

UNIVERSITY OF CALGARY

Architectures for the Finite Ridgelet Transform

by

Choudhury Ashiq Rahman

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JANUARY, 2004

© Choudhury Ashiq Rahman 2004

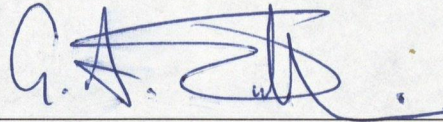
## FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Architectures for the Finite Ridgelet Transform" submitted by Choudhury Ashiq Rahman in partial fulfilment of the requirements of the degree of Master of Science.



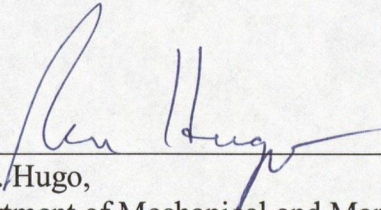
---

Supervisor, Dr. W. Badawy,  
Department of Electrical and Computer Engineering.



---

Dr. G. Jullien,  
Department of Electrical and Computer Engineering.



---

Dr. R. Hugo,  
Department of Mechanical and Manufacturing Engineering.

Date: Jan 23, 2003

---

## Abstract

---

Finite ridgelet transform (FRIT) has emerged very recently as a prospective transform for the next generation image compression standards. It is particularly suitable for natural images with lots of edges, where it proves its superiority over the 2-D discrete wavelet transform (DWT) by preserving the edge modeling of the image. There is no VLSI architecture for FRIT in the literature so far due to its relatively new introduction. Thus, this thesis introduces two original architectures for the finite ridgelet transform. The proposed architectures are prototype of FRIT for 7x7 block size images. The proposed architectures are coded in Verilog HDL, simulated by ModelSim and synthesized by Xilinx ISE development tools. The performance analysis of the two proposed architectures shows significant improvement over a direct implementation of the FRIT algorithm for real time applications.



---

## Acknowledgements

---

First of all, I would like to express my gratitude to my M.Sc thesis supervisor Dr. Wael Badawy for introducing me to this exciting world of image compression. His continuous guidance, encouragement and superintended support in course of my research work provided me a great deal of confidence. He also provided me with an environment conducive to learning and quality research.

I am very grateful to Dr. Graham Jullien, Department of Electrical and Computer Engineering and Dr. Ron Hugo, Department of Mechanical and Manufacturing Engineering for examining the thesis meticulously and giving their valuable comments.

I would also like to thank all the researchers of the Laboratory for Integrated Video Systems (LIVS). My special thanks to Mr. Mehboob Alam for interesting discussions and help during this research.

Last but not least, I would like to thank my parents, and my loving sister for their patience, affection, endless love, understanding, and constant support.



*To my family*

---

# Table of Contents

---

<b>Approval Page .....</b>	<b>ii</b>
<b>Abstract.....</b>	<b>iii</b>
<b>Acknowledgements .....</b>	<b>iv</b>
<b>Table of Contents .....</b>	<b>vi</b>
<b>List of Tables .....</b>	<b>viii</b>
<b>List of Figures.....</b>	<b>ix</b>
<b>List of Abbreviations .....</b>	<b>xi</b>
<b>Chapter 1: Introduction .....</b>	<b>1</b>
1.1 Introduction.....	1
1.2 Video and Image Compression – Why? .....	3
1.2.1 Principles of Compression .....	4
1.2.2 Compression Techniques .....	4
1.3 Research Objective .....	5
1.4 Thesis Outline .....	6
<b>Chapter 2: Brief Review of Image Compression .....</b>	<b>7</b>
2.1 Introduction.....	7
2.2 Transform.....	8
2.2.1 Discrete Cosine Transform (DCT).....	9
2.2.2 Wavelet Transform .....	9
2.2.2.1 Continuous Wavelet Transform (CWT) .....	11
2.2.2.2 Discrete Wavelet Transform (DWT) .....	13
2.2.3 Finite Ridgelet Transform (FRIT) .....	14
2.3 Quantization.....	15
2.4 Entropy Coding.....	15
2.5 Performance Measures.....	15
2.6 Summary .....	17
<b>Chapter 3: The Finite Ridgelet Transform .....</b>	<b>18</b>
3.1 Introduction.....	18
3.2 Ridgelet Transform .....	19
3.2.1 Continuous Ridgelet Transform (CRT) .....	20
3.2.2 Finite Ridgelet Transform (FRIT) .....	22
3.2.2.1 Finite Radon Transform (FRAT).....	23
3.2.2.2 Optimal Ordering of FRAT Coefficients.....	24

3.2.2.3	1-D Discrete Wavelet Transform (DWT) .....	27
3.3	Architectures for 1-D DWT .....	29
3.4	Distributed Arithmetic .....	31
3.4.1	DA Principle .....	32
3.5	Summary .....	33
<b>Chapter 4:</b>	<b>The Proposed FRIT Architectures .....</b>	<b>34</b>
4.1	Introduction.....	34
4.2	FRAT Architecture .....	35
4.2.1	Algorithm.....	35
4.2.2	Proposed FRAT Architecture.....	40
4.2.3	Proposed Memoryless FRAT Architecture.....	42
4.3	Proposed DWT Architecture.....	44
4.4	The FRIT Prototype .....	49
<b>Chapter 5:</b>	<b>Performance Analysis .....</b>	<b>51</b>
5.1	Introduction.....	51
5.2	Simulation Results .....	52
5.2.1	FRIT Architecture with Memory .....	52
5.2.2	Memoryless FRIT Architecture .....	54
5.3	Synthesis Results .....	58
<b>Chapter 6:</b>	<b>Conclusions and Future Work .....</b>	<b>62</b>
6.1	Summary of Accomplishments.....	62
6.2	Recommendations for Future Work.....	63
<b>References</b> .....		<b>65</b>
<b>Appendix A:</b>	<b>MATLAB Codes.....</b>	<b>69</b>
<b>Appendix B:</b>	<b>Verilog HDL Codes .....</b>	<b>75</b>
<b>Appendix C:</b>	<b>Publications and Presentations .....</b>	<b>123</b>



---

## List of Tables

---

Table 4.1: Normal vectors for 7x7 FRAT.....	35
Table 4.2: Radon coefficients of eight Radon slices; the pixels locations are given in (row, column) format for the 7x7 image block shown in Figure 4.5.....	38
Table 4.3: Values of $m$ , $r$ and $c$ for the Radon slices .....	41
Table 4.4: Values of $m1$ , $m2$ , $m3$ , $m4$ , $m5$ , $m6$ , $m7$ and $c$ for the Radon slices .....	43
Table 4.5: Operation performed by the adder butterfly network .....	47
Table 4.6: Assignments of inputs of the adder compressor array .....	48
Table 5.1: I/O signal description of the FRIT module with memory.....	53
Table 5.2: I/O signal description of the memoryless FRIT module.....	54
Table 5.3: Comparison of PSNR of “Lena” image for different compression ratios .....	56
Table 5.4: Comparison of time required for transforming CIF and QCIF images with a core speed of 50MHz.....	57
Table 5.5: Synthesis results of the proposed architectures .....	58
Table 5.6: Comparison of number of components used in the architectures .....	58

---

## List of Figures

---

Figure 2.1: Block diagram of transform based image compression technique. ....	8
Figure 2.2: A wavelet function, $\psi_{a,b}(t)$ .....	10
Figure 2.3: Convolution operation of five sample wavelet (W) and signal samples (S).10	
Figure 2.4: A five-tap filter for five-sample wavelet. ....	11
Figure 2.5: Two level signal decomposition. ....	14
Figure 3.1: Reconstructed image using DWT and FRIT from 256 most significant coefficients, out of 65536 coefficients [12]. ....	19
Figure 3.2: Block diagram of ridgelet transform.....	20
Figure 3.3: A ridgelet function, $\psi_{a,b,\theta}(x_1, x_2)$ .....	21
Figure 3.4: Process flow diagram for computing FRIT .....	22
Figure 3.5: Lines for 7x7 FRAT. One line per slope has been shown in shaded gray color. For each slope, there would be six more lines parallel to the line shown in the figure.....	23
Figure 3.6: The set of normal vectors for $p = 7$ .....	26
Figure 3.7: Daubechies $D_4$ scaling and wavelet functions. ....	29
Figure 3.8: DWT architecture proposed by Knowles [28].....	30
Figure 3.9: DWT architectures proposed by Parhi et. al. (3-level) [30].....	30
Figure 3.10: DWT architecture proposed by Chang et. al. (3-level) [32] .....	31
Figure 4.1: Simplified block diagram of the proposed FRAT architectures .....	35
Figure 4.2: A 7x7 image matrix $f[i]$ .....	36
Figure 4.3: Pseudo code for computing Radon coefficient of the image matrix shown in Figure 4.2 .....	36
Figure 4.4: Lines of FRAT for 7x7 block size image. Coefficient's orders are signified by increasing gray level for each direction. ....	37

Figure 4.5: 7x7 image block showing the address of the pixel locations in (row, column) format.....	39
Figure 4.6: Proposed FRAT architecture with memory .....	40
Figure 4.7: Proposed memoryless FRAT architecture .....	43
Figure 4.8: Proposed DWT architecture.....	44
Figure 4.9: Low pass and high pass filter coefficients matrices.....	46
Figure 4.10: Delay line .....	46
Figure 4.11: Adder butterfly network.....	47
Figure 4.12: Parallel adders of adder compressor array .....	48
Figure 4.13: Proposed FRIT architecture with memory.....	49
Figure 4.14: Proposed memoryless FRIT architecture.....	50
Figure 5.1: I/O ports of the FRIT module with memory .....	52
Figure 5.2: Snapshot of ModelSim simulation of the FRIT architecture with memory. ....	53
Figure 5.3: I/O ports of the memoryless FRIT module .....	54
Figure 5.4: Snapshot of ModelSim simulation of the memoryless architecture .....	55
Figure 5.5: Original and reconstructed “Lena” images of different compression.....	56
Figure 5.6: Plot of percentage of retained coefficients vs. PSNR.....	57
Figure 5.7: Xilinx ECS view of the proposed FRIT architecture with memory .....	59
Figure 5.8: Xilinx ECS view of the proposed memoryless FRIT architecture .....	59



---

## List of Abbreviations

---

1-D	One Dimension
2-D	Two Dimension
CIF	Common Intermediate Format
CRT	Continuous Ridgelet Transform
CWT	Continuous Wavelet Transform
DA	Distributed Arithmetic
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DHT	Discrete Hartley Transform
DSP	Digital Signal Processing
DST	Discrete Sine Transform
DWT	Discrete Wavelet Transform
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FRAT	Finite Radon Transform
FRIT	Finite Ridgelet Transform
HDL	Hardware Description Language
HT	Haar Transform
HVS	Human Visual System
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
KLT	Karhunen Løeve Transform
LUT	Look Up Table
MPEG	Motion Picture Experts Group
MSE	Mean Square Error
NEDA	New Distributed Arithmetic
PSNR	Peak Signal to Noise Ratio
PSTN	Public Switched Telephone Network
QCIF	Quarter Common Intermediate Format
RAM	Random Access Memory
RGB	Red-Green-Blue
RMSE	Root Mean Square Error
ROM	Read Only Memory
VLSI	Very Large Scale Integration
WTP	Wavelet Transform Processor

---

# Chapter 1

## *Introduction*

---

### **1.1 Introduction**

Over the last two decades, there has been an enormous increase in the storage and transmission of information. Digital video and still images are the most important medium for communicating visual information. Unfortunately, this requires large channel bandwidth for transmission and large storage space for archival. In addition, due to the large number of pixels in a high resolution image, manipulation of digital images is feasible only with low complexity algorithms. Because of this, reliable and fast compression techniques are desirable. Most of today's compression techniques are dependent on a mathematical process called transform. The major setback in using the transform is computational time. Even with the advent of the digital computer, the techniques to reduce computational time were generally unknown until 1965 when James

W. Cooley and John W. Tukey published their mathematical algorithm which has become known as the fast Fourier transform (FFT) [1][2]. This was a revolution in the field of digital signal processing (DSP). After that, over the years a wide range of orthogonal transforms have come into existence. Among them, the two most popular transforms used in the field of still image and video compression are the discrete cosine transform (DCT) [3] and the discrete wavelet transform (DWT) [4].

To meet the growing need for image compression and to ensure compatibility, the International Organization for Standardization (ISO) proposed the JPEG [5] and the MPEG [6] standards for image and video compression respectively almost a decade ago. These standards are based on discrete cosine transform (DCT) of small image blocks and are very effective in reducing the spatial redundancy in images. However, DCT coding has the drawbacks of blockiness and aliasing distortion in the reconstructed image at high compression ratios. Recently, JPEG2000 [7] has been proposed, which is the new generation standard of JPEG. This new standard is based on the discrete wavelet transform (DWT). DWT was first applied to image coding by Mallat [8][9]. The implementation of DWT is very similar to subband coding. However, subband coding emphasizes on improving the frequency selectivity of the filters whereas wavelet emphasizes the smoothness properties of the basis functions. Combining the advantages of multiresolution analysis and transform coding, wavelet offers a wide variety of useful features [10] and these are:

- Computational complexity of  $O(N)$ ; here  $N$  is the number of pixels.
- Efficient VLSI implementation.



- Reconstructed images without blocking artifacts.
- Lower aliasing distortion.
- Inherent scalability.

In this family of transform, a new member was introduced in 1999 by Candes and Donoho [11] of which a discrete version has been proposed very recently by Do and Vetterli [12] and is called the finite ridgelet transform (FRIT). FRIT preserves the edge modeling, which allows a better restoration mechanism for edges. In contrast to the 2-D DWT, which leads to a poor performance especially when the image has many edges, FRIT shows a significant visual enhancement. Surveying the literature, no VLSI implementation has been found so far for the finite ridgelet transform. In this thesis, we present two original VLSI architectures of the finite ridgelet transform for the first time.

## 1.2 Video and Image Compression – Why?

The term image compression refers to the process of reducing the amount of image data required to represent an image while maintaining an acceptable subjective quality. This is done in order to meet a certain bit rate requirement. Video as a sequence of video frames involves a huge amount of data. For an example, if we assume the video frame of common intermediate format (CIF) [13] resolution, which is 352 x 288 pixels, then to achieve real-time full motion video broadcasting we need a channel bandwidth of  $352 \times 288 \times 8 \times 3 \times 30 = 72,990,720$  bits per second (bps). Here, “3” is the number of colors of RGB color space and “8” is the number of bits to represent each color for each pixel. If we compare this bandwidth requirement with the present public switched telephone

network (PSTN) modem, which can operate at a maximum bit rate of 56,600 bps, then the revealed fact is that we need to compress the video data by at least 1290 times in order to accomplish video transmission over this medium.

### 1.2.1 Principles of Compression

Video and image compression is not only possible but also feasible because of the following two redundancies present in images. By eliminating these redundancies, we can achieve video and image compression [13].

**i. Statistical redundancy** – It is the correlation between neighboring pixels of an image frame (spatial redundancy) and between the pixels from successive frames in a temporal image or video sequence (temporal redundancy).

**ii. Psychovisual redundancy** – It is the redundancy that originates from the characteristics of the human visual system (HVS).

In the case of still image compression, which is the aim of this research, spatial redundancy is reduced as much as possible in order to achieve a low bit rate suitable for the intended application. But the human visual system can also be exploited to further increase the compression ratio.

### 1.2.2 Compression Techniques

Most of the image compression techniques are based on the concept of information theory first formulated by Shannon [14] and can be broadly classified into two groups:

**i. Lossless compression techniques** – These are Huffman coding, run-length coding and arithmetic coding, which preserve all the information present in an image, i.e.,

the original image is exactly recoverable. There is considerable interest in lossless techniques, especially in applications which require very high fidelity reconstructed images such as medical imaging.

**ii. Lossy compression techniques** – Predictive coding, transform, subband coding, vector quantization and fractal coding fall into this group, which provide a better coding performance compared to lossless techniques. Lossless techniques usually result in a low compression ratio (typically 2 to 3). Because of this, lossless compression techniques are not employed when a high compression ratio is required. In lossy compression techniques the objective is therefore to reduce the bit rate while maintaining some constraints on the image quality.

One of the computational stages in this lossy compression technique is the transform. There are two possible methods of implementation of this transform and these are software and hardware implementation. But a software encoder may not meet the real time processing requirement especially when the mathematical transform is very complex. So hardware implementation is the only solution to enhance the performance.

### **1.3 Research Objective**

The objective of the research behind this thesis is to present VLSI architectures for the finite ridgelet transform which has emerged as a prospective image compression technique for the next generation standards.

The motivation behind this research is based on the performance of the finite ridgelet transforms that have proven to be superior over the discrete wavelet transform (DWT) in the case with images that have many edges. The FRIT is similar to the majority



of other mathematical transformations where the direct implementation suffers from high computational cost. Though its computational complexity is higher than the DCT or DWT and requires much larger resource allocation for implementation in FPGAs, the introduction of fast and low cost VLSI techniques may prove the feasibility of the FRIT in the near future.

## 1.4 Thesis Outline

This thesis is organized in six chapters and three appendices. In Chapter 2, a brief review of video and image compression is given. Chapter 3 presents the finite ridgelet transform (FRIT). The theory and motivation is covered in detail in this chapter. Then the proposed architecture for FRIT follows in Chapter 4. Two architectures for this transform have been presented in Chapter 4. Simulation and synthesis results are shown in Chapter 5 where a performance analysis of the proposed designs, in terms of power, throughput, used components and real time analysis of the architectures is provided. A Comparison between the proposed architectures is also given in Chapter 5. Chapter 6 concludes the thesis by summarizing the accomplishments of the research and giving some recommendations for future work.

The appendices include additional information that complements the work presented in the thesis body. Appendix A and Appendix B contain the MATLAB and Verilog HDL codes used in modeling, simulation and synthesis of the proposed FRIT architectures. Finally, the thesis ends with Appendix C, which contains the list of publications and seminars that have resulted during this M.Sc research work.

---

# Chapter 2

## *Brief Review of Image Compression*

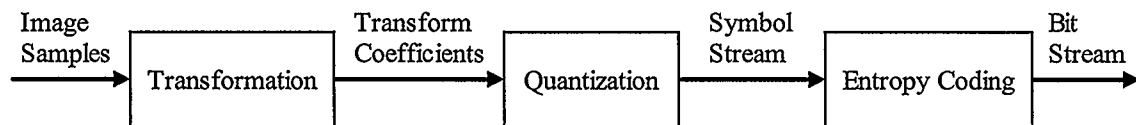
---

### 2.1 Introduction

Due to high degree of pixel correlation in the spatial domain, implementation of the lossless algorithms practically yields relatively low compression ratios. This correlation means that the energy is spread out over the entire image. If the energy of a pixel is directly related to its importance, then in the pixel domain there are many important coefficients making the selection for elimination extremely difficult. For this reason, most of the image compression literatures investigate transform based methods for compression. These systems use a reversible linear transform in combination with lossy techniques to achieve greater and more accurate compression than operating in the pixel domain. This improvement is due to the transform's ability to decorrelate the image data, thus packing the energy into a few significant coefficients. Karhunen Løve Transform

(KLT) [15] is considered the optimal transform for energy compaction, i.e., it places as much energy as possible in as few coefficients as possible. But it suffers from high computational cost. The KLT is a linear transform where the basis functions are taken from the statistics of the signal, and can thus be adaptive. The DCT is an approximation for the KLT with much less computation than the KLT. This is the reason why the DCT is so popular in video compression applications.

Three major steps comprise a typical transform based compression algorithm. These are transformation, quantization and entropy coding [15][16][17], as shown in Figure 2.1.



*Figure 2.1: Block diagram of transform based image compression technique.*

## 2.2 Transform

A transform is a map, or function, from an N-dimension space to an M-dimension space [18]. It creates a new set of values according to the definition used. No compression or information loss occurs in this step, but rather in the stages that follow. In addition to decorrelating image data for image compression, transforms such as the DCT and DWT are also used in signal processing to display characteristics of the signal not visible in the original representation. This allows for easier techniques to analyze and represent the signal.

### 2.2.1 Discrete Cosine Transform (DCT)

The DCT can be defined for any rectangular array of pixels, but in image compression the basic block is generally an 8x8 array or 64 pixels. The equation of the DCT, as used in JPEG and similar compression schemes, is given in equation 2.1.

$$F(u, v) = \frac{1}{4} C_u C_v \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right) \quad (2.1)$$

where,  $x$  and  $y$  are indices into an 8x8 array of samples, and  $u$  and  $v$  are indices into an 8x8 array of DCT coefficients.  $C_u$  and  $C_v$  are defined by

$$C_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{otherwise} \end{cases} \quad \left| \quad C_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } v = 0 \\ 1 & \text{otherwise} \end{cases} \quad (2.2)$$

So, a special case happens when both  $u$  and  $v$  are zero, which means that the top left DCT coefficient, according to equations 2.1 and 2.2, is

$$F(0,0) = \frac{1}{8} \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \quad (2.3)$$

So, the first DCT coefficient represents the DC value of all 64 samples.

### 2.2.2 Wavelet Transform

Wavelet means small wave and can be viewed as a burst of energy with a dominant frequency. Such a wavelet is shown in Figure 2.2. As with the Fourier transform, if this wavelet is multiplied with a signal and integrated, the result would give a nonzero coefficient. The multiplication by the wavelet picks out the detail from the signal.

Obviously, the result would depend on the placement of the wavelet on the original signal.

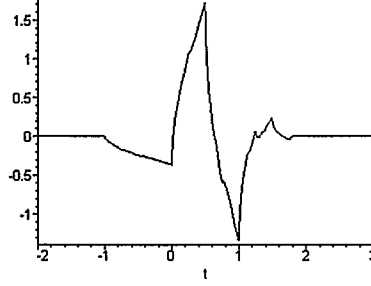


Figure 2.2: A wavelet function,  $\psi_{a,b}(t)$

So in the wavelet transform, the weighted moving average of the signal is calculated with the wavelet, i.e, the sequence of discrete values of the wavelet, flipped back to the front.

This process is known as convolution and is depicted in Figure 2.3.

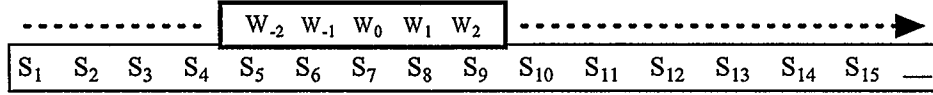


Figure 2.3: Convolution operation of five sample wavelet ( $W$ ) and signal samples ( $S$ ).

In Figure 2.3, the wavelet is shown operating on sample  $S_7$ , and the output of the convolution is a new value  $S_7'$

$$S_7' = W_{-2} \cdot S_5 + W_{-1} \cdot S_6 + W_0 \cdot S_7 + W_1 \cdot S_8 + W_2 \cdot S_9 \quad (2.4)$$

In this way, the convolution operation over the entire set of the signal samples results in a new set of values ( $S_1', S_2', S_3', \dots$ ). This process is similar to the operation of a digital filter of which the classical representation is shown in Figure 2.4. Here, the samples of the input signal,  $S$ , pass through four delays,  $Z^{-1}$ , equal to the sample interval. The output

of this filter is  $S_7'$  when sample  $S_7$  is at the center of the filter. The wavelet shape shown in Figure 2.2 is the impulse response of the filter shown in Figure 2.4. So, the convolution technique gives us a system's output when an input signal and the systems impulse response is given.

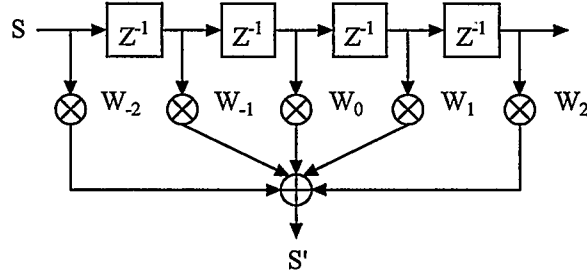


Figure 2.4: A five-tap filter for five-sample wavelet.

### 2.2.2.1 Continuous Wavelet Transform (CWT)

This idea of moving a wavelet over the image and picking out the detail shows how wavelets can give both frequency and location information. Now, let us take a look at the mathematical definition of the wavelet.

A function  $\psi(t)$  is called a mother wavelet if it satisfies the following properties [4][19]:

1. The function integrates to zero, or

$$\int_{-\infty}^{\infty} \psi(t) dt = 0 \quad (2.5)$$

2. The function is square integrable i.e. the function has finite energy

$$\int_{-\infty}^{\infty} |\psi(t)|^2 dt < \infty \quad (2.6)$$

3. The function satisfies the admissibility condition

$$C = \int_{-\infty}^{\infty} \frac{|\psi(\omega)|^2}{\omega} d\omega \quad 0 < C < \infty \quad (2.7)$$

The first property suggests a signal that oscillates and has a wavy appearance, hence the name “wavelet”. The second property suggests that for the wavelet most of the energy should be contained in a finite duration, thus giving rise to the locality property. The third property ensures the existence of an inverse transform.  $\psi(\omega)$  is the Fourier transform of  $\psi(t)$ . When a mother wavelet  $\psi(t)$  is found, the wavelet transform of a function  $f(t)$  is defined as

$$F(a, b) = \int_{-\infty}^{\infty} f(t) \psi_{a,b}(t) dt \quad (2.8)$$

Where the mother wavelet  $\psi_{a,b}(t)$ , with respect to the variable  $a$  and  $b$ , is defined as

$$\psi_{a,b}(t) = a^{-1/2} \psi\left(\frac{t-b}{a}\right) \quad (2.9)$$

Here  $a$  and  $b$  denote the scale and translation parameter of the wavelet, respectively. The  $a^{-1/2}$  term in equation 2.9 is the normalization term which ensures that the energy stays the same for all values of  $a$  and  $b$ . If  $a > 1$ ,  $\psi_{a,b}(t)$  stretches along the time axis and if  $0 < a < 1$ ,  $\psi_{a,b}(t)$  contracts along the time axis. On the other hand, by changing the translation parameter,  $b$ , the location of the wavelet with respect to the signal can be changed. So by changing  $a$ , different frequency ranges can be covered and by changing  $b$ , the length of the signal for analysis can be covered. These translated and scaled versions

of the mother wavelet constitute the basis function and are referred to as daughter wavelets.

### 2.2.2.2 Discrete Wavelet Transform (DWT)

The wavelet transform defined in equation 2.8 is highly redundant since here the one variable function  $\psi_{a,b}(t)$  is represented as a function of two variables,  $a$  and  $b$ . This redundancy can be removed by discretizing  $a$  and  $b$ , such that the dilation and translation parameters  $a$  and  $b$  take the form  $a = 2^k$  and  $b = 2^k l$  ( $k$  and  $l$  are non-negative integers), respectively. This method of sampling  $(a, b)$  coordinates is called dyadic sampling as the consecutive values of the discrete scale differ by a factor of two. The DWT of this type results in a non-redundant wavelet representation.

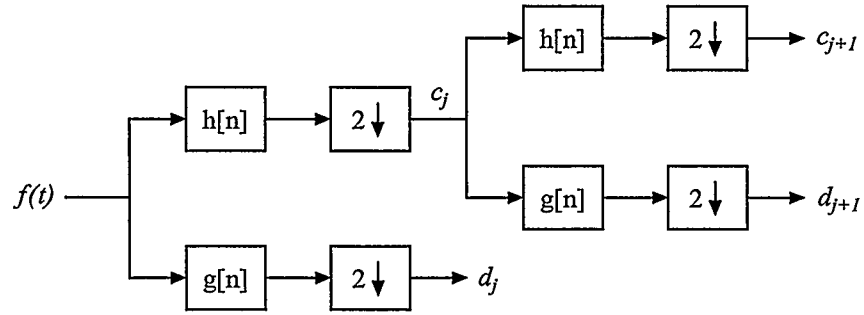
In 1989, Mallat [9] utilized the fact that the basis functions are dilated and translated versions of the mother wavelet to show that the wavelet coefficients of any scale or resolution could be computed from the wavelet coefficients of the previous stage, which is known as the Mallat's tree algorithm. This is the basic foundation of the implementation of the DWT and can be expressed by the following two equations

$$\begin{aligned} c_{j+1,k} &= \sum_m c_{j,m} h[m-2k] \\ d_{j+1,k} &= \sum_m c_{j,m} g[m-2k] \end{aligned} \quad (2.10)$$

where  $c_{p,q}$  and  $d_{p,q}$  are the low-pass or scaling coefficients and the high-pass or wavelet coefficients of  $p$ th scale and  $q$ th location respectively.  $h[n]$  and  $g[n]$  are the low-pass and high-pass filter coefficients corresponding to the mother wavelet respectively. Figure 2.5 shows a two level decomposition of signal  $f(t)$ . The symbol  $2 \downarrow$  stands for down-



sampling by a factor of two for decimating the filter results (i.e., to remove redundancies). The signal produced from the low-pass filter is called the approximation signal and is a smoothed version of the original and the high-pass filter produces the detailed signal which contains the high frequencies or sharp edges of the input signal.



*Figure 2.5: Two level signal decomposition.*

### 2.2.3 Finite Ridgelet Transform (FRIT)

In 1999, the ridgelet transform [11] was introduced as a sparse expansion for functions on continuous spaces that are smooth away from discontinuities along lines. Inspired by the performance of this ridgelet transform, Do and Vetterli proposed an orthonormal version of the ridgelet transform for discrete and finite size images in 2003, which is known as the finite ridgelet transform (FRIT) [12]. The FRIT preserves edge modeling, which allows a better restoration mechanism for edges. In contrast to the 2-D DWT, which leads to a poor performance especially when the image has many edges, the FRIT shows a significant visual enhancement. The finite ridgelet transform (FRIT), which is the main topic of this thesis, is discussed in detail in Chapter 3.

## 2.3 Quantization

If the transform step is effective then the energy of the signal is concentrated into the low-frequency coefficients while many of the remaining coefficients are small. The lossy compression step discards near-zero coefficients and rounds the remaining components to a smaller representative set of integers. Thus, the output of the quantization stage is a stream of small integers, many of which are zero, called the symbol stream. Techniques for accomplishing this task range from very simple to highly complex and include uniform quantization, scalar quantization and vector quantization [13].

## 2.4 Entropy Coding

This final stage of compression is a lossless step, which removes redundancies and provides a final measure of compaction. Entropy coding considers how often each symbol occurs in the stream and replaces the stream with a more efficient alphabet based on these occurrences. Symbols that appear more frequently are represented with shorter code words than rare symbols. Two of the more commonly used entropy coding methods are Huffman coding [20] and arithmetic coding [21].

## 2.5 Performance Measures

In order to measure the integrity of a compression algorithm, several performance measures are used. The first measure is the compression ratio, which measures the amount of compression obtained. It compares the original and compressed file size by using the following equation

$$CR = \frac{\text{Size of the original image}}{\text{Size of the compressed image}} \quad (2.11)$$

The bit rate offers an alternative measure for determining the amount of compression using the following equation

$$\text{bit rate} = \frac{\text{Compressed image size (bits)}}{\text{Number of pixels in original image}} \quad (2.12)$$

For example, an image of size 512x512x8 compressed to 16,384 bytes has a compression ratio of 16:1 or a bit rate of 0.5 bpp (i.e bits per pixel).

The second measure, called the mean square error (MSE), represents the amount of error present in the reconstructed image. In other words, it measures how closely a reconstructed image resembles the original. The formula for MSE is

$$MSE = \frac{\sum [f(i,j) - F(i,j)]^2}{N^2} \quad (2.13)$$

where  $f(i,j)$  is the original source image of size  $N \times N$  and  $F(i,j)$  is the reconstructed image.

Finally, the peak signal to noise ratio (PSNR) is probably the most commonly used metric of image quality in the literature. Closely related to the MSE, it measures the quality of a reconstructed image compared with an original image. Reconstructed images with higher metrics are judged better, and two identical images would have an infinite PSNR. This measure is calculated by

$$PSNR = 20 \log_{10} \left( \frac{255}{RMSE} \right) \quad (2.14)$$

where RMSE is simply the square root of MSE. It is important to keep in mind that this measure has a limited relationship with the perceived errors noticed by the human visual

system. In fact, two images could have identical PSNR values but one may look better than the other. So, higher PSNR values do not always mean a perceptually better image.

## **2.6 Summary**

The purpose of this chapter was to briefly review the image and video compression techniques. The technique that leads to a lossy compression has been discussed. Lossy compression is the only way of achieving high compression ratio for video broadcasting. The discussion has been limited to a transform based lossy compression technique. Finally, the different performance measures for measuring the integrity of a compression algorithm have been presented.

---

# Chapter 3

## *The Finite Ridgelet Transform*

---

### 3.1 Introduction

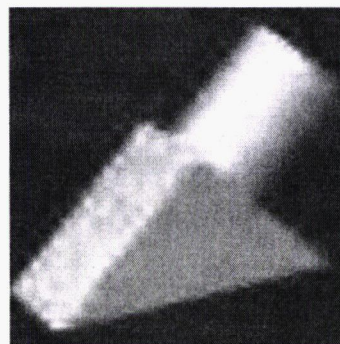
In 1999, the ridgelet transform was introduced as a sparse expansion for functions on continuous spaces that are smooth away from line discontinuities [11]. Inspired by the performance of this ridgelet transform, Do and Vetterli in 2003, proposed an orthonormal version of the ridgelet transform for discrete and finite size images, which is known as the finite ridgelet transform (FRIT) [12]. Their construction uses the finite Radon transform (FRAT) [22] as a building block and it has been shown that FRIT outperforms wavelet transforms in approximating and denoising images with straight edges.

The Radon transform [23] has long been used for many line detection applications within image processing, computer vision, and seismics. But it never drew much

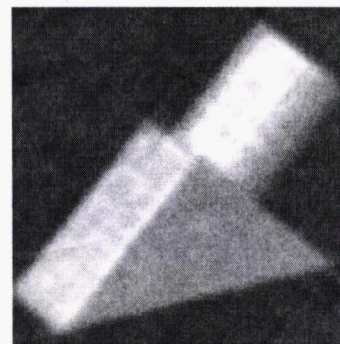
attention of researchers in the field of image compression until the finite Radon transform was introduced for image representation.

## 3.2 Ridgelet Transform

The ridgelet transform was proposed to overcome this weakness of wavelet transforms in 2-D. The wavelet transform has proved to be a good transform over the years mainly due to its strong performance for piecewise smooth functions in one dimension. However, in higher dimensions such as in 2-D, where singularities can be lines, or in 3-D, where singularities can be planes, the discrete wavelet transform does not provide good results. In essence, wavelets are good at catching zero dimensional or point singularities, but 2-D piecewise smooth signals of images may have one dimensional or line singularities; i.e, smooth regions are separated by edges and consequently, the DWT does not show good performance in reconstructing those edges. Figure 3.1 shows a reconstructed image using the DWT and the FRIT. From this figure, the smoothness along edges of the reconstructed image using the FRIT is apparent compared to the one using the DWT.



(a) Using DWT



(b) Using FRIT

*Figure 3.1: Reconstructed image using DWT and FRIT from 256 most significant coefficients, out of 65536 coefficients [12].*

The ridgelet transform is basically a conjunction of two transforms – the Radon transform and the wavelet transform, as shown in Figure 3.2. The idea is to map the line singularities into point singularities using the Radon transform and then to apply the wavelet transform. Since the wavelet transform can effectively handle the point singularities, the overall transform thus gives better performance than using only the wavelet transform in 2-D.

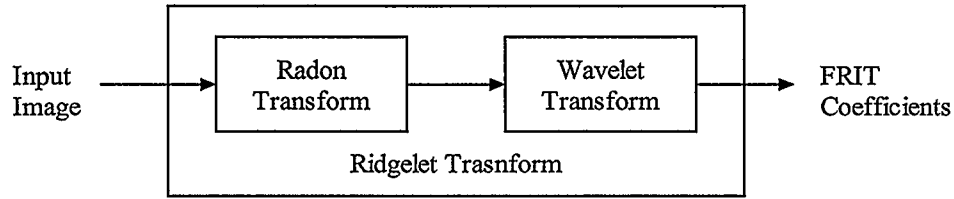


Figure 3.2: Block diagram of ridgelet transform

### 3.2.1 Continuous Ridgelet Transform (CRT)

The continuous ridgelet transform [11] of a bivariate integrable function  $f(x)$  is defined as

$$CRT_f(a, b, \theta) = \int_{\mathbb{R}^2} \psi_{a,b,\theta}(x) f(x) dx \quad (3.1)$$

where the ridgelets,  $\psi_{a,b,\theta}(x)$ , in 2-D are defined from a wavelet type function in 1-D,  $\psi(x)$  as

$$\psi_{a,b,\theta}(x) = a^{-1/2} \psi\left(\frac{x_1 \cos \theta + x_2 \sin \theta - b}{a}\right) \quad (3.2)$$

this function is constant along ridges  $x_1 \cos \theta + x_2 \sin \theta = \text{const}$  and wavelet transverses these ridges; hence the name “ridgelet”. So, a ridgelet can be thought of concatenating

1-D wavelets along lines, which actually motivated the use of ridgelets in image processing since, in images, point singularities are often joined together along edges or contours. As a result, the ridgelet transform can be very efficient in catching such singularities. Figure 3.3 shows an example of a ridgelet function, which is oriented at an angle  $\theta$ .

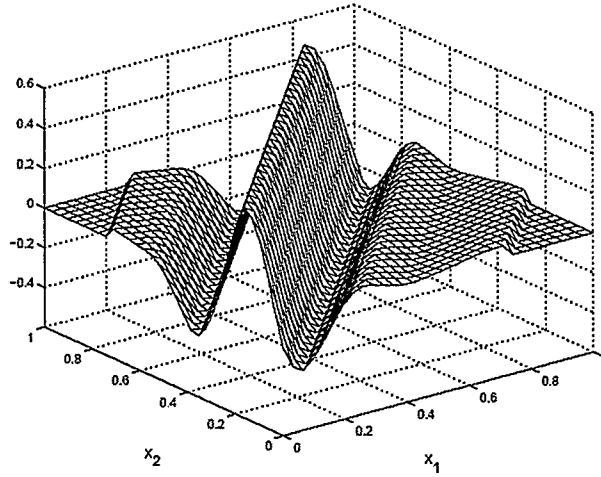


Figure 3.3: A ridgelet function,  $\psi_{a,b,\theta}(x_1, x_2)$

In 2-D, points and lines are related via the Radon transform, which is why equation 3.1 can be split into two, as shown in equations 3.3 and 3.4. Equation 3.3 is the Radon transform of the bivariate function,  $f(x)$ , and it produces slices (or projections) of Radon coefficients,  $R_f(\theta, t)$ . The ridgelet transform is the application of a 1-D wavelet transform to these slices, as shown in equation 3.4.

$$R_f(\theta, t) = \int_{\mathbb{R}^2} f(x) \delta(x_1 \cos \theta + x_2 \sin \theta - t) dx \quad (3.3)$$



$$CRT_f(a, b, \theta) = \int_{\mathbb{R}^2} \psi_{a,b}(t) R_f(\theta, t) dt \quad (3.4)$$

Here  $\psi_{a,b}(t)$  is the mother wavelet and is the same as defined in equation 2.9, in Chapter 2. For convenience, equation 2.9 is rewritten in the following as equation 3.5.

$$\psi_{a,b}(t) = a^{-1/2} \psi\left(\frac{t-b}{a}\right) \quad (3.5)$$

Here  $a$  and  $b$  denote the scale and translation parameters of the wavelet, respectively.

### 3.2.2 Finite Ridgelet Transform (FRIT)

The finite ridgelet transform is the discrete ridgelet transform applied to finite length signals. The finite ridgelet transform can be computed by using the finite Radon transform (FRAT) on the input signal samples and then applying the 1-D discrete wavelet transform (DWT) to the FRAT slices produced in the first stage, as shown in Figure 3.4.

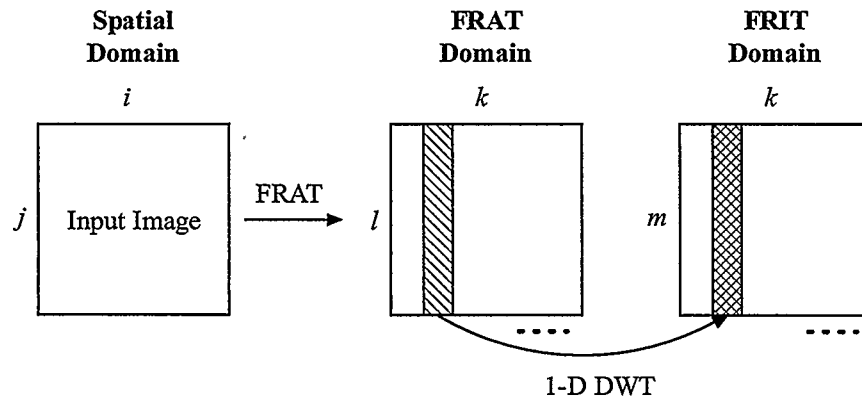
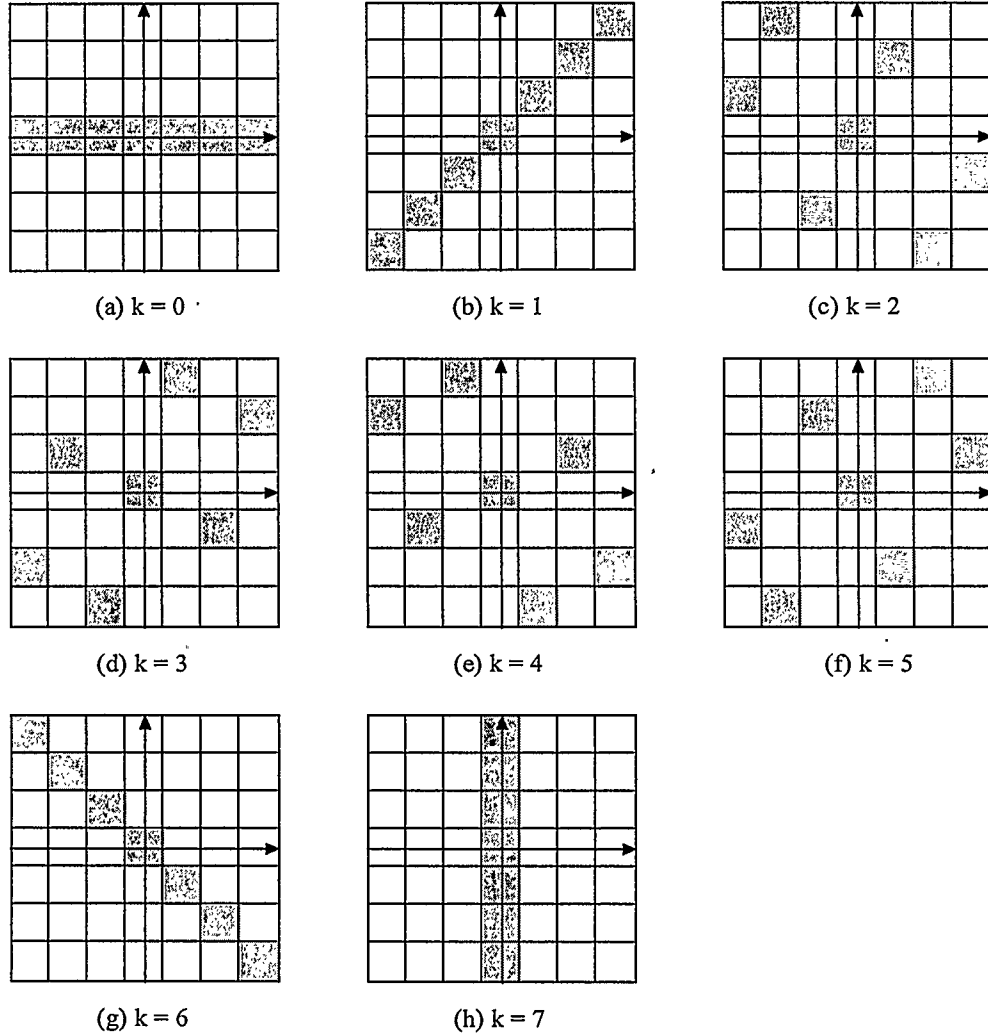


Figure 3.4: Process flow diagram for computing FRIT

### 3.2.2.1 Finite Radon Transform (FRAT)

The finite Radon transform [22][24] is defined as the summation of image pixels over a certain set of “lines”. These “lines” are defined in a finite geometry,  $Z_p^2$ , where  $p$  is a prime number. The equation of the FRAT is given below

$$r_k[l] = FRAT_f(k, l) = p^{-1/2} \sum_{(i,j) \in L_{k,l}} f[i, j] \quad (3.6)$$



*Figure 3.5: Lines for 7x7 FRAT. One line per slope has been shown in shaded gray color. For each slope, there would be six more lines parallel to the line shown in the figure.*

The prime dimension ensures that no two points of the 2-D array of pixels belong more than one line. This results unique projection patterns which makes the inverse transform simple and can be done by simple additive operations rather than more general algebraic transformation.

For best energy compaction, the mean value from the image  $f[i,j]$  is subtracted before calculating the FRAT coefficients.  $L_{k,l}$ , in equation 3.6, denotes the set of “lines” on  $Z_p^2$  and is defined as follows:

$$L_{k,l} = \begin{cases} \{(i, j) : j = ki + l \pmod{p}, i \in Z_p\} & \text{for } 0 \leq k < p \\ \{(l, j) : j \in Z_p\} & \text{for } k = p \end{cases} \quad (3.7)$$

Figure 3.5 shows examples of such lines (in shaded gray) for 7x7 size blocks. One line has been shown in the figure for each slope,  $k$ . So, for each slope, we obtain 7 lines defined in this way by changing the value of  $l$  from 0 to 6 (i.e., 0 to  $p-1$ ).

### 3.2.2.2 Optimal Ordering of FRAT Coefficients

The set of lines for the FRAT, defined by equations 3.7, is not the best way to describe lines on a finite grid over  $Z_p^2$ . The best way would be to define these lines in terms of normal vectors as follows:

$$L_{a,b,t} = \{(i, j) \in Z_p^2 : ai + bj - t = 0 \pmod{p}\} \quad a, b, t \in Z_p \text{ and } (a, b) \neq (0, 0) \quad (3.8)$$

Here  $(a, b)$  is the normal vector and  $t$  is the translation parameter. So, for a fixed normal vector,  $L_{a,b,t}$  is a set of  $p$  parallel lines since  $t \in Z_p$ . So, for the same slope  $k$ , where  $k = -a/b$ , equation 3.7 and 3.8 define the same set of  $p$  parallel lines. Moreover,  $k = 0$  signifies the horizontal lines and  $k = p$  signifies the vertical lines and the set of lines with

normal vector  $(a, b)$  is equal to the set of lines with the normal vector  $(na, nb)$ , for each  $n = 1, 2, 3, \dots, (p-1)$ . With this definition of lines the new FRAT equation can be written as

$$r_{a,b}[t] = FRAT_f(a, b, t) = p^{-1/2} \sum_{(i,j) \in L_{a,b,t}} f[i, j] \quad (3.9)$$

The usual FRAT expressed by equation 3.6 uses the set of  $(p+1)$  normal vectors  $u_k$ , where

$$u_k = \begin{cases} (-k, 1) & \text{for } k = 0, 1, 2, \dots, p-1 \\ (1, 0) & \text{for } k = p \end{cases} \quad (3.10)$$

For the new FRAT defined by equation 3.9,  $(p+1)$  normal vectors  $(a_k, b_k)$  are needed such that they cover all  $(p+1)$  directions as represented by  $u_k$ , and there are  $(p-1)$  possible choices for that. Do and Vetterli [12] showed that the best choice for the set of normal vectors can be defined as

$$(a_k^*, b_k^*) = \arg \min_{\substack{(a_k, b_k) \in \{nu_k : 1 \leq n \leq p-1\} \\ s.t. \ C_p(b_k) \geq 0}} \|C_p(a_k), C_p(b_k)\| \quad (3.11)$$

Here  $C_p(x)$  denotes the centralized function of period  $p$ , defined as  $C_p(x) = x - p \cdot \text{round}(x/p)$ . So,  $\|C_p(a_k), C_p(b_k)\|$  represents the length of the normal vectors and the optimal choice among these vectors for each  $k$  is the one with smallest length. Figure 3.6 shows the usual set and the optimal set of normal vectors for  $p = 7$ . As can be seen from the figure, the optimal set provides uniform angular coverage. It also ensures least wrap around effect due to periodization which in turns ensures that the FRAT projections are smooth or low frequency dominated so that it can be presented well by the wavelet transform later.

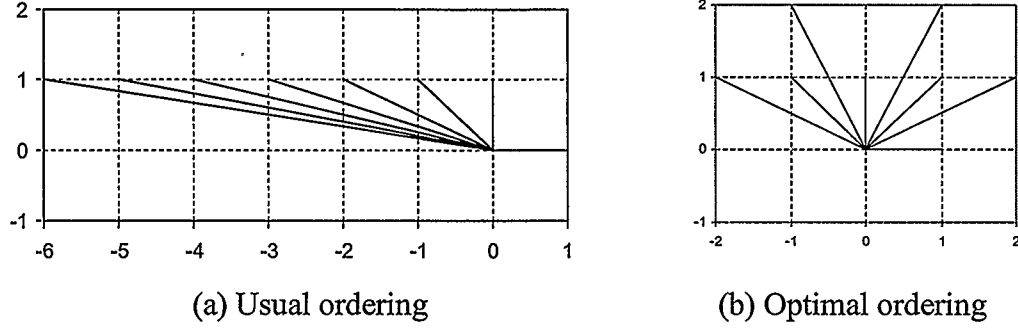


Figure 3.6: The set of normal vectors for  $p = 7$ .

Some points from this discussion which are worthy to note for calculating the FRAT are

1. Total number of defined lines are  $(p^2 + p)$
2. Each of these lines contains  $p$  points.
3. Any two distinct points belong to just one line.
4. For a particular slope, there are  $p$  parallel lines that provide the complete coverage of the plane,  $Z_p^2$ .

From the first three points stated above, the computational complexity of FRAT can be calculated. The third point suggests that the summation of any two distinct pixels can be used only for calculating a single FRAT coefficient and can not be used for calculating any of the remaining FRAT coefficients. Thus, the number of additions required is  $(p^2 + p)(p - 1)$  and the multiplication by a factor of  $p^{-1/2}$  for each FRAT coefficient gives us the computational complexity of the FRAT equal to  $O(p^2 M + p^3 A)$ . Here,  $p$  is the prime dimension of a square image, “M” denotes multiplication and “A” denotes addition operation. This third order complexity of the direct approach poses a huge workload and restricts the FRAT from being a potential candidate for use in image compression systems. VLSI architectures neither for the finite Radon transform nor for the finite

ridgelet transform have been found after surveying the literature at the time of this writing.

### 3.2.2.3 1-D Discrete Wavelet Transform (DWT)

The 1-D discrete wavelet transform (DWT) is used, on each of the FRAT slices ( $r_k[0]$ ,  $r_k[1]$ , ...,  $r_k[p-1]$ ) computed in the first stage for calculating the finite ridgelet transform. This is done after performing a periodic extension of the Radon slices to make them dyadic. The discrete wavelet transform is discussed in detail in Chapter 2. Here we shall discuss Daubechies  $D_4$  wavelet filter coefficients [25][26]. Several wavelet filters, such as Haar, Symlets, Gaussian, Mexican hat etc., have been defined over the past few years for compression algorithms. In this research, Daubechies  $D_4$  wavelet filter coefficients have been used. It is simple and the most localized member among Daubechies wavelets and provides excellent performance in image compression applications. The subscript “4” represents the number of filter taps or the number of the filter coefficients.

The properties of the scaling filter,  $h[n]$ , can be used as criteria in the design of a wavelet system. Given a scaling filter that satisfies the desired properties, the scaling and wavelet functions can be calculated. A very important class of wavelet systems is that with compact support. This gives rise to simple finite impulse response (FIR) filters with convenient time-localization properties. The most fundamental property of these filters is that the length of the filter must be even. For a filter length of 4, the minimal requirements of the scaling filter can be summarized as follows [27]:

1. Length of the filter,  $N = 4$

2.  $h[0] + h[1] + h[2] + h[3] = \sqrt{2}$

$$3. \quad h^2[0] + h^2[1] + h^2[2] + h^2[3] = 1$$

$$4. \quad h[0]h[2] + h[1]h[3] = 0$$

The degree of freedom here is  $N/2 - 1 = 1$ , which means there is still one degree of freedom remaining after the minimal requirements have been satisfied. Letting  $\alpha$  represent this degree of freedom parameter, we can formulate the scaling filter coefficient equations in the form:

$$\begin{aligned} h[0] &= \frac{1 - \cos \alpha + \sin \alpha}{2\sqrt{2}}, & h[1] &= \frac{1 + \cos \alpha + \sin \alpha}{2\sqrt{2}} \\ h[2] &= \frac{1 + \cos \alpha - \sin \alpha}{2\sqrt{2}}, & h[3] &= \frac{1 - \cos \alpha - \sin \alpha}{2\sqrt{2}} \end{aligned} \quad (3.12)$$

“ $\alpha$ ” in the above equations can be adjusted to give a wavelet system with the desired properties. The Daubechies wavelet with filter length 4 arises from  $\alpha = \pi/3$ , which gives the four scaling filter coefficients as follows

$$h[0] = \frac{1 + \sqrt{3}}{4\sqrt{2}}, \quad h[1] = \frac{3 + \sqrt{3}}{4\sqrt{2}}, \quad h[2] = \frac{3 - \sqrt{3}}{4\sqrt{2}}, \quad h[3] = \frac{1 - \sqrt{3}}{4\sqrt{2}} \quad (3.13)$$

From these low pass (scaling) filter coefficients, the high pass (wavelet) filter coefficients can be computed using the following relation

$$g[n] = (-1)^n h[N - n - 1] \quad (3.14)$$

which gives us the following four high pass (wavelet) filter coefficients

$$g[0] = \frac{1 - \sqrt{3}}{4\sqrt{2}}, \quad g[1] = -\frac{3 - \sqrt{3}}{4\sqrt{2}}, \quad g[2] = \frac{3 + \sqrt{3}}{4\sqrt{2}}, \quad g[3] = -\frac{1 + \sqrt{3}}{4\sqrt{2}} \quad (3.15)$$

Figure 3.7 shows the Daubechies  $D_4$  scaling,  $\phi(t)$  and wavelet,  $\psi(t)$  functions.

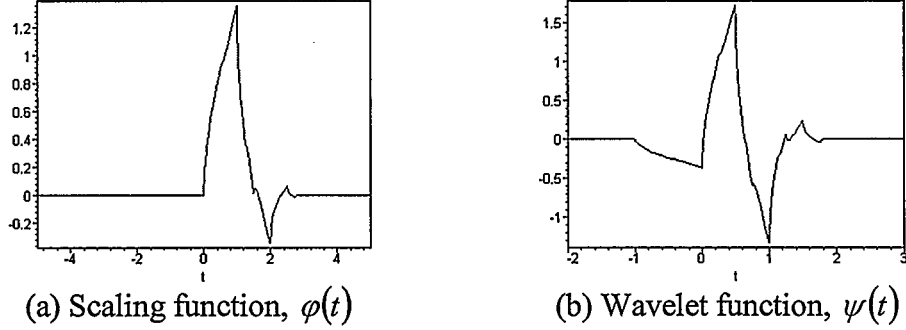


Figure 3.7: Daubechies  $D_4$  scaling and wavelet functions.

### 3.3 Architectures for 1-D DWT

This section presents recent 1-D DWT architectures introduced by various authors. The first one was proposed by Knowles [28] and is shown in Figure 3.8. It is a fully pipelined architecture but it is not particularly suitable for VLSI implementation as it requires large area, complex control and routing. The experimental results showed that implementing this circuit with 4 fixed coefficients and 3 octaves would require a NEC CMOS5 gate array with 1500 gates and it would be able to run at 6 MHz. Aware Inc. introduced a wavelet transform processor (WTP) [29], almost at the same time when Knowles proposed his architecture, which allows up to 6 coefficients and can operate at a speed of 30 MHz. The user chooses the wavelet coefficients, either specifying the coefficient values or the pre-loaded 6 coefficient Daubechies transform. Later, two architectures, folded and digit serial for 1-D DWT, were proposed by Parhi and Nishitani [30], as shown in Figure 3.9. These architectures assumed a filter of 4 taps. So, a wavelet with more coefficients requires more registers which ultimately affects the area and latency in



the final design and also the use of carry ripple adders affects the speed of the overall design. Vishwanath et. al. [31] proposed a linear systolic array architecture. But their architecture computes  $N$ -point DWT in  $2N$  cycles. This architecture suffers from a large delay (latency) and complex routing requirement. Recently, Chang et. al. [32] proposed an architecture claimed to be suitable for MPEG4 applications. Their architecture can compute  $N$ -point DWT in  $N/2$  cycles, but contains a large number of multipliers which increases the cost of the implementation. Figure 3.10 shows this architecture.

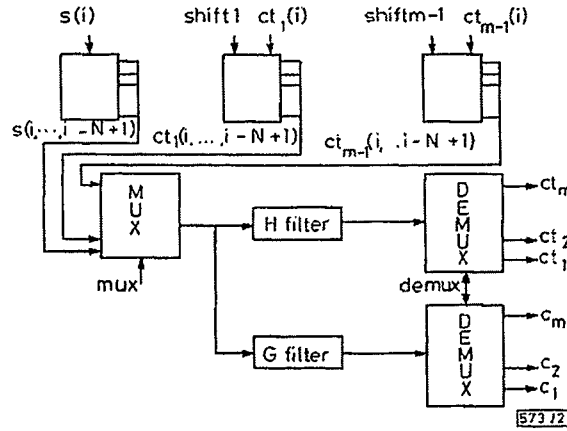
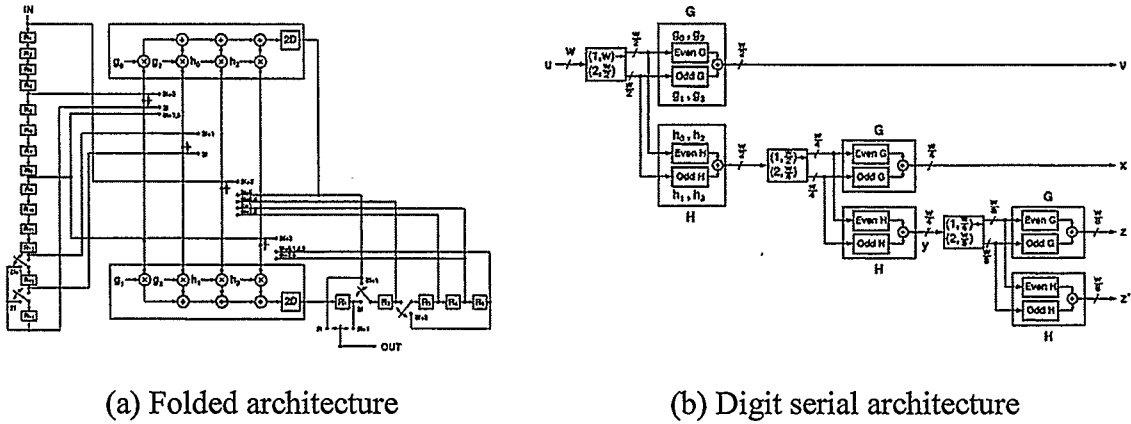


Figure 3.8: DWT architecture proposed by Knowles [28]



(a) Folded architecture

(b) Digit serial architecture

Figure 3.9: DWT architectures proposed by Parhi et. al. (3-level) [30]

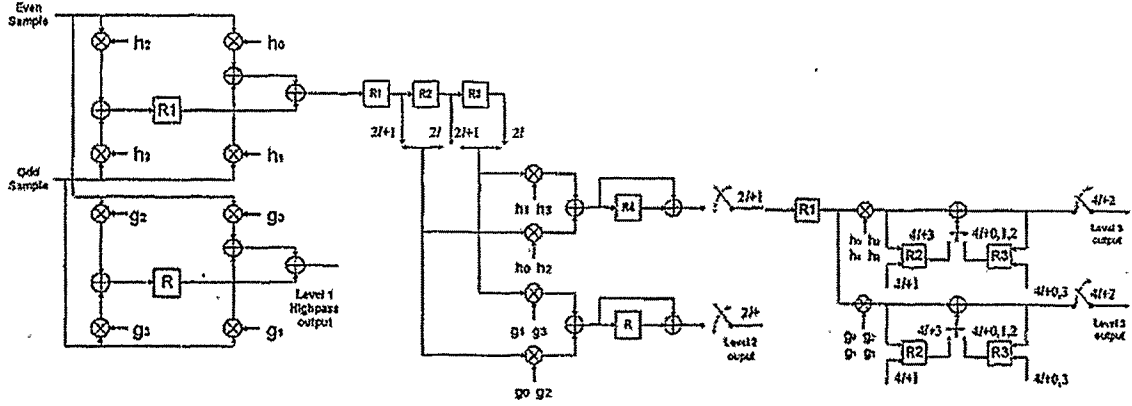


Figure 3.10: DWT architecture proposed by Chang et. al. (3-level) [32]

In this thesis, a distributed arithmetic (DA) based 1-D DWT architecture is used for computing the FRIT low pass and high pass coefficients. The main objective of the architecture was to keep it simple and fully multiplication free. The proposed architecture can compute one low pass and one high pass coefficient at every clock cycle, which inherently doubles its throughput. This means that the architecture can compute an  $N$ -point DWT in  $N/2$  clock cycles.

### 3.4 Distributed Arithmetic

Distributed arithmetic (DA) [33][34][35] has been used most often in the VLSI implementation of digital signal processing (DSP) architectures. It is an efficient method of computing vector inner products, which are required in many DSP systems. In most DSP algorithms the main computational block is a multiply / accumulate (MAC) structure [36], which is most often implemented using a standard multiplier and adder unit. The MAC unit can be implemented using DA, with the main advantage of pre-

computing all the possible products and storing them in a ROM. But the major drawback of this approach is the exponential growth of the ROM size with the number of inputs [37]. A different approach using DA is to distribute the coefficients to the input, one such example is the NEDA [38] architecture for the computation of the discrete cosine transform (DCT). This architecture relies on finding redundant computations in the vector inner product. In this thesis, for the 1-D DWT, we propose a DA based architecture where the DWT coefficient inner product is distributed over the input. The result is an efficient solution when the area of implementation is a concern for a given specification of input, output and coefficient word lengths. The architecture therefore is free of both multipliers and a ROM and is implemented using only adders.

### 3.4.1 DA Principle

Distributed arithmetic (DA) is an efficient strategy when one of the vectors is fixed. The one-dimensional inner product computation between two vectors  $x$  and  $c$ , where  $x$  is the input vector and  $c$  is the fixed coefficient vector can be represented by

$$y = c \cdot x = \sum_{j=0}^{k-1} c_j \cdot x_j \quad (4.1)$$

Here  $c = [c_0, c_1, c_2, \dots, c_{k-1}]$  is the fixed coefficient vector and  $x = [x_0, x_1, x_2, \dots, x_{k-1}]$  is the input vector. If  $c_j$  is represented in 2's complement form then

$$c_j = -c_{m,j} \cdot 2^m + \sum_{i=n}^{m-1} c_{i,j} \cdot 2^i \quad 0 \leq j \leq k-1 \quad (4.2)$$

Here  $m$  is the sign bit and  $n$  is the least significant bit. The output,  $y$ , can then be given by

$$y = -\sum_{j=0}^{k-1} c_{m,j} \cdot x_j \cdot 2^m + \sum_{i=n}^{m-1} \left[ \sum_{j=0}^{k-1} c_{i,j} \cdot x_j \right] \cdot 2^i \quad (4.3)$$

The above coefficient matrix is distributed, resulting in the following representation, which demonstrates the distribution of the bits of the coefficients over the input.

$$y = \begin{bmatrix} -2^m & 2^{m-1} & 2^{m-2} & \dots & 2^n \end{bmatrix} \begin{bmatrix} c_{m,0} & c_{m,1} & c_{m,2} & \dots & c_{m,k-1} \\ c_{m-1,0} & c_{m-1,1} & c_{m-1,2} & \dots & c_{m-1,k-1} \\ c_{m-2,0} & c_{m-2,1} & c_{m-2,2} & \dots & c_{m-2,k-1} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ c_{n,0} & c_{n,1} & c_{n,2} & \dots & c_{n,k-1} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{k-1} \end{bmatrix} \quad (4.4)$$

### 3.5 Summary

The theory of finite ridgelet transform has been presented in this chapter. The motivation behind the research has also been discussed. The recent architectures for the 1-D DWT which is one of the building blocks of the finite ridgelet transform have been shown in this chapter. Finally, the chapter ends with the discussion of distributed arithmetic; the technique that has been used in the proposed 1-D DWT architecture.

---

# Chapter 4

## *The Proposed FRIT Architectures*

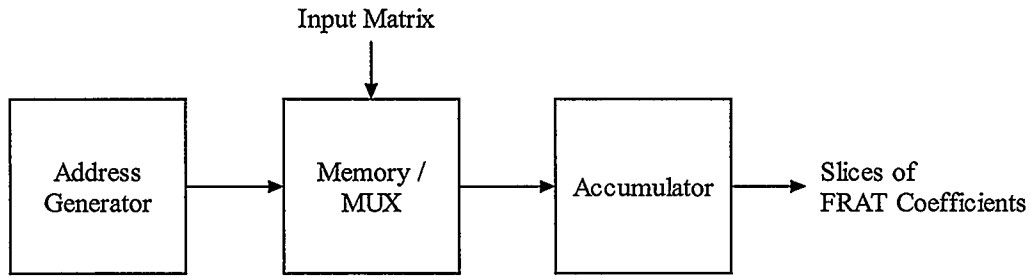
---

### 4.1 Introduction

As discussed in Chapter 3, the FRIT is a two-stage computational algorithm. In the first stage, the finite Radon transform is performed on input samples. This operation results in 1-D slices of Radon coefficients; in the second stage these 1-D slices of Radon coefficients are processed by a 1-D DWT that ultimately produces the FRIT low pass and high pass coefficients. So, the overall architecture for the FRIT can be presented in two distinct parts – the architecture for the finite Radon transform (FRAT) and the architecture for the discrete wavelet transform (DWT). In this thesis, these two distinct parts are first presented separately and then at the end of this chapter the overall look of the architecture is shown.

## 4.2 FRAT Architecture

The simplified block diagram of the architecture for the finite Radon transform is shown in Figure 4.1. Three distinct blocks are apparent from the figure – Address Generator, Memory Block or MUX depending on whether the architecture is with memory or without memory, respectively, and finally the Accumulator.



*Figure 4.1: Simplified block diagram of the proposed FRAT architectures*

### 4.2.1 Algorithm

In this research, we were interested in building a prototype for an  $7 \times 7$  size block of an image. From Chapter 3, the optimal ordered normal vectors for  $p = 7$ , in increasing angular sequence, can be tabulated as shown in Table 4.1.

*Table 4.1: Normal vectors for  $7 \times 7$  FRAT*

<b>k</b>	<b>A</b>	<b>b</b>
0	1	0
4	2	1
1	1	1
2	1	2
7	0	1
5	-1	2
6	-1	1
3	-2	1

If we index the elements of a 7x7 image matrix as shown in Figure 4.2, the simple pseudo code of Figure 4.3 will give us the Radon coefficient slices of eight directions.

0	7	14	21	28	35	42
1	8	15	22	29	36	43
2	9	16	23	30	37	44
3	10	17	24	31	38	45
4	11	18	25	32	39	46
5	12	19	26	33	40	47
6	13	20	27	34	41	48

Figure 4.2: A 7x7 image matrix  $f[i]$

```

For (  $\forall$  projections )
{
    Initialize  $ip = 0$  and  $R = 0$ ;
    If ( $a < 0$ )  $a = a \pmod{p}$ ; i.e., mapping  $a$  to  $Z_p^2$ .
    If ( $b < 0$ )  $b = b \pmod{p}$ ; i.e., mapping  $b$  to  $Z_p^2$ .

    For ( $j = 0$  to  $p-1$ )
    {
         $t = b * j \pmod{p}$ ; i.e., computing the starting index of the Radon slice  $R$ .

        For ( $i = 0$  to  $p-1$ )
        {
             $R[t] = R[t] + f[i + ip]$ ; i.e., adding the pixels for Radon coefficients.
             $t = (t + a) \pmod{p}$ ; i.e., advancing  $t$  for indexing next coefficient.
        }

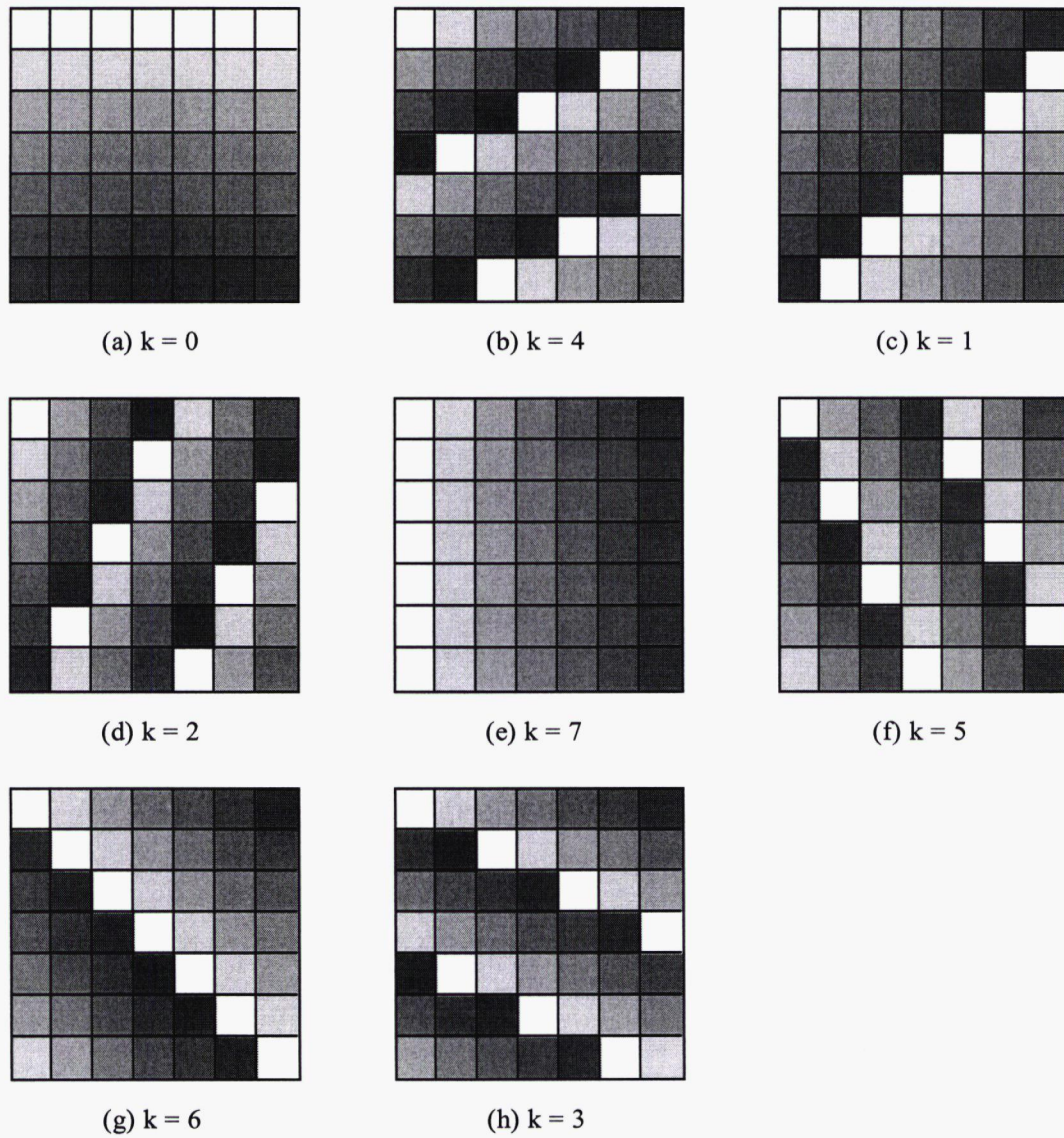
         $ip = ip + p$ ; i.e., advancing  $ip$  for indexing the start of next column of the
        image matrix  $f$ .
    }
}

```

Figure 4.3: Pseudo code for computing Radon coefficient of the image matrix shown in

Figure 4.2

This pseudo code computes the Radon coefficients of the eight Radon slices, shown in Figure 4.4, which are tabulated in Table 4.2.



*Figure 4.4: Lines of FRAT for 7x7 block size image. Coefficient's orders are signified by increasing gray level for each direction.*



Table 4.2: Radon coefficients of eight Radon slices; the pixels locations are given in (row, column) format for the 7x7 image block shown in Figure 4.5

Radon Slices ( <i>a</i> , <i>b</i> )	Radon Coefficients	Pixels of which values are added for each of the coefficients						
(1, 0)	C[1]	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)
	C[2]	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
	C[3]	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
	C[4]	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
	C[5]	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
	C[6]	(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)
	C[7]	(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)
(2, 1)	C[1]	(0, 0)	(3, 1)	(6, 2)	(2, 3)	(5, 4)	(1, 5)	(4, 6)
	C[2]	(4, 0)	(0, 1)	(3, 2)	(6, 3)	(2, 4)	(5, 5)	(1, 6)
	C[3]	(1, 0)	(4, 1)	(0, 2)	(3, 3)	(6, 4)	(2, 5)	(5, 6)
	C[4]	(5, 0)	(1, 1)	(4, 2)	(0, 3)	(3, 4)	(6, 5)	(2, 6)
	C[5]	(2, 0)	(5, 1)	(1, 2)	(4, 3)	(0, 4)	(3, 5)	(6, 6)
	C[6]	(6, 0)	(2, 1)	(5, 2)	(1, 3)	(4, 4)	(0, 5)	(3, 6)
	C[7]	(3, 0)	(6, 1)	(2, 2)	(5, 3)	(1, 4)	(4, 5)	(0, 6)
(1, 1)	C[1]	(0, 0)	(6, 1)	(5, 2)	(4, 3)	(3, 4)	(2, 5)	(1, 6)
	C[2]	(1, 0)	(0, 1)	(6, 2)	(5, 3)	(4, 4)	(3, 5)	(2, 6)
	C[3]	(2, 0)	(1, 1)	(0, 2)	(6, 3)	(5, 4)	(4, 5)	(3, 6)
	C[4]	(3, 0)	(2, 1)	(1, 2)	(0, 3)	(6, 4)	(5, 5)	(4, 6)
	C[5]	(4, 0)	(3, 1)	(2, 2)	(1, 3)	(0, 4)	(6, 5)	(5, 6)
	C[6]	(5, 0)	(4, 1)	(3, 2)	(2, 3)	(1, 4)	(0, 5)	(6, 6)
	C[7]	(6, 0)	(5, 1)	(4, 2)	(3, 3)	(2, 4)	(1, 5)	(0, 6)
(1, 2)	C[1]	(0, 0)	(5, 1)	(3, 2)	(1, 3)	(6, 4)	(4, 5)	(2, 6)
	C[2]	(1, 0)	(6, 1)	(4, 2)	(2, 3)	(0, 4)	(5, 5)	(3, 6)
	C[3]	(2, 0)	(0, 1)	(5, 2)	(3, 3)	(1, 4)	(6, 5)	(4, 6)
	C[4]	(3, 0)	(1, 1)	(6, 2)	(4, 3)	(2, 4)	(0, 5)	(5, 6)
	C[5]	(4, 0)	(2, 1)	(0, 2)	(5, 3)	(3, 4)	(1, 5)	(6, 6)
	C[6]	(5, 0)	(3, 1)	(1, 2)	(6, 3)	(4, 4)	(2, 5)	(0, 6)
	C[7]	(6, 0)	(4, 1)	(2, 2)	(0, 3)	(5, 4)	(3, 5)	(1, 6)
(0, 1)	C[1]	(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)
	C[2]	(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)
	C[3]	(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	(6, 2)
	C[4]	(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	(6, 3)
	C[5]	(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)	(6, 4)
	C[6]	(0, 5)	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)	(6, 5)
	C[7]	(0, 6)	(1, 6)	(2, 6)	(3, 6)	(4, 6)	(5, 6)	(6, 6)

Table continued on the next page

Radon Slices ( <i>a, b</i> )	Radon Coefficients	Pixels of which values are added for each of the coefficients						
(-1, 2)	C[1]	(0, 0)	(2, 1)	(4, 2)	(6, 3)	(1, 4)	(3, 5)	(5, 6)
	C[2]	(6, 0)	(1, 1)	(3, 2)	(5, 3)	(0, 4)	(2, 5)	(4, 6)
	C[3]	(5, 0)	(0, 1)	(2, 2)	(4, 3)	(6, 4)	(1, 5)	(3, 6)
	C[4]	(4, 0)	(6, 1)	(1, 2)	(3, 3)	(5, 4)	(0, 5)	(2, 6)
	C[5]	(3, 0)	(5, 1)	(0, 2)	(2, 3)	(4, 4)	(6, 5)	(1, 6)
	C[6]	(2, 0)	(4, 1)	(6, 2)	(1, 3)	(3, 4)	(5, 5)	(0, 6)
	C[7]	(1, 0)	(3, 1)	(5, 2)	(0, 3)	(2, 4)	(4, 5)	(6, 6)
(-1, 1)	C[1]	(0, 0)	(1, 1)	(2, 2)	(3, 3)	(4, 4)	(5, 5)	(6, 6)
	C[2]	(6, 0)	(0, 1)	(1, 2)	(2, 3)	(3, 4)	(4, 5)	(5, 6)
	C[3]	(5, 0)	(6, 1)	(0, 2)	(1, 3)	(2, 4)	(3, 5)	(4, 6)
	C[4]	(4, 0)	(5, 1)	(6, 2)	(0, 3)	(1, 4)	(2, 5)	(3, 6)
	C[5]	(3, 0)	(4, 1)	(5, 2)	(6, 3)	(0, 4)	(1, 5)	(2, 6)
	C[6]	(2, 0)	(3, 1)	(4, 2)	(5, 3)	(6, 4)	(0, 5)	(1, 6)
	C[7]	(1, 0)	(2, 1)	(3, 2)	(4, 3)	(5, 4)	(6, 5)	(0, 6)
(-2, 1)	C[1]	(0, 0)	(4, 1)	(1, 2)	(5, 3)	(2, 4)	(6, 5)	(3, 6)
	C[2]	(3, 0)	(0, 1)	(4, 2)	(1, 3)	(5, 4)	(2, 5)	(6, 6)
	C[3]	(6, 0)	(3, 1)	(0, 2)	(4, 3)	(1, 4)	(5, 5)	(2, 6)
	C[4]	(2, 0)	(6, 1)	(3, 2)	(0, 3)	(4, 4)	(1, 5)	(5, 6)
	C[5]	(5, 0)	(2, 1)	(6, 2)	(3, 3)	(0, 4)	(4, 5)	(1, 6)
	C[6]	(1, 0)	(5, 1)	(2, 2)	(6, 3)	(3, 4)	(0, 5)	(4, 6)
	C[7]	(4, 0)	(1, 1)	(5, 2)	(2, 3)	(6, 4)	(3, 5)	(0, 6)

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)

Figure 4.5: 7x7 image block showing the address of the pixel locations in (row, column) format.

$C[7]$  coefficients in Table 4.2 have to be calculated twice, once at the beginning and once at the end of each slice. This would make  $C[7]$  coefficients available for periodic extension in order to make the Radon slices dyadic.

### 4.2.2 Proposed FRAT Architecture with Memory

The proposed FRAT architecture with memory is shown in Figure 4.6. This is actually the detailed view of the block diagram shown in Figure 4.1.

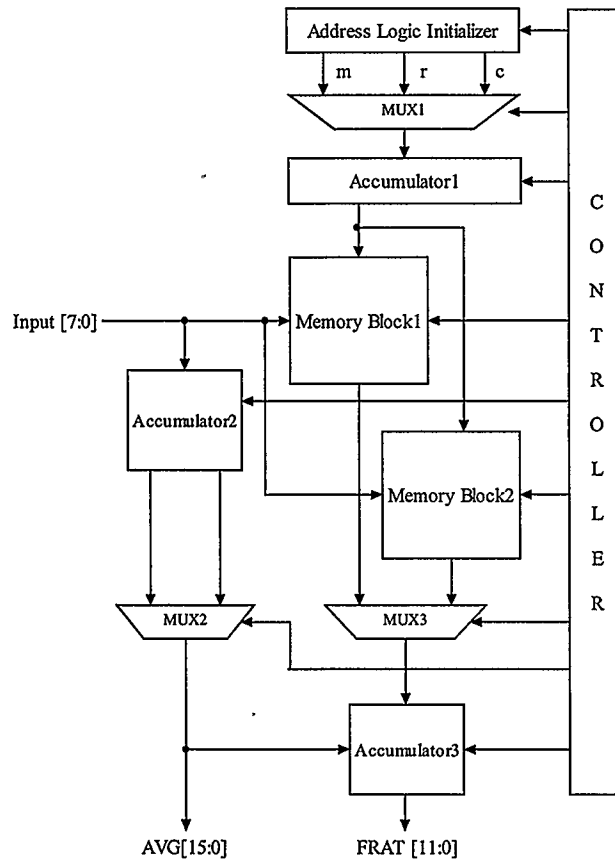


Figure 4.6: Proposed FRAT architecture with memory

Address Logic Initializer, MUX1 and Accumulator1 constitute the Address Generator. If we denote  $m$ ,  $c$  and  $r$  that satisfy the equations 4.1, 4.2 and 4.3 respectively, we can formulate the Table 4.3.

$$C_{i+1}[7]_1 = (C_i[7]_7 + m) \pmod{p} \quad (4.1)$$

$$C_i[j]_k = (C_j[j]_{k+1} + r) \pmod{p} \quad (4.2)$$

$$C_i[j+1]_1 = (C_i[j]_7 + c) \pmod{p} \quad (4.3)$$

In the above three equations,  $C_i[j]_k$  denotes the  $k$ th pixel for the  $j$ th coefficient of slice  $i$ .

*Table 4.3: Values of  $m$ ,  $r$  and  $c$  for the Radon slices*

<b>m</b>	<b>r</b>	<b>c</b>
6	0	1
4	3	0
6	6	0
6	5	6
6	1	1
2	2	1
2	1	0
4	4	0

From the architecture we see that the Address Logic Initializer outputs  $m$ ,  $r$  and  $c$  values according to the look up table (Table 4.3) for each FRAT slices. MUX1, controlled by the controller, outputs one of these values at every clock cycle and Accumulator1 accumulates the output value of MUX1. This gives the row address of the pixels. The column address is generated by a counter included in the controller. Accumulator1 is a 3-bit 1's complement accumulator which actually performs the mod 7 operation for generating the correct row address of each successive pixel of each successive coefficient. Accumulator2 and MUX2 give the mean of the image. This mean value is

subtracted from the image for best energy compaction. Actually, the output of MUX2 is seven times the mean value, so it is subtracted once after adding the seven pixels for each FRAT coefficient. The two Memory Blocks are of size  $7 \times 7$ , and for transforming images these two memory blocks are first loaded with successive image blocks of size  $7 \times 7$ . Two memory blocks have been used to keep the pipeline always full (a double buffered architecture). Computation of the Radon coefficients is immediately started when Memory Block1 is loaded with the first image block. While the computation of coefficients for Memory Block1 is carried out, Memory Block2 is loaded with the second image block. MUX2 is used to select one of the registered mean values of Accumulator2, because the loading of Memory Block2 and computation of the mean value for this block will be finished before the end of the computation of coefficients for Memory Block1. In this way, both the average and the image are available for the immediate start of computation for the second Memory Block at the end of the first one. Both registers and dual port block RAMs have been used as Memory Blocks in the simulation and synthesis of the architecture and the results are given in Chapter 5.

### 4.2.3 Proposed Memoryless FRAT Architecture

The proposed memoryless architecture for FRAT is shown in Figure 4.7. This is actually a parallel input architecture. The AGs are the Address Generator blocks for the seven pixels of each of the FRAT coefficients. Here the Address Logic Initializer outputs two values for each of the seven AGs. These two values are  $m$  and  $c$ , where  $m$  is the row address of the first pixel of the first coefficient i.e.,  $C[7]$  and  $c$  is the value that satisfies the equation 4.4.

$$C_i[j+1]_k = (C_i[j]_k + c) \pmod{p} \quad (4.4)$$

Here  $C_i[j]_k$  denotes the  $k$ th pixel for the  $j$ th coefficient of slice  $i$ . This gives us Table 4.4.

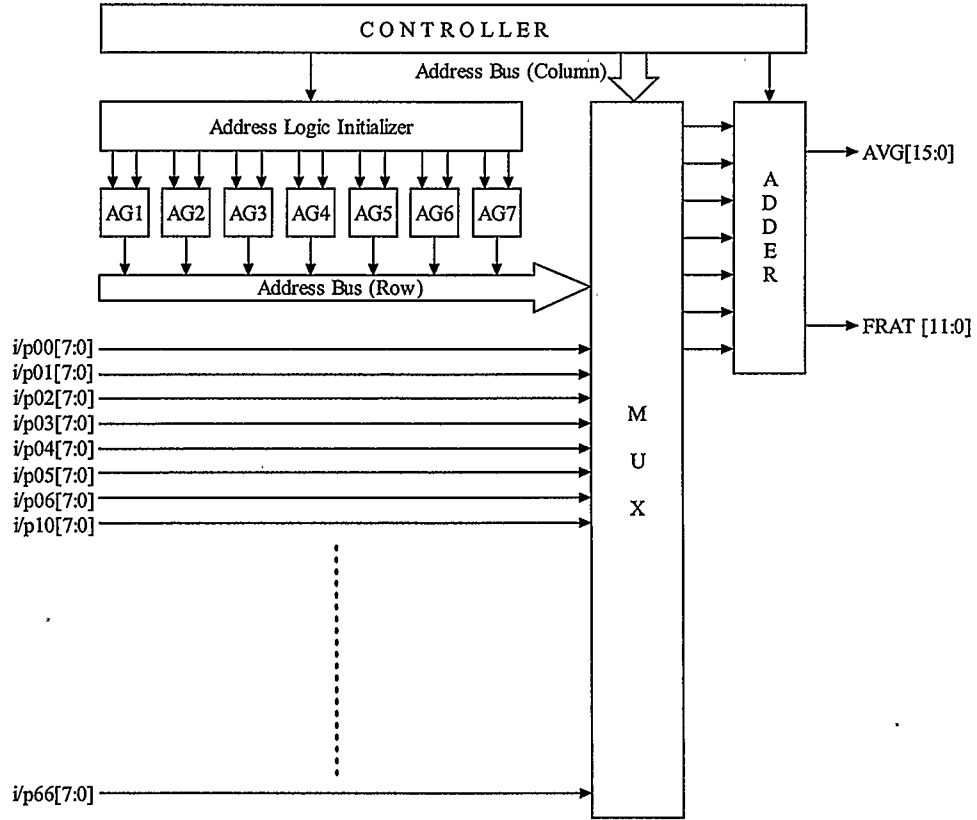


Figure 4.7: Proposed memoryless FRAT architecture

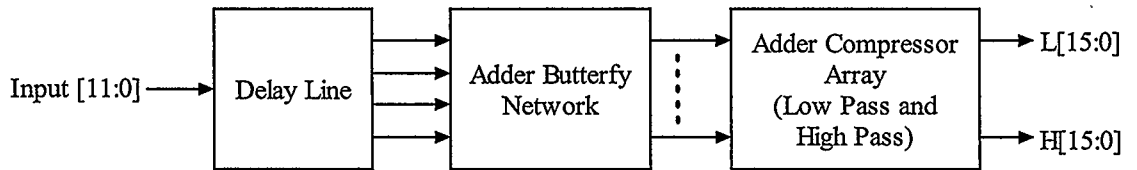
Table 4.4: Values of  $m1$ ,  $m2$ ,  $m3$ ,  $m4$ ,  $m5$ ,  $m6$ ,  $m7$  and  $c$  for the Radon slices

<b>m1</b>	<b>m2</b>	<b>m3</b>	<b>m4</b>	<b>m5</b>	<b>m6</b>	<b>m7</b>	<b>c (for every AG)</b>
6	6	6	6	6	6	6	1
3	6	2	5	1	4	0	4
6	5	4	3	2	1	0	1
6	4	2	0	5	3	1	1
0	1	2	3	4	5	6	0
1	3	5	0	2	4	6	6
1	2	3	4	5	6	0	6
4	1	5	2	6	3	0	3

The AGs are 3-bit 1's complement accumulators. So, the mod 7 operation is performed internally. The controller provides the column addresses which are fixed for each of the seven AGs except for the case of vertical lines. This special case is also handled by the controller. The (row, column) addresses are then used to select the inputs of the MUX. The seven pairs of (row, column) addresses select seven inputs for computing each of the Radon coefficients. These seven outputs of the MUX are then added by an adder compressor array and stored internally in a queue of seven registers. So, when all seven coefficients of the first slice are available, the Adder sequentially outputs the Radon coefficients and AVG of the image matrix. Again, this AVG is seven times the mean of the input image matrix.

### 4.3 Proposed DWT Architecture

The proposed DWT architecture is shown in Figure 4.8. The architecture is based on a Daubechies  $D_4$  wavelet filter bank.



*Figure 4.8: Proposed DWT architecture*

The forward transform uses two analysis filters  $h$  (low pass) and  $g$  (high pass) with filter coefficients as given in equations 4.5 and 4.6. These are actually the floating point representations of up to six decimal places of equations 3.13 and 3.15. These irrational

numbers forced us to choose a precision for the purpose of implementation. Hence we have chosen to represent the coefficients with an accuracy of 13 bits. The assumption is reasonable since 13 bits representation gives high enough accuracy for the fixed-point implementation.

$$h[0] = 0.482963, \quad h[1] = 0.836516, \quad h[2] = 0.224144, \quad h[3] = -0.129409 \quad (4.5)$$

$$g[0] = -0.129409, \quad g[1] = -0.224144, \quad g[2] = 0.836516, \quad g[3] = -0.482963 \quad (4.6)$$

One operation that we did not include in the FRAT architectures of the previous section is the multiplication of the FRAT coefficients by the normalization factor,  $p^{-1/2}$  as shown in equation 3.6. This operation can be equivalently performed by using the pre-divided filter coefficients shown in equation 4.7 and 4.8 in the DWT architecture.

$$h[0] = 0.182543, \quad h[1] = 0.316173, \quad h[2] = 0.084718, \quad h[3] = -0.048912 \quad (4.7)$$

$$g[0] = -0.048912, \quad g[1] = -0.084718, \quad g[2] = 0.316173, \quad g[3] = -0.182543 \quad (4.8)$$

The above coefficient matrices can be distributed into 13 bits (coefficient word length) as shown in Figure 4.9. “•” in the matrices represents the binary point.

For the computation of the DWT, the serial input data is passed through a delay line, as shown in Figure 4.10, which provides parallel data to the computational block-Adder Butterfly Network, shown in Figure 4.11. The two MUXs used in Figure 4.10 solve the dyadic problem of the FRAT slices. The adder butterfly network is found by finding the computational redundancy in the coefficient matrices (Figure 4.9) by considering the computation of both the low and high pass coefficients. Table 4.5 shows the adder butterfly network outputs. In total, there are fourteen partial products for low pass and high pass coefficients. These products are then passed through two identical



parallel arrays of adders (Figure 4.12) called the Adder Compressor Array. This finally gives the DWT coefficients, which are in fact the FRIT low pass and high pass coefficients of the combined (FRAT plus DWT) architecture. Table 4.6 shows how the outputs of the adder butterfly network are connected to various inputs of the adder compressor array.

$$h = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad g = \begin{bmatrix} 1 & 1 & 0 & 1 \\ \bullet & \bullet & \bullet & \bullet \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 4.9: Low pass and high pass filter coefficients matrices.

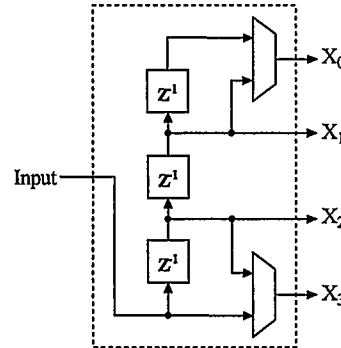


Figure 4.10: Delay line

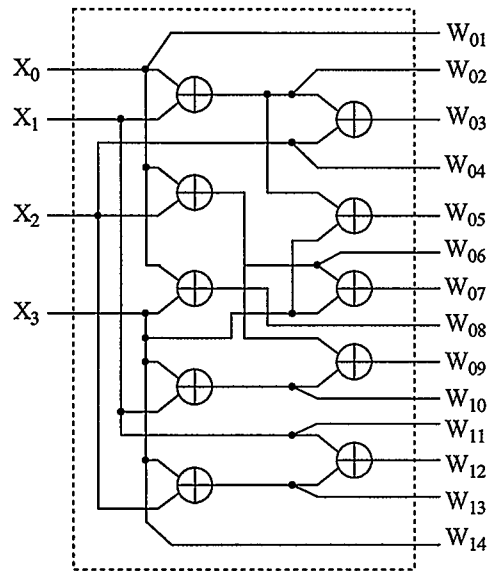


Figure 4.11: Adder butterfly network

Table 4.5: Operation performed by the adder butterfly network

Outputs	Expression
$W_{01}$	$X_0$
$W_{02}$	$X_0 + X_1$
$W_{03}$	$X_0 + X_1 + X_2$
$W_{04}$	$X_2$
$W_{05}$	$X_0 + X_1 + X_3$
$W_{06}$	$X_0 + X_2$
$W_{07}$	$X_0 + X_2 + X_3$
$W_{08}$	$X_0 + X_3$
$W_{09}$	$X_0 + X_1 + X_2 + X_3$
$W_{10}$	$X_1 + X_3$
$W_{11}$	$X_1$
$W_{12}$	$X_1 + X_2 + X_3$
$W_{13}$	$X_2 + X_3$
$W_{14}$	$X_3$

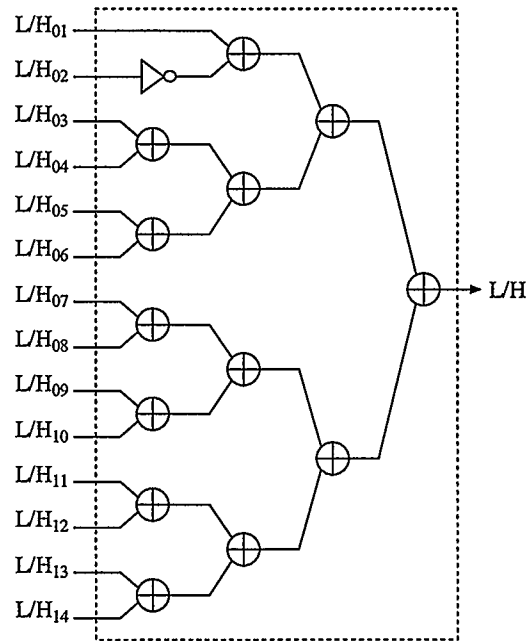


Figure 4.12: Parallel adders of adder compressor array

Table 4.6: Assignments of inputs of the adder compressor array

Inputs	For low pass	For high pass
L/H <sub>01</sub>	'1'	'1'
L/H <sub>02</sub>	W <sub>14</sub>	W <sub>05</sub>
L/H <sub>03</sub>	W <sub>14</sub>	W <sub>05</sub>
L/H <sub>04</sub>	W <sub>10</sub>	W <sub>09</sub>
L/H <sub>05</sub>	W <sub>08</sub>	W <sub>02</sub>
L/H <sub>06</sub>	W <sub>12</sub>	W <sub>07</sub>
L/H <sub>07</sub>	W <sub>01</sub>	W <sub>11</sub>
L/H <sub>08</sub>	W <sub>06</sub>	'0'
L/H <sub>09</sub>	W <sub>08</sub>	W <sub>02</sub>
L/H <sub>10</sub>	W <sub>13</sub>	W <sub>08</sub>
L/H <sub>11</sub>	W <sub>03</sub>	W <sub>04</sub>
L/H <sub>12</sub>	W <sub>10</sub>	W <sub>09</sub>
L/H <sub>13</sub>	W <sub>09</sub>	W <sub>06</sub>
L/H <sub>14</sub>	W <sub>09</sub>	W <sub>06</sub>

## 4.4 The FRIT Prototype

This section shows the proposed FRIT architecture by combining the proposed architectures for FRAT and DWT of the previous two sections. Figures 4.13 and 4.14 show the proposed FRIT architectures with memory and without memory, respectively. The proposed architectures are coded in Verilog HDL [39]. For simulation of the Verilog codes (Appendix B), the ModelSim HDL simulator [40] is used. In order to generate the image bit-stream for the hardware and to reconstruct the image from the output FRIT coefficient bit-stream of the hardware, MATLAB programs are written (Appendix A).

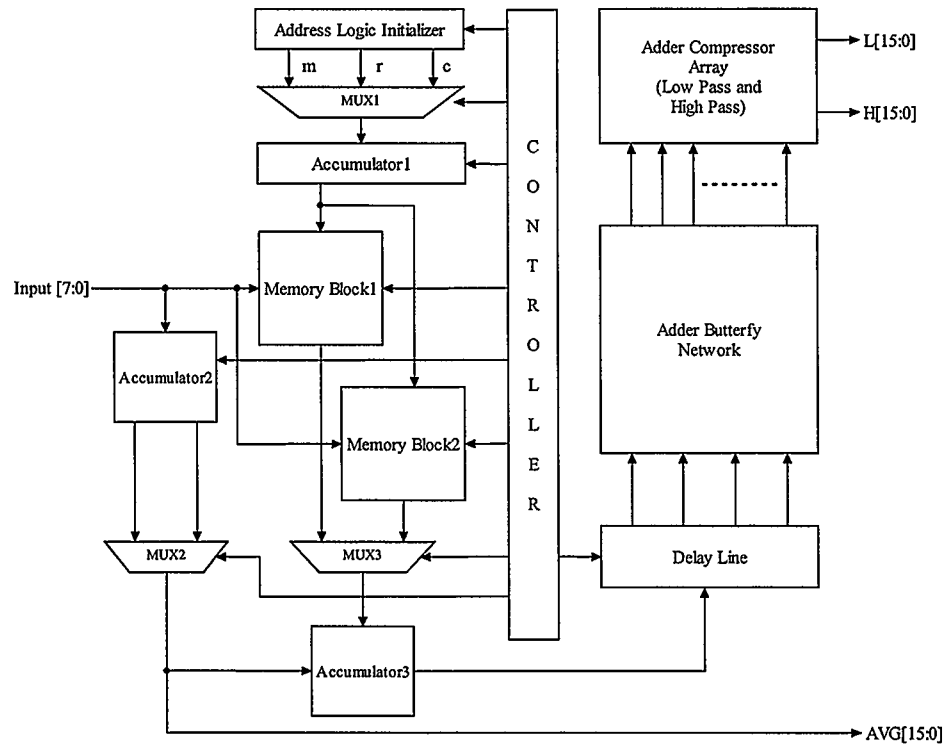


Figure 4.13: Proposed FRIT architecture with memory

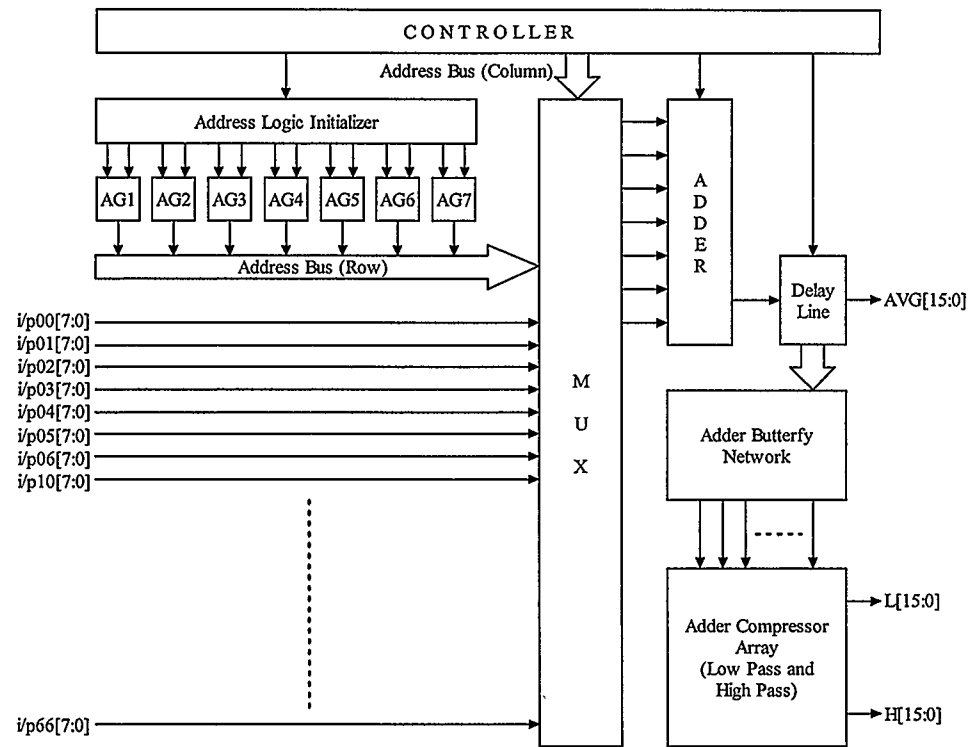


Figure 4.14: Proposed memoryless FRIT architecture

---

# Chapter 5

## *Performance Analysis*

---

### **5.1 Introduction**

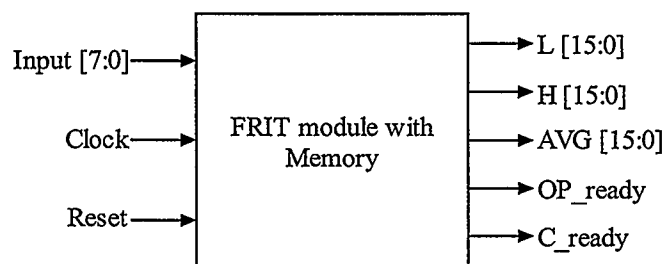
This chapter presents the simulation and synthesis results of the two proposed FRIT architectures discussed in the previous chapter. By doing only waveform simulation it is very difficult to judge whether or not the designed system meets the standard requirements. So a system simulation method has to be performed in order to confirm that the system meets the requirements. To do so, a system simulation test bench has been developed which provides an efficient way to process data and display the processed data dynamically during the simulation process. Several standard test bench images have been used. The performance measure that has been used is the peak signal to noise ratio (PSNR) discussed earlier in this thesis.

## 5.2 Simulation Results

The proposed architectures are coded using Verilog HDL. The Verilog codes are then compiled and simulated by ModelSim software. In order to interface the image with the hardware, MATLAB programs are written. CIF and QCIF image types are used for simulation. First the image is read by a MATLAB program that generates a binary input file. This file is used for simulation of the hardware by ModelSim, which generates two binary output files. One contains the low pass FRIT coefficients and the other contains the high pass FRIT coefficients. Then another MATLAB program is used for the inverse transform. Here, the number of retained coefficients can be specified. This overall process results in a reconstructed image of the original input, which is then used for performance measures; i.e., PSNR calculation.

### 5.2.1 FRIT Architecture with Memory

Figure 5.1 and Table 5.1 give the I/O interface description of the Verilog model. Figure 5.2 shows a snapshot of the simulation waveforms. The core latency is 76 cycles, hence the first pair of low and high pass output is available after 76 cycles and thereafter, the core outputs a pair of coefficients every 14 cycles.



*Figure 5.1: I/O ports of the FRIT module with memory*

Table 5.1: I/O signal description of the FRIT module with memory

Signal	I/O	Description
Input [7:0]	Input	8-bit wide pixel data-in of a 7x7 block
Clock	Input	Core clock signal
Reset	Input	Core reset signal, active low
L [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) low pass FRIT coefficient output
H [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) high pass FRIT coefficient output
AVG [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) average value output of the input image matrix
OP_ready	Output	Active high indicate the availability of each new FRIT coefficients at the output
C_ready	Output	Active high indicate core is ready to take input. Goes low as the memory is filled with the next image block.

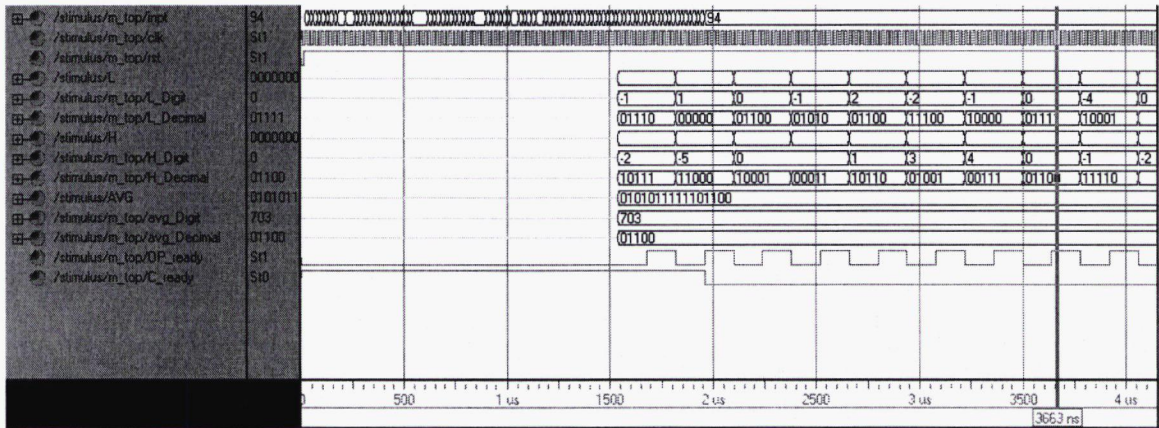


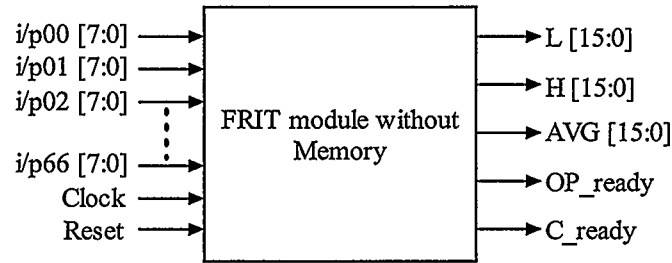
Figure 5.2: Snapshot of ModelSim simulation of the FRIT architecture with memory

After start/reset, “C\_ready” goes low when both of the memory blocks are loaded with successive blocks of image data. Thereafter it goes high when the core computes all the FRIT coefficients of one of the embedded memory blocks. This allows the memory block to be loaded with the next image block, while the core computes the FRIT coefficients of the other image block.



## 5.2.2 Memoryless FRIT Architecture

Figure 5.3 and Table 5.2 give the port description of the written code for the memoryless FRIT architecture.



*Figure 5.3: I/O ports of the memoryless FRIT module*

*Table 5.2: I/O signal description of the memoryless FRIT module*

Signal	I/O	Description
i/p00 [7:0] to i/p66 [7:0]	Inputs	49 8-bit wide pixel data-in of a 7x7 block
Clock	Input	Core clock signal
Reset	Input	Core reset signal, active low
L [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) low pass FRIT coefficient output
H [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) high pass FRIT coefficient output
AVG [15:0]	Output	16 bit (11 bit digit and 5 bit decimal places) average value output of the input image matrix
OP_ready	Output	Active high indicate the availability of each new FRIT coefficients at the output
C_ready	Output	Active high indicate core is ready to take input. Stays low as long as the core computes the FRIT coefficients of the input image block

Figure 5.4 shows a snapshot of the simulation waveforms. The core latency is 10 cycles, hence the first pair of low and high pass outputs is available after 10 cycles and thereafter the core outputs a pair of coefficients every 2 cycles.

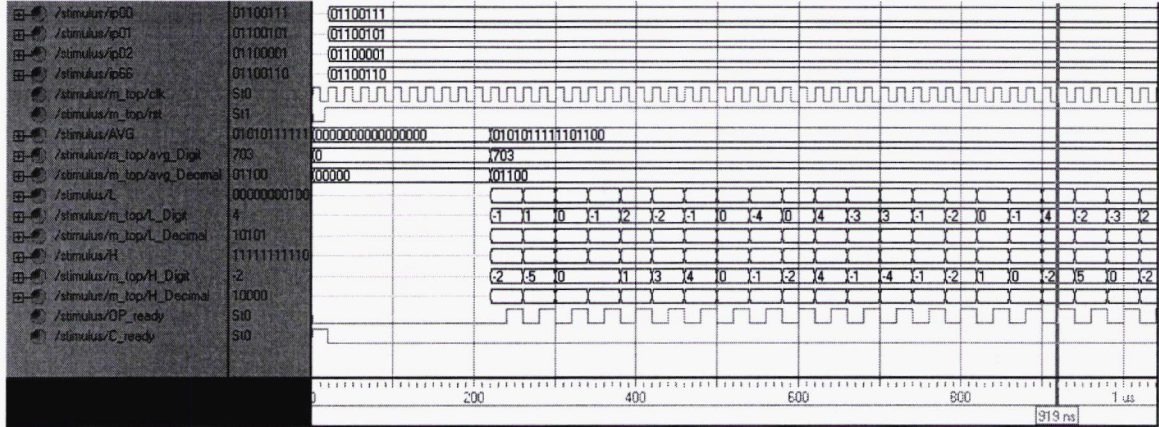
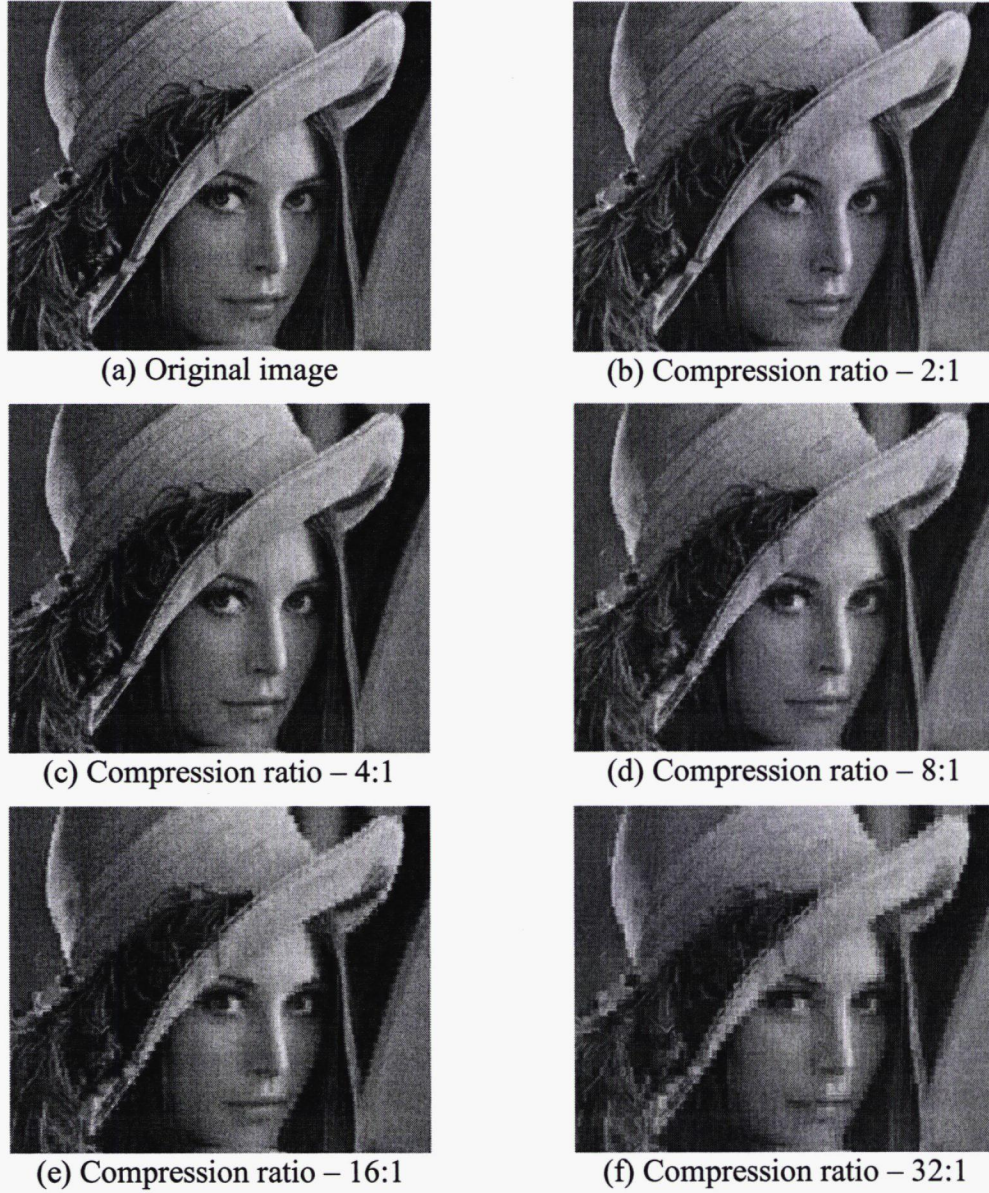


Figure 5.4: Snapshot of ModelSim simulation of the memoryless architecture

“C\_ready” stays low while the core computes the FRIT coefficients of an image block. After that “C\_ready” goes high to indicate the core is ready to process the next image block.

Since the I/O and internal signal precision in the proposed two architectures are kept equal, the simulation results of the two architectures produced identical reconstructed images; i.e., equal PSNR for the same compression ratios. Figure 5.5 shows the reconstructed “Lena” image of resolution 352x288 (CIF resolution), for five different compression ratios. Table 5.3 shows the PSNR values of the reconstructed “Lena” images achieved with the precision that has been used in the architectures for various compression ratios and Figure 5.6 shows this graphically.



*Figure 5.5: Original and reconstructed “Lena” images of different compression*

*Table 5.3: Comparison of PSNR of “Lena” image for different compression ratios*

Compression ratio	PSNR (dB)
2:1	39.38
4:1	33.08
8:1	29.54
16:1	26.94
32:1	25.15
64:1	23.95



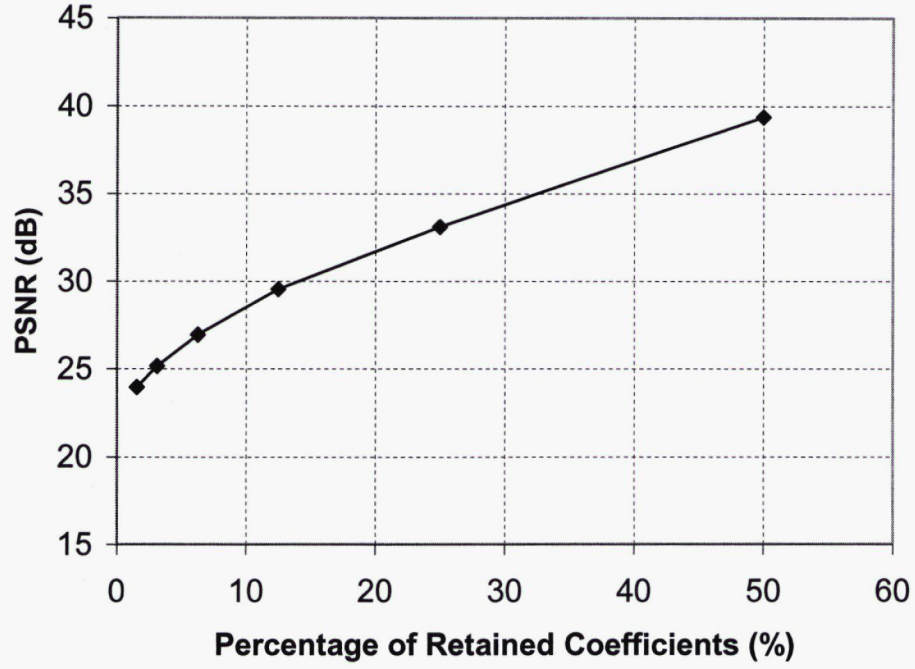


Figure 5.6: Plot of percentage of retained coefficients vs. PSNR

Table 5.4 shows another analysis of the proposed architectures. This table illustrates that the two proposed architectures conform to the real time processing requirement.

Table 5.4: Comparison of time required for transforming CIF and QCIF images with a core speed of 50MHz

Architecture	Latency (ns)	Image	1 frame (ms)	30 frames (ms)
With Memory	1,520	QCIF	4.8	144
		CIF	19.2	576
Memoryless	200	QCIF	0.7	21
		CIF	2.74	83

### 5.3 Synthesis Results

The proposed architectures have been synthesized using Xilinx ISE development tools [41]. The synthesis target device is the xc2v3000 of the Virtex-II device family. For power estimation, the Xilinx foundation series utility “XPower” has been used. The calculated power dissipation shown in Table 5.5, is for a 1.5 volt supply voltage with a clock frequency of 50 MHz and a capacitive load of 10 pF, assuming an ambient temperature of 25° C.

*Table 5.5: Synthesis results of the proposed architectures*

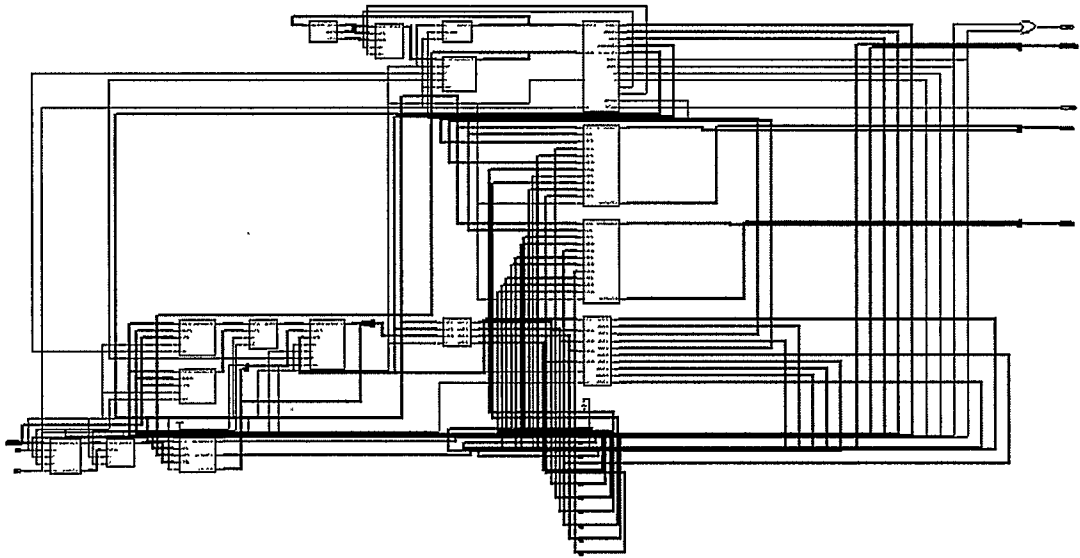
Architecture	# of Slices	# of 4-input LUTs	Gate Count	Speed (MHz)	Power (mW)
With Memory	826	1,187	17,629	84.33	140.05
Memoryless	1,115	2,098	21,485	66.35	290.06

Table 5.6 gives a comparison in terms of components used in the architectures. The table shows that the two proposed architectures are free of multipliers and implemented using only adders. This is the most attractive feature of the proposed architectures. Various bit length adders have been used throughout the architecture to meet the accuracy requirement.

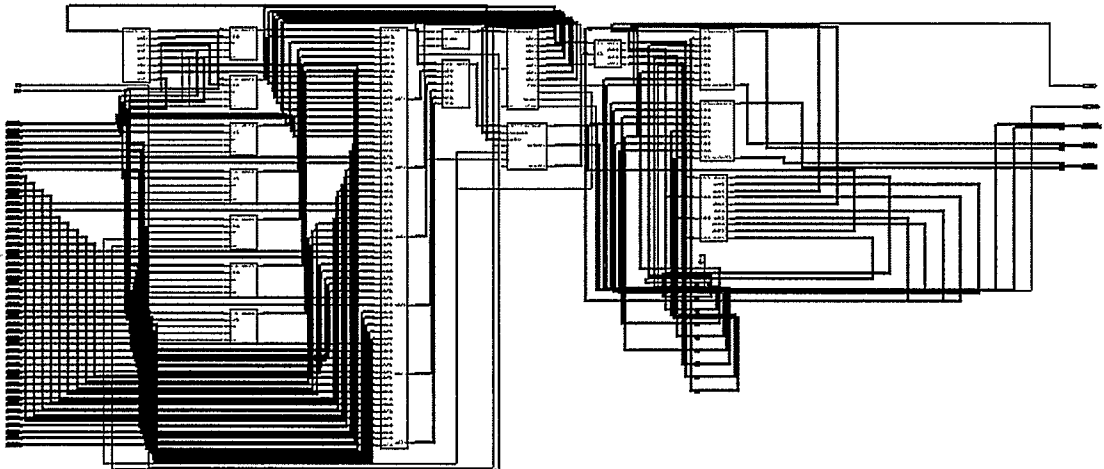
*Table 5.6: Comparison of number of components used in the architectures*

Architecture	# of BRAMs	# of Adders	# of REG. (FFs)
With Memory	2 (Eqv. 131,078 GC)	47 (Eqv. 1008, 1-bit cells)	672
Memoryless	-	69 (Eqv. 1090, 1-bit cells)	339

The proposed with memory and memoryless FRIT architectures require only 47 and 69 adders, respectively. This is equivalent to using 1008 and 1090 1-bit adder cells. Figure 5.7 and 5.8 show the Xilinx Engineering Capture System (ECS) view of the synthesized architectures.



*Figure 5.7: Xilinx ECS view of the proposed FRIT architecture with memory*



*Figure 5.8: Xilinx ECS view of the proposed memoryless FRIT architecture*

The most significant difference between the two proposed architectures is that they access the input matrix in two different ways. The first architecture (with memory) is a serial input architecture, while the second one (memoryless) is a parallel input architecture. This imposes a constraint on the inputs of the second architecture; the inputs (entire image block) should be available before the start of the computation and should not be changed until the computation of coefficients for the entire block is over. From this point of view the first architecture (with memory) is a more practical implementation. The architecture with memory uses two 49x8 dual port block RAMs while the other one uses a large MUX for inputs. Although there is not that much of a difference in speed, gate count and the number of adders used in the two proposed architectures, there is a substantial difference in the power consumption. This is because of the difference in the throughput of the two architectures at the same operating frequency. The second architecture's throughput is seven times that of the first one. So in order to compare the power dissipation, the second architecture was again simulated, while maintaining the same throughput as the first architecture, and the power dissipation was found to be 41.25 mW. This is almost 3.5 times less consumption than the first architecture, which consumes 140.05 mW for the same throughput. 36 adders have been used for computing the 1-D DWT coefficients and the rest (11 for the first and 33 for the second architecture) have been used for computing the FRAT coefficients and addressing pixels in both of the proposed two architectures. Direct implementation of the FRIT requires 7 adders plus 1 multiplier for computing the FRAT coefficients and 8 multipliers plus 6 adders for computing the 1-D DWT coefficients. This means that for 13 bit precision, the

improvement of the proposed two architectures over a direct implementation is 61.15% and 42.97%, respectively. From a quality point of view, the reconstructed images from the two proposed architectures are identical because same level of precision has been used in designing the architectures.



---

# Chapter 6

## *Conclusions and Future Work*

---

### **6.1 Summary of Accomplishments**

In this thesis, two original VLSI architectures for the finite ridgelet transform (FRIT) have been proposed. This is original in the sense that these two are the first ever proposed VLSI architectures for this transform based on a comprehensive literature survey. Moreover, within the proposed two FRIT architectures, two original architectures for the finite Radon transform (FRAT) and a state-of-the-art architecture for the 1-D discrete wavelet transform (DWT) are also proposed. Although the Radon transform has long been used in many edge detection applications, the finite Radon transform which was developed for representing images did not receive sufficient attention. This most likely explains why no VLSI architectures have been found in the literature for this transform. The first part of the two proposed FRIT architectures therefore present two new VLSI

architectures for the finite Radon transform. The DWT part of both architectures is also novel and is based on the NEDA architecture for computation of DCT coefficients. The simulation and synthesis results of the two proposed architectures conform to the real time processing requirements for QCIF as well as CIF image sequences.

## 6.2 Recommendations for Future Work

In order to reduce complexity, the CIF/QCIF images have been partitioned into 7x7 size image blocks for processing by the architectures. This has introduced block artifacts into the reconstructed images, similar to those introduced by the DCT, which are noticeable with a high compression ratio. One of the recommendations for future work is therefore to explore the feasibility of adaptive blocking which would allow a tradeoff between the complexity and the quality of the reconstructed images. This scheme would adaptively partition the image into various sizes of suitable blocks where edges look straight. A filtering approach may also be exploited for smoothing the artifacts.

The application of the FRIT results better reconstructed images over the DWT only in the case of images with many lines. This suggests that a hybrid type transform should be studied which would adaptively choose between the DWT and the FRIT depending on the image contents.

In this thesis, an architectural solution for the FRIT algorithm has been presented, which has been demonstrated for a 1-level forward transform. For more levels, the same proposed 1-D DWT framework can be used with the proposed FRAT architectures. This solution may be further explored, optimizing parameters of the architectures such as area, power, accuracy, etc.

Finally, an inverse transform architecture for the finite ridgelet transform is yet to be proposed. A low complexity solution suitable for integration with the proposed forward transform architectures would make them useful for the next generation standards of image compression.

---

## References

---

- [1] H. J. Nussbaumer, *Fast Fourier transform and convolution algorithms*, Heidelberg, Germany: Springer-Verlag, second edition, 1981, 1982.
- [2] H. V. Sorensen, C. S. Burrus, and M. T. Heideman, *Fast Fourier transform database*, Boston: PWS Publishing, 1995.
- [3] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *IEEE Transaction on Computers*, vol. C-23, pp. 90-93, January 1974.
- [4] R. M. Rao and A. S. Bopardikar, *Wavelet Transforms: Introduction to Theory and Application*, Addison Wesley Longman Inc. Reading, Massachusetts, 1998.
- [5] G. K. Wallace, "The JPEG still picture compression standard", *Communications of the ACM*, vol. 34, pp. 30-45, April 1991.
- [6] D. L. Gall, "MPEG: a video compression standard for multimedia applications", *Communications of the ACM*, vol. 34, pp. 46-58, April 1991.
- [7] ISO/ IEC JTC1/ SC29/ WG1, Document N390R, *New York Item: JPEG 2000 image coding system*, March 21, 1997.
- [8] S. Mallat, "Multifrequency channel decompositions of images wavelet models", *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37. no. 12, pp. 2091-2110, December 1989.
- [9] S. Mallat, "A theory for multiresolution signal decomposition: The Wavelet Representation", *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674-693, July 1989.
- [10] W. R. Zettler, J. Huffman and D. C. P. Linden, "Application of compactly supported wavelets to image compression", *Proceedings of SPIE: Image Processing Algorithms and Techniques*, vol. 1244, pp. 150-160, 1990.

- [11] E. J. Candes and D. L. Donoho, "Ridgelets: a key to higher- dimensional intermittency," *Phil. Trans. R. Soc. Lond. A.*, pp. 2495-2509, 1999.
- [12] M. N. Do and M. Vetterli, "The finite ridgelet transform for image representation," *IEEE Transactions on Image Processing*, vol-12, issue-1, pp. 16-28, Jan 2003.
- [13] A. M. Tekalp, *Digital video processing*, Prentice Hall, New Jersey, 1995.
- [14] C. E. Shannon, "A mathematical theory of communication", *Bell System Technical Journal*, vol. 27, pp. 379-423 (part I), pp. 623-656 (part II), 1948.
- [15] K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann Publishers Inc., San Francisco, California, 1996.
- [16] K. R. Castleman, *Digital Image Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.
- [17] M. V. Wickerhäuser, "High-resolution still picture compression", *Digital Signal Processing: A Review Journal*, vol. 2, pp. 204-226, 1992.
- [18] *Complex Number Operators, Waveforms, and Phasors*, class notes for EE 205, College of Engineering, University of Tennessee, Spring 1999.
- [19] O. Rioul and M. Vetterli, "Wavelets and signal processing," *IEEE Signal Processing Magazine*, pp. 14-38, October 1991.
- [20] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proceedings of the I.R.E.*, pp. 1098-1101, September 1952.
- [21] N. Abramson, *Information Theory and Coding*, McGraw Hill, New York, 1963.
- [22] F. Matus and J. Flusser, "Image representation via a finite Radon transform," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 15, no. 10, pp. 996-1006, October 1993.
- [23] S. R. Deans, *The Radon Transform and Some of its Applications*, John Wiley & Sons, 1983.
- [24] E. D. Bolker, "The finite Radon transform," in *Integral Geometry (Contemporary Mathematics, vol. 63)*, S. Helgason, R. L Bryant, V. Guillemin and R. O. Wells Jr., Eds., pp. 27-50, 1987.

- [25] I. Daubechies, "Orthonormal bases for compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909-996, November 1988.
- [26] I. Daubechies, "Ten lectures on wavelets," *Regional Conference Series in Applied Mathematics*, vol. 61, SIAM, 1992.
- [27] S. Mallat, *A wavelet tour of signal processing*, Academic Press, London, UK, second edition, 1999.
- [28] G. Knowles, "VLSI architecture for the discrete wavelet transform", *IEE Electronics Letters*, vol. 26, no. 15, pp. 1184-1185, July 1990.
- [29] *Aware Wavelet Transform Processor (WTP) Preliminary*, Aware Inc., Cambridge, MA, 1991.
- [30] K. Parhi and T. Nishitani, "VLSI architectures for discrete wavelet transforms", *IEEE Transactions on VLSI Systems*, vol. 1, no. 2, pp. 191-202, June 1993.
- [31] M. Vishwanath, R. M. Owens and M. J. Irwin, "VLSI architectures for the discrete wavelet transform," *IEEE Transaction on Circuits and Systems- II*, vol. 42, no. 5, pp. 305-316, May 1995.
- [32] S. J. Chang; M. H. Lee and J. Y. Park, "A high speed VLSI architecture of discrete wavelet transform for MPEG-4," *IEEE Transaction on Consumer Electronics*, vol. 43, issue. 3, pp. 623-627, August 1997.
- [33] A. Peled and B. Liu, "A new hardware realization of digital filters," *IEEE Transactions on ASSP*, vol. 22, no. 6, pp. 456-462, December 1974.
- [34] D. F. Elliott, *Handbook of Digital Signal Processing*, Academic Press, pp. 964-972, 1987.
- [35] W. P. Burleson, "A VLSI Design Methodology for Distributed Arithmetic," *Kluwer Journal of VLSI Signal Processing*, vol. 2, pp. 235-252, 1991.
- [36] G. K. Ma and F. J. Taylor, "Multiplier policies for digital signal processing," *IEEE ASSP Magazine*, pp. 6-19, January 1990.
- [37] S. A. White, "Applications of distributed arithmetic to digital signal processing: a tutorial review," *IEEE ASSP Magazine*, pp. 4-19, July 1989.

- [38] A. Shams, W. Pan, A. Chidanandan, and M.A Bayoumi, "A low power high performance distributed DCT architecture," *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 21 –27, April 2002.
- [39] 1364-2001 *IEEE Standard for Verilog Hardware Description Language*, IEEE, ISBN 0-7381-2827-9, 2001.
- [40] <http://www.model.com/>, ModelSim, A Mentor Graphics Products, 2003.
- [41] *Synthesis and Simulation Design Guide*, Xilinx Development System, Xilinx Inc., 2002.

---

# Appendix A

## *MATLAB Codes*

---

### A.1 Introduction

This section of appendix contains the MATLAB files that have been used for interfacing between image data and hardware designed in Verilog HDL.

### A.2 MATLAB Codes

There are 3 MATLAB .m files – “IMGinput.m”, “IMGoutput.m” and “PSNR.m”.

#### A.2.1 “IMGinput.m”

This program reads CIF/QCIF images and generates binary input file name 'ipframe.txt'.

```
%*****
*
% Reads CIF/QCIF images and generates binary input file name
%'ipframe.txt'.
%
```



```
% File Name: IMGinput.m
% Design Type: MATLAB .m file
%
% Input:
%     image    : Input CIF/QCIF image
%
% Output:
%     x        : Input CIF/QCIF image matrix
%
% This utility program is originally developed by Choudhury Ashiq
Rahman
% {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
%*****
*
```

```
function x = IMGinput (image)

%Read image and save in in x

[x] = imread(image);

if (size(x) == [288, 352])
    BLOCKS = 2142; %for CIF
end
if (size(x) == [144, 176])
    BLOCKS = 546; %for QCIF
end

%Create binary input stream file
fid = fopen('ipframe.txt','w');

B= im2col(x,[7 7],'distinct');

for i = 1:BLOCKS

    y = B(:,i);
    y1 = [y(1:7) y(8:14) y(15:21) y(22:28) y(29:35) y(36:42) y(43:49)];
    y1 = y1';
    y1 = y1(:);

    for j = 1:49
        ytemp = double(y1);
        yb0 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb1 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb2 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb3 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb4 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb5 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb6 = mod(ytemp(j),2); ytemp(j) = fix(ytemp(j)/2);
        yb7 = mod(ytemp(j),2);

        %Write image to the binary input file
```

```

        fprintf(fid, '%d%d%d%d%d%d%d ', yb7, yb6, yb5, yb4, yb3, yb2,
yb1, yb0);
    end

    fprintf(fid, '\n');

end

fclose(fid);

```

### A.2.2 “IMGoutput.m”

This program reconstructs images with the specified number of most significant coefficients from the two binary input files 'frit\_LP.txt' and 'frit\_HP.txt'. 'frit\_LP.txt' contains the low pass FRIT coefficients and the average value of the input image and 'frit\_HP.txt' contains the high pass FRIT coefficients. It uses the function 'ifrit', originally developed by Minh N. Do for inverse FRIT transform. This function is available in the frit toolbox that can be downloaded from the following link

<http://www.ifp.uiuc.edu/~minhdo/software/>

```

%*****
%
% Reconstructs images with the specified number of most significant
% coefficients from the two binary input files 'frit_LP.txt' and
% 'frit_HP.txt'. 'frit_LP.txt' contains the low pass FRIT coefficients
% and the average value of the input image and 'frit_HP.txt' contains
% the high pass FRIT coefficients. It uses the function 'ifrit',
% originally developed by Minh N. Do for inverse FRIT transform. This
% function is available in the frit toolbox that can be downloaded from
% the following link
% http://www.ifp.uiuc.edu/~minhdo/software/
%
% File Name: IMGoutput.m
% Design Type: MATLAB .m file
%
% Input:
%     nofCOFF : Number of most significant coefficients to use for
%               inverse transform
%
% Output:

```



```

if ((HPDi10(i)-48) == 1)
    HP(z) = HP(z)-2048;
end

if (mod(i,32)==0)
    Y = Y + 1;
    z = 1;
    A = [LP(1:4) LP(5:8) LP(9:12) LP(13:16) LP(17:20) LP(21:24)
LP(25:28) LP(29:32); HP(1:4) HP(5:8) HP(9:12) HP(13:16) HP(17:20)
HP(21:24) HP(25:28) HP(29:32)];
    DC = (AVGDi10(i)-48)*1024 + (AVGDi9(i)-48)*512 + (AVGDi8(i)-
48)*256 + (AVGDi7(i)-48)*128 + (AVGDi6(i)-48)*64 + (AVGDi5(i)-48)*32 +
(AVGDi4(i)-48)*16 + (AVGDi3(i)-48)*8 + (AVGDi2(i)-48)*4 + (AVGDi1(i)-
48)*2 + (AVGDi0(i)-48) + (AVGDe1(i)-48)*0.5 + (AVGDe2(i)-48)*0.25 +
(AVGDe3(i)-48)*0.125 + (AVGDe4(i)-48)*0.0625 + (AVGDe5(i)-48)*0.03125;

    %Taking most significant coefficients

    C = A(:);
    for m = 1:64
        indx = 0;
        for n = 1:64
            if (abs(C(m))<abs(C(n)))
                indx = indx + 1;
            end
        end
        if (indx > (nofCOFF-1)) C(m) = 0;
    end
end

C = [C(1:8) C(9:16) C(17:24) C(25:32) C(33:40) C(41:48)
C(49:56) C(57:64)];

im7x7 = ifrit (C, 1, DC, 'db2');
img(:,y) = im7x7(:);

else z = z + 1;
end
i = i + 1;
end

if (size(LPDi10,1) == 68544)
X= col2im(img,[7 7],[288 352], 'distinct'); %for CIF
image = 'CIF';
end

if (size(LPDi10,1) == 17472)
X= col2im(img,[7 7],[144 176], 'distinct'); %for QCIF
image = 'QCIF';
end

X = uint8(X);
imshow(X)

```

```

n2 = int2str(nofCOFF);
name = strcat(image, '_', n2, '.jpg');
imwrite(X, name, 'jpg');

```

### A.2.3 “PSNR.m”

This program computes the peak signal to noise ratio (PSNR) of the reconstructed image.

```

%*****
%
% Gives the Peak Signal to Noise Ratio (PSNR) of the reconstructed
% image.
%
% File Name: PSNR.m
% Design Type: MATLAB .m file
%
% Input:
%     image      : Input image
%     est        : Reconstructed image matrix
%
% Output:
%     s          : Peak Signal to Noise Ratio (PSNR)
%
% This utility program is originally developed by Choudhury Ashiq
% Rahman
% {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
%*****
%

function s = PSNR (image, est)
in= imread(image);

in = double(in);
est = double(est);

error = in - est;
[x y] = size(in);

temp = sum(error(:).^2)/(x*y);

s = 20 * log10 (255 / sqrt(temp));

```

---

# Appendix B

## *Verilog HDL Codes*

---

### **B.1 Introduction**

This section of appendix contains the Verilog HDL files of the two proposed FRIT architectures. Verilog 2000 syntax has been using in coding.

### **B.2 Codes for FRIT Architecture with Memory**

There are 15 Verilog HDL files in total for this architecture. These are – “stimulus.v”, “module\_top.v”, “logic\_INIT.v”, “mux3to1.v”, “acc\_row.v”, “memoryblock.v”, “sum\_for\_avg.v”, “avg\_perline.v”, “mux2to1.v”, “acc\_FRAT.v”, “delay\_line.v”, “adder\_bfly.v”, “adr\_compressor.v”, “counter.v” and “controller.v”.

### B.2.1 “stimulus.v”

```

//*****
// Stimulus for simulation
//
// File Name: stimulus.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module stimulus
#(parameter Width = 8, Qblocks = 546, BElements = 49, TElements =
104958);
// TElements = 104958 for CIF and 26754 for QCIF
reg [Width-1:0] memory_input [0:TElements-1];
reg [Width-1:0] IP;

reg clk, rst;
wire [Width+7:0] L, H, AVG;

integer i, k, file1, file2;

module_top m_top(L, H, AVG, OP_ready, C_ready, IP, clk, rst);

initial
begin
    rst = 1'b0;
    clk = 1'b1;

    i = 0;
    k = 0;

    $readmemb("ipframe.txt",memory_input);
    file1 = $fopen("frit_LP.txt");
    file2 = $fopen("frit_HP.txt");

    #15 rst = ~rst;
end

always #10 clk = ~clk;

always@(AVG)
    if (i == TElements && k) begin
        $fclose(file1);
        $fclose(file2);
        $stop;
    end

```

```

        end else k = ~k;

always@(OP_ready)
    if (OP_ready && rst) begin
        $fdisplay(file1, "%b %b %b %b", L[15:5], L[4:0], AVG[15:5],
        AVG[4:0]);
        $fdisplay(file2, "%b %b", H[15:5], H[4:0]);
    end

always@(posedge clk)
    if (C_ready)
        begin
            if (i == TElements) IP = 0;
            else begin
                IP = memory_input[i];
                i = i + 1;
            end
        end
    end
endmodule

```

## B.2.2 “module\_top.v”

```

//*****
// Top Module
//
// File Name: module_top.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module module_top
#(parameter Width = 8)
    (output [Width+7:0] L, H, AVG,
     output OP_ready, C_ready,
     input [Width-1:0] inpt,
     input clk, rst);

    wire [Width-6:0] count, m, c, r, op_mux, col, row, col_dummy, count_j;
    wire [Width-1:0] val, val1, val2;
    wire [Width+3:0] FRAT;
    wire [Width+5:0] M1, M2, M1or2;
    wire [Width+15:0] x0, x1, x3;

    wire [Width+13:0] x2;

```



```

wire [Width+14:0] n_avg;

wire [Width+7:0] w3;
wire [Width+12:0] w7, w12, w13;
wire [Width+13:0] w8;
wire [Width+14:0] w6, w9, w10;
wire [Width+16:0] w2, w5;
wire [Width-3:0] ADDW, ADDR;
wire [Width+2:0] L_Digit, H_Digit, avg_Digit;
wire [Width-4:0] L_Decimal, H_Decimal, avg_Decimal;

assign L = {L_Digit, L_Decimal};
assign H = {H_Digit, H_Decimal};
assign AVG = {avg_Digit, avg_Decimal};

or (C_ready, e1, e2);

sum_for_avg sfavg (M1, M2, inpt, e1, e2, clk, rst);
mux2to1 #(Width+6) mux_avg (M1or2, M1, M2, en1, en2);
avg_perline avgline (avg_Digit, avg_Decimal, n_avg, M1or2, count_j,
row, col, clk);

counter countr (count, nc, clk, rst);
logicINIT lINIT (m, c, r, count);
mux3to1 mux_init (op_mux, m, c, r, i, j);
acc_row acc1 (row, op_mux, clk, rst, e1, e2);
controller contrl (col, col_dummy, i, j, nc, count_j, ADDW, ADDR,
e1, e2, en1, en2, OP_ready, row, count, clk, rst);

memoryblock mem7x7_1 (val1, inpt, ADDW, ADDR, e1, clk);
memoryblock mem7x7_2 (val2, inpt, ADDW, ADDR, e2, clk);
mux2to1 mux_op (val, val1, val2, en1, en2);

acc_FRAT acc2 (FRAT, val, {1'b1, n_avg[22:12]}, col_dummy, clk, rst,
e1, e2);
delay_line dl (x0, x1, x2, x3, {FRAT, n_avg[11:0]}, count_j, col_dummy,
clk);

adder_bfly bfly (w2, w3, w5, w6, w7, w8, w9, w10, w12, w13, x0, x1, x2,
x3, col_dummy, clk, rst);

adr_compressor lowPass (L_Digit, L_Decimal, {x3[23], x3}, w10, w8, w12,
{x0[23], x0[23:5]}, w6[22:4], w13[20:4], w3, w9[22:9], count_j,
col_dummy, clk);
adr_compressor highPass (H_Digit, H_Decimal, w5, w9, w2[24:3], w7,
{x1[23], x1[23:5]},
19'b00000000000000000000, w8[21:5], {x2[21], x2[21:7]}, w6[22:9],
count_j, col_dummy, clk);

endmodule

```

A modified version of this “module\_top.v” file is given below when using array of registers as memory.

```
//*****
// Top Module
//
// File Name: module_top.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module module_top
#(parameter Width = 8)
  (output [Width+7:0] L, H, AVG,
   output OP_ready, C_ready,
   input [Width-1:0] inpt,
   input clk, rst);

  wire [Width-6:0] count, m, c, r, op_mux, col, row, col_dummy, count_j;
  wire [Width-1:0] val, val1, val2;
  wire [Width+3:0] FRAT;
  wire [Width+5:0] M1, M2, M1or2;
  wire [Width+15:0] x0, x1, x3;

  wire [Width+13:0] x2;
  wire [Width+14:0] n_avg;

  wire [Width+7:0] w3;
  wire [Width+12:0] w7, w12, w13;
  wire [Width+13:0] w8;
  wire [Width+14:0] w6, w9, w10;
  wire [Width+16:0] w2, w5;
  wire [Width+2:0] L_Digit, H_Digit, avg_Digit;
  wire [Width-4:0] L_Decimal, H_Decimal, avg_Decimal;

  assign L = {L_Digit, L_Decimal};
  assign H = {H_Digit, H_Decimal};
  assign AVG = {avg_Digit, avg_Decimal};

  or (C_ready, e1, e2);

  sum_for_avg sfavg (M1, M2, inpt, e1, e2, clk, rst);
  mux2to1 #(Width+6) mux_avg (M1or2, M1, M2, en1, en2);
```

```

avg_perline avgline (avg_Digit, avg_Decimal, n_avg, M1or2, count_j,
row, col, clk);

counter countr (count, nc, clk, rst);
logicINIT lINIT (m, c, r, count);
mux3to1 mux_init (op_mux, m, c, r, i, j);
acc_row acc1 (row, op_mux, clk, rst, e1, e2);
controller contrl (col, col_dummy, i, j, nc, count_j, OP_ready, row,
clk, e1, e2, rst);

memoryblock mem7x7_1 (val1, e1, en1, 1'b0, 1'b1, e2, inpt, row, col,
count, i, j, clk, rst);
memoryblock mem7x7_2 (val2, e2, en2, 1'b1, 1'b0, e1, inpt, row, col,
count, i, j, clk, rst);
mux2to1 mux_op (val, val1, val2, en1, en2);

acc_FRAT acc2 (FRAT, val, {1'b1, n_avg[22:12]}, col_dummy, clk, rst,
e1, e2);
delay_line dl (x0, x1, x2, x3, {FRAT, n_avg[11:0]}, count_j, col_dummy,
clk);

adder_bfly bfly (w2, w3, w5, w6, w7, w8, w9, w10, w12, w13, x0, x1, x2,
x3, col_dummy, clk, rst);

adr_compressor lowPass (L_Digit, L_Decimal, {x3[23], x3}, w10, w8, w12,
{x0[23],x0[23:5]}, w6[22:4], w13[20:4], w3, w9[22:9], count_j,
col_dummy, clk);
adr_compressor highPass (H_Digit, H_Decimal, w5, w9, w2[24:3], w7,
{x1[23],x1[23:5]},
19'b00000000000000000000, w8[21:5], {x2[21],x2[21:7]}, w6[22:9],
count_j, col_dummy, clk);

endmodule

```

### B.2.3 “logic\_INIT.v”

```

//*****
// Address Logic Initializer, generates m, c and r values.
//
// File Name: logicINIT.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

```

```

module logicINIT
#(parameter width = 3)
  (output reg [width-1:0] m, c, r,
   input [width-1:0] count);

always@(count)
  case(count)
    3'b000: begin
      m = 3'b110; c = 3'b001; r = 3'b000;
    end
    3'b001: begin
      m = 3'b100; c = 3'b000; r = 3'b011;
    end
    3'b010: begin
      m = 3'b110; c = 3'b000; r = 3'b110;
    end
    3'b011: begin
      m = 3'b110; c = 3'b110; r = 3'b101;
    end
    3'b100: begin
      m = 3'b110; c = 3'b001; r = 3'b001;
    end
    3'b101: begin
      m = 3'b010; c = 3'b001; r = 3'b010;
    end
    3'b110: begin
      m = 3'b010; c = 3'b000; r = 3'b001;
    end
    3'b111: begin
      m = 3'b100; c = 3'b000; r = 3'b100;
    end
  endcase
endmodule

```

## B.2.4 “mux3to1.v”

```

//*****
// MUX1, selects one of the inputs (m, c or r) for Accumulator1
//
// File Name: mux3to1.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

```

```

module mux3to1
#(parameter Width = 3)
  (output reg [Width-1:0] out,
   input [Width-1:0] in1, in2, in3,
   input select_i, select_j);

always@(in1, in2, in3, select_i, select_j)
  begin
    casex ({select_j, select_i})
      2'b00: out = in1;
      2'b01: out = in2;
      2'b1x: out = in3;
    endcase
  end
endmodule

```

### B.2.5 “acc\_row.v”

```

//*****
// Accumulator1, generates row addresses of the pixels.
//
// File Name: acc_row.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

```

```

module acc_row
#(parameter Width = 3)
  (output [Width-1:0] acc_out,
   input [Width-1:0] acc_in,
   input clock, reset, e1, e2);

reg carry;
reg [Width-1:0] sum;

always@(posedge clock, negedge reset)
  begin
    if (~reset)
      begin
        sum <= 1;
        carry <= 0;
      end
    else if (~e1 || ~e2)
      begin

```

```

        {carry, sum} <= sum + acc_in + carry;
    end
end

assign acc_out = sum + carry;

endmodule

```

## B.2.6 “memoryblock.v”

```

//*****
// Memory Block (Xilinx Dual Port Block RAM)
//
// File Name: memoryblock.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module memoryblock
#(parameter Element = 49, BitLength = 8)
(output reg [BitLength-1:0] memvalue,
 input [BitLength-1:0] inpt,
 input [5:0] ADDW,
 input [5:0] ADDR,
 input empty,
 input clk);

reg [BitLength-1:0] memory [Element-1: 0];

always@(posedge clk)
begin
    if (empty) memory[ADDW] <= inpt;
    memvalue <= memory[ADDR];
end

endmodule

```

A modified version of this “memoryblock.v” file is given below when using array of registers as memory.

```

//*****

```

```
// Memory Block (using array of 8 bit registers)
//
// File Name: memoryblock.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****
```

```
module memoryblock
#(parameter Column = 7, BitLength = 8)
(output reg [BitLength-1:0] memvalue,
 output reg empty,
 output reg enable,
 input p1, p2, e_other,
 input [BitLength-1:0] inpt,
 input [2:0] row, col, count,
 input i, j,
 input clk, reset);

reg [5:0] ADDR;
reg [BitLength-1:0] memory_r0 [Column-1: 0];
reg [BitLength-1:0] memory_r1 [Column-1: 0];
reg [BitLength-1:0] memory_r2 [Column-1: 0];
reg [BitLength-1:0] memory_r3 [Column-1: 0];
reg [BitLength-1:0] memory_r4 [Column-1: 0];
reg [BitLength-1:0] memory_r5 [Column-1: 0];
reg [BitLength-1:0] memory_r6 [Column-1: 0];
```

```
always@(posedge clk, negedge reset)
begin
  if (~reset)
  begin
    ADDR <= 1;
    empty <= 1;
    enable <= 0;
    memvalue <= 0;
    memory_r0[0] <= 0;
    memory_r0[1] <= 0;
    memory_r0[2] <= 0;
    memory_r0[3] <= 0;
    memory_r0[4] <= 0;
    memory_r0[5] <= 0;
    memory_r0[6] <= 0;
    memory_r1[0] <= 0;
    memory_r1[1] <= 0;
    memory_r1[2] <= 0;
    memory_r1[3] <= 0;
```

```

memory_r1[4] <= 0;
memory_r1[5] <= 0;
memory_r1[6] <= 0;
memory_r2[0] <= 0;
memory_r2[1] <= 0;
memory_r2[2] <= 0;
memory_r2[3] <= 0;
memory_r2[4] <= 0;
memory_r2[5] <= 0;
memory_r2[6] <= 0;
memory_r3[0] <= 0;
memory_r3[1] <= 0;
memory_r3[2] <= 0;
memory_r3[3] <= 0;
memory_r3[4] <= 0;
memory_r3[5] <= 0;
memory_r3[6] <= 0;
memory_r4[0] <= 0;
memory_r4[1] <= 0;
memory_r4[2] <= 0;
memory_r4[3] <= 0;
memory_r4[4] <= 0;
memory_r4[5] <= 0;
memory_r4[6] <= 0;
memory_r5[0] <= 0;
memory_r5[1] <= 0;
memory_r5[2] <= 0;
memory_r5[3] <= 0;
memory_r5[4] <= 0;
memory_r5[5] <= 0;
memory_r5[6] <= 0;
memory_r6[0] <= 0;
memory_r6[1] <= 0;
memory_r6[2] <= 0;
memory_r6[3] <= 0;
memory_r6[4] <= 0;
memory_r6[5] <= 0;
memory_r6[6] <= 0;
end

else begin
if (empty && (p1<p2 || ~e_other))
begin
enable <= 0;
case (ADDR)
6'b000001 : memory_r0[0] <= inpt;
6'b000010 : memory_r0[1] <= inpt;
6'b000011 : memory_r0[2] <= inpt;
6'b000100 : memory_r0[3] <= inpt;
6'b000101 : memory_r0[4] <= inpt;
6'b000110 : memory_r0[5] <= inpt;
6'b000111 : memory_r0[6] <= inpt;
6'b001000 : memory_r1[0] <= inpt;
6'b001001 : memory_r1[1] <= inpt;

```



```

        6'b001010 : memory_r1[2] <= inpt;
        6'b001011 : memory_r1[3] <= inpt;
        6'b001100 : memory_r1[4] <= inpt;
        6'b001101 : memory_r1[5] <= inpt;
        6'b001110 : memory_r1[6] <= inpt;
        6'b001111 : memory_r2[0] <= inpt;
        6'b010000 : memory_r2[1] <= inpt;
        6'b010001 : memory_r2[2] <= inpt;
        6'b010010 : memory_r2[3] <= inpt;
        6'b010011 : memory_r2[4] <= inpt;
        6'b010100 : memory_r2[5] <= inpt;
        6'b010101 : memory_r2[6] <= inpt;
        6'b010110 : memory_r3[0] <= inpt;
        6'b010111 : memory_r3[1] <= inpt;
        6'b011000 : memory_r3[2] <= inpt;
        6'b011001 : memory_r3[3] <= inpt;
        6'b011010 : memory_r3[4] <= inpt;
        6'b011011 : memory_r3[5] <= inpt;
        6'b011100 : memory_r3[6] <= inpt;
        6'b011101 : memory_r4[0] <= inpt;
        6'b011110 : memory_r4[1] <= inpt;
        6'b011111 : memory_r4[2] <= inpt;
        6'b100000 : memory_r4[3] <= inpt;
        6'b100001 : memory_r4[4] <= inpt;
        6'b100010 : memory_r4[5] <= inpt;
        6'b100011 : memory_r4[6] <= inpt;
        6'b100100 : memory_r5[0] <= inpt;
        6'b100101 : memory_r5[1] <= inpt;
        6'b100110 : memory_r5[2] <= inpt;
        6'b100111 : memory_r5[3] <= inpt;
        6'b101000 : memory_r5[4] <= inpt;
        6'b101001 : memory_r5[5] <= inpt;
        6'b101010 : memory_r5[6] <= inpt;
        6'b101011 : memory_r6[0] <= inpt;
        6'b101100 : memory_r6[1] <= inpt;
        6'b101101 : memory_r6[2] <= inpt;
        6'b101110 : memory_r6[3] <= inpt;
        6'b101111 : memory_r6[4] <= inpt;
        6'b110000 : memory_r6[5] <= inpt;
        6'b110001 : begin
                        memory_r6[6] <= inpt;
                        empty <= 0;
                    end
        endcase
        ADDR <= ADDR + 1;
    end

    if ((enable || e_other) && ~empty)
    begin
        enable <= 1;
        if (ADDR == 50) ADDR <= 1;

        case (row)
        3'b001:      memvalue <= memory_r0[col-1];

```

```

        3'b010: memvalue <= memory_r1[col-1];
        3'b011:      memvalue <= memory_r2[col-1];
        3'b100:      memvalue <= memory_r3[col-1];
        3'b101:      memvalue <= memory_r4[col-1];
        3'b110:      memvalue <= memory_r5[col-1];
        default: memvalue <= memory_r6[col-1];
    endcase

    if (count == 0 && row == 1 && col == 7 && ~i && ~j)
        empty <= 1;
    end
end
endmodule

```

## B.2.8 “avg\_perline.v”

```

//*****
// 2nd part of Accumulator2 (Accumulator2 has been divided into two
// parts, the other part is sum_for_avg.v), it computes the 7 times
// mean value of the input image.
//
// File Name: avg_perline.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module avg_perline
#(parameter Width = 23)
    (output reg [Width-13:0] avg_Digit,
     output reg [Width-19:0] avg_Decimal,
     output [Width-1:0] n_avg,
     input [Width-10:0] z,
     input [2:0] cj, row, col,
     input clk);

    wire [Width-1:0] w1, w2, w3;

    assign      w1 = {z, 9'b00000000} + {3'b000,z,6'b0000000};
    assign      w2 = {6'b000000,z,3'b000} + {9'b000000000,z};
    assign w3 = w1 + w2;
    assign n_avg = ~w3 + 1;

    always@(posedge clk)

```

```

        if (cj == 3 && row == 3 && col == 6)
        begin
            avg_Digit <= w3[22:12];
            avg_Decimal <= w3[11:7];
        end
    endmodule

```

### B.2.9 “mux2to1.v”

```

//*****
// MUX2 or MUX3, for selecting input to the Accumulator3
//
// File Name: mux2to1.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module mux2to1
#(parameter Width = 8)
    (output [Width-1:0] out,
     input [Width-1:0] in1, in2,
     input en1, en2);

    assign out = en1 ? in1 : (en2 ? in2 : 8'bxxxxxxxx);

endmodule

```

A modified version of this “mux2to1.v” file is given below when using array of registers as memory.

```

//*****
// MUX2 or MUX3, for selecting input to the Accumulator3
//
// File Name: mux2to1.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003

```

```
// Copyright (c) 2004
// All Right Reserved.
//*****

module mux2to1
#(parameter Width = 8)
  (output reg [Width-1:0] out,
   input [Width-1:0] in1, in2,
   input en1, en2);

always@(in1, in2, en1, en2)
  begin
    out = 0;
    if (en1) out = in1;
    else if (en2) out = in2;
  end
endmodule
```

### B.2.10 “acc\_FRAT.v”

```
//*****
// Accumulator3, generates FRAT coefficients
//
// File Name: acc_FRAT.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ualgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module acc_FRAT
#(parameter Width = 8)
  (output reg signed [Width+3:0] acc_out,
   input [Width-1:0] acc_in,
   input [Width+3:0] avg,
   input [2:0] col,
   input clock, reset, e1, e2);

reg [Width+3:0] sum;

always@(posedge clock, negedge reset)
  begin
    if (~reset)
      begin
        sum <= 0;

```

```

        acc_out <= 0;
    end

    else if (~e1 || ~e2)
    begin
        if (col != 1) sum <= sum + acc_in;
        else
            begin
                acc_out <= sum + acc_in + avg;
                sum <= 0;
            end
        end
    end

end

endmodule

```

### B.2.11 “delay\_line.v”

```

//*****
// Delay Line module
//
// File Name: delay_line.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module delay_line
#(parameter Width = 24)
    (output reg [Width-1:0] x0, x1,
     output reg [Width-3:0] x2,
     output reg [Width-1:0] x3,
     input signed [Width-1:0] ip,
     input [2:0] c1, c2,
     input clk);

    reg [Width-1:0] z1, z2, z3, z4, z2temp, z3temp, iptemp;
    reg [Width-3:0] z1temp;

    always@(posedge clk)
    begin
        if (c2 == 2)
            begin
                z4 <= z3;

```

```

        z3 <= z2;
        if (c1 == 1)      z2 <= ip;
            else z2 <= z1;
        z1 <= ip;
    end

    if (c2 == 3)
    begin

        if (~(c1[0] || c1[1] || c1[2]))
        begin
            z3temp <= z3;
            z2temp <= z2;
            z1temp <= z1[23:2];
            iptemp <= ip;
        end

        if (c1[0] && (c1[1] || c1[2]))
        begin
            x0 <= z4;
            x1 <= z3;
            x2 <= z2[23:2];
            x3 <= z1;
        end

        else if (c1[0] && ~c1[1] && ~c1[2])
        begin
            x0 <= z3temp;
            x1 <= z2temp;
            x2 <= z1temp;
            x3 <= iptemp;
        end
    end
end
endmodule

```

### B.2.12 “adder\_bfly.v”

```

//*****
// Adder Butterfly, generates the partial products for low pass and
// high pass FRIT coefficients.
//
// File Name: adder_bfly.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004

```

```
// All Right Reserved.
//*****

module adder_bfly
#(parameter Width = 25)
  (output reg signed [Width-1:0] w2,
   output reg signed [Width-10:0] w3,
   output reg signed [Width-1:0] w5,
   output reg signed [Width-3:0] w6,
   output reg signed [Width-5:0] w7,
   output reg signed [Width-4:0] w8,
   output reg signed [Width-3:0] w9,
   output reg signed [Width-3:0] w10,
   output reg signed [Width-5:0] w12,
   output reg signed [Width-5:0] w13,
   input signed [Width-2:0] x0, x1,
   input signed [Width-4:0] x2,
   input signed [Width-2:0] x3,
   input [2:0] c1,
   input clk, reset);

always@(posedge clk, negedge reset)
begin
  if (~reset)
  begin
    w2 <= 0; w6 <= 0;
    w8 <= 0; w10 <= 0;
    w13 <= 0; w3 <= 0;
    w5 <= 0; w7 <= 0;
    w9 <= 0; w12 <= 0;
  end
  else begin

    if (c1 == 4)
    begin
      w2 <= {x0[23],x0} + {x1[23],x1};
      w6 <= {x0[23],x0[23:2]} + {x2[21],x2};
      w8 <= {x0[23],x0[23:3]} + {x3[23],x3[23:3]};
      w10 <= {x1[23],x1[23:2]} + {x3[23],x3[23:2]};
      w13 <= {x2[21],x2[21:2]} + {x3[23],x3[23:4]};
    end
    else if (c1 == 5)
    begin
      w3 <= w2[24:9] + {x2[21],x2[21:7]};
      w5 <= w2 + {x3[23],x3};
      w7 <= w6[22:2] + {x3[23],x3[23:4]};
      w9 <= w6 + w10;
      w12 <= w13 + {x1[23],x1[23:4]};
    end
  end
end
endmodule
```

### B.2.13 “adr\_compressor.v”



```

        {z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z
9[13], z9[13], z9} +
        {z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z
9[13], z9[13], z9[13], z9[13:1]};

always@(posedge clk)
begin
    if (c1 != 0 && c2 == 6)
    begin
        LorH_Digit <= LH[22:12];
        LorH_Decimal <= LH[11:7];
    end
end

endmodule

```

### B.2.14 “counter.v”

```

//*****
// Counter module, this is actually a part of controller, coded
// separately.
//
// File Name: counter.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module counter
#(parameter width = 3)
(output reg [width-1:0] count,
input next_count, clk, rst);

always@(posedge clk, negedge rst)
    if (~rst) count <= 3'b000;
    else if (~next_count)
    begin
        if (count == 3'b111) count <= 3'b000;
        else count <= count + 3'b001;
    end
end

endmodule

```

### B.2.15 “controller.v”

```

//*****
// CONTROLLER
//
// File Name: controller.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module controller
#(parameter Width = 3)
  (output reg [Width-1:0] col,
   output reg [Width-1:0] col_dummy,
   output reg i, j,
   output reg next_count,
   output reg [Width-1:0] int_count_j,
   output reg [5:0] ADDW,
   output [5:0] ADDR,
   output reg empty_1, empty_2, en1, en2,
   output reg OP_ready,
   input [Width-1:0] row, count,
   input clk, reset);

  reg complete_sc, enable_1, enable_2;
  reg [Width-1:0] count_sc;
  reg switch, ready;
  reg [5:0] int_col;

  assign ADDR = int_col + col;

  always@(row)
    case (row)
      3'b001: int_col <= 6'b111111;
      3'b010: int_col <= 6'b000110;
      3'b011: int_col <= 6'b001101;
      3'b100: begin
        int_col <= 6'b010100;
        ready <= 1'b1;
      end
      3'b101: int_col <= 6'b011011;
      3'b110: int_col <= 6'b100010;
      3'b111: int_col <= 6'b101001;
      default: int_col <= 6'bxxxxxx;
    endcase

```

```

always@(posedge clk, negedge reset)
begin

    if (~reset)
    begin
        col <= 3'b000;
        i <= 0; j <= 0;
        next_count <= 1;
        int_count_j <= 0;
        count_sc <= 4;
        col_dummy <= 0;
        complete_sc <= 0;
        switch <= 0;

        ADDW <= 0;
        empty_1 <= 1;
        empty_2 <= 1;
        enable_1 <= 0;
        enable_2 <= 0;
        OP_ready <= 0;
        ready <= 0;
        en1 <= 0;
        en2 <= 0;

    end

    else begin

        en1 <= enable_1;
        en2 <= enable_2;

        if (count_sc == 0 && (~empty_1 || ~empty_2))
        begin
            if (col_dummy == 3'b110)
            begin
                j <= 0;
                next_count <= 1;
                col_dummy <= col_dummy +1;
                if (ready) OP_ready <= ~OP_ready;

                if (int_count_j == 3'b111 )
                begin
                    i <= 0;
                    int_count_j <= 0;
                    count_sc <= 1;
                end
                else int_count_j <= int_count_j + 1;
            end
            else if (int_count_j == 3'b111 && col_dummy == 3'b101)
            begin
                next_count <= 0;
                col_dummy <= col_dummy +1;
            end
        end
    end
end

```

```

        j <= 1;
        i <= 1;
    end

    else if (col_dummy == 3'b111)
    begin
        col_dummy <= 3'b001;
        if (complete_sc)
            begin
                if (col == 7) col <= 3'b001;
                else col <= col +1;
            end
        else complete_sc <= ~complete_sc;

        j <= 1;
        i <= 1;
        next_count <= 1;
    end

    else
    begin
        col_dummy <= col_dummy +1;
        j <= 1;
        i <= 1;
        next_count <= 1;
    end
end

else if (col_dummy == 3'b110 && (~empty_1 || ~empty_2))
begin
    j <= 0;
    next_count <= 1;
    col_dummy <= col_dummy +1;
    col <= col + 'b1;
    if (ready) OP_ready <= ~OP_ready;

    if (int_count_j == 3'b111 )
    begin
        i <= 0;
        int_count_j <= 0;
        if (count_sc != 7)
            count_sc <= count_sc + 1;

        else
        begin
            count_sc <= 0;
            complete_sc <= 0;
        end
    end
    else int_count_j <= int_count_j + 1;
end

else if (int_count_j == 3'b111 && col_dummy == 3'b101 &&
(~empty_1 || ~empty_2))

```

```

begin
    next_count <= 0;
    col_dummy <= col_dummy +1;
    col <= col + 'b1;
    j <= 1;
    i <= 1;
end

else if (col_dummy == 3'b111 && (~empty_1 || ~empty_2))
begin
    col_dummy <= 3'b001;
    col <= 3'b001;
    j <= 1;
    i <= 1;
    next_count <= 1;
end

else if (~empty_1 || ~empty_2)
begin
    col_dummy <= col_dummy +1;
    col <= col + 'b1;
    j <= 1;
    i <= 1;
    next_count <= 1;
end

if (empty_1 || empty_2)
begin
    if (ADDW == 48)
begin
    ADDW <= 0;

    if (~switch) begin
        empty_1 <= 0;
        switch <= ~switch;
    end
    else begin
        empty_2 <= 0;
        switch <= ~switch;
    end
end
    else ADDW <= ADDW + 1;
end

if (~empty_1 && ~enable_2)
begin
    if (count == 0 && row == 1 && col == 7 && ~i && ~j)
begin
    empty_1 <= 1;
    enable_1 <= 0;
    enable_2 <= 1;
end
    else begin
        enable_1 <= 1;
    end
end

```

```

        enable_2 <= 0;
    end
end

if (~empty_2 && ~enable_1)
    begin
        if (count == 0 && row == 1 && col == 7 && ~i && ~j)
            begin
                empty_2 <= 1;
                enable_1 <= 1;
                enable_2 <= 0;
            end
        else begin
            enable_1 <= 0;
            enable_2 <= 1;
        end
    end
end

end

endmodule

```

A modified version of this “controller.v” file is given below when using array of registers as memory.

```

//*****
// CONTROLLER
//
// File Name: controller.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module controller
#(parameter Width = 3)
    (output reg [Width-1:0] col,
    output reg [Width-1:0] col_dummy,
    output reg i, j,
    output reg next_count,
    output reg [Width-1:0] int_count_j,
    output reg OP_ready,
    input [Width-1:0] row,

```

```

input clk, e1, e2, reset);

reg complete_sc, ready;
reg [Width-1:0] count_sc;

always@(posedge clk, negedge reset)
begin
    if (~reset)
        begin
            col <= 3'b000;
            i <= 0; j <= 0;
            next_count <= 1;
            int_count_j <= 0;
            count_sc <= 4;
            col_dummy <= 0;
            complete_sc <= 0;
            OP_ready <= 0;
            ready <= 0;
        end

    else if (count_sc == 0 && (~e1 || ~e2))
        begin
            if (row == 4) ready <= 1'b1;
            if (col_dummy == 3'b110)
                begin
                    j <= 0;
                    next_count <= 1;
                    col_dummy <= col_dummy +1;
                    if (ready) OP_ready <= ~OP_ready;

                    if (int_count_j == 3'b111 )
                        begin
                            i <= 0;
                            int_count_j <= 0;
                            count_sc <= 1;
                        end
                    else int_count_j <= int_count_j + 1;
                end
            else if (int_count_j == 3'b111 && col_dummy == 3'b101)
                begin
                    next_count <= 0;
                    col_dummy <= col_dummy +1;
                    j <= 1;
                    i <= 1;
                end
            end

    else if (col_dummy == 3'b111)
        begin
            col_dummy <= 3'b001;
            if (complete_sc)
                begin
                    if (col == 7) col <= 3'b001;
                    else col <= col +1;
                end
        end
end

```

```

        else complete_sc <= ~complete_sc;

        j <= 1;
        i <= 1;
        next_count <= 1;
    end

    else
    begin
        col_dummy <= col_dummy +1;
        j <= 1;
        i <= 1;
        next_count <= 1;
    end
end

else if (col_dummy == 3'b110 && (~e1 || ~e2))
begin
    if (row == 4) ready <= 1'b1;
    j <= 0;
    next_count <= 1;
    col_dummy <= col_dummy +1;
    col <= col + 'b1;
    if (ready) OP_ready <= ~OP_ready;

    if (int_count_j == 3'b111 )
    begin
        i <= 0;
        int_count_j <= 0;
        if (count_sc != 7)
            count_sc <= count_sc + 1;

        else
        begin
            count_sc <= 0;
            complete_sc <= 0;
        end
    end
    else int_count_j <= int_count_j + 1;
end

else if (int_count_j == 3'b111 && col_dummy == 3'b101 && (~e1 ||
~e2))
begin
    if (row == 4) ready <= 1'b1;
    next_count <= 0;
    col_dummy <= col_dummy +1;
    col <= col + 'b1;
    j <= 1;
    i <= 1;
end

else if (col_dummy == 3'b111 && (~e1 || ~e2))
begin

```



```

        if (row == 4) ready <= 1'b1;
        col_dummy <= 3'b001;
        col <= 3'b001;
        j <= 1;
        i <= 1;
        next_count <= 1;
    end

    else if (~e1 || ~e2)
    begin
        if (row == 4) ready <= 1'b1;
        col_dummy <= col_dummy +1;
        col <= col + 'b1;
        j <= 1;
        i <= 1;
        next_count <= 1;
    end

end
endmodule

```

## B.3 Codes for Memoryless FRIT Architecture

There are 12 Verilog HDL files in total for this architecture. These are – “stimulus.v”, “module\_top.v”, “logic\_INIT.v”, “add\_gen.v”, “muxip.v”, “add\_pixels.v”, “frat.v”, “delay\_line.v”, “adder\_bfly.v”, “adr\_compressor.v”, “counter.v” and “controller.v”.

### B.3.1 “stimulus.v”

```

//*****
// Stimulus for simulation
//
// File Name: stimulus.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module stimulus

```

```

#(parameter Width = 8, Qblocks = 546, BElements = 49, TElements =
104958);
// TElements = 104958 for CIF and 26754 for QCIF
reg [Width-1:0] memory_input [0:TElements-1];
reg [Width-1:0] ip00, ip01, ip02, ip03, ip04, ip05, ip06,
            ip10, ip11, ip12, ip13, ip14, ip15, ip16,
            ip20, ip21, ip22, ip23, ip24, ip25, ip26,
            ip30, ip31, ip32, ip33, ip34, ip35, ip36,
            ip40, ip41, ip42, ip43, ip44, ip45, ip46,
            ip50, ip51, ip52, ip53, ip54, ip55, ip56,
            ip60, ip61, ip62, ip63, ip64, ip65, ip66;

reg clk, rst;
wire [Width+7:0] L, H, AVG;

integer i, k, file1, file2;

module_top m_top(L, H, AVG, OP_ready, C_ready,
            ip00, ip01, ip02, ip03, ip04, ip05, ip06,
            ip10, ip11, ip12, ip13, ip14, ip15, ip16,
            ip20, ip21, ip22, ip23, ip24, ip25, ip26,
            ip30, ip31, ip32, ip33, ip34, ip35, ip36,
            ip40, ip41, ip42, ip43, ip44, ip45, ip46,
            ip50, ip51, ip52, ip53, ip54, ip55, ip56,
            ip60, ip61, ip62, ip63, ip64, ip65, ip66, clk, rst);

initial
begin
    rst = 1'b0;
    clk = 1'b1;

    i = 0;
    k = 0;

    $readmemb("ipframe.txt",memory_input);
    file1 = $fopen("frit_LP.txt");
    file2 = $fopen("frit_HP.txt");

    #15 rst = ~rst;
end

always #10 clk = ~clk; // a clock of 10 tu period

always@(AVG)
    if (i == TElements && ~k) begin
        $fclose(file1);
        $fclose(file2);
        $stop;
    end else k = ~k;

always@(OP_ready)
    if (OP_ready && rst) begin
        $fdisplay(file1, "%b %b %b %b", L[15:5], L[4:0], AVG[15:5],
AVG[4:0]);
    end

```

```

        $fdisplay(file2, "%b %b", H[15:5], H[4:0]);
    end

always@(posedge clk)
    if (C_ready)
        begin
            if (i == TElements)
                begin
                    ip00 = 0;
                    ip01 = 0;
                    ip02 = 0;
                    ip03 = 0;
                    ip04 = 0;
                    ip05 = 0;
                    ip06 = 0;
                    ip10 = 0;
                    ip11 = 0;
                    ip12 = 0;
                    ip13 = 0;
                    ip14 = 0;
                    ip15 = 0;
                    ip16 = 0;
                    ip20 = 0;
                    ip21 = 0;
                    ip22 = 0;
                    ip23 = 0;
                    ip24 = 0;
                    ip25 = 0;
                    ip26 = 0;
                    ip30 = 0;
                    ip31 = 0;
                    ip32 = 0;
                    ip33 = 0;
                    ip34 = 0;
                    ip35 = 0;
                    ip36 = 0;
                    ip40 = 0;
                    ip41 = 0;
                    ip42 = 0;
                    ip43 = 0;
                    ip44 = 0;
                    ip45 = 0;
                    ip46 = 0;
                    ip50 = 0;
                    ip51 = 0;
                    ip52 = 0;
                    ip53 = 0;
                    ip54 = 0;
                    ip55 = 0;
                    ip56 = 0;
                    ip60 = 0;
                    ip61 = 0;
                    ip62 = 0;
                    ip63 = 0;
                end
            else
                begin
                    ip00 = 0;
                    ip01 = 0;
                    ip02 = 0;
                    ip03 = 0;
                    ip04 = 0;
                    ip05 = 0;
                    ip06 = 0;
                    ip10 = 0;
                    ip11 = 0;
                    ip12 = 0;
                    ip13 = 0;
                    ip14 = 0;
                    ip15 = 0;
                    ip16 = 0;
                    ip20 = 0;
                    ip21 = 0;
                    ip22 = 0;
                    ip23 = 0;
                    ip24 = 0;
                    ip25 = 0;
                    ip26 = 0;
                    ip30 = 0;
                    ip31 = 0;
                    ip32 = 0;
                    ip33 = 0;
                    ip34 = 0;
                    ip35 = 0;
                    ip36 = 0;
                    ip40 = 0;
                    ip41 = 0;
                    ip42 = 0;
                    ip43 = 0;
                    ip44 = 0;
                    ip45 = 0;
                    ip46 = 0;
                    ip50 = 0;
                    ip51 = 0;
                    ip52 = 0;
                    ip53 = 0;
                    ip54 = 0;
                    ip55 = 0;
                    ip56 = 0;
                    ip60 = 0;
                    ip61 = 0;
                    ip62 = 0;
                    ip63 = 0;
                end
            i = i + 1;
        end
    end
end

```

```

        ip64 = 0;
        ip65 = 0;
        ip66 = 0;
    end
else
    begin
        ip00 = memory_input[i];
        ip01 = memory_input[i+1];
        ip02 = memory_input[i+2];
        ip03 = memory_input[i+3];
        ip04 = memory_input[i+4];
        ip05 = memory_input[i+5];
        ip06 = memory_input[i+6];
        ip10 = memory_input[i+7];
        ip11 = memory_input[i+8];
        ip12 = memory_input[i+9];
        ip13 = memory_input[i+10];
        ip14 = memory_input[i+11];
        ip15 = memory_input[i+12];
        ip16 = memory_input[i+13];
        ip20 = memory_input[i+14];
        ip21 = memory_input[i+15];
        ip22 = memory_input[i+16];
        ip23 = memory_input[i+17];
        ip24 = memory_input[i+18];
        ip25 = memory_input[i+19];
        ip26 = memory_input[i+20];
        ip30 = memory_input[i+21];
        ip31 = memory_input[i+22];
        ip32 = memory_input[i+23];
        ip33 = memory_input[i+24];
        ip34 = memory_input[i+25];
        ip35 = memory_input[i+26];
        ip36 = memory_input[i+27];
        ip40 = memory_input[i+28];
        ip41 = memory_input[i+29];
        ip42 = memory_input[i+30];
        ip43 = memory_input[i+31];
        ip44 = memory_input[i+32];
        ip45 = memory_input[i+33];
        ip46 = memory_input[i+34];
        ip50 = memory_input[i+35];
        ip51 = memory_input[i+36];
        ip52 = memory_input[i+37];
        ip53 = memory_input[i+38];
        ip54 = memory_input[i+39];
        ip55 = memory_input[i+40];
        ip56 = memory_input[i+41];
        ip60 = memory_input[i+42];
        ip61 = memory_input[i+43];
        ip62 = memory_input[i+44];
        ip63 = memory_input[i+45];
        ip64 = memory_input[i+46];
        ip65 = memory_input[i+47];
    end
end

```

```

        ip66 = memory_input[i+48];

        i = i + 49;
    end
end
endmodule

```

### B.3.2 “module\_top.v”

```

//*****
// Top Module
//
// File Name: module_top.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahmanc@enl.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module module_top
#(parameter Width = 8)
(output [Width+7:0] L, H, AVG,
 output OP_ready, C_ready,
 input [Width-1:0] ip00, ip01, ip02, ip03, ip04, ip05, ip06,
        ip10, ip11, ip12, ip13, ip14, ip15, ip16,
        ip20, ip21, ip22, ip23, ip24, ip25, ip26,
        ip30, ip31, ip32, ip33, ip34, ip35, ip36,
        ip40, ip41, ip42, ip43, ip44, ip45, ip46,
        ip50, ip51, ip52, ip53, ip54, ip55, ip56,
        ip60, ip61, ip62, ip63, ip64, ip65, ip66,
 input clk, rst);

wire [Width-6:0] col1, col2, col3, col4, col5, col6, col7,
        add1, add2, add3, add4, add5, add6, add7,
        count, m1, m2, m3, m4, m5, m6, m7, c, col_dummy;

wire [Width-4:0] L_Decimal, H_Decimal, avg_Decimal;
wire [Width-1:0] op1, op2, op3, op4, op5, op6, op7;

wire signed [Width+2:0] ap_out, L_Digit, H_Digit, avg_Digit;
wire [Width+7:0] w3;
wire [Width+12:0] w7, w12, w13;
wire [Width+13:0] w8;
wire [Width+14:0] w6, w9, w10;
wire [Width+16:0] w2, w5;

```

```

wire [Width+13:0] x2;
wire [Width+15:0] x0, x1, x3, opfrat;

assign L = {L_Digit, L_Decimal};
assign H = {H_Digit, H_Decimal};
assign AVG = {avg_Digit, avg_Decimal};

counter countr (count, nc, clk, rst);
logicINIT lINIT (m1, m2, m3, m4, m5, m6, m7, c, count);

add_gen add_1 (add1, m1, c, i, clk, rst);
add_gen add_2 (add2, m2, c, i, clk, rst);
add_gen add_3 (add3, m3, c, i, clk, rst);
add_gen add_4 (add4, m4, c, i, clk, rst);
add_gen add_5 (add5, m5, c, i, clk, rst);
add_gen add_6 (add6, m6, c, i, clk, rst);
add_gen add_7 (add7, m7, c, i, clk, rst);

muxip mux_ip (op1, op2, op3, op4, op5, op6, op7, ip00, ip01, ip02,
ip03, ip04, ip05, ip06,
            ip10, ip11, ip12, ip13, ip14, ip15, ip16, ip20, ip21, ip22,
ip23, ip24, ip25, ip26,
            ip30, ip31, ip32, ip33, ip34, ip35, ip36, ip40, ip41, ip42,
ip43, ip44, ip45, ip46,
            ip50, ip51, ip52, ip53, ip54, ip55, ip56, ip60, ip61, ip62,
ip63, ip64, ip65, ip66,
            add1, add2, add3, add4, add5, add6, add7, col1, col2, col3,
col4, col5, col6, col7);

add_pixels ap (ap_out, op1, op2, op3, op4, op5, op6, op7);
frat fratop (opfrat, avg_Digit, avg_Decimal, ap_out, count, col_dummy,
nc, i, clk, rst);

controller contrl (col1, col2, col3, col4, col5, col6, col7, col_dummy,
C_ready, OP_ready, i, nc, count, clk, rst);
delay_line dl (x0, x1, x2, x3, opfrat, col_dummy, clk);

adder_bfly bfly (w2, w3, w5, w6, w7, w8, w9, w10, w12, w13, x0, x1, x2,
x3);

adr_compressor lowPass (L_Digit, L_Decimal, {x3[23], x3}, w10, w8, w12,
{x0[23], x0[23:5]}, w6[22:4], w13[20:4], w3, w9[22:9]);
adr_compressor highPass (H_Digit, H_Decimal, w5, w9, w2[24:3], w7,
{x1[23], x1[23:5]},
19'b00000000000000000000, w8[21:5], {x2[21], x2[21:7]}, w6[22:9]);

endmodule

```

### B.3.3 “logic\_INIT.v”

```

//*****
// Address Logic Initializer, generates m1, m2, m3, m4, m5, m6, m7
// and c values.
//
// File Name: logicINIT.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module logicINIT
#(parameter width = 3)
  (output reg [width-1:0] m1, m2, m3, m4, m5, m6, m7, c,
   input [width-1:0] count);

always@(count)
  case(count)
    3'b000: begin
      m1 <= 3'b111; m2 <= 3'b111; m3 <= 3'b111; m4 <= 3'b111;
      m5 <= 3'b111; m6 <= 3'b111; m7 <= 3'b111; c <= 3'b001;
    end
    3'b001: begin
      m1 <= 3'b100; m2 <= 3'b111; m3 <= 3'b011; m4 <= 3'b110;
      m5 <= 3'b010; m6 <= 3'b101; m7 <= 3'b001; c <= 3'b100;
    end
    3'b010: begin
      m1 <= 3'b111; m2 <= 3'b110; m3 <= 3'b101; m4 <= 3'b100;
      m5 <= 3'b011; m6 <= 3'b010; m7 <= 3'b001; c <= 3'b001;
    end
    3'b011: begin
      m1 <= 3'b111; m2 <= 3'b101; m3 <= 3'b011; m4 <= 3'b001;
      m5 <= 3'b110; m6 <= 3'b100; m7 <= 3'b010; c <= 3'b001;
    end
    3'b100: begin
      m1 <= 3'b001; m2 <= 3'b010; m3 <= 3'b011; m4 <= 3'b100;
      m5 <= 3'b101; m6 <= 3'b110; m7 <= 3'b111; c <= 3'b000;
    end
    3'b101: begin
      m1 <= 3'b010; m2 <= 3'b100; m3 <= 3'b110; m4 <= 3'b001;
      m5 <= 3'b011; m6 <= 3'b101; m7 <= 3'b111; c <= 3'b110;
    end
    3'b110: begin
      m1 <= 3'b010; m2 <= 3'b011; m3 <= 3'b100; m4 <= 3'b101;
      m5 <= 3'b110; m6 <= 3'b111; m7 <= 3'b001; c <= 3'b110;
    end
  endcase
end

```

```

        3'b111: begin
            m1 <= 3'b101; m2 <= 3'b010; m3 <= 3'b110; m4 <= 3'b011;
            m5 <= 3'b111; m6 <= 3'b100; m7 <= 3'b001; c <= 3'b011;
        end
    endcase
endmodule

```

### B.3.4 “add\_gen.v”

```

//*****
// AG, generates row addresses of the pixels.
//
// File Name: add_gen.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

```

```

module add_gen
#(parameter Width = 3)
(output reg [Width-1:0] addr,
 input [Width-1:0] m, c,
 input j, clock, reset);

reg carry;
reg [Width-1:0] sum;

always@(posedge clock, negedge reset)
begin
    if (~reset)
    begin
        addr <= 0;
        carry <= 0;
        sum <= 0;
    end
    else
    begin
        if (~j)
        begin
            addr <= m;
            {carry, sum} <= m + c;
        end
        else
        begin
            addr <= sum + carry;

```



```

        {carry, sum} <= sum + c + carry;
    end
end
end
endmodule

```

### B.3.5 “muxip.v”

```

//*****
// MUX module, selects 7 inputs every cycle for computing FRAT
// coefficients.
//
// File Name: muxip.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module muxip
#(parameter Width = 8)
    (output reg [Width-1:0] op1, op2, op3, op4, op5, op6, op7,
    input [Width-1:0] ip00, ip01, ip02, ip03, ip04, ip05, ip06, ip10,
    ip11, ip12, ip13, ip14, ip15, ip16,
    ip20, ip21, ip22, ip23, ip24, ip25, ip26, ip30, ip31, ip32,
    ip33, ip34, ip35, ip36,
    ip40, ip41, ip42, ip43, ip44, ip45, ip46, ip50, ip51, ip52,
    ip53, ip54, ip55, ip56,
    ip60, ip61, ip62, ip63, ip64, ip65, ip66,
    input [Width-6:0] add1, add2, add3, add4, add5, add6, add7, col1,
    col2, col3, col4, col5, col6, col7);

    always@(ip00, ip01, ip02, ip03, ip04, ip05, ip06, ip10, ip11, ip12,
    ip13, ip14, ip15, ip16,
    ip20, ip21, ip22, ip23, ip24, ip25, ip26, ip30, ip31, ip32, ip33,
    ip34, ip35, ip36,
    ip40, ip41, ip42, ip43, ip44, ip45, ip46, ip50, ip51, ip52, ip53,
    ip54, ip55, ip56,
    ip60, ip61, ip62, ip63, ip64, ip65, ip66, add1, add2, add3, add4,
    add5, add6, add7,
    col1, col2, col3, col4, col5, col6, col7)
    begin
        case({add1, col1})
            6'b001001: op1 = ip00;
            6'b010001: op1 = ip10;

```

```

6'b011001: op1 = ip20;
6'b100001: op1 = ip30;
6'b101001: op1 = ip40;
6'b110001: op1 = ip50;
6'b111001: op1 = ip60;
6'b001010: op1 = ip01;
6'b001011: op1 = ip02;
6'b001100: op1 = ip03;
6'b001101: op1 = ip04;
6'b001110: op1 = ip05;
6'b001111: op1 = ip06;
default: op1 = 0;
endcase

```

```

case({add2, col2})
6'b001010: op2 = ip01;
6'b010010: op2 = ip11;
6'b011010: op2 = ip21;
6'b100010: op2 = ip31;
6'b101010: op2 = ip41;
6'b110010: op2 = ip51;
6'b111010: op2 = ip61;
6'b010001: op2 = ip10;
6'b010011: op2 = ip12;
6'b010100: op2 = ip13;
6'b010101: op2 = ip14;
6'b010110: op2 = ip15;
6'b010111: op2 = ip16;
default: op2 = 0;
endcase

```

```

case({add3, col3})
6'b001011: op3 = ip02;
6'b010011: op3 = ip12;
6'b011011: op3 = ip22;
6'b100011: op3 = ip32;
6'b101011: op3 = ip42;
6'b110011: op3 = ip52;
6'b111011: op3 = ip62;
6'b011001: op3 = ip20;
6'b011010: op3 = ip21;
6'b011100: op3 = ip23;
6'b011101: op3 = ip24;
6'b011110: op3 = ip25;
6'b011111: op3 = ip26;
default: op3 = 0;
endcase

```

```

case({add4, col4})
6'b001100: op4 = ip03;
6'b010100: op4 = ip13;
6'b011100: op4 = ip23;
6'b100100: op4 = ip33;
6'b101100: op4 = ip43;

```

```

6'b110100: op4 = ip53;
6'b111100: op4 = ip63;
6'b100001: op4 = ip30;
6'b100010: op4 = ip31;
6'b100011: op4 = ip32;
6'b100101: op4 = ip34;
6'b100110: op4 = ip35;
6'b100111: op4 = ip36;
default: op4 = 0;
endcase

```

```

case({add5, col5})
6'b001101: op5 = ip04;
6'b010101: op5 = ip14;
6'b011101: op5 = ip24;
6'b100101: op5 = ip34;
6'b101101: op5 = ip44;
6'b110101: op5 = ip54;
6'b111101: op5 = ip64;
6'b101001: op5 = ip40;
6'b101010: op5 = ip41;
6'b101011: op5 = ip42;
6'b101100: op5 = ip43;
6'b101110: op5 = ip45;
6'b101111: op5 = ip46;
default: op5 = 0;
endcase

```

```

case({add6, col6})
6'b001110: op6 = ip05;
6'b010110: op6 = ip15;
6'b011110: op6 = ip25;
6'b100110: op6 = ip35;
6'b101110: op6 = ip45;
6'b110110: op6 = ip55;
6'b111110: op6 = ip65;
6'b110001: op6 = ip50;
6'b110010: op6 = ip51;
6'b110011: op6 = ip52;
6'b110100: op6 = ip53;
6'b110101: op6 = ip54;
6'b110111: op6 = ip56;
default: op6 = 0;
endcase

```

```

case({add7, col7})
6'b001111: op7 = ip06;
6'b010111: op7 = ip16;
6'b011111: op7 = ip26;
6'b100111: op7 = ip36;
6'b101111: op7 = ip46;
6'b110111: op7 = ip56;
6'b111111: op7 = ip66;
6'b111001: op7 = ip60;

```

```

        6'b111010: op7 = ip61;
        6'b111011: op7 = ip62;
        6'b111100: op7 = ip63;
        6'b111101: op7 = ip64;
        6'b111110: op7 = ip65;
        default: op7 = 0;
    endcase
end
endmodule

```

### B.3.6 “add\_pixels.v”

```

//*****
// 1st part of Adder block, (Adder block has been divided into two
// parts, the other part is frat.v), adds 7 outputs of MUX module
// every cycle for computing FRAT coefficients.
//
// File Name: add_pixels.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahmanc@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module add_pixels
#(parameter Width = 8)
    (output [Width+2:0] op,
     input [Width-1:0] x1, x2, x3, x4, x5, x6, x7);

    wire [Width+2:0] w1, w2, w3, w4, w5;

    assign      w1 = x1 + x2;
    assign      w2 = x3 + x4;
    assign      w3 = x5 + x6;
    assign      w4 = w1 + x7;
    assign      w5 = w2 + w3;
    assign      op = w4 + w5;

endmodule

```

### B.3.7 “frat.v”

```

//*****
// 2st part of Adder block, (Adder block has been divided into two
// parts, the other part is add_pixels.v), computes 7 times mean
// value of the input image and the FRAT coefficients.
//
// File Name: frat.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module frat
#(parameter Width = 8)
(output [Width+15:0] opfrat,
 output reg [Width+2:0] avg_Digit,
 output reg [Width-4:0] avg_Decimal,
 input [Width+2:0] ap_out,
 input [Width-6:0] count,
 input [Width-6:0] col_dummy,
 input nc, i, clk, rst);

reg [Width+2:0] q1, q2, q3, q4, q5, q6, q7;
wire [Width+3:0] opfrat_temp;
reg [Width+5:0] sum, sum_out;
wire [Width+14:0] avg, w1, w2, w3, op;

assign      w1 = {sum_out,9'b000000000} + {3'b000,sum_out,6'b0000000};
assign      w2 = {6'b0000000,sum_out,3'b000} + {9'b0000000000,sum_out};
assign      avg = w1 + w2;

assign op = ~avg + 1;
assign opfrat_temp = q1 + {1'b1,op[22:12]};
assign opfrat = {opfrat_temp, op[11:0]};

always@(posedge clk, negedge rst)
begin
    if (~rst)
        begin
            sum <= 0;
            sum_out <= 0;
            q1 <= 11'bxxxxxxxxxxxx;
            q2 <= 11'bxxxxxxxxxxxx;
            q3 <= 11'bxxxxxxxxxxxx;
            q4 <= 11'bxxxxxxxxxxxx;
            q5 <= 11'bxxxxxxxxxxxx;

```

```

        q6 <= 11'bxxxxxxxxxxxx;
        q7 <= 11'bxxxxxxxxxxxx;
        avg_Digit <= 0;
        avg_Decimal <= 0;
    end
else
    begin
        if (col_dummy == 2)
            begin
                avg_Digit <= avg[22:12];
                avg_Decimal <= avg[11:7];
            end

        if (count == 0 && i)
            begin
                if (nc) sum <= sum + ap_out;
                else
                    begin
                        sum_out <= sum + ap_out;
                        sum <= 0;
                    end
            end

        end

        q1 <= q2;
        q2 <= q3;
        q3 <= q4;
        q4 <= q5;
        q5 <= q6;
        q6 <= q7;
        q7 <= ap_out;
    end
end
endmodule

```

### B.3.8 “delay\_line.v”

```

//*****
// Delay Line module
//
// File Name: delay_line.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

```

```

module delay_line
#(parameter Width = 24)
  (output reg [Width-1:0] x0, x1,
   output reg [Width-3:0] x2,
   output reg [Width-1:0] x3,
   input signed [Width-1:0] ip,
   input [Width-22:0] c1,
   input clk);

  reg [Width-1:0] z1, z2, z3;

  always@(posedge clk)
  begin
    z3 <= z2;
    z2 <= z1;
    z1 <= ip;

    if (c1 == 4 || c1 == 6)
      begin
        x0 <= z3;
        x1 <= z2;
        x2 <= z1[23:2];
        x3 <= ip;
      end
    else if (c1 == 2)
      begin
        x0 <= z2;
        x1 <= z2;
        x2 <= z1[23:2];
        x3 <= ip;
      end
    else if (c1 == 0)
      begin
        x0 <= z3;
        x1 <= z2;
        x2 <= z1[23:2];
        x3 <= z1;
      end
  end

end
endmodule

```

### B.3.9 “adder\_bfly.v”

```

//*****
// Adder Butterfly, generates the partial products for low pass and
// high pass FRIT coefficients.
//
// File Name: adder_bfly.v
// Design Type: Verilog .v file

```

```
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****
```

```
module adder_bfly
#(parameter Width = 25)
(output signed [Width-1:0] w2,
 output signed [Width-10:0] w3,
 output signed [Width-1:0] w5,
 output signed [Width-3:0] w6,
 output signed [Width-5:0] w7,
 output signed [Width-4:0] w8,
 output signed [Width-3:0] w9,
 output signed [Width-3:0] w10,
 output signed [Width-5:0] w12,
 output signed [Width-5:0] w13,
 input signed [Width-2:0] x0, x1,
 input signed [Width-4:0] x2,
 input signed [Width-2:0] x3);

assign w2 = {x0[23],x0} + {x1[23],x1};
assign w6 = {x0[23],x0[23:2]} + {x2[21],x2};
assign w8 = {x0[23],x0[23:3]} + {x3[23],x3[23:3]};
assign w10 = {x1[23],x1[23:2]} + {x3[23],x3[23:2]};
assign w13 = {x2[21],x2[21:2]} + {x3[23],x3[23:4]};
assign w3 = w2[24:9] + {x2[21],x2[21:7]};
assign w5 = w2 + {x3[23],x3};
assign w7 = w6[22:2] + {x3[23],x3[23:4]};
assign w9 = w6 + w10;
assign w12 = w13 + {x1[23],x1[23:4]};

endmodule
```

### B.3.10 “adr\_compressor.v”

```
//*****
// Adder Compressor Array, gives the low pass and high pass FRIT
// coefficients.
//
// File Name: adr_compressor.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
```



```

//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module adr_compressor
#(parameter Width = 8)
  (output reg signed [Width+2:0] LorH_Digit, //11bit
   output reg [Width-4:0] LorH_Decimal, //5bit
   input signed [Width+16:0] z1,
   input signed [Width+14:0] z2,
   input signed [Width+13:0] z3,
   input signed [Width+12:0] z4,
   input signed [Width+11:0] z5,
   input signed [Width+10:0] z6,
   input signed [Width+8:0] z7,
   input signed [Width+7:0] z8,
   input signed [Width+5:0] z9);

  wire [24:0] LH, z1_inv;

  assign z1_inv = ~z1 + 1;

  assign LH = z1_inv +
    {z1[24], z1[24:1]} +
    {z2[22], z2[22], z2} +
    {z3[21], z3[21], z3[21], z3} +
    {z4[20], z4[20], z4[20], z4[20], z4} +
    {z5[19], z5[19], z5[19], z5[19], z5[19], z5} +
    {z6[18], z6[18], z6[18], z6[18], z6[18], z6[18], z6} +
    {z3[21], z3[21], z3[21], z3[21], z3[21], z3[21], z3[21], z3[21:4]} +
    {z7[16], z7[16], z7[16], z7[16], z7[16], z7[16], z7[16], z7[16], z7} +
    {z8[15], z8[15], z8[15], z8[15], z8[15], z8[15], z8[15], z8[15], z8[15], z
  8} +
    {z2[22], z2[22], z2[22], z2[22], z2[22], z2[22], z2[22], z2[22], z2[22], z
  2[22], z2[22:8]} +
    {z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z
  9[13], z9[13], z9} +
    {z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z9[13], z
  9[13], z9[13], z9[13], z9[13:1]};

  always@(LH)
  begin
    LorH_Digit <= LH[22:12];
    LorH_Decimal <= LH[11:7];
  end

endmodule

```

### B.3.11 “counter.v”

```

//*****
// Counter module, this is actually a part of controller, coded
// separately.
//
// File Name: counter.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module counter
#(parameter width = 3)
  (output reg [width-1:0] count,
   input next_count, clk, rst);

always@(posedge clk, negedge rst)
  if (~rst) count <= 3'b000;
  else if (~next_count)
    begin
      if (count == 3'b111) count <= 3'b000;
      else count <= count + 3'b001;
    end
endmodule

```

### B.3.12 “controller.v”

```

//*****
// CONTROLLER
//
// File Name: controller.v
// Design Type: Verilog .v file
//
// This hardware utility is originally developed by Choudhury Ashiq
// Rahman {rahman@enel.ucalgary.ca} as a part of M.Sc research work.
//
// Last Modified: Dec, 2003
// Copyright (c) 2004
// All Right Reserved.
//*****

module controller
#(parameter Width = 3)

```

```

(output reg [Width-1:0] col1,
output reg [Width-1:0] col2,
output reg [Width-1:0] col3,
output reg [Width-1:0] col4,
output reg [Width-1:0] col5,
output reg [Width-1:0] col6,
output reg [Width-1:0] col7,
output reg [Width-1:0] col_dummy,
output reg C_ready, OP_ready, i,
output reg next_count,
input [Width-1:0] count,
input clk, reset);

reg [Width-1:0] count_sc;
reg ready;

always@(posedge clk, negedge reset)
begin
    C_ready <= 0;
    if (~reset)
        begin
            col1 <= 3'b001;
            col2 <= 3'b010;
            col3 <= 3'b011;
            col4 <= 3'b100;
            col5 <= 3'b101;
            col6 <= 3'b110;
            col7 <= 3'b111;
            i <= 0;
            next_count <= 1;
            count_sc <= 4;
            col_dummy <= 0;
            C_ready <= 1;
            OP_ready <= 0;
            ready <= 0;
        end

    else if (count_sc == 0 )
        begin
            if (count[0] && col_dummy[1]) ready <= 1;
            if (ready) OP_ready <= ~OP_ready;
            if (count == 7 && ~next_count) C_ready <= 1;
            if (col_dummy == 0)
                begin
                    col1 <= 3'b111;
                    col2 <= 3'b111;
                    col3 <= 3'b111;
                    col4 <= 3'b111;
                    col5 <= 3'b111;
                    col6 <= 3'b111;
                    col7 <= 3'b111;
                    col_dummy <= col_dummy +1;
                    next_count <= 1;
                    i <= 1;
                end
        end
end

```

```

end
else
  begin
    if (col_dummy == 3'b111)
      begin
        next_count <= 1;
        i <= 0;
        count_sc <= count_sc + 1;
        col_dummy <= 3'b000;

        end
      else
        begin
          if (col_dummy == 3'b110) next_count <= 0;
          else next_count <= 1;
          col_dummy <= col_dummy +1;
          i <= 1;

          end

        if (col1 == 7)
          begin
            col1 <= 3'b001;
            col2 <= 3'b001;
            col3 <= 3'b001;
            col4 <= 3'b001;
            col5 <= 3'b001;
            col6 <= 3'b001;
            col7 <= 3'b001;

            end
          else
            begin
              col1 <= col1 + 3'b001;
              col2 <= col2 + 3'b001;
              col3 <= col3 + 3'b001;
              col4 <= col4 + 3'b001;
              col5 <= col5 + 3'b001;
              col6 <= col6 + 3'b001;
              col7 <= col7 + 3'b001;

              end
            end
          end

        end if (col_dummy == 3'b111 )
        begin
          if (count[0] && col_dummy[1]) ready <= 1;
          if (ready) OP_ready <= ~OP_ready;
          if (count == 7 && ~next_count) C_ready <= 1;
          col1 <= 3'b001;
          col2 <= 3'b010;
          col3 <= 3'b011;
          col4 <= 3'b100;
          col5 <= 3'b101;
          col6 <= 3'b110;
          col7 <= 3'b111;
        end
      end
    end
  end
end

```

```

col_dummy <= 3'b000;
i <= 0;
next_count <= 1;

if (count_sc == 3'b111 )
    begin
        count_sc <= 0;
    end
else count_sc <= count_sc + 1;

end

else
    begin
        if (count[0] && col_dummy[1]) ready <= 1;
        if (ready) OP_ready <= ~OP_ready;
        if (count == 7 && ~next_count) C_ready <= 1;
        col1 <= 3'b001;
        col2 <= 3'b010;
        col3 <= 3'b011;
        col4 <= 3'b100;
        col5 <= 3'b101;
        col6 <= 3'b110;
        col7 <= 3'b111;
        col_dummy <= col_dummy +1;
        if (col_dummy == 3'b110) next_count <= 0;
        else next_count <= 1;
        i <= 1;
    end

end

endmodule

```

---

# Appendix C

## *Publications and Presentations*

---

### C.1 Publications

1. **C. A. Rahman** and W. Badawy, "A VLSI Architecture for Finite Ridgelet Transform", *Accepted for publication in the proceedings of the 46th IEEE Midwest Symposium on Circuits and Systems- MWSCAS 2003*, Dec 27-30, Cairo, Egypt, 2003.
2. **C. A. Rahman** and W. Badawy, "VLSI Architectures for Finite Radon Transform", *Accepted for publication in the proceedings of the Canadian Conference on Electrical and Computer Engineering- CCECE 2004*, May 2-5, Niagara Falls, Ontario, Canada, 2004.
3. M. Alam, **C. A. Rahman**, W. Badawy and G. Jullien, "Efficient Distributed Arithmetic Based DWT Architecture for Multimedia Applications", *Proceedings*

*of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications- IWSOC 2003*, June 30- July 2, Calgary, AB, Canada, 2003, Page(s): 333 -336.

4. **C. A. Rahman**, W. Badawy and Ahmad Radmanesh, "A Real Time Vehicle's License Plate Recognition System", *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance- AVSS 2003*, July 21-22, Miami, FL, USA, 2003, Page(s): 163-166.

## **C.2 Presentations / Workshops / Seminars**

1. **Presenter**, "A VLSI Architecture for Finite Ridgelet Transform (FRIT)", *Graduate Seminar Course (ENEL 605)*, November 5, 2003, Department of Electrical and Computer Engineering, University of Calgary, Canada.
2. **Instructor**, "Image Processing and Digital System Design", *Workshop*, August 10-20, 2003, Organized by Department of Electrical and Electronic Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh.
3. **Presenter**, "The Finite Ridgelet Transform – An Overview", *LIVS Research Group Seminar*, June 3, 2003, Organized by Laboratory for Integrated Video Systems (LIVS), Department of Electrical and Computer Engineering, University of Calgary, Canada.