coeff_L, dwt_coeff_H: OUT                  std_logic_vector(width-1 downto 0));
END rs97_pe;

ARCHITECTURE str OF rs97_pe IS
for  delay2,delay3,delay4,delay5:delay_rs use entity comp.delay_rs;
for mac1,mac2,mac3,mac4: mac_sym use entity comp.mac_sym;
for mult1,mult2: mult use entity comp.mult;
for reg1,reg2,reg3: reg use entity comp.reg;

signal alpha,beta,gama,delta,zeta,one,zeta_inv:std_logic_vector(width-1 downto 0);
signal e1,e2,e3,e4,e5,o1,o2,o3,
        o4,o5,s1,s2,s3,s:std_logic_vector(width-1 downto 0);
signal NInEn,S1en,S2sw2,S3sw2,set0dl:std_logic;
signal lowout,highout:std_logic_vector(width*2-1 downto 0);
signal delayout1,delayout2,delayout3,delayout4,delayout5:std_logic_vector(width*stage-1 downto 0);
signal o1_delay,r1_in,r2_in,r3_in,r4_in:std_logic_vector(width-1 downto 0);
signal NoCtrlSig:std_logic_vector(stage-1 downto 0);

BEGIN
        alpha<="1001101001111101"; -- right shift the original parameters for 14bits
        beta<="1111110010011100";
        gama<="0011100010000010";
        delta<="0001110001100010";
        zeta<="0100100110010011";
        one<="0100000000000000";
        zeta_inv<="0011011110101100";
        NoCtrlSig<=(others=>'0');

        e1<=E;
        o1<=O;

        -- enable signal for the stage1 data flow
        -- S1en
-- step1
        --delay1: delay_rs generic map(width=>width, len=>fifo_len, stage=>stage,
               rc=>rc)

        -- port map(clk=>clock,clr=>clear,enable=>step1en,input=>o1,output=>delayout1);

        --mux1: for index in 1 to stage generate

        --begin
        --
        o1_delay<= delayout1(width*index-1 downto width*(index-1)) when step1sw(index-
        1)='1' else

        --
                        (others=>'0') when step1sw=NoCtrlSig else
        --                            (others=>'Z');
        --end generate mux1;

        delay2: delay_rs generic map(width=>width,len=>fifo_len,stage=>stage,rc=>rc)
                port map(clk=>clock,clr=>clear,enable=>step1en,input=>e1,output=>delayout2);

        mux2: for index in 1 to stage generate
        begin
                r1_in<= delayout2(width*index-1 downto width*(index-1)) when step1sw(index-1)='1' else
                        (others=>'0') when step1sw=NoCtrlSig else
                        (others=>'Z');
        end generate mux2;

        mac1:mac_sym generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,Set0=>Set0(0),acc=>o1,in1=>e1,in2=>r1_in,amp=>alpha,
                        output=>o2);

        reg1:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>r1_in,output=>e2);

-- step2

        delay3: delay_rs generic map(width=>width,len=>fifo_len,stage=>stage,rc=>rc)
                port map(clk=>clock,clr=>clear,enable=>step2en,input=>o2,output=>delayout3);

        mux3: for index in 1 to stage generate
        begin
                r2_in<= delayout3(width*index-1 downto width*(index-1)) when step2sw(index-1)='1' else
                                (others=>'0') when step2sw=NoCtrlSig else
                                (others=>'Z');
        end generate mux3;

        mac2:mac_sym generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,Set0=>Set0(1),acc=>e2,in1=>o2,in2=>r2_in,amp=>beta,
                output=>e3);

        reg2:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>r2_in,output=>o3);

-- step3
        delay4: delay_rs generic map(width=>width,len=>fifo_len,stage=>stage,rc=>rc)
                port map(clk=>clock,clr=>clear,enable=>step3en,input=>e3,output=>delayout4);

        mux4: for index in 1 to stage generate
        begin
                r3_in<= delayout4(width*index-1 downto width*(index-1)) when step3sw(index-1)='1' else
                                (others=>'0') when step3sw=NoCtrlSig else
                                (others=>'Z');
        end generate mux4;

        mac3:mac_sym generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,Set0=>Set0(2),acc=>o3,in1=>e3,in2=>r3_in,amp=>gama,
                        output=>o4);

        reg3:reg generic map(width=>width)
```

```
                        port map(clk=>clock,clr=>clear,en=>InEn,input=>r3_in,output=>e4);

-- step4
        delay5: delay_rs generic map(width=>width,len=>fifo_len,stage=>stage,rc=>rc)
                        port map(clk=>clock,clr=>clear,enable=>step4en,input=>o4,output=>delayout5);

        mux5: for index in 1 to stage generate
        begin
                        o5<= delayout5(width*index-1 downto width*(index-1)) when step4sw(index-1)='1' else
                                        (others=>'0') when step4sw=NoCtrlSig else
                                        (others=>'Z');
        end generate mux5;

        mac4:mac_sym generic map(width=>width,add_sc=>14,res_sc=>14)
                        port map(clock=> clock,clear=>clear,Set0=>Set0(3),acc=>e4,in1=>o4,in2=>o5,amp=>delta,
                        output=>e5);

        mult1:mult generic map(width=>width)
                        port map(a=>e5,b=>zeta,p=>lowout);

        mult2:mult generic map(width=>width)
                        port map(a=>o5,b=>zeta_inv,p=>highout);

        scale1:dwt_coeff_L<=lowout(width*2-3 downto width-2)when lowout(width-3)='0' else
                                        lowout(width*2-3 downto width-2)+1;
        scale2:dwt_coeff_H<=highout(width*2-3 downto width-2) when highout(width-3)='0' else
                                        highout(width*2-3 downto width-2)+1;
END str;


-- input switch for 1-D 97 lifting filter
-- Hongyu Liao Aug12/2002
--
-- Last update: /2002
LIBRARY ieee;
library comp;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use comp.liftcomp.all;

ENTITY rs97_in_sw IS
        GENERIC(width : positive:= 16; stage:positive:=3);
        PORT(
                InSignal1,Insignal2     : IN            std_logic_vector(width-1 downto 0);
                clock,clear: in std_logic;
                even_en,odd_en,sw:std_logic_vector(stage-1 downto 0);
                Even,Odd: OUT           std_logic_vector(width-1 downto 0));
END rs97_in_sw;

ARCHITECTURE str OF rs97_in_sw IS
for reg1 : reg use entity comp.reg;
for even_regs,odd_regs:delay_rs use entity comp.delay_rs;

signal odd_reg_out,even_reg_out:std_logic_vector(width*stage-1 downto 0);
signal NoCtrlSig:std_logic_vector(stage-1 downto 0);
BEGIN
        NoCtrlSig<=(others=>'0');
        -- synchronize the even and odd samples
        reg1:reg generic map(width=>width)
            port map(clk=>clock,clr=>clear,en=>even_en(0),input=>Insignal1,output=>even_reg_out(width-1 downto 0));

        odd_reg_out(width-1 downto 0)<=Insignal1;

        -- for stage 2 and up
        even_regs: delay_rs generic map(width=>width, len=>1, stage=>stage-1, rc=>1)
                port map(clk=>clock,clr=>clear,enable=>even_en(stage-1 downto
                1),input=>Insignal2,output=>even_reg_out(width*stage-1 downto width));

        odd_regs: delay_rs generic map(width=>width, len=>1, stage=>stage-1, rc=>1)
```

```
                port map(clk=>clock,clr=>clear,enable=>odd_en(stage-1 downto
                1),input=>Insignal2,output=>odd_reg_out(width*stage-1 downto width));

        -- there is 1 cycle delay between the switch signals and the enable signals
        -- for the data flows of stage 2 and up, which allows the data flows to get to
        -- the next step
        mux_even: for i in 1 to stage generate
        begin
                Even<=even_reg_out(width*i-1 downto width*(i-1)) when sw(i-1)='1' else
                        (others=>'0') when sw=NoCtrlSig else
                        (others=>'Z');
        end generate mux_even;

        mux_odd: for i in 1 to stage generate
        begin
                Odd<=odd_reg_out(width*i-1 downto width*(i-1)) when sw(i-1)='1' else
                        (others=>'0') when sw=NoCtrlSig else
                        (others=>'Z');
        end generate mux_odd;

END str;
```

2. 1-D Daub-4 Recursive Architecture

```
-- a recursive lift arch. for multi-stage D-4
-- Hongyu Liao feb17/02
-- Last update
LIBRARY ieee;
library my_lib;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use my_lib.liftcomp.all;

ENTITY rs_1d IS
        GENERIC(width : positive:= 16; fifo_len:positive:=1;SigLen:positive:=32);
        PORT(
                InSignal   : IN        std_logic_vector(width-1 downto 0);
                clock,reset,start: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT          std_logic_vector(width-1 downto 0);
                Done: out std_logic);
END rs_1d;

ARCHITECTURE beh OF rs_1d IS
component d4lift_rs_dp
        GENERIC(width : positive; fifo_len:positive);
        PORT(
                InSignal1,Insignal2    : IN        std_logic_vector(width-1 downto 0);
                clock,clear,InEn,Set0,S2en1,S3en1,S2en2,S3en2,S2sw1,S3sw1,S3en1b: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT          std_logic_vector(width-1 downto 0));
END component;

component RSControl
        generic(SigLen: positive);
        PORT(
                clk        : IN        STD_LOGIC;
                reset      : IN        STD_LOGIC;
                start      : IN        STD_LOGIC;
                InEn,clear,Set0,S2en1,S3en1,S2en2,S3en2,S2Sw1,S3sw1,S3en1b,Done: OUTSTD_LOGIC);
END component;

signal clr,InSel,S2en1,S3en1,S2en2,S3en2,InputEn,Setzero,ClkDivEn,Lout:std_logic;
signal S2sw1,S3sw1,S3en1b:std_logic;
signal zeropad:std_logic_vector(5 downto 0):=(others=>'0');
signal input_sc,Lsig,Lfeedbk1,Lfeedbk2:std_logic_vector(width-1 downto 0);
BEGIN
```

```
        input_sc<=InSignal(width-7 downto 0) & zeropad;

        core: d4lift_rs_dp GENERIC map(width=>width,fifo_len=>fifo_len)
                PORT map(
                InSignal1=>input_sc,InSignal2=>Lfeedbk1,clock=>clock, clear=>clr,InEn=>InputEn,Set0=>Setzero,
                S2en1=>S2en1,S3en1=>S3en1,S2en2=>S2en2,S3en2=>S3en2,S2sw1=>S2sw1,
                S3sw1=>S3sw1,S3en1b=>S3en1b,dwt_coeff_L=>Lsig, dwt_coeff_H=>dwt_coeff_H);

        controller:RSControl generic map(SigLen=>SigLen)
                PORT map(
                clk=>clock,reset=>reset,start=>start, InEn=>InputEn,clear=>clr,set0=>Setzero,
                S2en1=>S2en1,S3en1=>S3en1,S2en2=>S2en2,S3en2=>S3en2,S2sw1=>S2sw1,
                S3sw1=>S3sw1,S3en1b=>S3en1b,Done=>Done);

        LSel:d_ff port map(d=>S3en2,clk=>clock,clr=>reset,q=>Lout);
        Lfeedbk1<=LSig when Lout='0' else
                (others=>'0');
        dwt_coeff_L<=LSig when Lout='1' else
                (others=>'0');
END beh;


-- a recursive lift arch. datapath for D-4
-- Hongyu Liao feb12/2001
-- the module should be cleared before use
-- Change port map
-- Last update: Apr09/2002
LIBRARY ieee;
library my_lib;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use my_lib.liftcomp.all;

ENTITY d4lift_rs_dp IS
        GENERIC(width : positive:= 16; fifo_len:positive:=1);
        PORT(
                InSignal1,Insignal2     : IN            std_logic_vector(width-1 downto 0);
                clock,clear,InEn,Set0,S2en1,S3en1,S2en2,S3en2,S2sw1,S3sw1,S3en1b: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT                   std_logic_vector(width-1 downto 0));
END d4lift_rs_dp;

ARCHITECTURE beh OF d4lift_rs_dp IS

signal alpha,beta,gama,one,K,K_inv,zero:std_logic_vector(width-1 downto 0);
signal e1_in,e1_s1,e1_s2,e1_s3,e1,e2,e3,e4_in,e4,e5,o1,o1_s1,o1_s2,o1_s3,o2,o3,
        o4,o5_in,o5,delay_in,s1,s2,s3,s1up,s:std_logic_vector(width-1 downto 0);
signal NInEn,S1en,S2sw2,S3sw2,clrM3:std_logic;
signal lowout,highout:std_logic_vector(width*2-1 downto 0);
BEGIN
        alpha<="1001000100100111"; -- right shift the original parameters for 14bits
        beta<="0001101110110110";
        gama<="1111101110110111";
        one<="0100000000000000";
        K<="0111101110100011";
        K_inv<="0010000100100000";
        zero<=(others=>'0');

        o1_s1<=InSignal1;
        o1_s2<=InSignal2;
        NInEn<=not InEn;
        -- enable signal for the stage1 data flow
        -- S1en
        s1sel:T_FF port map(clk=>clock,clr=>NInEn ,q=>S1en);

        -- synchronize the even and odd samples
        reg1:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>Insignal1,output=>e1_s1);
        reg1_2:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>S2en1,input=>Insignal2,output=>e1_s2);
        reg1_3:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>S3en1,input=>Insignal2,output=>e1_s3);
        reg1_4:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>S3en1b,input=>Insignal2,output=>o1_s3);

        -- there is 1 cycle delay between the switch signals and the enable signals
        -- for the data flows of stage 2 and up, which allows the data flows to get to
        -- the next step
        einmux: e1<= e1_s1 when S1en='1' else
                        e1_s2 when S2sw1='1' else
                        e1_s3 when S3sw1='1' else
                        zero;
        oinmux: o1<= o1_s1 when S1en='1' else
                        o1_s2 when S2sw1='1' else
                        o1_s3 when S3sw1='1' else
                        zero;

        mac1:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>o1,mul=>e1,amp=>alpha, output=>o2);

        reg2:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>e1,output=>e2);

        mac2:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>e2,mul=>o2,amp=>beta, output=>e3);

        reg3:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>o2,output=>delay_in);

        -- delay unit, 3 delay registers
        d1:delay generic map(width=>width,len=>fifo_len)
                port map(clk=>clock,clr=>clear,en=>S1en,input=>delay_in,output=>s1);
        d2:delay generic map(width=>width,len=>fifo_len)
                port map(clk=>clock,clr=>clear,en=>S2en2,input=>delay_in,output=>s2);
        d3:delay generic map(width=>width,len=>fifo_len)
                port map(clk=>clock,clr=>clear,en=>S3en2,input=>delay_in,output=>s3);

        delaymux: o3<= s1 when S1en='1' else
                        s2 when S2en2='1' else
                        s3 when S3en2='1' else
                        zero;

        -- this mac can be cleared for the last high frequency DWT coefficient
        mac3:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clrM3,acc=>e3,mul=>o3,amp=>gama, output=>e4);
        clrM3<=clear or set0;

        reg4:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>o3,output=>o4);

        mac4:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>o4,mul=>e4,amp=>one, output=>o5);

        mult1:mult
                generic map(width=>width)
                port map(a=>e4,b=>K,p=>lowout);

        mult2:mult
                generic map(width=>width)
                port map(a=>o5,b=>K_inv,p=>highout);

        scale1:dwt_coeff_L<=lowout(width*2-3 downto width-2)when lowout(width-3)='0' else
                                lowout(width*2-3 downto width-2)+1;
        scale2:dwt_coeff_H<=highout(width*2-3 downto width-2) when highout(width-3)='0' else
                                highout(width*2-3 downto width-2)+1;

END beh;


-- a controller for 1-D Recursive
```

```
--created feb13/02
-- Last update
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

ENTITY RSControl IS
        generic(SigLen: positive:=8);
        PORT(
                clk         : IN        STD_LOGIC;
                reset       : IN        STD_LOGIC;
                start       : IN        STD_LOGIC;
                InEn,clear,Set0,S2en1,S3en1,S2en2,S3en2,S2sw1,S3sw1,S3en1b,Done: OUTSTD_LOGIC);
END RSControl;

ARCHITECTURE a OF RSControl IS
        TYPE STATE_TYPE IS (idle,analyzing,Stg1End,Stg2End,Stg3End,Finish);
        SIGNAL state: STATE_TYPE;
        signal InEn_sig:std_logic;
BEGIN
        PROCESS (clk)
        variable FifoDep: positive:=SigLen;
        variable Count,cntD1,cntD2,cntD3,cntD4,cntS2a,cntS3a,cntS3a2,cntS3sw,cntS2b,cntS3b,cntS2sw: natural:=0;
        variable counting: std_logic;
        BEGIN
                IF reset = '1' THEN
                        state <=idle;
                ELSIF clk'EVENT AND clk = '1' THEN
                        CASE state IS
                                WHEN idle =>
                                        counting:='0';
                                        count:=0;
                                        cntS2a:=0;
                                        cntS2sw:=0;
                                        cntS3a:=0;
                                        cntS3sw:=0;
                                        cntS3a2:=0;
                                        cntS2b:=0;
                                        cntS3b:=0;
                                        S2en1<='0';
                                        S3en1<='0';
                                        S2en2<='0';
                                        S3en2<='0';
                                        set0<='0';
                                        IF start='1' THEN
                                        state <= analyzing;
                                        counting:='1';
                                        END IF;

                                WHEN analyzing =>
                                        IF count=SigLen+3 THEN
                                                state <= stg1End;
                                                Set0<='1';
                                        END IF;

                                WHEN Stg1End =>
                                        Set0<='0';
                                        if count=SigLen+8 then
                                                state<=Stg2End;
                                                Set0<='1';
                                        end if;

                                WHEN Stg2End =>
                                        Set0<='0';
                                        if count=SigLen+18 then
                                                state <= Stg3End;
                                                Set0<='1';
                                        END IF;
```

```
                    WHEN Stg3End =>
                              Set0<='0';
                              if count=SigLen+20 then
                                        state <= Finish;
                              END IF;
                    when Finish =>
                                        state<=Idle;
          end case;
          if counting='1' then
                    count:=count+1;
                    if count >= 2 then
                              cntS2a:=cntS2a+1;
                              if cntS2a = 4 then
                                        S2en1<='1';
                                        cntS2a:=0;
                              else
                                        S2en1<='0';
                              end if;
                    end if;
                    if count >= 4 then
                              cntS2sw:=cntS2sw+1;
                              if cntS2sw = 4 then
                                        S2sw1<='1';
                                        cntS2sw:=0;
                              else
                                        S2sw1<='0';
                              end if;
                    end if;
                    if count>=6 then
                              cntS2b:=cntS2b+1;
                              if cntS2b = 4 then
                                        S2en2<='1';
                                        cntS2b:=0;
                              else
                                        S2en2<='0';
                              end if;
                    end if;
                    if count>=3 then
                              cntS3a:=cntS3a+1;
                              if cntS3a = 8 then
                                        S3en1<='1';
                                        cntS3a:=0;
                              else
                                        S3en1<='0';
                              end if;
                    end if;
                    if count>=7 then
                              cntS3a2:=cntS3a2+1;
                              if cntS3a2 = 8 then
                                        S3en1b<='1';
                                        cntS3a2:=0;
                              else
                                        S3en1b<='0';
                              end if;
                    end if;
                    if count>=10 then
                              cntS3sw:=cntS3sw+1;
                              if cntS3sw = 8 then
                                        S3sw1<='1';
                                        cntS3sw:=0;
                              else
                                        S3sw1<='0';
                              end if;
                    end if;
                    if count>= 12 then
                              cntS3b:=cntS3b+1;
                              if cntS3b = 8 then
                                        S3en2<='1';
                                        cntS3b:=0;
                              else
```

```
                                                        S3en2<='0';
                                                end if;
                                        end if;
                                end if;
                        END IF;
                END PROCESS;

                WITH state SELECT
                        InEn_sig <= '0' WHEN        Idle,
                                    '1'     WHEN    others;
                WITH state SELECT
                        clear    <= '1' WHEN        Idle,
                                    '0'     WHEN    others;

                InEn<=InEn_sig;
END a;
```

3. 1-D Daub-4 Dual-Scan Architecture

```
-- a double scan lift arch. for multi-stage D-4
-- Hongyu Liao 1/23/02
-- Last update: Jun28/2001
LIBRARY ieee;
library comp;

use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use comp.liftcomp.all;

ENTITY d4lift_ms IS
        GENERIC(width : positive:= 16; fifo_len:positive:=1;SigLen:positive:=32);
        PORT(
                firstline, secondline   : IN        std_logic_vector(width-1 downto 0);
                clock,reset,start: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT               std_logic_vector(width-1 downto 0);
                Done: out std_logic);
END d4lift_ms;

ARCHITECTURE beh OF d4lift_ms IS
component d4lift
        GENERIC(width : positive; fifo_len:positive);
        PORT(
                firstline, secondline   : IN        std_logic_vector(width-1 downto 0);
                clock, enable,clear,InEn,Set0: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT               std_logic_vector(width-1 downto 0));
END component;

component DSControl
        generic(SigLen: positive);
        PORT(
                clk             : IN        STD_LOGIC;
                reset           : IN        STD_LOGIC;
                start           : IN        STD_LOGIC;
                Sel_EO,Sel_FIFO,Sel_In,ClkCtr,InEn,Set0,Done: OUT STD_LOGIC);
END component;

signal first_int,second_int,ToFirst,ToSecond, Lout, ToFifo,Fifo1_in,Fifo2_in: std_logic_vector(width-1 downto 0);
signal InSel, EOSel,FSel,FClk,FClk2,clkx2,clksel,InputEn,Setzero,high:std_logic;
signal zeropad:std_logic_vector(5 downto 0):=(others=>'0');
signal first_sc,second_sc:std_logic_vector(width-1 downto 0);

for in_sw1,in_sw2: mux use entity comp.mux;
for fifo1,fifo2: delay use entity comp.delay;
for l_sw1,l_sw2: switch use entity comp.switch;
for D_Freq:T_FF use entity comp.t_ff;
for core: d4lift use entity work.d4lift;
for controller:DSControl use entity work.dscontrol.
BEGIN
            high<='1';
            first_sc<=firstline(width-7 downto 0) & zeropad;
            second_sc<=secondline(width-7 downto 0) & zeropad;
            in_sw1: mux GENERIC map(width=>width)
                        PORT map(
                        input1=>first_sc,input2=>first_int, Sel=>InSel,  output=>ToFirst);

            in_sw2: mux GENERIC map(width=>width)
                        PORT map(
                        input1=>second_sc,input2=>second_int, Sel=>InSel,  output=>ToSecond);

            core: d4lift GENERIC map(width=>width,fifo_len=>fifo_len)
                        PORT map(
                        firstline=>ToFirst,secondline=>ToSecond,clock=>clock,
                        enable=>high,clear=>reset,InEn=>InputEn,Set0=>Setzero,
                        dwt_coeff_L=>Lout, dwt_coeff_H=>dwt_coeff_H);

            controller:DSControl generic map(SigLen=>SigLen)
                        PORT map(
                        clk=>clock,reset=>reset,start=>start,
                        Sel_EO=>EOSel,Sel_FIFO=>FSel,Sel_In=>InSel,ClkCtr=>clksel,
                        InEn=>InputEn,set0=>Setzero,Done=>Done);

            L_sw1:switch GENERIC map(width=>width)
                        PORT map(
                        input=>Lout,Sel=>EOSel, output1=>dwt_coeff_L, output2=>ToFifo);

            L_sw2:switch GENERIC map(width=>width)
                        PORT map(
                        input=>ToFifo,Sel=>FSel, output1=>Fifo1_in, output2=>Fifo2_in);

            D_Freq:T_FF port map (clk=>clock,NotEn=>reset,Q=>clkx2);

            with clksel select
                        FClk<= clock when '1' ,
                                    clkx2 when others;

-- 180 phase shift for fifo1 clock signal, so that the Lcoeff for the first signal can be latched
            FClk2<=not FClk when clksel='0' else
                        FClk when clksel='1'; -- after 1 ns;
-- A bug in the simulator switching the content of these two fifos
            Fifo1: delay generic map(width=>width,len=>SigLen/2)
                        port map(
                        clk=>FClk2,clr=>reset,input=>Fifo1_in, output=>second_int);--first_int feb07/02

            Fifo2: delay generic map(width=>width,len=>SigLen/2)
                        port map(
                        clk=>FClk,clr=>reset,input=>Fifo2_in, output=>first_int);--second_int
END beh;


-- a double scan lift arch. for D-4
-- Hongyu Liao 11/01/2001
-- modified for synopsys
-- Last update: jun08/2002
LIBRARY ieee;
library comp;

use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use comp.liftcomp.all;

ENTITY d4lift IS
            GENERIC(width : positive:= 16; fifo_len:positive:=1);
            PORT(
                        firstline, secondline   : IN        std_logic_vector(width-1 downto 0);
                        clock, enable,clear,InEn,Set0: in std_logic;
                        dwt_coeff_L, dwt_coeff_H: OUT              std_logic_vector(width-1 downto 0));
END d4lift;

ARCHITECTURE beh OF d4lift IS

signal alpha,beta,gama,one,K,K_inv:std_logic_vector(width-1 downto 0);
signal e1_in,e1,e2,e3,e4_in,e4,e5,o1,o2,o3,
        o4,o5_in,o5,delay_in,zero:std_logic_vector(width-1 downto 0);
signal lowout,highout,lowout2,highout2:std_logic_vector(width*2-1 downto 0);

for reg1,reg2,reg3,reg4,reg5:reg use entity comp.reg;
for mac1,mac2,mac3,mac4: mac use entity comp.mac;
for mult1,mult2: mult use entity comp.mult;
for delay1: delay use entity comp.delay;
for mux1: mux use entity comp.mux;
for in_circuit: in_switch use entity comp.in_switch;
BEGIN
        alpha<="1001000100100111"; -- right shift the original parameters for 14bits
        beta<="0011011101101101";
        gama<="1111011101101101";
        one<="0100000000000000";
        K<="0111101110100011";
        K_inv<="0100001001000010";

        zero<=(others=>'0');

        in_circuit:in_switch generic map(width=>width)
                        port map( firstline=>firstline,secondline=>secondline, clk=>clock,enable=>InEn,
                                odd=>o1,even=>e1_in);

        reg1:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,input=>e1_in,output=>e1);

        mac1:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>o1,mul=>e1,amp=>alpha, output=>o2);

        reg2:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,input=>e1,output=>e2);

        mac2:mac generic map(width=>width,add_sc=>15,res_sc=>15)
                port map(clock=> clock,clear=>clear,acc=>e2,mul=>o2,amp=>beta, output=>e3);

        reg3:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,input=>o2,output=>delay_in);

        delay1:delay generic map(width=>width,len=>fifo_len)
                port map(clk=>clock,clr=>clear,input=>delay_in,output=>o3);

        mac3:mac generic map(width=>width,add_sc=>15,res_sc=>15)
                port map(clock=> clock,clear=>clear,acc=>e3,mul=>o3,amp=>gama, output=>e4_in);

        mux1:mux generic map(width=>width)
                port map(input1=>e4_in,input2=>zero,Sel=>Set0,output=>e4);

        reg4:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,input=>o3,output=>o4);

        mac4:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>o4,mul=>e4,amp=>one, output=>o5);

        reg5:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,input=>e4,output=>e5);

        mult1:mult generic map(width=>width)
                port map(a=>e5,b=>K,p=>lowout);

        mult2:mult generic map(width=>width)
                port map(a=>o5,b=>K_inv,p=>highout);
                scale1:dwt_coeff_L<=lowout(width*2-3 downto width-2);
                scale2:dwt_coeff_H<=highout(width*2-2 downto width-1);

END beh;

-- a controller for 1-D DoubleScan
--created 1/23/02
-- Last update
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

ENTITY DSControl IS
        generic(SigLen: positive:=8);
        PORT(
                clk             : IN            STD_LOGIC;
                reset           : IN            STD_LOGIC;
                start           : IN            STD_LOGIC;
                Sel_EO,Sel_FIFO,Sel_In,ClkCtr,InEn,Set0,Done: OUT STD_LOGIC);
END DSControl;

ARCHITECTURE a OF DSControl IS
        TYPE STATE_TYPE IS (idle,init,ToFifo1,ToFifo2,FinalStg,Finish);
        SIGNAL state: STATE_TYPE;

BEGIN
        PROCESS (clk)
        variable FifoDep: positive:=SigLen;
        variable Count: natural:=0;
        variable counting: std_logic;
        BEGIN
                IF reset = '1' THEN
                        state <=idle;
                ELSIF clk'EVENT AND clk = '1' THEN

                        CASE state IS
                                WHEN idle =>
                                        IF start='1' THEN
                                        state <= init;
                                        InEn<='1';
                                        counting:='1';
                                        END IF;
                                        Set0<='0';

                                WHEN init =>
                                        IF count=6 THEN
                                                state <= ToFifo1;
                                                Set0<='1';
                                        end if;
                                        state <= ToFifo2;

                                WHEN ToFifo2 =>
                                        if count=FifoDep+6 then
                                                state <= FinalStg;
                                                InEn<='0';
                                        else
                                                state<=ToFifo1;
                                        END IF;
                                when FinalStg =>
                                                InEn<='1';
                                        if count=FifoDep+7 then
                                                Set0<='0';
                                        end if;
                                        if count=FifoDep+FifoDep/2+12 then
                                                Set0<='1';
                                        end if;
                                        if count=FifoDep+FifoDep/2+13 then
                                                state<=Finish;
                                                InEn<='0';
                                        end if;
                                when Finish =>
                                        state<= idle;
                                END CASE;
                        if counting='1' then
                                count:=count+1;
                        end if;
                END IF;
        END PROCESS;

        WITH state SELECT
                Sel_In      <= '1' WHEN         FinalStg,
                             '0'        WHEN    others;
        WITH state SELECT
                ClkCtr      <= '1' WHEN         FinalStg,
                             '0'        WHEN    others;
        with state select
                Sel_EO      <= '1' WHEN         ToFifo1,
                                '1' when ToFifo2,
                             '0'        WHEN    others;
        with state select
                Sel_FIFO<= '1' WHEN             ToFifo2,
                             '0'        WHEN    others;
        with state select
                Done <='1' When finish,
                        '0' when others;
END a;
```

4. 2-D Daub-4 Recursive Architecture

```
-- a recursive lift arch. for multi-stage 2D (NxM) D-4
-- Hongyu Liao Apr11/02
-- the column pe processes the low frequency component first
-- and then the hi frequency component so that the data flow
-- can be more regular. Apr14/02
-- Last update: May15/02
LIBRARY ieee;
library my_lib;
library lib;
library std;

use ieee.std_logic_1164.all;
use my_lib.liftcomp.all;
use std.textio.all;
use lib.tb_utilities.all;
use lib.io_utils.all;

ENTITY rs_2d IS
GENERIC (width: positive:= 16; RowSiz:positive:=32; CoLSiz:positive:=32);
PORT(
        InSignal    : IN        std_logic_vector(width-1 downto 0);
        clock,reset,start,RowEnd,FrameEnd: in std_logic;
        dwt_coeff_L, dwt_coeff_H: inOUT         std_logic_vector(width-1 downto 0);
        Done: inout std_logic);
END rs_2d;

ARCHITECTURE beh OF rs_2d IS
component rs_pe
        GENERIC(width : positive; fifo_len1:positive;fifo_len2:positive;fifo_len3:positive);
        PORT(
                E,O             : IN        std_logic_vector(width-1 downto 0);
                clock,clear,InEn,Set0S1,Set0S2,Set0S3,S2en2,S3en2: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT           std_logic_vector(width-1 downto 0));
END component;
```

```
component RS2dCtrl
            generic(RowLen: positive;ColLen:positive);
            PORT(
                        clk          : IN         STD_LOGIC;
                        reset        : IN         STD_LOGIC;
                        start,RowEnd,FrameEnd           : IN        STD_LOGIC;
                        RowEnd_ColumnPE,clear:inout std_logic;
                        InEn,Set0row: OUT   STD_LOGIC;
                        S3en1bc: OUT        STD_LOGIC);
END component;

constant input_lsh:natural:=4;
constant scaler:natural:=14;
signal clr,S2enOrg,S2en1r,S3enOrg,S3en1r,S2en2r,S2en2r_int,S3en2r,InputEn,NInEn,Lout:std_logic;
signal
S1EnR,S1EnC,S2EnR,S2EnC,S2enDelay,S2EnRDL,S2en2c,S2en2C_int,S2en2C_last,S3en2c,S3en2C_int,S3en2C_last,Set0S1C:st
d_logic;
signal Set0S1CEn, Set0S2CEn, Set0S3CEn,Set0S1CEn_org, Set0S2CEn_org, Set0S3CEn_org: std_logic;
signal S1ColswEn,S2ColInputEn, S3ColInputEn,FifoS1En:std_logic;
signal S2sw1r,S3sw1r,S3en1br,S3en1br_int,S3en1r_int:std_logic;
signal S3en1bc,S3EvenRow,S3enDelay:std_logic;
signal RowEnd_ColumnPE,RowEndD1,SelOddRow,SelOddRowS2_int,SelOddRowS3_int,
         SelOddRowS3_int2,SelOddRowS2,SelOddRowS3:std_logic;
signal
SetzeroR,set0S1R,S2en2rLast,S3en2rLast,set0S2R,Set0S2C,Set0S2C_int,set0S3R,SetzeroC,Set0S3C,Set0S3RFirst,Set0S3RLast
         :std_logic;
signal Set0S2C_DL   :std_logic;
signal clkx8Org,clkx8Orgdel :std_logic;
signal LLImageEnS1,done_int     :std_logic;
signal
S2CFifoEn,S3CFifoEn,SetOddR_DL,SetOddR_DL2,SetOddR_DL3,SetOddR_DL4,Set0S2C_H,Set0S2C_H_int,S3en2r_DL:std_lo
gic;
signal zeropad:std_logic_vector(input_lsh-1 downto 0):=(others=>'0');
signal input_sc,LRow,HRow,Lsig:std_logic_vector(width-1 downto 0);
signal LRowS1,S1RFifoIn,S1RowE,S1RowH,S1RLowOdd:std_logic_vector(width-1 downto 0);
signal LRowS2,S2RFifoIn,S2RowE,S2RowH,S2RLowOdd:std_logic_vector(width-1 downto 0);
signal LRowS3,S3RFifoIn,S3RowE,S3RowH,S3RLowOdd:std_logic_vector(width-1 downto 0);
signal EcolIn,OColIn,RowEIn_s1,RowEIn_s2,RowEIn_s3,RowOIn_s3:std_logic_vector(width-1 downto 0);
signal ERowIn,ORowIn:std_logic_vector(width-1 downto 0);
signal S2OutLEn,S2OutHEn,S3OutLEn,S3OutHEn: std_logic;

signal S1RowCnt,S2RowCnt,S3RowCnt,S1ColCnt,S2ColCnt,S3ColCnt: natural;
constant Lstage:positive:=3;
BEGIN
         input_sc<=InSignal(width-input_lsh-1 downto 0) & zeropad;

         ------------------------------------------
         ---- input switches for Row Processor ----
         ------------------------------------------
         NInEn<=not InputEn;
         -- enable signal for the stage 1 data flow
         -- S1en identifier:
         s1sel:T_FF port map(clk=>clock,clr=>NInEn ,q=>S1enR); -- 2i+1

         -- enable signal for stage 2 data flow, Delay+2kM+4i+2
         -- generate 4xClk signal
         s2clkdiv:T_FF port map(clk=>S1EnC,clr=>NInEn ,q=>S2enR); --4i+2,3
         -- generate a signal with duty=25%, f=4xClk
         S2EnRDelay:d_ff port map(d=>S2enR,clk=>clock,clr=>NInEn,q=>S2enRDL); --4i+3,4
         S2enOrg<=(not S2enRDL) and S2enR; --4i+2
         S2en1r <= LLImageEnS1 and S2enOrg ; -- (2k+1)M+10+4i+2

         -- enable signal for stage 3
         -- generate 8xclk signal
         s3clkdiv:T_FF port map(clk=>S2enOrg,clr=>NInEn ,q=>clkx8Org); -- 8i+2,3,4,5
         S3clkdivdel:d_ff port map(d=>clkx8Org,clk=>clock,clr=>NInEn,q=>clkx8Orgdel); --8i+3,4,5,6
         S3enOrg<= (not clkx8Orgdel) and clkx8Org; --8i+2
         S3enDel1: bit_delay generic map (del=>5)
                  port map(clk=>clock,clr=>clr,d=>S3enOrg,q=>S3en1r_int); -- 8i+7
         S3EvenRow<=SelOddRowS2_int and (not SelOddRowS2);
```

```
S3OddRoeDel: bit_delay generic map (del=>3)
            port map(clk=>clock,clr=>clr,d=>S3EvenRow,q=>SelOddRowS3_int); --4kM+23?
S3en1r<=  S3en1r_int and SelOddRowS3_int; --4kM+23+8i+7
S3enDel2: bit_delay generic map (del=>4)
            port map(clk=>clock,clr=>clr,d=>S3en1r,q=>S3en1br); -- 4kM+23+8i+11
-- stage 3: row pe input switch signal = S3en1br+3
S3RinSw: bit_delay generic map (del=>3)
            port map(clk=>clock,clr=>clr,d=>S3en1br,q=>S3sw1r); --4kM+23+8i+14
S3RDelEn:bit_delay generic map (del=>2)
            port map(clk=>clock,clr=>clr,d=>S3sw1r,q=>S3en2r); --4kM+23+8i+16

-- synchronize the even and odd samples
reg1:reg generic map(width=>width)
            port map(clk=>clock,clr=>clr,en=>InputEn,input=>input_sc,output=>RowEIn_s1);
reg1_2:reg generic map(width=>width)
            port map(clk=>clock,clr=>clr,en=>S2en1r,input=>Lsig,output=>RowEIn_s2);
reg1_3:reg generic map(width=>width)
            port map(clk=>clock,clr=>clr,en=>S3en1r,input=>Lsig,output=>RowEIn_s3);
reg1_4:reg generic map(width=>width)
            port map(clk=>clock,clr=>clr,en=>S3en1br,input=>Lsig,output=>RowOIn_s3);

-- there are delays between the switch signals and the enable signals
-- for the data flows of stage 2 and up, which allows the data flows to get to
-- the next step:
--          Stage 2 switch signal is a 2 cycle-delay version of S2en1r;
S2RInSel:bit_delay generic map (del=>2)
            port map(clk=>clock,clr=>clr,d=>S2en1r,q=>S2sw1r); -- (2k+1)M+10+4i+4

reinmux: ERowIn<= RowEIn_s1 when S1enR='1' else
                   RowEIn_s2 when S2sw1r='1' else
                   RowEIn_s3 when S3sw1r='1' else
                   (others=>'0');
roinmux: ORowIn<= input_sc when S1enR='1' else
                   Lsig when S2sw1r='1' else
                   RowOIn_s3 when S3sw1r='1' else
                   (others=>'0');

-- Enable signals for the delay units
-- 1. Enable signal for the delay unit of Stage 2
--          is a 2 cycle delay version of S2sw1r
S2RDelayEn:bit_delay generic map (del=>2)
            port map(clk=>clock,clr=>clr,d=>S2sw1r,q=>S2en2r_int); -- (2k+1)M+10+4i+6

------------------------------------------------------------------
-- Set0 signal for Row PE: set at the same time as the last                         --
--                       enable signals for the delay units
------------------------------------------------------------------
-- Set0 for Stage 2 = last S2 input switch signal+6, where 2cycles for delay from input to delay unit
-- and 4 cycles for the delay of this stage.
S2en2rLast<=(not SelOddRow) and S2en2r_int; -- (2k+2)M+10+6, M=4i, which is true in almost all cases
S2set0R:bit_delay generic map (del=>4)
            port map(clk=>clock,clr=>clr,d=>S2en2rLast,q=>Set0S2R); -- (2k+2)M+20

-- Set0 for stage 3: The last S3en2r delay 8 cycles
S3en2rLast<=( not S3EvenRow) and S3en2r;
S3set0RLast:bit_delay generic map (del=>8)
            port map(clk=>clock,clr=>clr,d=>S3en2rLast,q=>Set0S3RLast);
-- set0 for the first coefficient of each row
S3OddRDelay4:bit_delay generic map(del=>12)
                  port map(d=>SetOddR_DL2,clk=>clock, clr=>clr,q=>SetOddR_DL4);
Set0S3RFirst<=S3en2R and SetOddR_DL4;
Set0S3R<=Set0S3RFirst or Set0S3RLast;

SetzeroR<=set0S1R or set0S2R or set0S3R;

------------------------------
------- Row Processor --------
------------------------------
RowPe: rs_pe GENERIC map(width=>width,fifo_len1=>1,fifo_len2=>1,fifo_len3=>1)
            PORT map(
            E=>ERowIn,O=>ORowIn,clock=>clock,
            clear=>clr,InEn=>InputEn,Set0S1=>Set0S1R,Set0S2=>Set0S2R,Set0S3=>Set0S3R,
            S2en2=>S2enOrg,S3en2=>S3en2r,
            dwt_coeff_L=>LRow, dwt_coeff_H=>HRow);


------------------------------
--- Row Counters -----------
------------------------------
process(clock)
begin
            if rising_edge(clock) then
                        if InputEn='1' then
                                    if set0S1R='1' then
                                                S1RowCnt<= S1RowCnt+1 ;
                                    end if;
                                    if set0S2R='1' then
                                                S2RowCnt<= S2RowCnt+1 ;
                                    end if;
                                    if set0S3R='1' then
                                                S3RowCnt<= S3RowCnt+1 ;
                                    end if;
                        else
                                    S1RowCnt<= 0 ;
                                    S2RowCnt<= 0 ;
                                    S3RowCnt<= 0 ;
                        end if;
            end if;
end process;

--------------------------------------------------
-------- Input circuitry for Column PE ---------
--------------------------------------------------
-- Delays for synchronizing the low and high frequency coefficients
-- Stage 1
S1EnC<=not S1EnR; -- 2i+1
-- L components delay 3 cycles to synchronize with the h components
SynDelS1: delay generic map(width=>width,len=>3)
            port map(clk=>clock,clr=>clr,en=>InputEn,input=>LRow,output=>LRowS1);

-- Stage 2,S2en2R delay 1 cycle as stage2 Low output delay enable
-- 2kM+2i+2
S2RHoutEnable:bit_delay generic map (del=>6)
                        port map (clk=>clock,clr=>clr,d=>S2en2r_int, q=>S2EnC); --(2k+1)M+10+4i+12
-- delay 5 cycles to synchronize with the h components, 2 cycles from DelS1 output
SynDelS2: delay generic map(width=>width,len=>2)
            port map(clk=>clock,clr=>clr,en=>InputEn,input=>LRowS1,output=>LRowS2);

-- low frequency DWT row coefficient delay one cycle to sychronize with high
-- frequency coefficient
SynDelS3: delay generic map(width=>width,len=>4)
                                    port map (clk=>clock, clr=>clr, en=>InputEn, input=>LRowS2, output=>LRowS3);

-- EndofRow signal delayed 8 cycle and passed through a T-FF to generate
-- a row select signal for colume PE
RowEndDelay:bit_delay generic map (del=>8)
                        port map(clk=>clock,clr=>clr,d=>RowEnd_ColumnPE,q=>RowEndD1); --kM+7, k>0.

S1RowSel: T_FF port map(clk=>RowEndD1,clr=>clr,q=>SelOddRow); --(2k+1)M+7~(2k+2)M+7, k>=0

-- Switches for exchanging the low and high frequency coeff.s
-- Stage 1
S1ExA: S1RFifoIn<= LRowS1 when SelOddRow='0' else
            S1RowH;
S1ExB: S1RLowOdd<= LRowS1 when SelOddRow='1' else
            S1RowH;
-- Stage 2
-- the select signal frequenct is LLImageEnS1, delay 11 cycles
SelRowDelS2: bit_delay generic map (del=>10)
                        port map (d=>LLImageEnS1,clk=>clock,clr=>clr,q=>SelOddRowS2_int); --
```

```
(2k+1)M+20~(2k+2)M+20
            S2RowSel: T_FF port map(clk=>SelOddRowS2_int, clr=>clr,q=>SelOddRowS2); -- (4k+1)M+20~(4k+3)M+20

            S2ExA: S2RFifoIn<= LRowS2 when SelOddRowS2='1' else
                     S2RowH;
            S2ExB: S2RLowOdd<= LRowS2 when SelOddRowS2='0' and SelOddRowS2_int='1' else
                     S2RowH;

            -- Stage 3
            -- the select signal frequenct is seloddrows3_int, delay 11 cycles
            SelRowDelS3: bit_delay generic map (del=>19)
                                  port map (d=>SelOddRowS3_int,clk=>clock,clr=>clr,q=>SelOddRowS3_int2);
            S3RowSel: T_FF port map(clk=>SelOddRowS3_int2, clr=>clr,q=>SelOddRowS3);
            S3ExA: S3RFifoIn<= LRowS3 when SelOddRowS3='1' else
                     S3RowH;
            S3ExB: S3RLowOdd<= LRowS3 when SelOddRowS3='0' and SelOddRowS3_int2='1' else
                     S3RowH;

            ------------------------------
            ---- Col PE Stage 2 Enable ----
            ------------------------------
            -- Stage 2
            -- The column DWT coefficients are processed when SelOddRowS2=0
            -- Fifo enable signal
            DelayS2en: bit_delay generic map(del=>2)
                                                  port map(clk=>clock,clr=>clr,d=>S2enOrg,q=>S2enDelay);--4i+4

            ColPES2en: bit_delay generic map(del=>2)
                                                  port map(clk=>clock,clr=>clr,d=>S2enDelay,q=>S2en2c_int);--4i+6
            S2Idle:d_ff port map(d=>SetOddR_DL,clk=>clock, clr=>clr,q=>SetOddR_DL2);
            S2en2C<= ((not SetOddR_DL2) and S2en2C_int) or S2en2C_last;


            S2ColInputEn<= (not SelOddRowS2) and S2enDelay;
            S2CFifoEn<=S2ColInputEn or S2enC;

            -- Stage 3
            DelayS3en: bit_delay generic map(del=>3)
                                                  port map(clk=>clock,clr=>clr,d=>S3en1r_int,q=>S3enDelay);

            ---------------------------------------------------
            -- Fifo2 for storing the even row coefficients ------
            ---------------------------------------------------
            -- Stage 1
            FifoS1En<=S1ColswEn and S1enR;
            LFifoS1:delay generic map(width=>width,len=>RowSiz/2)
                                  port map(clk=>clock,clr=>clr,en=>FifoS1En,input=>S1RFifoIn,output=>S1RowE);

            HFifoS1:delay generic map(width=>width,len=>RowSiz/2)
                                  port map(clk=>clock,clr=>clr,en=>FifoS1En,input=>HRow,output=>S1RowH);


            LFifoS2:delay generic map(width=>width,len=>RowSiz/4)
                                  port map(clk=>clock,clr=>clr,en=>S2CFifoEn,input=>S2RFifoIn,output=>S2RowE);

            HFifoS2:delay generic map(width=>width,len=>RowSiz/4)
                                  port map(clk=>clock,clr=>clr,en=>S2CFifoEn,input=>HRow,output=>S2RowH);


            ---- Column PE input fifos for Stage 3
            S3en2rDel: bit_delay generic map(del=>10)
                                                  port map(clk=>clock,clr=>clr,d=>S3en2r,q=>S3en2r_DL);
            S3Idle:bit_delay generic map(del=>19)
                                  port map(d=>SetOddR_DL2,clk=>clock, clr=>clr,q=>SetOddR_DL3);
            S3ColInputEn<= (not SelOddRowS3) and (not SetOddR_DL3) and S3enDelay;
            S3en2C_int<= (((not SelOddRowS3) and (not SetOddR_DL3)) or Set0S3CEn_org) and S3enDelay;-- extra time for
shifting out the last row

            S3en2CSig:bit_delay generic map(del=>2)
                                                  port map(clk=>clock,clr=>clr,d=>S3en2C_int,q=>S3en2C);
```

```
S3CFifoEn<=S3ColInputEn or (S3en2r_DL and SelOddRowS3);

LFifoS3:delay generic map(width=>width,len=>RowSiz/8)
                    port map(clk=>clock,clr=>clr,en=>S3CFifoEn,input=>S3RFifoIn,output=>S3RowE);

HFifoS3:delay generic map(width=>width,len=>RowSiz/8)
                    port map(clk=>clock,clr=>clr,en=>S3CFifoEn,input=>HRow,output=>S3RowH);

-- Column processor input switches
EColIn<=S1RowE when S1ColswEn='1' and S1EnR='1' else
                         S2RowE when S2ColInputEn='1' else
                         S3RowE when S3ColInputEn='1' else
               (others=>'0');
OColIn<=S1RLowOdd when S1ColswEn='1' and S1EnR='1' else
                         S2RLowOdd when S2ColInputEn='1' else
                         S3RLowOdd when S3ColInputEn='1' else
               (others=>'0');

------------------------------------------
------- Set 0 for Column PE ------------
------------------------------------------
S2set0C2:d_ff port map(d=>SelOddRowS2,clk=>clock, clr=>clr,q=>SetOddR_DL);

-- set0 signals for stage 3

-- Enable set 0 signal for Stage 1
Set0S1CEn_org<='1' when S1RowCnt>=RowSiz+1 else
                                                              '0';
Set0S1CEnable:bit_delay generic map (del=>5)           -- 5 cycles
               port map(clk=>clock,clr=>clr,d=>Set0S1CEn_org,q=>Set0S1CEn);
Set0S1C<=Set0S1CEn and S1enR;

-- Enable set 0 signal for Stage 2
Set0S2CEn_org<= Set0S1CEn_org and SetOddR_DL;
S2en2C_last<=Set0S2CEn_org and S2en2C_int;
Set0S2C<=            S2en2C_last;

-- Enable set 0 signal for Stage 3
Set0S3CEn_org<= Set0S1CEn_org and SetOddR_DL3;
S3en2C_last<= Set0S3CEn_org and S3en2C;
Set0S3C<=S3en2C_last;

------------------------------------------
--------- Column Processor -------------
------------------------------------------
ColPe: rs_pe GENERIC map(width=>width,fifo_len1=>RowSiz,fifo_len2=>RowSiz/2,fifo_len3=>RowSiz/4)
          PORT map(
          E=>EColIn,O=>OColIn,clock=>clock,
          clear=>clr,InEn=>InputEn,Set0S1=>Set0S1C,Set0S2=>Set0S2C,Set0S3=>Set0S3C,
          S2en2=>S2en2c,S3en2=>S3en2c,
          dwt_coeff_L=>Lsig, dwt_coeff_H=>dwt_coeff_H);

------------------------------
--- Column Counters -----------
------------------------------
process(clock)
begin
          if rising_edge(clock) then
                    if InputEn='1' then
                              if set0S1C='1' then
                                        S1ColCnt<= S1ColCnt+1 ;
                              end if;
                              if set0S2C='1' then
                                        S2ColCnt<= S2ColCnt+1 ;
                              end if;
                              if set0S3C='1' then
                                        S3ColCnt<= S3ColCnt+1 ;
                              end if;
                    else
                              S1ColCnt<= 0 ;
```

```
                                        S2ColCnt<= 0 ;
                                        S3ColCnt<= 0 ;
                                end if;
                        end if;
        end process;

        ----------------------------------------------------------
        ---------- Column PE Low Frequency Coeff selector ----------
        ----------------------------------------------------------
        -- enable signal for LL DWT coefficients, 3 cycle delay of SelOddRow signal
        S1LLSel:bit_delay generic map (del=>3)
                        port map(clk=>clock,clr=>clr,d=>SelOddRow,q=>LLImageEnS1); --(2k+1)M+10~(2k+2)M+10

        -- LL Subimage
        --LLImage: Lcomp <= Lsig

        -- LH Subimage, condition 1: LH for Stage 1
        LHImage: dwt_coeff_L <= Lsig when (LLImageEnS1='0' and S1EnC='1') else
                                (others=>'0');

        ---------------------------------
        ------- Controller --------------
        ---------------------------------
        controller:RS2dCtrl generic map(RowLen=>RowSiz,ColLen=>ColSiz)
                PORT map(
                clk=>clock,reset=>reset,start=>start,RowEnd=>RowEnd,FrameEnd=>FrameEnd,
                RowEnd_ColumnPE=>RowEnd_ColumnPE,clear=>clr,InEn=>InputEn,sel0row=>Sel0S1R,
                S3en1bc=>S3en1bc);

        ---------------------------------------------
        ------- counters and control signals ---------
        ------- for switching data flows of ---------
        ------- different DWT stages         ---------
        ---------------------------------------------
        process(clock)
        variable cntS1RStart,cntS2RStart,cntS3RStart:natural;
        begin
                if InputEn='0' then
                        cntS1RStart:=0;
                        cntS2RStart:=0;
                        cntS3RStart:=0;
                        S1ColswEn<='0';
                else
                        if cntS1RStart=6 then
                                S1ColswEn<='1';
                        elsif rising_edge(clock) then
                                cntS1RStart:=cntS1RStart+1;
                        end if;

                end if;
        end process;

        done_int<= '1' when S1RowCnt>=RowSiz+4 else  --S3ColCnt>=ColSiz/(2**(Lstage-1)) else
                                '0';
        doneSig:bit_delay generic map (del=>2)
                        port map(clk=>clock,clr=>clr,d=>done_int,q=>done);

        ----------------------------------------
        ----- Writing results to data files -----
        ----------------------------------------
        S2LOutEnable: bit_delay generic map (del=>1)
                        port map(clk=>clock,clr=>clr,d=>S2en2C,q=>S2OutLEn);
        S2HOutEnable: bit_delay generic map (del=>2)
                        port map(clk=>clock,clr=>clr,d=>S2en2C,q=>S2OutHEn);
        S3LOutEnable: bit_delay generic map (del=>1)
                        port map(clk=>clock,clr=>clr,d=>S3en2C,q=>S3OutLEn);
        S3HOutEnable: bit_delay generic map (del=>2)
                        port map(clk=>clock,clr=>clr,d=>S3en2C,q=>S3OutHEn);

        output: process(clock)
```

```
                file testoutputS3L:text open write_mode is "S3outputL.txt";
                file testoutputS3h:text open write_mode is "S3outputH.txt";
                file testoutputS2L:text open write_mode is "S2outputLH.txt";
                file testoutputS2h:text open write_mode is "S2outputH.txt";
                file testoutputS1l:text open write_mode is "S1outputLH.txt";
                file testoutputS1h:text open write_mode is "S1outputH.txt";
                variable OutLS1,OutHS1,OutLS2,OutHS2,OutLS3,OutHS3: Line;
    --          variable E: boolean:=True;
                begin

                if rising_edge(clock) then
                        -- 1H
                        if (S1RowCnt>=3 and done='0' ) and FifoS1En='1' then
                                write(OutHS1,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS1h,OutHS1);
                        -- 1HL
                        elsif (S1RowCnt>=2 and S1RowCnt<RowSiz+2 and LLImageEnS1='0') and FifoS1En='0' then
                                write(OutLS1,slv_to_bv(Lsig),right,8,decimal,false);
                                        writeline(testoutputS1l,OutLS1),
                        end if;
                        -- 2H
                        if (S1RowCnt>=7 and done='0') and S2OutHEn='1' then
                                write(OutHS2,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS2h,OutHS2);
                        -- 2HL
                        elsif S2OutLEn='1' and (S1RowCnt>=4 and S1RowCnt<RowSiz+2) and SelOddRowS2_int='0' then
                                write(OutLS2,slv_to_bv(Lsig),right,8,decimal,false);
                                        writeline(testoutputS2L,OutLS2);
                        end if;
                        -- 3H
                        if (S1RowCnt>=15 and done='0') and S3OutHEn='1' then
                                write(OutHS3,slv_to_bv(dwt_coeff_H),right,8,decimal,false);
                                        writeline(testoutputS3h,OutHS3);
                        -- 3L
                        elsif S3OutLEn='1'  and (S1RowCnt>=7 and S1RowCnt<RowSiz+2) then
                                write(OutLS3,slv_to_bv(Lsig),right,8,decimal,false);
                                        writeline(testoutputS3L,OutLS3);
                        end if;
                        --              writeline(testoutputh,OutH);
                end if;
                end process output;
END beh;


-- a process element for implementing 2_D DWT
-- Hongyu Liao feb12/2001
-- the module should be cleared before use
-- Change port map
-- Last update: Apr11/2002
LIBRARY ieee;
library my_lib;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use my_lib.liftcomp.all;

ENTITY rs_pe IS
        GENERIC(width : positive:= 16; fifo_len1:positive:=1;fifo_len2:positive:=1;fifo_len3:positive:=1);
        PORT(
                E,O             : IN            std_logic_vector(width-1 downto 0);
                clock,clear,InEn,Set0S1,Set0S2,Set0S3,S2en2,S3en2: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT                   std_logic_vector(width-1 downto 0));
END rs_pe;

ARCHITECTURE beh OF rs_pe IS

signal alpha,beta,gama,one,K,K_inv:std_logic_vector(width-1 downto 0);
signal e1,e2,e3,e4,o1,o2,o3,
        o4,o5,delay_in,s1,s2,s3,s:std_logic_vector(width-1 downto 0);
signal NInEn,S1en,S1sw2,S2sw2,S3sw2,set0,set0dl:std_logic;
signal lowout,highout:std_logic_vector(width*2-1 downto 0);

--for all:mac use entity my_lib.mac;
--for all:mac_b use entity my_lib.mac_b;
for all:mult use entity my_lib.mult;
for all:reg use entity my_lib.reg;
for all:delay use entity my_lib.delay;
for all:d_ff use entity my_lib.d_ff;
BEGIN
        alpha<="1001000100100111"; -- right shift the original parameters for 14bits
        beta<="0001101110110110";
        gama<="1111101110110111";
        one<="0100000000000000";
        K<="0111101110100011";
        K_inv<="0010000100100000";

        e1<=E;
        o1<=O;

        NInEn<=not InEn;
        -- enable signal for the stage1 data flow
        -- S1en
        s1sel:T_FF port map(clk=>clock,clr=>NInEn ,q=>S1en);

        mac1:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>o1,mul=>e1,amp=>alpha,
                        output=>o2);

        reg2:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>e1,output=>e2);

        mac2:mac generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,acc=>e2,mul=>o2,amp=>beta,
                        output=>e3);

        reg3:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>o2,output=>delay_in);

        -- delay unit, 3 delay registers
        d1:delay generic map(width=>width,len=>fifo_len1)
                port map(clk=>clock,clr=>clear,en=>S1en,input=>delay_in,output=>s1);
        d2:delay generic map(width=>width,len=>fifo_len2)
                port map(clk=>clock,clr=>clear,en=>S2en2,input=>delay_in,output=>s2);
        d3:delay generic map(width=>width,len=>fifo_len3)
                port map(clk=>clock,clr=>clear,en=>S3en2,input=>delay_in,output=>s3);

        -- In order to get the last data in the delay units
        -- the switch signal should has one more cycle than the enable signal
        S1sw2<=S1en or Set0S1;
        S2sw2<=S2en2 or Set0S2;
        S3sw2<=S3en2 or Set0S3;

        delaymux: o3<= s1 when S1sw2='1' else
                        s2 when S2sw2='1' else
                        s3 when S3sw2='1' else
                        (others=>'0');

        -- break the even and odd parts when set0
        mac3:mac_b generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,break=>set0,acc=>e3,mul=>o3,amp=>gama, output=>e4);

        reg4:reg generic map(width=>width)
                port map(clk=>clock,clr=>clear,en=>InEn,input=>o3,output=>o4);

        Set0<=Set0S1 or Set0S2 or Set0S3;
        set0delay:d_ff port map(d=>set0,clk=>clock,clr=>clear,q=>set0dl);
        mac4:mac_b generic map(width=>width,add_sc=>14,res_sc=>14)
                port map(clock=> clock,clear=>clear,break=>set0dl,acc=>o4,mul=>e4,amp=>one, output=>o5);

        mult1:mult
```

```
                        generic map(width=>width)
                        port map(a=>e4,b=>K,p=>lowout);

            mult2:mult
                        generic map(width=>width)
                        port map(a=>o5,b=>K_inv,p=>highout);

            scale1:dwt_coeff_L<=lowout(width*2-3 downto width-2)when lowout(width-3)='0' else
                                    lowout(width*2-3 downto width-2)+1;
            scale2:dwt_coeff_H<=highout(width*2-3 downto width-2) when highout(width-3)='0' else
                                    highout(width*2-3 downto width-2)+1;
END beh;


-- a controller for 1-D Recursive
--created feb13/02
-- Last update may15/02
library ieee;
library my_lib;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use my_lib.liftcomp.t_ff;

ENTITY RS2dCtrl IS
            generic(RowLen: positive:=8;ColLen:positive:=8);
            PORT(
                        clk         : IN        STD_LOGIC;
                        reset       : IN        STD_LOGIC;
                        start,RowEnd,FrameEnd           : IN        STD_LOGIC;
                        RowEnd_ColumnPE,clear:inout std_logic;
                        InEn,Set0row: OUT   STD_LOGIC;
                        S3en1be: OUT        STD_LOGIC);
END RS2dCtrl;

ARCHITECTURE a OF RS2dCtrl IS
            TYPE STATE_TYPE IS (idle,analyzing,endofrow,analyzing2,Finish);
            SIGNAL state: STATE_TYPE;
            signal InEn_sig,FrameEndLatch,AppendRowEnd:std_logic;
BEGIN
            PROCESS (clk)
            variable Count,cntset0r,cntset0c,LStage,CntLastRow: natural:=0;
            variable counting: std_logic;
            BEGIN
                        -- Number of stage
                        Lstage:=3;
                        IF reset = '1' THEN
                                    state <=idle;
                        ELSIF clk'EVENT AND clk = '1' THEN
                                    CASE state IS
                                                WHEN idle =>
                                                            -- reset everything
                                                            IF start='1' THEN
                                                                        state <= analyzing;
                                                            END IF;
                                                WHEN analyzing =>
                                                            if RowEnd_ColumnPE='1' then
                                                                        state<=endofrow;
                                                            end if;

                                                when endofrow =>
                                                            cntset0r:=0;
                                                            Set0row<='0';
                                                            if count=RowLen*ColLen+RowLen*Lstage*5+20 then
                                                                        state <= Finish;
                                                            else
                                                                        state<=analyzing2;
                                                            END IF;

                                                when analyzing2 =>
```

```
                                        cntset0r:=cntset0r+1;
                                        if RowEnd_ColumnPE='1' then
                                                state<=endofrow;
                                        end if;
                                        Set0row<='0';
--                                      Set0col<='0';
                                        if cntset0r=2 then
                                                Set0row<='1';
                                        end if;

                                when Finish =>
                                        state<=Idle;
                        end case;

                        -- add (L+1+2+...+2^L) more lines for processing the column DWT
                        -- latch FrameEnd
                        if FrameEndLatch='1' then
                                CntLastRow:=CntLastRow+1;
                        Else
                                CntLastRow:=0;
                        end if;
                        if CntLastRow=RowLen then
                                AppendRowEnd<='1';
                                CntLastRow:=0;
                        else
                                AppendRowEnd<='0';
                        end if;

                END IF;
        END PROCESS;

        LatchFrameEnd: t_ff port map (clk=>FrameEnd,clr=>clear,q=>FrameEndLatch);
        RowEnd_ColumnPE<=RowEnd or AppendRowEnd;
        WITH state SELECT
                InEn_sig <= '0' WHEN            Idle,
                            '1'     WHEN        others;
        WITH state SELECT
                clear       <= '1' WHEN         Idle,
                            '0'     WHEN        others;

        InEn<=InEn_sig;
END a;
```

5. 2-D Dual-Scan Architecture

```
- a 2-D double scan lift arch. for one-stage D-4
-- Hongyu Liao 4/23/02
--
--
-- Last update:
Jun28/2001
LIBRARY ieee;
library comp;

use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use comp.liftcomp.all;


ENTITY d4lift_2d IS
        GENERIC(width : positive:= 16; fifo_len_Row:positive:=1;fifo_len_Column:positive:=2;RowSize:

positive:=128;ColSize: positive:=128);
        PORT(
                firstrow, secondrow   : IN        std_logic_vector(width-1 downto 0);
```

```
                        clock,reset,start: in std_logic;
                        dwt_coeff_L, dwt_coeff_H: OUT            std_logic_vector(width-1 downto 0);
                        Done: out std_logic);
END d4lift_2d;

ARCHITECTURE beh OF d4lift_2d IS
component d4lift
        GENERIC(width : positive; fifo_len:positive);
        PORT(
                firstline, secondline   : IN        std_logic_vector(width-1 downto 0);
                clock, enable,clear,InEn,Set0: in std_logic;
                dwt_coeff_L, dwt_coeff_H: OUT              std_logic_vector(width-1 downto 0));
END component;

component DS2DControl
        generic(RowSize: positive:=8;ColSize: positive:=8);
        PORT(
                clk         : IN        STD_LOGIC;
                reset       : IN        STD_LOGIC;
                start       : IN        STD_LOGIC;
                InEn,RowSet0,ColSet0, Done: OUT          STD_LOGIC);
END component;

signal firstrow_sc,secondrow_sc,HRow,LRow: std_logic_vector(width-1 downto 0);
signal ColOdd1, ColOdd2, ColPEIn1, ColPEIn2,Lout, Hout: std_logic_vector(width-1 downto 0);
signal clkx2, InputEn,RowSetzero,ColSetzero,high:std_logic;
signal zeropad:std_logic_vector(5 downto 0):=(others=>'0');


for D_Freq:T_FF use entity comp.t_ff;
for RowPE, ColumnPE: d4lift use entity work.d4lift;
for controller:DS2DControl use entity work.DS2Control;
BEGIN
        high<='1';
        firstrow_sc<=firstrow(width-7 downto 0) & zeropad;
        secondrow_sc<=secondrow(width-7 downto 0) & zeropad;

        RowPE: d4lift GENERIC map(width=>width,fifo_len=>fifo_len_Row)
                PORT map(
                firstline=> firstrow_sc,secondline=> secondrow_sc,clock=>clock,
                enable=>high,clear=>reset,InEn=>InputEn,Set0=>RowSetzero,
                dwt_coeff_L=>LRow, dwt_coeff_H=>HRow);

        ColumnPE: d4lift GENERIC map(width=>width,fifo_len=>fifo_len_Column)
                PORT map(
                firstline=>LRow,secondline=>HRow,clock=>clock,
                enable=>high,clear=>reset,InEn=>InputEn,Set0=>ColumnSetzero,
                dwt_coeff_L=>dwt_coeff_L, dwt_coeff_H=>dwt_coeff_H);

        controller:DS2DControl generic map(RowSize=>RowSize,ColSize=>ColSize)
                PORT map(
                clk=>clock,reset=>reset,start=>start,
                InEn=>InputEn,Rowset0=>RowSetzero,ColSet0=>ColSetzero,Done=>Done);


        D_Freq:T_FF port map (
                clk=>clock,NotEn=>reset,Q=>clkx2);

END beh;


-- a controller for 2-D DoubleScan
--created 3/23/02
-- Last update
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

ENTITY DS2DControl IS
```

```
		generic(RowSize: positive:=8,ColSize: positive:=8);
		PORT(
				clk		: IN		STD_LOGIC;
				reset		: IN		STD_LOGIC;
				start		: IN		STD_LOGIC;
				InEn,RowSet0,ColSet0, Done: OUT		STD_LOGIC);
	END DS2DControl;

	ARCHITECTURE a OF DS2DControl IS
		TYPE STATE_TYPE IS (idle,RowPro, ColPro, Finish);
		SIGNAL state: STATE_TYPE;

	BEGIN
		PROCESS (clk)
		variable Count: natural:=0;
		variable Count2: natural:=0;
		variable counting: std_logic;
		BEGIN
			IF reset = '1' THEN
				state <=idle;
			ELSIF clk'EVENT AND clk = '1' THEN

				CASE state IS
					WHEN idle =>
						IF start='1' THEN
							state <= init;
							InEn<='1';
							counting:='1';
						END IF;
						Set0<='0';

					WHEN RowPro =>
						IF count=6 THEN
							state <= ColPro;
						END IF;
						ColSet0<='1';

					WHEN ColPro =>
						if count2=RowSize+3 then
							RowSet0<='1';
						end if;

						ColSet0<='0';

						if count=RowSize*ColSize+13 then
							state<=Finish;
							InEn<='0';
						end if;

					when Finish =>
						state<= idle;
					END CASE;
				if counting='1' then
					count:=count+1;
					count2:=count2+1;
					if count2=RowSize+3 then
						count2:=0;
					end if;
				end if;
			END IF;
		END PROCESS;
	END a;
```

6. Component Library: Liftcomp

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
library lib;
use lib.datalib.all;

PACKAGE liftcomp IS
        COMPONENT mult
                GENERIC(width : positive:= 8);
                PORT(
                        a, b                    : IN        std_logic_vector(width-1 downto 0);
                        p: OUT    std_logic_vector(width*2-1 downto 0));
        END COMPONENT;

        COMPONENT mac
                GENERIC(width : positive:= 8;keep:positive:=8;add_sc:positive:=1;res_sc:positive:=1);
                PORT(   clock,clear : in          std_logic;
                        acc, mul,amp            : IN        std_logic_vector(width-1 downto 0);
                        output: OUT             std_logic_vector(width-1 downto 0));
        END COMPONENT;

        component shifter
        GENERIC(width : positive:= 8;keep:positive:=8);
        PORT(
                input       : IN        std_logic_vector(width*2-1 downto 0);
                clk,clr                 :in std_logic;
                output: OUT             std_logic_vector(width-1 downto 0));
        END component;

        component reg
        generic(width:positive:=8);
        port(
                clk,clr:in std_logic;
                input: in std_logic_vector(width-1 downto 0);
                output: out std_logic_vector(width-1 downto 0));
        end component;

        component in_switch
        GENERIC(width : positive:= 8);
        PORT(
                firstline,secondline: IN            std_logic_vector(width-1 downto 0);
                clk,enable: in std_logic;
                odd,even: OUT           std_logic_vector(width-1 downto 0));
        end component;

        component delay
        generic(width:positive:=8;len:positive:=1);
        port(
                clk,clr:in std_logic;
                input: in std_logic_vector(width-1 downto 0);
                output: out std_logic_vector(width-1 downto 0));
        end component;

        component delay_en
        generic(width:positive:=8;len:positive:=1);
        port(
                clk,clr,en:in std_logic;
                input: in std_logic_vector(width-1 downto 0);
                output: out std_logic_vector(width-1 downto 0));
        end component;

        component trunc
        GENERIC(width : positive;keep:positive;nbit:positive);
        PORT(
                input       : IN        std_logic_vector(width-1 downto 0);
                clk,clr     : in        std_logic;
```

```
                        output: OUT             std_logic_vector(keep-1 downto 0));
                END component;

                component switch
                GENERIC(width : positive);
                PORT(
                        input: IN   std_logic_vector(width-1 downto 0);
                        Sel: in std_logic;
                        output1, output2: OUT           std_logic_vector(width-1 downto 0));
                END component;

                component mux
                GENERIC(width : positive:= 8);
                PORT(
                        input1,input2: IN       std_logic_vector(width-1 downto 0);
                        Sel: in std_logic;
                        output: OUT             std_logic_vector(width-1 downto 0));
                END component;

                component T_FF
                port(
                        clk,NotEn:in std_logic;
                        Q: out std_logic);
                end component;
        END liftcomp;

        -- a signed adder
        LIBRARY ieee;
        use ieee.std_logic_1164.all;
        use ieee.std_logic_arith.all;
        use ieee.std_logic_signed.all;


        ENTITY adder IS
                GENERIC(width : positive:=16);
                PORT(
                        a, b                    : IN        std_logic_vector(width-1 downto 0);
                        s: OUT      std_logic_vector(width-1 downto 0));
        END adder;

        ARCHITECTURE beh OF adder IS

        BEGIN
                s<=a+b;

        END beh;

        -- A clock divider
        -- Hongyu Liao
        -- last update: feb12/2002
        library ieee;
        use ieee.std_logic_1164.all;
        use ieee.std_logic_arith.all;

        entity clk_gen is
        port(
                clk,En:in std_logic;
                clkx2,clkx4: inout std_logic);
        end clk_gen;

        architecture beh of clk_gen is
        signal FB1: std_logic;
        begin
                c1:process(clk)
                begin
                        if En='1' then
                                if rising_edge(clk) then
                                        clkx2<= not clkx2;
                                end if;
                        else
                        clkx2<='0';
                end if;
        end process;

        c2:process(clkx2)
        begin
                if En='1' then
                        if rising_edge(clk) then
                                clkx4<= not clkx4;
                        end if;
                else
                        clkx4<='0';
                end if;
        end process;
end beh;

-- A symple FIFO used as a Delay unit
-- Hongyu Liao
-- last update: 11/23/2001
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity delay is
generic(width:positive:=8;len:positive:=1);
port(
        clk,clr:in std_logic;
        input: in std_logic_vector(width-1 downto 0);
        output: out std_logic_vector(width-1 downto 0));
end delay;

architecture beh of delay is
        constant MAX: positive:=len;
        subtype depth is positive range 1 to MAX;--Row Length;
        type reg_array is array(depth) of std_logic_vector(width-1 downto 0);
        signal reg:reg_array;
begin
        fifo: process(clk)
        begin
                if rising_edge(clk) then
                        if clr='1' then
                                output<=(others=>'0');
                                for index in depth loop
                                        reg(index)<=(others=>'0');
                                end loop;
                        else
                                reg(1)<=input;
                                for index in depth loop
                                        if index<MAX then
                                                reg(index+1)<=reg(index);
                                        end if;
                                end loop;
                                output<=reg(MAX);
                        end if;
                end if;
        end process fifo;
end beh;


-- A symple FIFO used as a Delay unit
-- Hongyu Liao
-- last update: 11/23/2001
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity delay_en is
generic(width:positive:=8;len:positive:=1);
```

```
port(
        clk,clr,en:in std_logic;
        input: in std_logic_vector(width-1 downto 0);
        output: out std_logic_vector(width-1 downto 0));
end delay_en;

architecture beh of delay_en is
        constant MAX: positive:=len;
        subtype depth is positive range 1 to MAX;--Row Length;
        type reg_array is array(depth) of std_logic_vector(width-1 downto 0);
        signal reg:reg_array;
begin
        fifo: process(clk)
        begin
                if rising_edge(clk) then
                        if clr='1' then
                                output<=(others=>'0');
                                for index in depth loop
                                        reg(index)<=(others=>'0');
                                end loop;
                        else
                                if en='1' then
                                        reg(1)<=input;
                                                for index in depth loop
                                                        if index<MAX then
                                                                reg(index+1)<=reg(index);
                                                        end if;
                                                end loop;
                                        output<=reg(MAX);
                                end if;
                        end if;
                end if;
        end process fifo;
end beh;


-- a switch performs the lazy wavelet for double scan arch.
-- last update:11/26/2001
LIBRARY ieee;
use ieee.std_logic_1164.all;

ENTITY in_switch IS
        GENERIC(width : positive:= 8);
        PORT(
                firstline,secondline: IN         std_logic_vector(width-1 downto 0);
                clk,enable: in std_logic; --
                odd,even: OUT        std_logic_vector(width-1 downto 0));
END in_switch;

ARCHITECTURE beh OF in_switch IS
        signal Sel: std_logic:='0';
        signal secondlineDelay: std_logic_vector(width-1 downto 0);
begin

        process(clk)
        begin
                -- a T flip-flop
                if rising_edge(clk) then
                        if enable='1' then -- enable sets when data comes
                                Sel<=not Sel;
                                secondlineDelay<=secondline;
                                if Sel='0' then
                                        even<=firstline;
                                        odd<=secondlineDelay;
                                else
                                        odd<=firstline;
                                        even<=secondlineDelay;
                                end if;
                        else
                                Sel<='0'; --make sure the data shifting in correct order
```

```
                                                secondlineDelay<=(others=>'0');
                                                odd<=(others=>'0');
                                                even<=(others=>'0');
                                        end if;

                                end if;
                        end process;
end beh;


-- A MAC
-- Hongyu Liao, 10/20/2001
-- add shifter for the accumulater input, 1/12/02
-- Last update:1/12/2001

LIBRARY ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


ENTITY mac IS
GENERIC(width : positive:= 8; keep:positive:=8;add_sc:positive:=9;res_sc:positive:=1);
-- add_sc: number of bits shifted for the adder input
-- res_sc: number of bits shifted for the mac result
PORT(   clock,clear: in         std_logic;
        acc, mul,amp            : IN        std_logic_vector(width-1 downto 0);
        output: OUT             std_logic_vector(keep-1 downto 0));
END mac;

ARCHITECTURE str OF mac IS
        COMPONENT mult
        GENERIC(width : positive);
        PORT(
                a, b            : IN        std_logic_vector(width-1 downto 0);
                p: OUT    std_logic_vector(width*2-1 downto 0));
        END COMPONENT;

        COMPONENT adder
        GENERIC(width : positive);
        PORT(
                a, b            : IN        std_logic_vector(width-1 downto 0);
                s: OUT    std_logic_vector(width-1 downto 0));
        END COMPONENT;

        component trunc
        GENERIC(width : positive;keep:positive;nbit:positive);
        PORT(
                input           : IN        std_logic_vector(width-1 downto 0);
                clk,clr         : in        std_logic;
                output: OUT                 std_logic_vector(keep-1 downto 0));
        END component;

        signal zero_pad:std_logic_vector(add_sc-1 downto 0);
        signal sig_pad:std_logic_vector(width-1 downto 0);
        signal acc_int:std_logic_vector(width*2-1 downto 0);
        signal mul_out:std_logic_vector(width*2-1 downto 0);
        signal add_in_a,add_out:std_logic_vector(width*2-1 downto 0);

BEGIN
        zero_pad<=(others=>'0');
        --signal extension
        sig_pad<=(others=>acc(width-1));

        --shift acc
        acc_int<=sig_pad & acc;
        add_in_a<= acc_int(width*2-1-add_sc downto 0) & zero_pad;

        multiplier: mult
                generic map(width=>width)
```

```
                        port map(a=>amp,b=>mul,p=>mul_out);

                add: adder
                        generic map(width=>width*2)
                        port map(a=>add_in_a,b=>mul_out,s=>add_out);

                --truncate the result
                trunc_reg: trunc
                        GENERIC map(width=> width*2,keep=>keep,nbit=>res_sc)
                        port map(input=>add_out,clk=>clock,clr=>clear,output=>output);

        END str;

        -- MAC library
        -- Hongyu Liao, 10/20/2001
        -- Last update:11/12/2001

        library ieee;
        use ieee.std_logic_arith.all;
        use ieee.std_logic_1164.all;
        use ieee.std_logic_signed.all;

        PACKAGE mac_comp IS
                COMPONENT mult
                        GENERIC(width : positive:= 8);
                        PORT(
                                a, b        : IN        std_logic_vector(width-1 downto 0);
                                p: OUT    std_logic_vector(width*2-1 downto 0));
                END COMPONENT;

                COMPONENT adder
                        GENERIC(width : positive:= 8);
                        PORT(
                                a, b        : IN        std_logic_vector(width-1 downto 0);
                                s: OUT    std_logic_vector(width-1 downto 0));
                END COMPONENT;

                component shifter
                GENERIC(width : positive:= 8;keep:positive:=8);
                PORT(
                        input       : IN        std_logic_vector(width*2-1 downto 0);
                        clk,clr                 :in std_logic;
                        output: OUT             std_logic_vector(keep downto 0));
                END component;

        END mac_comp;

        -- A signed multiplier
        -- Hongyu Liao, 10/20/2001
        -- Last update: 22/20/2001
        LIBRARY ieee;
        use ieee.std_logic_1164.all;
        use ieee.std_logic_arith.all;
        use ieee.std_logic_signed.all;

        ENTITY mult IS
                GENERIC(width : positive:= 16);
                PORT(
                        a, b        : IN        std_logic_vector(width-1 downto 0);
                        p: OUT    std_logic_vector(width*2-1 downto 0));
        END mult;

        ARCHITECTURE beh OF mult IS
        BEGIN
                p<=a*b;
        END beh;

        -- a switch detouring the data flow.
        -- created 1/23/02
        -- last update:1/23/2002
```

```
LIBRARY ieee;
use ieee.std_logic_1164.all;

ENTITY mux IS
	GENERIC(width : positive:= 8);
	PORT(
		input1,input2: IN	std_logic_vector(width-1 downto 0);
		Sel: in std_logic;
		output: OUT		std_logic_vector(width-1 downto 0));
END mux;

ARCHITECTURE beh OF mux IS
begin
	output<= input1 when Sel='0' else
		input2 when Sel='1' else
		(others=>'Z');
end beh;

-- A D flipflop with syn-clear
-- Hongyu Liao
-- last update: 11/22/2001
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity reg is
generic(width:positive:=8);
port(
	clk,clr:in std_logic;
	input: in std_logic_vector(width-1 downto 0);
	output: out std_logic_vector(width-1 downto 0));
end reg;

architecture beh of reg is
begin
	process(clk)
	begin
		if rising_edge(clk) then
			if clr='1' then
				output<=(others=>'0');
			else
				output<=input;
			end if;
		end if;
	end process;
end beh;

-- A simplified shift register
-- Hongyu Liao, 11/29/2001
-- Last update:11/12/2001
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;

ENTITY sh_reg IS
	GENERIC(width : positive:= 8);
	PORT(
		input	: IN	std_logic_vector(width-1 downto 0);
		clk,clr	: in	std_logic;
		num: in positive:=2;
		output: OUT	std_logic_vector(width-1 downto 0));
END sh_reg;

ARCHITECTURE beh OF sh_reg IS
	signal zeros: std_logic_vector(width-1 downto 0):=(others=>'0');
BEGIN
	process(clk)
	begin
		if rising_edge(clk) then
			if clr='1' then
```

```
                                        output<=(others=>'0');
                              else
                                        output<=input(width-num-1 downto 0) & zeros(num-1 downto 0);
                              end if;
                    end if;
          end process;

END beh;

-- A simplified right shift only register
-- Hongyu Liao, 10/20/2001
-- Last update:11/9/2001
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

ENTITY shift IS
          GENERIC(width : positive:= 8;rsh:positive:=4);
          PORT(
                    input       : IN        std_logic_vector(width-1 downto 0);
                    output: OUT             std_logic_vector(width-1 downto 0));
END shift;

ARCHITECTURE beh OF shift IS
signal fac:std_logic_vector(width-1 downto 0);
BEGIN
                    fac<="0010000000000000";
                    output<= sra 1;
END beh;

-- A simplified right shift only register
-- Hongyu Liao, 10/20/2001
-- change range,1/9
-- Last update:1/9/2002
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
library lib;
use lib.datalib.all;

ENTITY shifter IS
          GENERIC(width : positive:= 16;keep:positive:=16);
          PORT(
                    input       : IN        std_logic_vector(width*2-1 downto 0);
                    clk,clr     : in        std_logic;
                    output: OUT             std_logic_vector(keep-1 downto 0));
END shifter;

ARCHITECTURE beh OF shifter IS
BEGIN
process(clk)
begin
          if rising_edge(clk) then
                    if clr='1' then
                              output<=(others=>'0');
                    else
                    -- scale the partial result back
                              output<=input(width*2-1-(width-Factor) downto width*2-keep-(width-Factor));
                    end if;
          end if;
end process;

END beh;

-- a switch detouring the data flow.
-- created 1/23/02
-- last update:1/23/2002
LIBRARY ieee;
use ieee.std_logic_1164.all;
```

```
ENTITY switch IS
        GENERIC(width : positive:= 8);
        PORT(
                input: IN   std_logic_vector(width-1 downto 0);
                Sel: in std_logic;
                output1, output2: OUT            std_logic_vector(width-1 downto 0));
END switch;

ARCHITECTURE beh OF switch IS
begin
        output1<= input when Sel='0' else
                (others=>'0');
        output2<= input when Sel='1' else
                (others=>'0');
end beh;

-- A T flipflop
-- Hongyu Liao
-- last update: 1/24/2002
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity T_FF is

port(
        clk,NotEn:in std_logic;
        Q: out std_logic);
end T_FF;

architecture beh of T_FF is
signal FB: std_logic;
begin
        process(clk)
        begin
                if NotEn='0' then
                        if rising_edge(clk) then
                                FB<= not FB;
                        end if;
                else
                        FB<='0';
                end if;
        end process;
        Q<=FB;
end beh;

-- A register keeps the truncated result
-- input: 32bit;output:16bit
-- Hongyu Liao, 1/12/2002
-- change range
-- Last update:1/12/2002
LIBRARY ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--library lib;
--use lib.datalib.all;

ENTITY trunc IS
        GENERIC(width : positive:= 32;keep:positive:=16;nbit:positive:=1);
        -- nbit:number of bits to be shifted
        PORT(
                input      : IN      std_logic_vector(width-1 downto 0);
                clk,clr    : in      std_logic;
                output: OUT          std_logic_vector(keep-1 downto 0));
END trunc;

ARCHITECTURE beh OF trunc IS
BEGIN
```

```
process(clk)
begin
        if rising_edge(clk) then
                if clr='1' then
                        output<=(others=>'0');
                else
                        output<=input(keep-1+nbit downto nbit);
                end if;
        end if;
end process;

END beh;
```