Wavelet Transform based adaptive image compression on FPGA

by

Sarin George Mathen

B.Tech. (Computer Science & Engineering),
Regional Engineering College, Calicut,
University of Calicut, Calicut, India, 1996

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Professor in Charge

_____

_____

Committee Members

_____

Date Thesis Accepted

To all my good friends

Acknowledgments

At the end of two years at the Adaptive Computing Systems Lab, I believe I have become a lot more adaptable than before. Countless people have helped me in many different ways. Let me try to remember a few. To those I am sure to miss out, my sincerest apologies.

Thanks to Dr. Joe Evans, my advisor, for his guidance, support, and freedom through this project. I will always remember his amazingly fast and short e-mail replies. I awe at his 'never say no' attitude, and hope to carry it along. My special thanks to Dr. Gary Minden for his support and suggestions. My thanks also to Dr. John Gauch for his suggestions and for serving on my committee.

Thanks to Dr. Mike Ashley, my previous advisor, for introducing me to this project and the scheme programming language. I still marvel how quick he used to debug my scheme programs. I will certainly miss his antics while debugging. 'I am baffled,' comes to mind.

Thanks to my friends here, who have made me feel the two year stay less than two years. Special mention goes to Sandeep Mukthavaram, Harish Sitaraman, Hemang Parekh and Pramodh Mallipatna. Thanks also to my old pals Binu Anand, Sunil Chandran and Sajan Skaria who has made life a lot easier when times were difficult.

Special thanks to Ashish Sirasao of Synplicity Inc. for introducing me to Logic Synthesis.

Thanks to Dr. Kissan Joseph and family for making my stay here a lot more tasteful. My thanks also for his help in editing this thesis.

Thanks to my parents. Thanks to my little brother for all his wit and humor. Finally, thanks to God, for helping me out at difficult times.

Abstract

Image processing systems can encode raw images with different degrees of precision, achieving varying levels of compression. Different encoders with different compression ratios can be built and used for different applications. The need to dynamically adjust the compression ratio of the encoder arises in many applications. One example involves the real-time transmission of encoded data over a packet switched network.

To suitably adapt the encoder to varying compression requirements, adaptive adjustments of the compression parameters are required. This involves reconfiguring the encoder in an efficient manner. Our approach exploits the reconfigurable nature of Field Programmable Gate Arrays (FPGA), to adapt the encoder to the varying requirements in real time. A Wavelet transform based image compression scheme is implemented for encoding gray-scale frames of 512 by 512 pixels on FPGAs. By varying the zero thresholds, the encoder can achieve varying compression levels.

The complete design of the encoder on FPGA is presented. Implementation details of the individual blocks are discussed in great detail. Finally, results from testing are reported and discussed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Several aspects of high performance embedded computing and radio processing functions are considered as part of the 'Adaptive Computing Systems' project* at the University of Kansas. The motivation is to use reconfigurable hardware for implementing high performance computation blocks. Essentially, the same piece of silicon is reprogrammed to achieve different functionalities. In this regard, a set of re-programmable hardware resources, namely Field Programmable Gate Arrays (FPGA), with embedded memory [WILDFORCE], is hosted on a computer with real time operating system. This has been successfully used to accelerate computational intensive functions [SHA].

1.1.1 Why adaptive image compression?

Image processing systems can encode raw images with different degrees of precision, achieving varying levels of compression. Encoding can be achieved with different encoders with varying compression ratios. The need to dynam-

*This work is funded by DARPA contract number DABT63-97-C-0032 as part of the Adaptive Computing Systems initiative.

ically adjust the compression ratio of the encoder arises in many situations. One example involves the real-time transmission of encoded data over a packet switched network. On detecting network congestion, the encoder can cut down the precision and gain more compression, rather than waiting for some packets to be dropped.

To suitably adapt the encoder to the varying compression requirements, adaptive adjustments of the compression parameters are required. This involves reconfiguring the encoder in some sense. In this thesis, reconfigurable hardware, namely FPGA, is used in the implementation of an adaptive encoder.

1.1.2 Scope of reconfigurable hardware

System performance is a major concern while designing complex systems. The Von Neumann style of fetch-operate-writeback computation fails to exploit the inherent parallelism in the algorithm. For example, a 30 tap FIR filter implemented on a DSP microprocessor would require 30 MAC (Multiply Accumulate) cycles for advancing one unit of real-time. Further, each MAC operation may consist of more than one cycle as it involves a memory fetch, the multiply accumulate operation, and the memory write back. In contrast, a hardware implementation can store the data in registers and perform the 30 MAC operations in parallel over a single cycle. Thus, high throughput requirements of real-time digital systems often dictate hardware intensive solutions.

FPGAs provide a rapid prototyping platform. They can be reprogrammed to achieve different functionalities without incurring the non-reccuring engineering costs typically associated with custom IC fabrication. One drawback, however, is that due to the rather coarse grained reconfigurable blocks, an implementation on an FPGA is often not as efficient, in terms of space and time, as on a custom IC.

1.2 Thesis Overview

The remainder of this document is organized as follows. Chapter one explains related work in this field. Chapter two describes Wavelet transform based image compression schemes. Next, chapter three explains the design and implementation of the encoder. Then, chapter four summarizes the results obtained. Finally, chapter five concludes.

1.3 Related work

There have been other efforts to implement Wavelet transform based image compression systems on FPGA. In one implementation [BRIAN], the discrete wavelet transform coefficients are computed for 256x256 grayscale frames. This implementation also supports a multiplierless quantizer and a run length encoder. The frame rates quoted are 20 frames / second on Xilinx 4008 FPGAs with on-board embedded memory.

There are many other implementations using ASICs and custom ICs. There have also been many software based image compression kits like [GEOFF], which utilizes wavelet based compression techniques.

Chapter 2

Wavelets

2.1 Wavelets

The following introduction on wavelets is based on the paper by mathematician Gilbert Strang [STRANG]. A wavelet is a localized function in time (or space in the case of images) with mean zero. A wavelet basis is derived from the wavelet (small wave) by its own dilations and translations.

$$w_{j,k}(t) = 2^{\frac{-j}{2}} w(2^{-j}t - k)$$

Let the original wavelet start at $t = 0$ and end at $t = N$. The shifted wavelet $w_{0,k}$, starts at $t = k$ and ends at $t = k + N$. The rescaled wavelet $w_{j,0}$ starts at $t = 0$ and ends at $t = N/2^j$. At a given resolution j, the basis functions are $w_{j,k}(t)$, and the time steps at that level are 2^{-j}. At the next finer resolution, $j+1$, the time steps are $2^{-(j+1)}$. Frequencies shift upward by an octave, when time is rescaled by 2.

Functionally, Discrete Wavelet Transform (DWT) is very much similar to the Discrete Fourier Transform, in that the transformation function is orthogonal. A signal passed twice through the orthogonal function is unchanged. As the input signal is a set of samples, both transforms are convolutions. While the basis function of the Fourier transform is a sinusoid, the wavelet basis is a set

waves obtained by the dilations and translations of the mother wavelet.

2.1.1 Compact support, Vanishing moments, and Smoothness

Wavelets are localized functions and zero outside a bounded interval. This compact support corresponds to an FIR implementation. Another way to characterize wavelets by the number of coefficients and the level of iteration. If the frequency response of the corresponding filter has p zeroes at π, the approximation order is p. In other words, a wavelet basis with p vanishing moments can give a p^{th} order approximation for any signal. The smoothness of the transfer functions is measured by the number of its derivatives.

2.1.2 Orthogonal and Bi-orthogonal Wavelets

The wavelet basis forms an orthogonal basis if the basis vectors are orthogonal to its own dialations and translations. A less stringent condition is that the vectors be bi-orthogonal. The DWT and inverse DWT can be implemented by filter banks. This includes an analysis filter and a synthesis filter. When the analysis and synthesis filters are transposes as well as inverses of each other, the whole filter bank is orthogonal. When they are inverses, but not necessarily transposes, the filter bank is bi-orthogonal.

2.1.3 A simple example - the Haar wavelet

One of the first wavelet was that of Haar. The Haar scaling function is shown below.

$$w(n) = \begin{cases} 1, 0 \leq t \leq 1 \\ 0, otherwise \end{cases}$$

Applying the Haar wavelet on a sequence of values computes its sums and differences. For example, a sequence of values a, b would be replaced by $s = (a + b)/2$ and $d = (b - a)$. The values of a and b can be reconstructed as $a = s - d/2$ and $b = s + d/2$. The input signal with 2^n samples is replaced with 2^{n-1} averages ($s_0(i)$) and 2^{n-1} differences ($d_0(i)$). The averages can be thought of as a coarser representation of the signal and the differences as the information needed to go back to the original resolution. The averages and differences are now computed on the coarser signal ($s_0(i)$) of length 2^{n-1}. This gives ($s_1(i)$) and ($d_1(i)$) of length 2^{n-2} each. This operation can be performed n times, till we run out of samples. The inverse operation starts by computing $s_{n-2}(j)$ from $s_{n-1}(j)$ and $d_{n-1}(j)$.

2.1.4 Lifting scheme

The above computation of the Haar wavelet needs intermediate storage to store the average and difference. The average computed, cannot be written back in place of a, till the difference has been computed. Lifting scheme on the other hand allows for an in place computation. In the first step, we compute only the difference $d = (b - a)$ and store it in place of b. Next, the average value is computed in terms of a and the newly computed difference, b, as $s = a + b/2$. The inverse can be computed by reversing the order and flipping the signs. This is a simple instance of lifting.

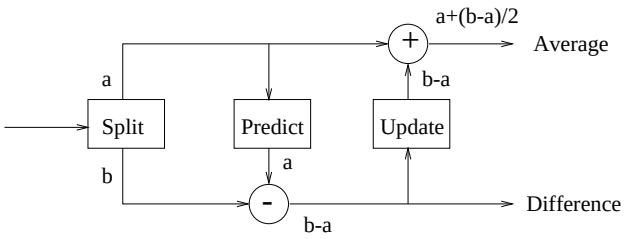

Figure 2.1: Lifting Scheme

A more general lifting scheme consists of three steps - split, predict and up-

date, figure 2.1. The splitting stage splits the signal into two disjoint sets of samples. In the above example, it consists of even numbered samples and odd numbered samples. Each group contains half as many samples as the original signal. If the signal has a local correlation the consecutive samples will be highly correlated. In other words, given one set it should be able to predict the other. In the diagram, the even samples are used to predict the odd samples. Then the detail is the difference between the odd sample and its prediction.

In the Haar case the prediction is simple, every even value is used to predict the next odd value. The order of the predictor in the Haar case is 1 and it eliminates zeroth order correlation. The reverse operation is done as undo-update, undo-predict and merge.

2.1.5 Wavelets that map Integer to Integer

We return to the Haar transform. Because of the division by 2 in the average computation, it is not an integer transform. A simple alternative is to calculate the sum instead of the average. Another solution known as the S (sequential) transform is to round off the average value to an integer value. The sum and difference of two integers are both even or both odd. So, the last bits of the difference and average should be identical. Hence the last bit from average can be omitted, with out loosing information. In the general case, though rounding may add a non-linearity to the transform, it has been shown to be invertible, [CALDERBANK].

2.1.6 (2,2) Bi-orthogonal Cohen Daubechies Feauveau Wavelet

The main intent of wavelet transform is to decompose a signal f, in terms of its basis vectors.

$$f = \sum a_i W_i$$

To have an efficient representation of signal f using only a few coefficients a_i, the basis functions should match the features of the signal we want to represent. The (2,2) Cohen Daubechies Feauveau Wavelet [COHEN] is widely used for image compression because of its good compression characteristics. The original filters have $5+3=8$ filter coefficients, whereas an implementation with the lifting scheme has only $2+2=4$ filter coefficients. The forward and reverse filters are shown in table 2.1. Fractional numbers are converted to integers at each stage. Though such an operation adds non-linearity to the transform, the transform is fully invertible as long as the rounding is deterministic.

Forward transform	
s_i	$\leftarrow x_{2i}$
d_i	$\leftarrow x_{2i+1}$
d_i	$\leftarrow d_i - (s_i + s_{i+1})/2$
s_i	$\leftarrow s_i + (d_{i-i} + d_i)/4$
Inverse transform	
s_i	$\leftarrow s_i - d_i/2$
d_i	$\leftarrow d_i + s_i$
x_{2i}	$\leftarrow s_i$
x_{2i+1}	$\leftarrow d_i$

Table 2.1: (2,2) CDF wavelet with lifting scheme

2.1.7 Boundary treatment

Real world signals are limited to a finite interval. However filter bank algorithms assume infinite lengths. The computation of s and d coefficients refer to k signal samples before and after the current sample, depending on the filter length k. Different methods of extending the signal at the boundaries has been suggested. One scheme that is widely used is the symmetric extension. It extends the finite signal by mirroring it around its boundaries.

2.1.8 Advantages of Wavelets

Real time signals are both time-limited (or space limited in the case of images) and band-limited. Time-limited signals can be efficiently represented by a basis of block functions (Dirac delta functions for infinitesimal small blocks). But block functions are not band-limited. Band limited signals on the other hand can be efficiently represented by a Fourier basis. But sines and cosines are not time-limited. Wavelets are localized in both time (space) and frequency (scale) domains. Hence it is easy to capture local features in a signal.

Another advantage of a wavelet basis is that it supports multi resolution. Consider the windowed Fourier transform. The effect of the window is to localize the signal being analyzed. Because a single window is used for all frequencies, the resolution of the analysis is same at all frequencies. To capture signal discontinuities (and spikes), one needs shorter windows, or shorter basis functions. At the same time, to analyze low frequency signal components, one needs longer basis functions. With a wavelet based decomposition, the window sizes vary. Thus it allows to analyze the signal at different resolution levels.

Chapter 3

Design and Implementation

This chapter explains aspects of design and implementation of the encoder.

3.1 Hardware platform

Xilinx 4000 series FPGAs[XC4000] were available and used for the implementation. These are look-up table based FPGAs. Each basic block called a CLB (Configurable Logic Block) consists of two 4 input look-up tables and one 3 input look-up table (figure A.1). Each CLB also has 2 flip flops. There are multiplexers within a CLB to achieve internal connectivity among the flip flops and look-up tables. The CLBs are arranged as a matrix. In addition to CLBs, these FPGAs have horizontal and vertical interconnects and switches (routing resources) to achieve connectivity between different ports of different CLBs. The look-up tables can be programmed with truth tables of 4 input or 3 input logic functions. The routing resources can be programmed to achieve the required connectivity between the CLBs.

The hardware platform used[WILDFORCE] is a PCI plug-in board with five Xilinx 4085 FPGAs, also referred to as PEs (Processing Elements). The board is stacked with five 1MB SRAM chips. Each of the five SRAM chips are directly connected to one of the five PEs. The embedded memory is accessible for

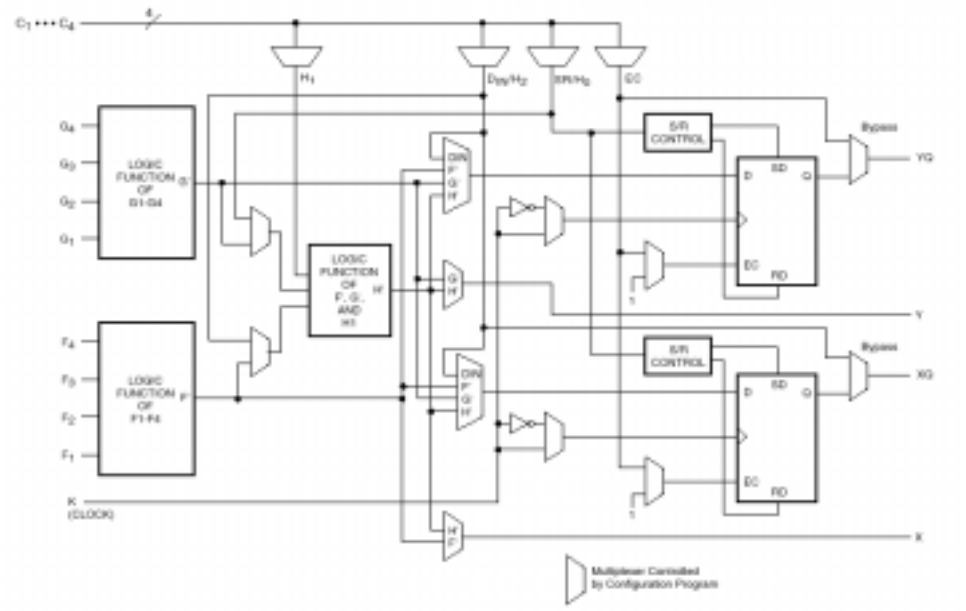

Figure 3.1: Configurable Logic Block (CLB) in XC4000 series FPGA

read / write from both the host computer as well as from the corresponding PE. Each of the 1MB memory chip is organized as 262144 words of 32 bits each.

3.2 Design parameters and constraints

3.2.1 Memory read/write

The input image to the encoder is raw gray scale frames of 512 by 512 pixels. Each pixel is represented by 256 gray scale levels (8 bits). Input frames are loaded to the embedded memory by the host computer and results are read back, once the PE has processed it. The PE also uses the embedded memory as intermediate storage to hold results between different stages of processing.

The memory has a read latency of 2 cycles while memory writes are completed in the same cycle. Memory reads can be pipelined so that the effects of this latency is minimized. However, a clock cycle is wasted when there is a read to write turn around. The design concerns are to minimize memory

read / write turn arounds and to allow longer spells of read or write cycles instead. Attempts have also been made to minimize memory operations.

3.2.2 Real time performance

While the conventional television standards require 30 frames / second, many multimedia applications like video conferencing run at much lower frame rates.

In general, a good system clock ensures a good throughput. Other contributing factors to throughput include the time taken by the operating system driver routines to read / write from the embedded memory.

3.2.3 Design partitioning

The whole computation is partitioned into two stages. The first stage computes discrete wavelet transform coefficients of the input image frame and writes it back to the embedded memory. The second stage, operates on this result to complete the rest of the processing. The second stage does dynamic quantization, zero thresholding, run length encoding for zeroes, and entropy encoding on the coefficients. The two stages are implemented on two separate FPGAs.

3.3 Stage 1: Discrete Wavelet Transform

Discrete Wavelet transform is implemented by filter banks. The filter used is the (2,2) Cohen-Debuchies-Feaveu wavelet filter. Though much longer filters are common for audio data, relatively short filters are used for video.

3.3.0.1 (2, 2) wavelet

A modified form of the Bi-orthogonal (2,2) Cohen-Debuchies-Feaveu wavelet filter is used. The analysis filter equations are shown below.

$$\begin{array}{ll} \text{High pass coefficients:} & g(k) = 2x(2k+1) - x(2k) - x(2k+2) \\ \text{Low pass coefficients:} & f(k) = x(2k) + (g(k-1) + g(k)/8 \end{array}$$

The boundary conditions are handled by symmetric extension of the coefficients as shown below:

$$x[2],\ x[1],\ \ [\ x[0], x[1], ..., x[n-1], x[n]\ ],\ \ x[n-1],\ x[n-2]$$

The synthesis filter equations are shown below.

$$\begin{array}{ll} \text{Even samples:} & x(2k) = f(k) - (g(k-1) + g(k+1)/8 \\ \text{Odd samples:} & x(2k+1) = (g(k) + f(k) + f(k+1))/2 \end{array}$$

3.3.0.2 DWT in X and Y directions

Each pixel in the input frame is represented by 16 bits, accounting for 2 pixels per memory word. Thus, each memory read brings in two consecutive pixels of a row. Each clock cycle generates one value each of f and g coefficients. These have to be written back in place. The f coefficients are used again in the next stage of wave-letting. Two consecutive values of f are written back in one memory location (figure 3.2). This saves on memory reads of the f coefficients in the next stage. In the next stage, where only the fs are processed, only alternate memory words are read from. Thus, the f and g coefficients are written back in an interleaved fashion. Another way to write back the coefficients is to put all the low frequency coefficients (f) ahead of the high frequency coefficients (g). This scheme of ordering the coefficients is called Mallot ordering. It allows progressive image transmission / reconstruction. The bulk of the 'average' information is ahead, followed by the minor 'difference' information. However, this ordering scheme requires temporary storage to hold the computed coefficients until the they can be written back. In our design, we use

the in-place ordering scheme described above which is optimized for memory read / write operation. Once the three stages of wave-letting is done, we resort back to Mallot ordering.

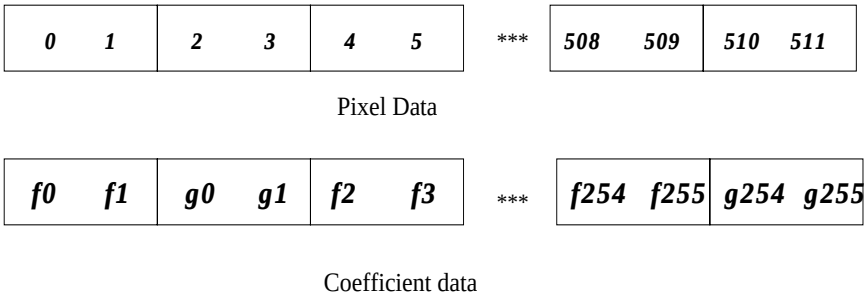

Figure 3.2: Coefficient ordering along X direction

Once the filter has been applied along all rows in a stage, the same filter is applied along the columns. With the afore mentioned interleaved ordering scheme, alternate columns are all fs or all gs. Unlike the row traversal, the two values obtained in a memory read on a column traversal, are not consecutive values of the same column. Rather, they are corresponding values from two different vertically parallel streams (figure 3.3).

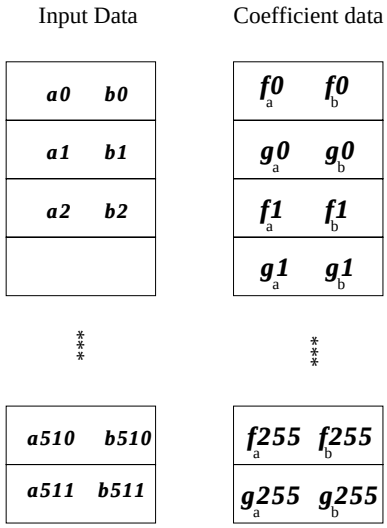

Figure 3.3: Coefficient ordering along Y direction

These differences along the row and column computations are accounted

by having two separate data flow blocks along the two directions. The data flow block in X direction (ForwardWaveletX) accepts two successive values of the same row and outputs either two consecutive fs or two consecutive gs, in alternate fashion. The data flow block in Y direction (ForwardWaveletY) accepts one value each from two parallel streams and outputs either the fs for the two streams or the gs in an alternate manner, (figure 3.4). These blocks also need information on when a row / column starts / ends to handle the boundary conditions. They also have a pipeline latency of 3 cycles.

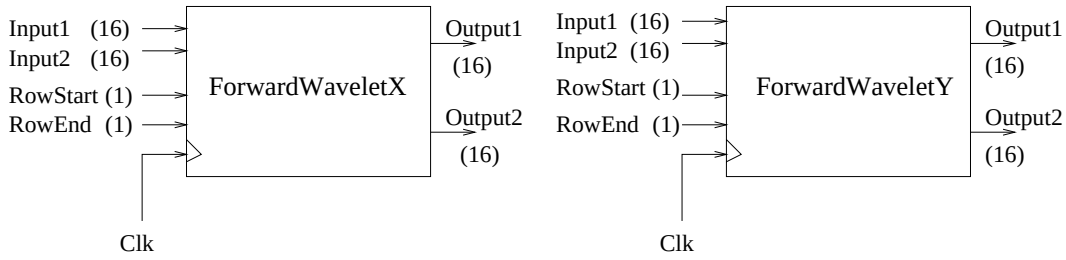

Figure 3.4: Fast Wavelet transform data flow blocks

3.3.0.3 3 stages of wave-letting

The 512 by 512 pixel input image frame is processed with three stages of wave-letting. In the first stage, 512 pixels of each row are used to compute 256 high pass coefficients (g) and 256 low pass coefficients (f), figure 3.5. The coefficients are written back in place of the original row.

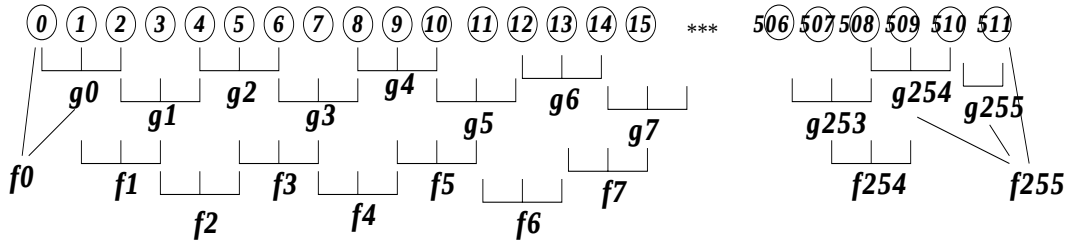

Figure 3.5: High pass and Low pass coefficients at stage 1, X direction

Once all the 512 rows are processed, the filters are applied in the Y direction.

This completes the first stage of wave-letting. While conventional Mallot ordering scheme aggregates coefficients into the 4 quadrants, our ordering scheme interleaves the coefficients in the memory. The second stage of wave-letting only processes the low frequency coefficients from the first stage. This corresponds to the upper left hand quadrant in the Mallot scheme. Thus, second stage operates on row and columns of length 256, while the third stage operates on rows and columns of length 128. The aggregation of coefficients along the 3 stages under Mallot ordering is shown in figure 3.6. The memory map with the interleaved ordering is shown in figure 3.7.

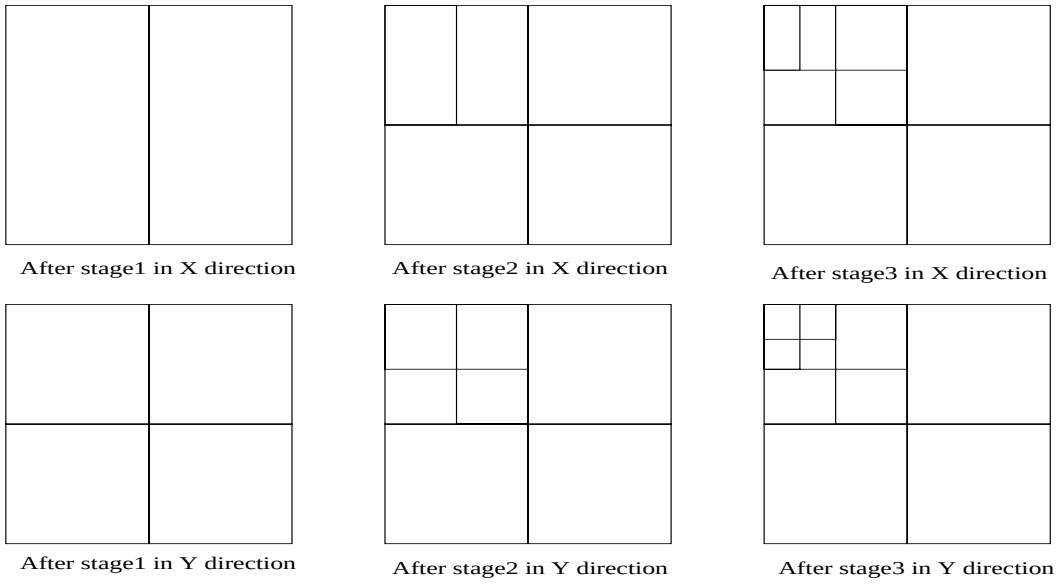

Figure 3.6: Mallot ordering along the 3 stages of wave-letting

3.3.0.4 Over all architecture of Stage 1

Stage one starts with a raw frame and does three stages of wave-letting. The over all architecture is shown in figure 3.8. Memory addressing is done with a pair of address registers - read and write address registers. The difference between write and read registers is the latency of the pipelined data-flow blocks. The maximum and minimum coefficient values for each block (each quadrant

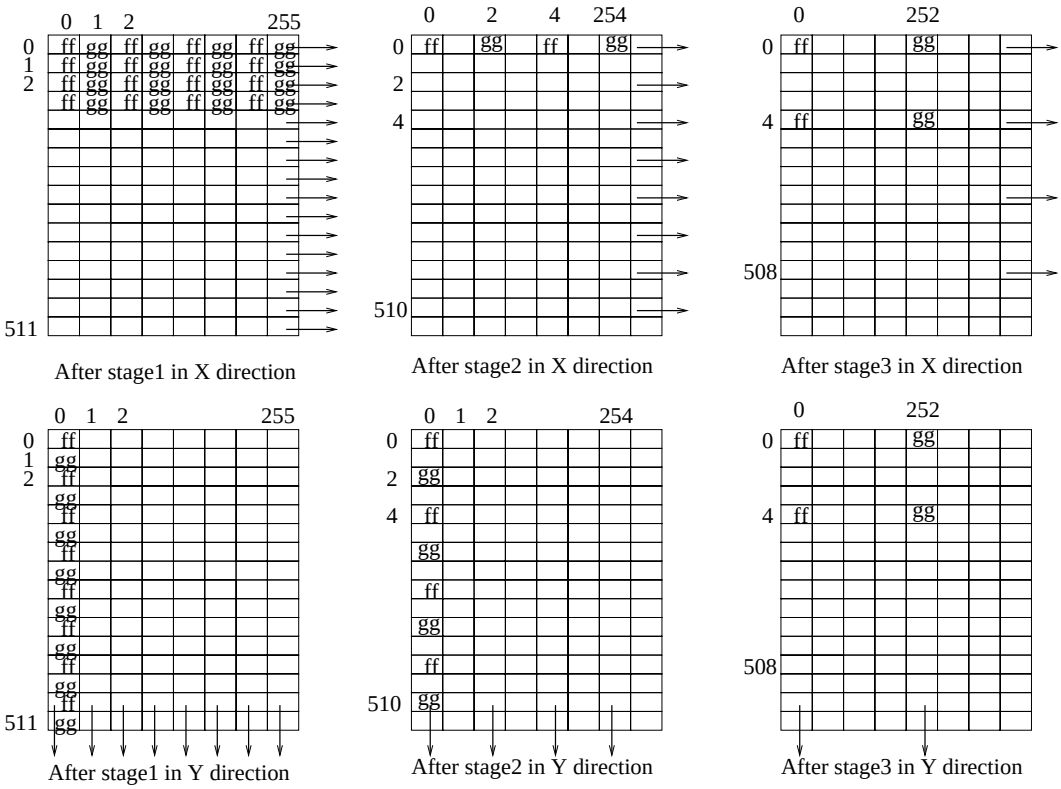

Figure 3.7: Interleaved ordering along the 3 stages of wave-letting

in the multi stage wave-letting) are maintained on the FPGA. These values are written back to a known location in the lower half (lower 0.5MB) of the embedded memory. The second stage, uses these values for the dynamic quantization of the coefficients.

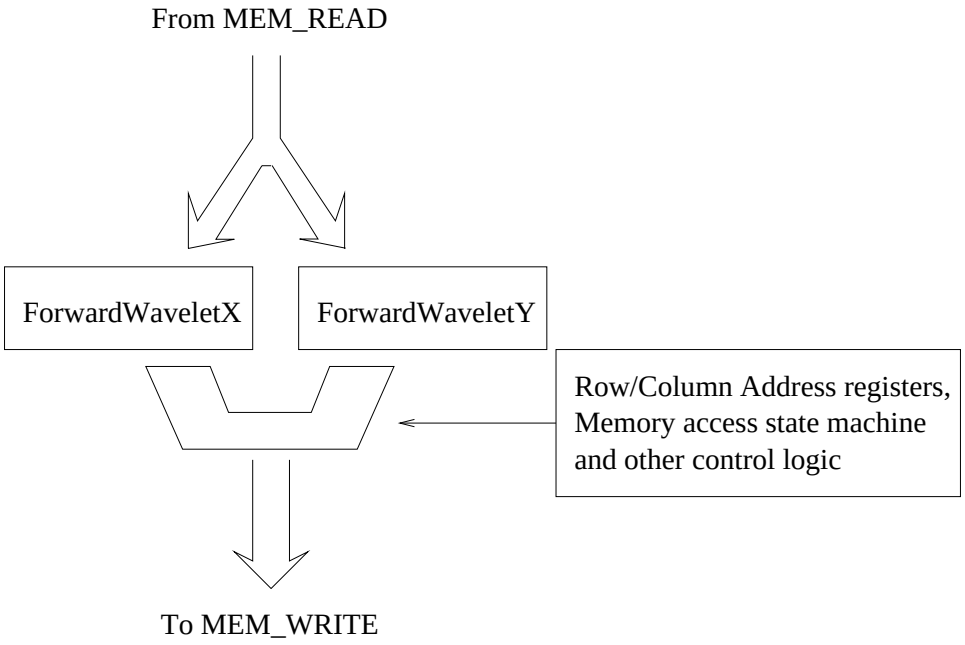

Figure 3.8: Stage 1 architecture

3.4 Stage 2

Stage 2 does the rest of the processing on the wavelet coefficients computed in the first stage. The coefficients, are quantized, zero-thresholded, zeroes run length encoded, and entropy encoded to get the final compressed image.

3.4.1 Dynamic quantization

The coefficients from different sub-bands (different quadrants with the Mallot ordering scheme) are quantized separately. The dynamic range of the coefficients for each sub-band (computed in first stage) is divided into 16 quantiza-

tion levels. The coefficients are quantized into one of the 16 possible levels. The maximum and minimum value of the coefficients for each sub-band is also needed while decoding the image.

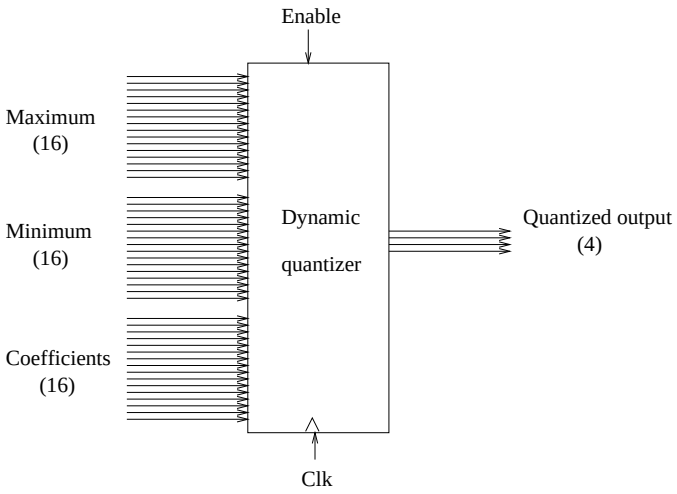

Figure 3.9: Dynamic Quantizer

The dynamic quantizer is implemented as a binary search tree look up in hardware (figure 3.9). A table look up based quantization scheme is not feasible since the range is dynamic - different for each sub-band, and different for each frame. The incoming stream of coefficients in the range [min:max] is translated to [0,max-min] by adding (or subtracting) the minimum. The shifted incoming value is then compared with half the dynamic range ($\frac{r}{2}$) to determine whether it lies in the lower eight or upper eight quantization levels. The result forms the first bit (most significant bit) of the quantizer output. Depending on the outcome, the value is then compared with $\frac{r}{2}+\frac{r}{4}$ or $\frac{r}{2}-\frac{r}{4}$. This forms the second bit of the quantized output. The next two comparisons provide the remaining bits. The quantizer is a pipelined design, with 4 stages.

3.4.2 Zero thresholding and RLE on zeroes

Regions with abrupt changes will have larger wavelet coefficients while regions of little or no change would have smaller coefficients. Coefficients of small magnitude can be neglected without considerable distortion to the image. The error introduced is proportional to the magnitude of the coefficient being neglected. Coefficients are truncated to zero, based on a threshold. Different thresholds are used for different sub-bands, resulting in different resolution in different sub-bands. Further, different sets of thresholds are used to achieve different levels of compression. Three different set of thresholds are used for each sub-band to get three different variants of the encoder with different compression levels. The corresponding levels for the three configurations of the encoder are shown in the appendix.

After the zero thresholding a large number of coefficients are truncated to zero. Long sequences of zeroes can be effectively compressed by run length encoding, which replaces each individual occurrence of a zero in a continuous spell with a count indicating the length of the spell. To decode a run length encoded stream, this count has to be distinguishable from other characters of the input data set. The other valid character are the 4 bit output from the quantizer. Sixteen numbers 0 to 15 are reserved for the quantizer output values, while numbers 16 to 255 (240 numbers) are free. Thus, any continuous spell of zeroes ranging from 1 (represented by the number 16) to 240 (represented by the number 255) can be replaced by the corresponding count. Longer spells have to be broken down to fall within this range. Table 3.1 shows the bit range allocation.

The run length encoder, might not have an output on every cycle. The succeeding block has to be signalled as to when to read the RLE count, and when to wait for a spell to finish. Whenever RLE detects a zero, it asserts 'RLErunning,' and starts counting the sequence of continuous zeroes. The current sum of zeroes is always available on 'RLEout.' When the continuous spell of zeroes

00000000 ... 00001111	16 numbers allocated for the output of quantizer, 16 quantization levels.
00010000 ... 11111111	256-16=240 numbers available for RLE. RLE can count upto 240 continuous zeroes.

Table 3.1: Bit range allocation for RLE

end, 'RLErunning' is deasserted, and 'RLEspellEnd' is asserted for one cycle to allow the next block to read off the RLE count. The RLE counter is also reset to 15.

In this set-up, there is look ahead problem. Before RLE can signal the end of a spell, it needs to see the next value is the stream. But, RLE is used in conjunction with the dynamic quantizer, (RLE and quantizer are connected in parallel) which is a 4 staged pipeline.

RLE might face an arbitrarily long sequence of zeroes. RLE can count only upto a maximum of 240 zeroes. Thus, when RLE has seen 240 continuous zeroes and still more zeroes are arriving, 'RLEspellEnd' would be asserted for one

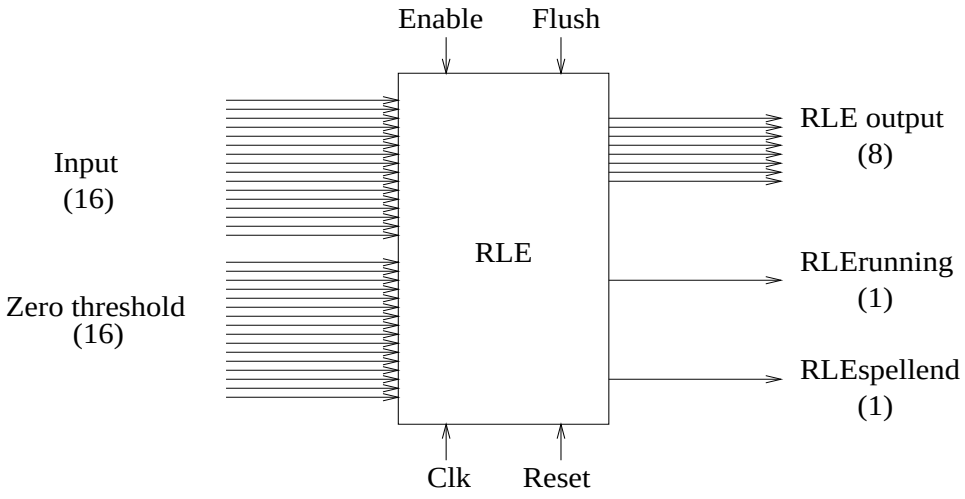

Figure 3.10: Run Length Encoder for continuous zeroes

clock cycle, and the internal counter is reset to 15. Here, 'RLErunning' would be high through out the spell.

The logic followed by the succeeding block is as follows. If 'RLErunning' is asserted then wait till 'RLEspellEnd' is asserted and read the 'RLEout'. Else, read the output of the dynamic quantizer.

3.4.3 Entropy encoding

Entropy encoding involves assigning a smaller length encoding for more frequently used characters in the data set and a larger length encoding for infrequently used characters in the data set. This involves variable length encoding of the input data. To efficiently retrieve the original data, an encoded word should not be a proper prefix of any other encoded word. Huffman trees are an efficient way of coming up with a variable length encoding for a set of characters, given the relative frequencies. Further, for a Huffman tree based encoding, decoding can be done in linear time (linear in the length of the encoded word). Various other schemes of encoding using different levels of context sensitive information exits. This might incur a costlier decoding function.

3.4.3.1 Encoding scheme

In our implementation, we use an encoding scheme which is not a Huffman tree based code. The bit allocation is shown in figure 3.11. Eight bit inputs are variable length encoded between 3 to 18 bits. The complete encoding table is shown in the appendix. The encoding is implemented by two look-up tables on the FPGA. Given an eight bit input, the first look-up table (LUT), provides information about the size of encoding. The second LUT gives the actual encoding. Only the relevant bits from the second LUT should be used. The rest of the bits in the output are don't care and are either chosen as logic 0 or 1 during logic optimization. The VHDL description of the encoder can be found in the

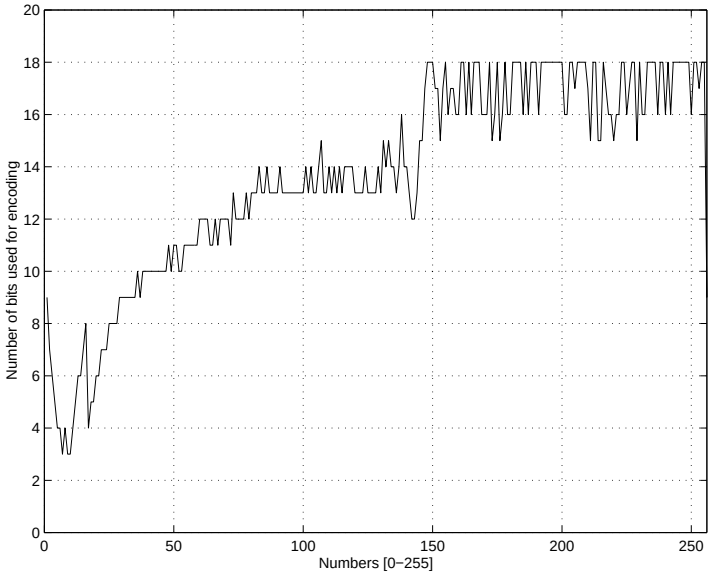

Figure 3.11: Entropy encoding, bit allocation

appendix, huffman.vhd.

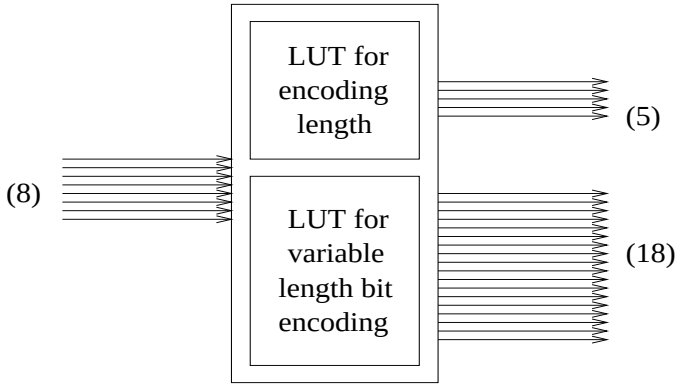

Figure 3.12: Entropy encoder

3.4.3.2 Bit packing

The output of the entropy encoder varies from 3 to 18 bits. The bits need to be packed into 32 bit words before being written back to the embedded memory. This is achieved by the shifter discussed below. This shifter is inspired from the Xtetris computer game and the binary search algorithm.

3.4.3.3 Shifter

The shifter consists of 5 register stages, each 32 bits wide. The input data can be shifted (rotated) by 16 or latched without shifting, to stage 1. The data can be shifted by 8 or passed on straight from stage 1 to stage 2. Similarly data can be shifted by 4, 2, and 1 when moving between the remaining stages. Data is shifted from stage to stage, and is accumulated at the last stage. When the last stage has 32 bits of data, a memory write is initiated and the last stage is flushed.

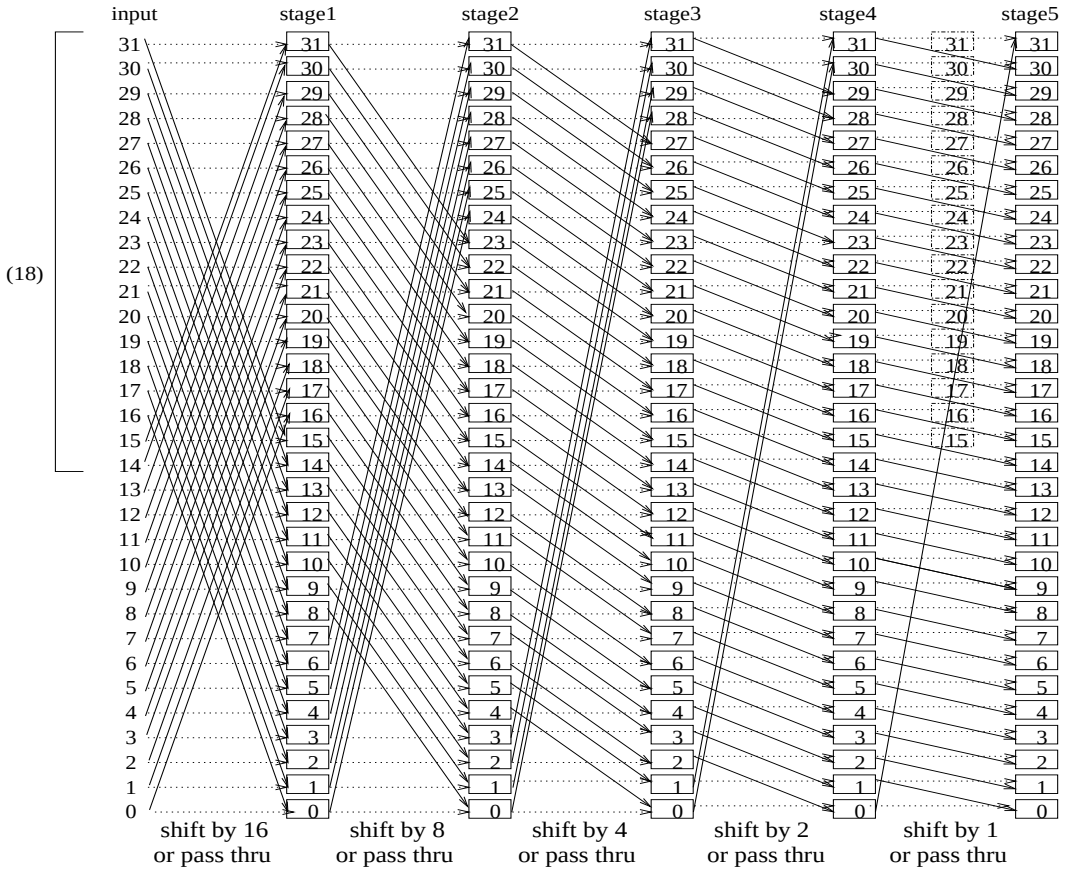

Figure 3.13: Binary Shifter for bit packing

The data is shifted to the right place over the 5 stages in order to complete a word at the last stage. The key decision is whether to shift or not at each stage. A 5 bit counter is maintained to store the length of the data currently held. For

example, let the lengths of the words arriving at stage 1 be a_1, a_2, a_3, etc. The counter will have values 0, a1, a1 + a2, etc. in the corresponding clock cycles. The counter is allowed to overflow once it reaches 31. Thus, the counter value indicates where the next word should start by the time it reaches the last stage. Different bits of the counter (delayed appropriately) are used to decide whether to shift or not at each stage.

Part of the last stage needs double buffering. To determine the size of the double buffer needed, consider the worst case. The last stage already has 31 bits and the next data coming from stage 4 is of maximum size (18 bits). Only 1 out of the 18 bits can be added to the last stage and a memory write initiated. The rest of the 17 bits need to be buffered for this cycle, and brought out in the next cycle. Thus, 17 out of the 32 bits in the last stage are double buffered. Thus, whenever an overflow is detected, the double buffer is loaded with the excess bits and taken out during the next cycle. The detailed hardware implementation may be found in the appendix in the file shifter.vhd.

3.4.4 Output file format

At the end of the second stage, the upper memory (upper 0.5MB) contains the packed bit stream. The total count of the bit stream approximated to the nearest WORD is written to memory location 0. To reconstruct the data from the bit stream, the following information is needed.

- The actual bit stream. On Huffman decoding, the actual 8 bit codes are retrieved. These codes are either the quantizer output, or the RLE count. On expanding the RLE count to the corresponding number of zeroes, we get the actual quantized stream.
- The four quadrants of the final stage of wave-letting can be located at the first four 128*128 byte blocks. The three quadrants of the next stage

can be located at at next three blocks sized at 256*256 bytes each. Each quadrant (sub-band) is quantized separately. The dynamic range of each of the quadrant should be known to reconstruct the original stream.

The output file written has all the information needed to reconstruct the image. The format of the output file generated is shown in figure 3.14.

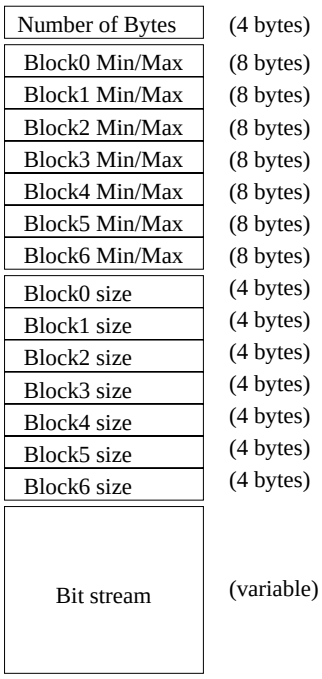

Figure 3.14: Outfile format

3.4.5 Stage 2, Overall architecture

The top level data flow diagram of the second stage is shown in figure 3.15. Wavelet coefficients from memory are read from the lower half of the embedded memory. The block (sub-band) minimum and maximum is also read from the memory. The packed bit stream output is written to the upper memory, and the bit stream length is written to memory location 0. The control software, reads the embedded memory and generates the compressed image file.

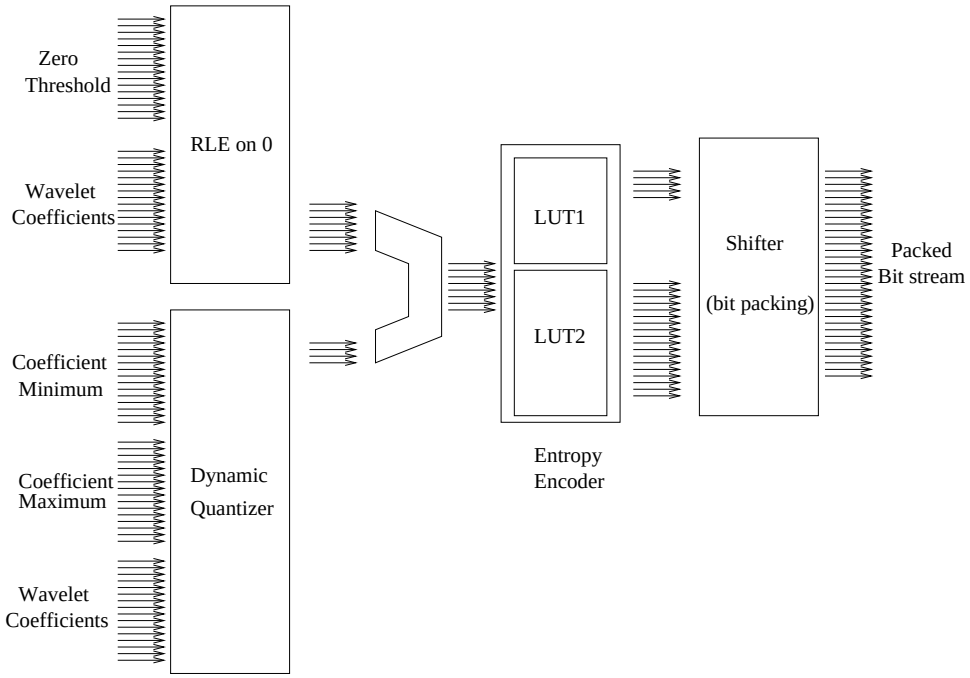

Figure 3.15: Stage 2, data flow diagram

The control flow is show in figure 3.16. Before reading the wavelet coefficients, the maximum and minimum of coefficients in each sub-band are read from the lower memory. The coefficients are then read and processed for each sub-band, starting with the lowest frequency band. As shown in the state diagram, a memory read is fired in stage *Read_001*. Memory read has a latency of 2 clock cycles. The results of the read is finally available in state *Read_100*. Memory writes are completed in the same cycle. The two intermediate states, *Read_010* and *Write* can be used to write back the output, if output is available. Each memory read brings in two wavelet coefficients. Consider the worst case, where the two coefficients gets expanded to 18 bits each. There are two memory write cycles before the next read. When ever a memory write is performed, the memory address register is incremented.

The read address generators, read each sub-band from the interleaved memory pattern. The address ranges for each sub-band with the interleaved ordering scheme is shown in appendix. The output is written as a continuous stream,

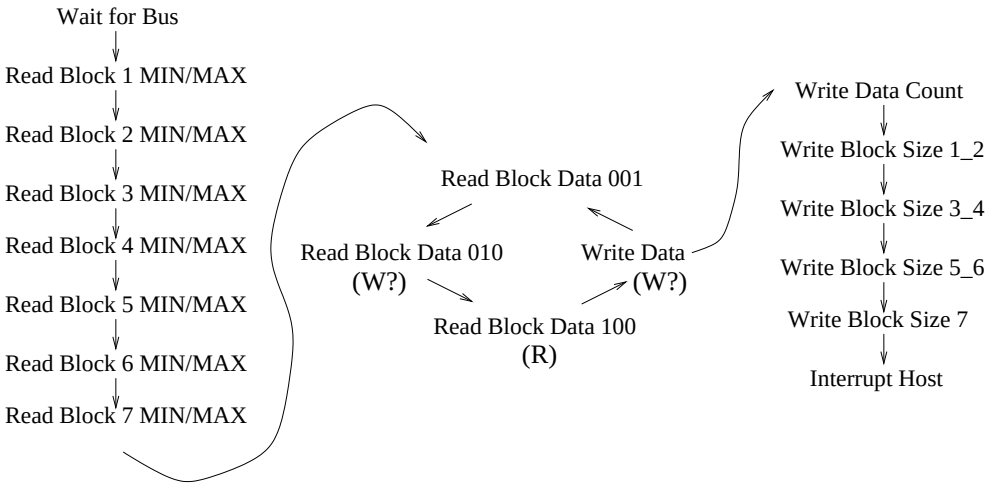

Figure 3.16: Stage 2, control flow diagram

starting with the lowest sub-band. Thus the output is effectively in Mallot ordering and can be progressively transmitted / decoded.

Chapter 4

Results

4.1 Metrics for testing

The metrics on which encoder is graded include the compression ratio, throughput, processing noise, and implementation costs. Further, the adaptivity of the encoder to support different compression levels at different noise levels is also measured.

4.1.1 Throughput

The encoder runs in two stages. A raw frame of 512 by 512 pixels is loaded to the embedded memory. After stage 1 finishes its processing on this memory, the memory image is used as input for the second stage. The two hardware configurations, corresponding to the two stages, can be run at a system clock of 25MHz. The two hardware configurations are loaded onto two different FPGAs on the same board.

4.1.1.1 Embedded memory performance

The embedded memory is loaded and unloaded by the host computer using the operating system driver routines. The measured memory access times are

listed below and it quantifies the time taken by the DMA based read/write APIs provided by the board vendor. The operating system running on the host computer is Linux, kernel version 2.2.5.

Read from host 0.5 MB	4.244 ms
Write from host 0.5 MB	4.017 ms
Read from host 1.0 MB	8.398 ms
Write from host 1.0 MB	7.981 ms

Table 4.1: Embedded memory access times from host computer

The host computer loads a raw frame in the embedded memory corresponding to PE1 and signals it to start. On finishing the processing of stage1, PE1 interrupts the host. The host then reads back the contents of the embedded memory and loads it into the embedded memory of PE2. PE2 then finishes the rest of the processing and interrupts the host. The host reads back the contents of the PE2's memory.

4.1.1.2 Effective throughput

The control software utilized, for loading the memory and servicing the PE interrupts is written in C. The device driver routines provided by the board vendor are employed for this task. A threaded implementation was used in which both the PEs we run simultaneously in a pipelined fashion. Assuming a realtime image source, the file system overheads are neglected. Consequently, 3 input frames were pre-loaded in the host computer's main memory and used as the frame source. For measuring the throughput, the processed image is read back to host computer's main memory and discarded. A write to secondary storage media is avoided to exclude host computer performance and secondary

storage media charecteristics. With the above set up, the sustained throughout observed varied from 10 to 12 frames per second. The detailed break of the times along a single thread are displayed in table 4.2. With multi threaded control software, few of the opreations are run concurrently. This achieves speed up.

Memory write from host 0.5 MB	4.017 ms
PE 1 running time @24MHz (stage 1)	57.402 ms
Memory read from host 1.0 MB	8.401 ms
Memory write from host 1.0 MB	7.948 ms
PE 2 running time @24MHz (stage 2)	5.51 ms
Memory read from host 1.0 MB	8.394 ms

Table 4.2: Delay along a single thread

4.1.2 Compression level Vs noise

Three different hardware configurations with different compression levels were built and tested. The characteristics of the three configurations over three different frames are displayed in tables (4.4, 4.5, 4.6). A software decoder available was used to recontruct the encoded image in order to compare with the original. Noise figures from a software encoder (using 32 bit integer arithmetic) are also quoted (in braces). The PSNR and RMSE metrics are computed as per the equation given below. Percentage compression is the ratio of compressed image size to the original image size (512x512 bytes). Bits per pixel (bpp) is the ratio of image size in bits to number of pixels. The corresponding reconstructed images are also shown in figures 4.3, 4.4.

(a) lena.pgm

(b) barbara.pgm

(c) goldhill.pgm

Figure 4.1: Original Images

(a) lena.pgm

(b) barbara.pgm

(c) goldhill.pgm

Figure 4.2: Configuration 1, Minimum compression

(a) lena.pgm (b) barbara.pgm (c) goldhill.pgm

Figure 4.3: Configuration 2, Medium compression

(a) lena.pgm (b) barbara.pgm (c) goldhill.pgm

Figure 4.4: Configuration 3, Maximum compression

$$\text{MSE} = \frac{1}{512x512} \sum_{x=1}^{x=512} \sum_{y=1}^{y=512} [p(x,y) - p'(x,y)]^2$$

$$\text{RMSE} = \sqrt{MSE}$$

$$\text{PSNR} = 20log10(255/RMSE)$$

Table 4.3: PSNR and RMSE equations

4.1.3 Implementation costs on the hardware

These designs were implemented on Xilinx 4085XLA FPGAs. The device usage is normally reported in terms of number of look-up tables, I/O buffers, and flip flops. The routing resource usage is not given by the place and route tools - a higher device utilization implies a greater routing resource utilization. The total number of CLBs used are a function of the the device resources used and how densely they are packed. For example, on a CLB only a flip flop could be used, but still it adds to the CLB count. The usages for both the stages for displayed in table 4.7.

Configuration	Compressed size (bytes)	Compression ratio	bpp	PSNR (dB)	RMS
Configuration 1, Least compression	28775	9.11	0.878	30.894 (31.102)	7.274 (7.102)
Configuration 2, Medium compression	5556	47.18	0.169	29.530 (30.015)	8.511 (8.049)
Configuration 3, Most compression	3767	69.58	0.114	28.059 (28.120)	10.082 (10.012)

Table 4.4: Compression levels and noise neasurements for 'lena'

Configuration	Compressed size (bytes)	Compression ratio	bpp	PSNR (dB)	RMS
Configuration 1, Least compression	29708	8.82	0.906	25.017 (25.024)	14.310 (14.299)
Configuration 2, Medium compression	8187	32.01	0.249	24.472 (24.589)	15.237 (15.033)
Configuration 3, Most compression	4915	53.33	0.149	23.427 (23.556)	17.185 (16.932)

Table 4.5: Compression levels and noise neasurements for 'barbara'

Configuration	Compressed size (bytes)	Compression ratio	bpp	PSNR (dB)	RMS
Configuration 1, Least compression	30024	8.7	0.916	30.038 (30.045)	8.027 (8.021)
Configuration 2, Medium compression	6070	43.18	0.185	28.119 (28.014)	10.013 (10.134)
Configuration 3, Most compression	3636	72.09	0.110	26.417 (26.446)	12.181 (12.140)

Table 4.6: Compression levels and noise neasurements for 'goldhill'

Block	LUTs (4)	LUTs (3)	CLB Flops	Total CLBs	I/O Buf	I/O Flops	Gate count	Timing (MHz)
Stage 1,	547	109	406	399 (12%)	75	88	8244	26.553
Stage 2, config. 1	1248	356	924	890 (28%)	77	88	17058	36.381
Stage 2, config. 2	1297	367	975	948 (30%)	77	88	17937	31.254
Stage 2, config. 3	1297	373	965	925 (29%)	77	88	17830	34.632

Table 4.7: Device usage and Timing statistics

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We have designed and implemented a Wavelet transform based image encoder on re-programmable hardware - FPGA. The encoder has multiple configurations which support different compression levels. The effective frame rate achieved ranges between 10 and 12. The major conclusions are as follows:

- Wavelet based image compression is ideal for adaptive compression since it is inherently a multi-resolution scheme. Variable levels of compression can be easily achieved. The number of wave-letting stages can be varied, resulting in different number of sub bands. The zero thresholds for truncating coefficients of small magnitude can be varied. Different filter banks with different characteristics can be used. For example, audio data has much longer correlation and hence longer filter are used for audio, compared to video. Filters tuned to the nature of the data achieve much higher compression.

- Efficient fast algorithm (pyramidal computing scheme) for the computation of discrete wavelet coefficients, makes a wavelet transform based encoder computationally efficient.

- Computationally intensive problems often require a hardware intensive solution. Unlike a microprocessor with a single MAC unit, a hardware implementation achieves greater parallelism, and hence higher throughput.

- Reconfigurable hardware is best suited for rapid prototyping applications where the lead time for implementation can be critical. It is an ideal development environment, since bugs can be fixed and multiple design iterations can be done, with out incurring any non recurring engineering costs.

- Reconfigurable hardware is also suited for applications with rapidly changing requirements. In effect, the same piece of silicon can be reused.

- With respect to limitations, achieving good timing/area performance on these FPGAs is much harder, when compared to an ASIC or a custom IC implementation. There are two reasons for this. The first pertains to the fixed size look-up tables. This leads to under utilization of the device. The second reason is that the pre-fabricated routing resources, run out fast with higher device utilization.

5.2 Future work

The lessons learned from this experience will help us enhance similar implementations in the future. Few of the improvements that we now foresee are listed below:

- Build a corresponding decoder on the FPGA and demonstrate the adaptability of the encoder-decoder pair. The encoder would need to signal the decoder on which codec is being used.

- Data movement from host to embedded memory and back to host takes a significant amount of the processing time. Data movement could have been minimized. By implementing both the stages of the encoder on a single FPGA, one read / write memory cycle could have been avoided. On the other side, when these FPGAs are utilized more than about 40%, the timing performance drops sharply. This is because it runs out of routing resources; consequently many long and circuitous routes result. Hence the over all system clock drops. This tradeoff can be better optimized.

- An alternate architecture would be to use the two PEs for the two stages (to get good timing), but use the local bus on the board to transfer data from PE1 to PE2.

- A suggestion with respect to embedded memory architecture is to have two embedded memory chips attached to each PE, so that is can work as a double buffer. Here, the host can refill the next frame on one of the memory chips, while the PE is still working with the other chip.

Appendix A

Design parameters

A.1 Zero threshold levels for different codecs

subband	config. 1	config. 2	config. 3
0	0	0	0
1	39	78	156
2	27	54	108
3	104	208	416
4	79	158	316
5	50	100	200
6	191	382	764

Table A.1: Zero threshold levels for different configurations

A.2 Throughput comparison with a software encoder

The software encoder distributed as part of the ACS bench mark suite was used to obtain time stamps. The encoder was run on a Linux based computer with Pentium 2 processor, running at 333MHz, and having a main memory of 256MB. Time stamps were inserted at points which demarcate the 2 stages. As for the FPGA implementation, timing measurements do not include secondary storage media latencies.

stage	time
1	181.046 ms
2	132.331 ms

Table A.2: Throughput measured from the software encoder

A.3 Design flow

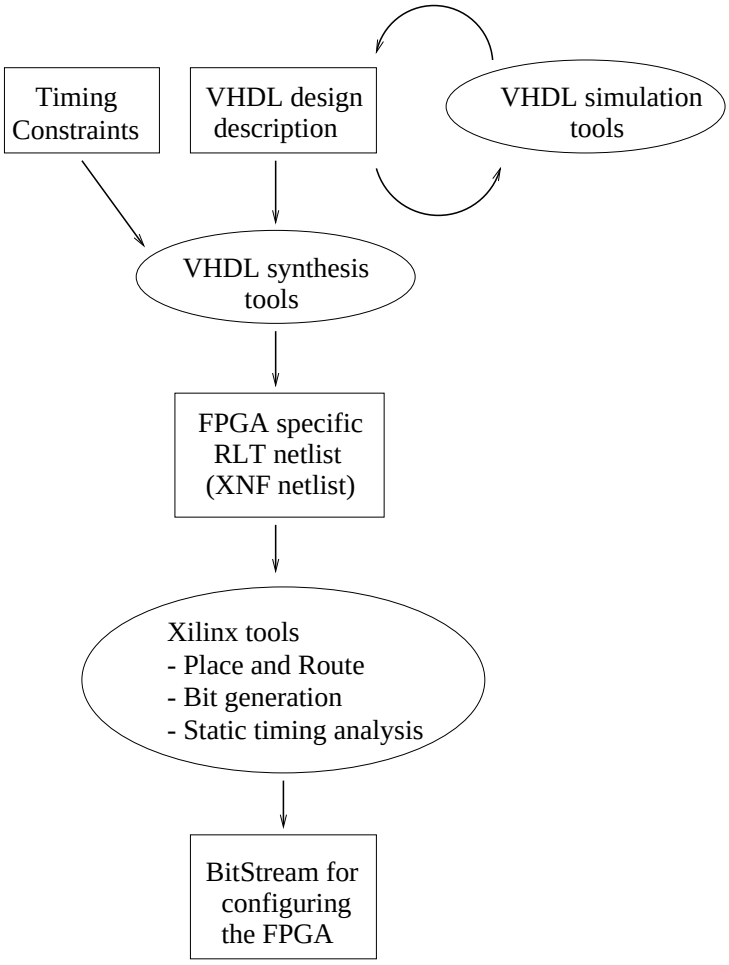

Figure A.1: Design flow

Appendix B

Source code listings

B.1 Stage 1 - VHDL source code

B.1.1 waveletX.vhd

```
--
--                WAVELET TRANSFORM IMPLEMENTATION
--                    Stage1 - Forward Wavelet (in X direction)
--
--
--  Author: sarin@ittc.ukans.edu, 03/27/2000
--
--   File : waveletX.vhd
--
--  Input : A 512x512 pixel image, streamed row wise, two pixels
--          at a time, 'p2' and 'p3'; two previous samples are
--          held in 'p0' and 'p1'.
--
--  Output: 'f' and 'g' are two weighted difference functions,
--          the data dependency is as shown below:
--
--   pixel:  0  1  2  3  4  5  6  7  8  ... 506 507 508 509 510 511
--           |\ | / \ | / \ | / \ | /        \  |   /  \  |  /\ / |
--       g:  |  g0    g1   g2     g3    ...    g253     g254 g255 |
--           | / \   /  \ /  \   /               \     /   \  |  |
--       f:  f0   f1    f2    f3        ...         f254      f255
--
--          The output is 256 values of 'f' and 256 values of 'g'.
--          Note that 'f' and 'g' at the boundary are slightly
--          different, due to which we need two additional signals
--          to signal row begining and ending.
--
--          Note that output is send back as 'f0', 'f1', 'g0', 'g1' ...
--          instead of 'f0', 'g0', 'f1', 'g1' ...
--          This is because of the order it is written back into
--          memory in the higher level module.
--
--  Pipeline latency .... (timing)
--
--         Input pixels        Output     FwavStart   FwavEnd  ForG
--         -------------+-------------+-----------+---------+-------
--            0   1            **   **        1         0       X
--            2   3            **   **        0         0       f (0)
--            4   5            **   **        0         0       g (1)
--            6   7            f0   f1        0         0       f (0)
--            8   9            g0   g1        0         0       g (1)
--           10  11            f2   f3        0         0       f (0)
--           12  13            g2   g3        0         0       g (1)
--           ......            .....         .         .       .
--          506 507          f250 f251        0         0       f (0)
--          508 509          g250 g251        0         0       g (1)
--          510 511          f252 f253        0         1       f (0)
--          *** ***          f252 g253        0         0       g (1)
--          *** ***          f254 f255        0         0       f (0)
--          *** ***          g254 g255        0         0       g (1)

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
use ieee.std_logic_signed.all;

entity ForwardWaveletX is
   port
   (  FwavClk      : in  std_logic;
      FwavEnbl     : in  std_logic;
      FwavStart    : in  std_logic;
      FwavEnd      : in  std_logic;
      Fwav_p3      : in  std_logic_vector (15 downto 0);
      Fwav_p2      : in  std_logic_vector (15 downto 0);
      Fwav_f       : out std_logic_vector (15 downto 0);
      Fwav_g       : out std_logic_vector (15 downto 0)
   );
end ForwardWaveletX;

architecture structural of ForwardWaveletX is
   constant prop_delay : time := 5 ns;
   subtype std16 is std_logic_vector (15 downto 0);

   signal p0, p1, g_out, g_prev1, g_prev2 : std16;
   signal f_tmp1, f_tmp2, f_out, f_prev   : std16;
   signal ForG, FwavStart1, FwavEnd1 : std_logic;

 begin

   run : process(FwavClk)
    begin
      if(rising_edge(FwavClk)) then
         if(FwavEnbl = '1') then
            p0         <= Fwav_p2    after prop_delay;
            p1         <= Fwav_p3    after prop_delay;

            if(ForG='1') then
               Fwav_f <= f_prev      after prop_delay;
               Fwav_g <= f_out       after prop_delay;
            else
               Fwav_f <= g_prev2     after prop_delay;
               Fwav_g <= g_prev1     after prop_delay;
            end if;

            if(FwavStart='1') then
                ForG  <= '0'         after prop_delay;
            else
                ForG  <= not(ForG)   after prop_delay;
            end if;

            g_prev2    <= g_prev1    after prop_delay;
            g_prev1    <= g_out      after prop_delay;
            f_prev     <= f_out      after prop_delay;

            FwavEnd1   <= FwavEnd    after prop_delay;
            FwavStart1 <= FwavStart  after prop_delay;
         end if;
      end if;
    end process;


   computeg : process(Fwav_p3, p1, p0, FwavEnd1)
     begin
       if(FwavEnd1='1') then
          g_out <= (p0(15) & p0(13 downto 0) & '0') -
                   (p1(15) & p1(13 downto 0) & '0');
       else
          g_out <= (p0(15) & p0(13 downto 0) & '0')-
                    (p1 + Fwav_p3);
       end if;
    end process;


   computef : process(FwavStart1, g_out, g_prev1,
                      f_tmp1, f_tmp2, p1)
    begin
       if(FwavStart1='1') then
          f_tmp1 <= g_out   +
                    g_out;
       else
          f_tmp1 <= g_prev1 +
                    g_out;
       end if;

    -- if((f_tmp1(15) = '1') and
    --    ((f_tmp1(2)='1') or (f_tmp1(1)='1') or (f_tmp1(0)='1'))) then
    --    f_tmp2 <= (f_tmp1(15) & f_tmp1(15) & f_tmp1(15) & f_tmp1(15 downto 3)) + 1;
    -- else
          f_tmp2 <= (f_tmp1(15) & f_tmp1(15) & f_tmp1(15) & f_tmp1(15 downto 3));
    -- end if;

       f_out <= p1 + f_tmp2;
```

```
   end process;

end structural;
```

B.1.2 waveletY.vhd

```
--
--                 WAVELET TRANSFORM IMPLEMENTATION
--                     Stage1 - Forward Wavelet (in Y direction)
--
--
--  Author: sarin@ittc.ukans.edu, 03/27/2000
--
--  File  : waveletY.vhd
--
--  Input : A 512x512 pixel image, streamed row wise, two pixels
--          at a time, 'p2' and 'p3'; two previous samples are
--          held in 'p0' and 'p1'.
--
--  Output: 'f' and 'g' are two weighted difference functions,
--          the data dependency is as shown below:
--
--   pixel:  0  1  2  3  4  5  6  7  8  ... 506 507 508 509 510 511
--           |\ | / \ | / \ | / \ | /        \  |   /  \  |  /\ / |
--       g:  |  g0    g1   g2     g3    ...    g253     g254 g255 |
--           | / \   /  \ /  \   /                \     /   \  |  |
--       f:  f0   f1    f2    f3        ...         f254      f255
--
--          The output is 256 values of 'f' and 256 values of 'g'.
--          Note that 'f' and 'g' at the boundary are slightly
--          different, due to which we need two additional signals
--          to signal row begining and ending.
--
--  Pipeline latency .... (timing)
--
--         Input pixels      Output     FwavStart   FwavEnd  ForG
--         ------------+--------------+-----------+---------+-------
--            0   0          *    *        1           0
--            1   1          *    *        0           0       0 (f)
--            2   2          *    *        0           0       1 (g)
--            3   3         f0   f0        0           0       0 (f)
--            4   4         g0   g0        0           0       1 (g)
--            5   5         f1   f1        0           0       0 (f)
--            6   6         g1   g1        0           0       1 (g)
--            7   7         f2   f2        0           0       0 (f)
--            8   8         g2   g2        0           0       1 (g)
--           ......        ...  ...        .           .       .
--          509 509       f253 f253        0           0       0 (f)
--          510 510       g253 g253        0           0       1 (g)
--          511 511       f254 f254        0           1       0 (f)
--          *** ***       f254 g254        0           0       1 (g)
--          *** ***       f255 f255        0           0       0 (f)
--          *** ***       g255 g255        0           0       1 (g)

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

entity ForwardWaveletY is
   port
   (  FwavClk      : in  std_logic;
      FwavEnbl     : in  std_logic;
      FwavStart    : in  std_logic;
      FwavEnd      : in  std_logic;
      Fwav_a4      : in  std_logic_vector (15 downto 0);
      Fwav_b4      : in  std_logic_vector (15 downto 0);
      Fwav_a       : out std_logic_vector (15 downto 0);
      Fwav_b       : out std_logic_vector (15 downto 0);
      Fwav_Max     : out std_logic
   );
end ForwardWaveletY;

architecture structural of ForwardWaveletY is
   constant prop_delay : time := 5 ns;
   subtype std1  is std_logic;
   subtype std16 is std_logic_vector (15 downto 0);

   signal a2, a3    : std16; -- 2 previous values of a4
   signal b2, b3    : std16; -- 2 previous values of b4

   signal a_g, b_g, a_gd    : std16; -- g outputs of 2 streams,
   signal b_gd, a_gdd, b_gdd: std16; -- latched, double latched ...
```

```
  signal a_f1, a_f2, a_f3, a_f3d : std16; -- f, partial outputs
  signal b_f1, b_f2, b_f3, b_f3d : std16; -- f, partial outputs

  signal FwavStart1, FwavStart2  : std1;  -- delayed FwavStart's
  signal FwavEnd1, FwavEnd2      : std1;  -- and FwavEnd's

  signal ForG : std1;              -- keep track of whether f or g
                                   -- is going out

begin

  run : process(FwavClk)
   begin
     if(rising_edge(FwavClk)) then
        if(FwavEnbl = '1') then
           a2      <= a3           after prop_delay;
           a3      <= Fwav_a4      after prop_delay;

           b2      <= b3           after prop_delay;
           b3      <= Fwav_b4      after prop_delay;

           FwavStart2<= FwavStart1 after prop_delay;
           FwavStart1<= FwavStart  after prop_delay;
           FwavEnd1  <= FwavEnd    after prop_delay;
           FwavEnd2  <= FwavEnd1   after prop_delay;

           a_gdd   <=  a_gd        after prop_delay;
           b_gdd   <=  b_gd        after prop_delay;

           a_gd    <=  a_g         after prop_delay;
           b_gd    <=  b_g         after prop_delay;

           a_f3d   <=  a_f3        after prop_delay;
           b_f3d   <=  b_f3        after prop_delay;

           if(FwavStart='1') then
               ForG  <= '0'        after prop_delay;
           else
               ForG  <= not(ForG)  after prop_delay;
           end if;

        end if;
     end if;
   end process;

  computeg : process(a2, a3, Fwav_a4,
                     b2, b3, Fwav_b4,
                     FwavEnd1)
    begin
      if(FwavEnd1='1') then
         a_g <= (a3(15) & a3(13 downto 0) & '0')-
                (a2(15) & a2(13 downto 0) & '0');
         b_g <= (b3(15) & b3(13 downto 0) & '0')-
                (b2(15) & b2(13 downto 0) & '0');
      else
         a_g <= (a3(15) & a3(13 downto 0) & '0')-
                 a2 -
                 Fwav_a4;
         b_g <= (b3(15) & b3(13 downto 0) & '0')-
                 b2 -
                 Fwav_b4;
      end if;
   end process;

  computef : process(FwavStart2, a2, b2,
                     a_g, b_g, a_gdd, b_gdd,
                     a_f1, b_f1, a_f2, b_f2)
   begin

      if(FwavStart2='1') then
         a_f1 <= a_g   +    -- current g
                 a_g;       -- current g
         b_f1 <= b_g   +    -- current g
                 b_g;       -- current g
      else
         a_f1 <= a_gdd +    -- prev    g
                 a_g;       -- current g
         b_f1 <= b_gdd +    -- prev    g
                 b_g;       -- current g
      end if;

      -- divide by 8 and drop fractional part,
      -- because of two's compliment representation, if number is
      -- negative and there is a non zero fractional value, we need to
      -- add 1 after dropping the fractional part.

   -- if((a_f1(15) = '1') and
```

```
    --    ((a_f1(2)='1') or (a_f1(1)='1') or (a_f1(0)='1'))) then
    --    a_f2 <= (a_f1(15) & a_f1(15) & a_f1(15) & a_f1(15 downto 3)) + 1;
    -- else
          a_f2 <= (a_f1(15) & a_f1(15) & a_f1(15) & a_f1(15 downto 3));
    -- end if;

    -- if((b_f1(15) = '1') and
    --    ((b_f1(2)='1') or (b_f1(1)='1') or (b_f1(0)='1'))) then
    --    b_f2 <= (b_f1(15) & b_f1(15) & b_f1(15) & b_f1(15 downto 3)) + 1;
    -- else
          b_f2 <= (b_f1(15) & b_f1(15) & b_f1(15) & b_f1(15 downto 3));
    -- end if;

       a_f3 <= a_f2 + a2;
       b_f3 <= b_f2 + b2;
    end process;


   out_mux : process(ForG, a_f3d, a_gdd, b_f3d, b_gdd)
    begin
        if(ForG = '0') then
            Fwav_a <= a_f3d;
            Fwav_b <= b_f3d;

            if(a_f3d > b_f3d) then
               Fwav_Max <= '0';
            else
               Fwav_Max <= '1';
            end if;

        else
            Fwav_a <= a_gdd;
            Fwav_b <= b_gdd;

            if(a_gdd > b_gdd) then
               Fwav_Max <= '0';
            else
               Fwav_Max <= '1';
            end if;

        end if;
    end process;

end structural;
```

B.1.3 pe1lca.vhd (top level for stage1)

```
--
--              WF4 memory interface for the wavelet dataflow
--                          block - Forward Wavelet
--
--  File  : pe1lca.vhd
--
--  Author: sarin@ittc.ukans.edu, 01/11/2000
--
--  Description:
--          This file along with "waveletX.vhd" & "waveletY.vhd" implements
--          DWT (discrete wavelet transform) / multi resolution encoding
--          of the input image.
--
--          The input image is 512x512 pixels, with each memory WORD
--          holding 2 pixels (12 bits each) the input is a 512x256
--          memory array (0.5 MB).
--
--          Stage 1: Process each row (512 pixels), extract 256 'f's and
--                   256 'g's from each row, write it back in place:
--                     [pppppppp...pppp] => [fgfgfgfg...fgfg]
--
--                   Instead, it is actually written back as:
--                     [pppppppp...pppp] => [ffggffgg...ffgg].
--
--                   Then same operation along Y direction.
--
--          Stage 2: Only 'f's from first stage are input to second stage.
--                   Thus we have rows of length 256.
--                   (see why f/g outputs from stage1 was written back
--                    jumbled? need only 256 memory READS, else it would
--                    have taken 512 memory READS).
--
--          Stage 3: The third stage follows similarly, processing only
--                   the 'f's from second stage. Each stage has to be
--                   done in both X and Y directions.
--
```

```
--          It is smooth sailing in X direction with two pixels of a row
--          arriving on each memory READ and two values being written back
--          in each memory WRITE.
--
--          In Y direction, we have to perform two memory READs to get two
--          consecutive values of a stream (column). By then we also get
--          two consecutive values from the next (vertically parallel) stream.
--          Hence, two different ForwardWavelet blocks (ForwardWaveletX and
--          ForwardWaveletY) are used for the X and Y directions.
--
--          ForwardWaveletX: accepts two successive values of the same row
--                           and outputs either two consecutive f's or
--                           two consecutive g's (alternately).
--
--          ForwardWaveletY: accepts one pixel each from two columns and
--                           outputs either one f each of the two columns or
--                           one g each of the two columns (alternately).


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

architecture Memory_Access of PE1_Logic_Core is

    component ForwardWaveletX is
       port ( FwavClk     : in  std_logic;
              FwavEnbl    : in  std_logic;
              FwavStart   : in  std_logic;
              FwavEnd     : in  std_logic;
              Fwav_p3     : in  std_logic_vector (15 downto 0);
              Fwav_p2     : in  std_logic_vector (15 downto 0);
              Fwav_f      : out std_logic_vector(15 downto 0);
              Fwav_g      : out std_logic_vector(15 downto 0));
    end component;

    component ForwardWaveletY is
       port ( FwavClk     : in  std_logic;
              FwavEnbl    : in  std_logic;
              FwavStart   : in  std_logic;
              FwavEnd     : in  std_logic;
              Fwav_a4     : in  std_logic_vector (15 downto 0);
              Fwav_b4     : in  std_logic_vector (15 downto 0);
              Fwav_a      : out std_logic_vector (15 downto 0);
              Fwav_b      : out std_logic_vector (15 downto 0);
              Fwav_Max    : out std_logic);
    end component;

    type MemoryStates is
      ( MemWaitforBus,
        MemRead001,       -- READ fired in this cycle, results later
        MemRead010,       -- READ fired last cycle, still waiting for results
        MemRead100,       -- READ result arrives
        MemWrite,         -- normal coefficients
        MemWriteMinMax1,  -- coefficient min/max block1 (done at end of each stage)
        MemWriteMinMax2,  -- coefficient min/max block2
        MemWriteMinMax3,  -- coefficient min/max block3
        MemWriteMinMax4,  -- coefficient min/max block4
        MemInterrupt,
        MemDone);         -- Black hole state!

    signal Mem_PState   : MemoryStates;    -- Present state
    signal Mem_NState   : MemoryStates;    -- Next state
    signal Enbl         : std_logic;
    signal nPass        : std_logic_vector(1 downto 0); -- 00, 01, 10

    signal ENDofROWx    : std_logic;       -- Row BEGIN and END signals
    signal STARTofROWx  : std_logic;

    signal ENDofROWy    : std_logic;       -- Col BEGIN and END signals
    signal STARTofROWy  : std_logic;

    signal ROWorCOL     : std_logic;       -- 0=>row, 1=>col.
    signal ROWorCOL1    : std_logic;       -- delayed versions of
    signal ROWorCOL2    : std_logic;       -- ROWorCOL
    signal ROWorCOL3    : std_logic;

    signal cntrROW      : std_logic_vector(8 downto 0); -- ROW, COL address
    signal cntrCOL      : std_logic_vector(7 downto 0); -- registers for READ

    signal cntrROW_old1 : std_logic_vector(8 downto 0);
    signal cntrCOL_old1 : std_logic_vector(7 downto 0);

    signal cntrROW_old2 : std_logic_vector(8 downto 0);
    signal cntrCOL_old2 : std_logic_vector(7 downto 0);

    signal cntrROW_old3 : std_logic_vector(8 downto 0); -- ROW, COL address
```

```
   signal cntrCOL_old3 : std_logic_vector(7 downto 0); -- registers for WRITE.

   signal ROW_limit    : std_logic_vector(8 downto 0); -- 511, 510, 508
   signal COL_limit    : std_logic_vector(7 downto 0); -- 255, 254, 252

   signal ROW_skip     : std_logic_vector(8 downto 0); -- 1, 2, 4
   signal COL_skip     : std_logic_vector(7 downto 0); -- 1, 2, 4

   signal userInputU   : std_logic_vector(15 downto 0); -- Input, from mem READ
   signal userInputL   : std_logic_vector(15 downto 0);

   signal userOutputUx : std_logic_vector(15 downto 0); -- Output from
   signal userOutputLx : std_logic_vector(15 downto 0); -- ForwardWaveletX

   signal userOutputUy : std_logic_vector(15 downto 0); -- Output from
   signal userOutputLy : std_logic_vector(15 downto 0); -- ForwardWaveletY

   signal normalCoeffU : std_logic_vector(15 downto 0); --
   signal normalCoeffL : std_logic_vector(15 downto 0); --
   signal Fwav_MaxY    : std_logic;

   signal block1min    : std_logic_vector(15 downto 0); -- Collect coefficient
   signal block1max    : std_logic_vector(15 downto 0); -- MIN/MAX at each stage of
   signal block2min    : std_logic_vector(15 downto 0); -- Waveletting, to be used
   signal block2max    : std_logic_vector(15 downto 0); -- the next stage - quantizer
   signal block3min    : std_logic_vector(15 downto 0);
   signal block3max    : std_logic_vector(15 downto 0); -- This saves an additional
   signal block4min    : std_logic_vector(15 downto 0); -- pass over the data.
   signal block4max    : std_logic_vector(15 downto 0);

begin
     wlet_x : ForwardWaveletX
        port map ( PE_Pclk,           -- here is the forward Wavelet
                   Enbl,              -- transform block for X direction
                   STARTofROWx,
                   ENDofROWx,
                   userInputU,
                   userInputL,
                   userOutputUx,
                   userOutputLx);

     wlet_y : ForwardWaveletY
        port map ( PE_Pclk,           -- here is the forward Wavelet
                   Enbl,              -- transform block for Y direction
                   STARTofROWy,
                   ENDofROWy,
                   userInputU,
                   userInputL,
                   userOutputUy,
                   userOutputLy,
                   Fwav_MaxY);


     memdata_mux : process (ROWorCOL3, Mem_PState,
                            userOutputUx, userOutputLx,
                            userOutputUy, userOutputLy,
                            normalCoeffU, normalCoeffL,
                            block1max, block1min)
      begin

       if(ROWorCOL3='0') then
          normalCoeffU <= userOutputUx;
          normalCoeffL <= userOutputLx;
       else
          normalCoeffU <= userOutputUy;
          normalCoeffL <= userOutputLy;
       end if;

       if(Mem_PState=MemWrite) then
          PE_MemData_OutReg(31 downto 16)<= normalCoeffU;
          PE_MemData_OutReg(15 downto 0) <= normalCoeffL;
       else
          PE_MemData_OutReg(31 downto 16)<= block1max;
          PE_MemData_OutReg(15 downto 0) <= block1min;
       end if;

     end process memdata_mux;



     st_update : process (PE_Pclk, PE_Reset)
       variable xtest, ytest: std_logic;
      begin

       if (PE_Reset = '1') then

          Mem_PState      <= MemWaitforBus; -- Initialize current state
```

```
   cntrROW        <= "000000000";   -- Initialize ROW and COL
   cntrCOL        <= "00000000";    -- address registers.
   cntrROW_old1   <= "000000000";
   cntrCOL_old1   <= "00000000";
   cntrROW_old2   <= "000000000";
   cntrCOL_old2   <= "00000000";
   cntrROW_old3   <= "000000000";
   cntrCOL_old3   <= "00000000";

   ROWorCOL       <= '0';           -- Initialize ROW / COL
   ROWorCOL1      <= '0';           -- direction indicator
   ROWorCOL2      <= '0';           -- to ROW
   ROWorCOL3      <= '0';

   userInputU     <= (others => '0');
   userInputL     <= (others => '0');

   block1max      <= (others => '0');
   block1min      <= (others => '0');
   block2max      <= (others => '0');
   block2min      <= (others => '0');
   block3max      <= (others => '0');
   block3min      <= (others => '0');
   block4max      <= (others => '0');
   block4min      <= (others => '0');

   nPass          <= "00";

   ROW_skip      <= "000000001";    --  1, 2, 4
   COL_skip      <=  "00000001";    --  1, 2, 4


   -- Pass 1 covers:
   --   ROWS [0,1,2,3, ..., 511] and COLS [0,1,2,3, ..., 255]
   --
   -- Pass 2 covers:
   --   ROWS [0,2,4,6, ..., 510] and COLS [0,2,4,6, ..., 254]
   --
   -- Pass 3 covers:
   --   ROWS [0,4,8,12,..., 508] and COLS [0,4,8,12,..., 252]

   ROW_limit      <= "111111111";   -- Initialize to 511
   COL_limit      <= "11111111";    -- Initialize to 255

elsif (rising_edge(PE_Pclk)) then

   Mem_PState     <= Mem_NState;

   if (Mem_PState = MemWrite) then

      if((cntrROW = ROW_limit) and        -- Switch between
         (cntrCOL = COL_limit)) then      -- X,Y directions
          ROWorCOL  <=  not(ROWorCOL);    -- (ROWorCOL=0) => X,
                                          -- (ROWorCOL=1) => Y.

          if(ROWorCOL = '1') then
             ROW_skip   <= (ROW_skip(7 downto 0) & '0');
             COL_skip   <= (COL_skip(6 downto 0) & '0');

             ROW_limit  <= (ROW_limit(7 downto 0) & '0');
             COL_limit  <= (COL_limit(6 downto 0) & '0');

             nPass      <= UNSIGNED(nPass) + 1;
          end if;
      end if;

      ROWorCOL1  <= ROWorCOL;             -- update delayed
      ROWorCOL2  <= ROWorCOL1;            -- versions of
      ROWorCOL3  <= ROWorCOL2;            -- ROWorCOL


      if(ROWorCOL = '0') then
          cntrCOL <= UNSIGNED(cntrCOL) + UNSIGNED(COL_skip);
          if (cntrCOL = COL_limit) then
              cntrROW <= UNSIGNED(cntrROW) + UNSIGNED(ROW_skip);
          end if;
      else
          cntrROW <= UNSIGNED(cntrROW) + UNSIGNED(ROW_skip);
          if (cntrROW = ROW_limit) then
              cntrCOL <= UNSIGNED(cntrCOL) + UNSIGNED(COL_skip);
          end if;
      end if;

      cntrROW_old1 <= cntrROW;            -- 2 sets of address
      cntrCOL_old1 <= cntrCOL;            -- regsisters, one for
                                          -- memory READ and another
      cntrROW_old2 <= cntrROW_old1;       -- for memory WRITE.
      cntrCOL_old2 <= cntrCOL_old1;       --
```

```
                                        -- WRITE lags the READ
   cntrROW_old3 <= cntrROW_old2;        -- by the latency of
   cntrCOL_old3 <= cntrCOL_old2;        -- ForwardWavelet.
end if;

if (Mem_PState = MemRead100) then
   userInputU <= PE_MemData_InReg(31 downto 16);
   userInputL <= PE_MemData_InReg(15 downto 0);
end if;

if ((Mem_PState = MemWriteMinMax1) or
    (Mem_PState = MemWriteMinMax2) or
    (Mem_PState = MemWriteMinMax3) or
    (Mem_PState = MemWriteMinMax4)) then
   block1max <= block2max;
   block1min <= block2min;

   block2max <= block3max;
   block2min <= block3min;

   block3max <= block4max;
   block3min <= block4min;

   block4max <= (others => '0');
   block4min <= (others => '0');

elsif (Mem_PState = MemWrite) then

   if (nPass = "00") then
     xtest := cntrROW_old3(0);
     ytest := cntrCOL_old3(0);
   elsif (nPass = "01") then
     xtest := cntrROW_old3(1);
     ytest := cntrCOL_old3(1);
   else
     xtest := cntrROW_old3(2);
     ytest := cntrCOL_old3(2);
   end if;

   if ((xtest = '0') and (ytest = '0') and (ROWorCOL3='1')) then
     if(Fwav_MaxY='0') then
        if(SIGNED(block1max) < SIGNED(normalCoeffU)) then
           block1max <= normalCoeffU;
        end if;
        if(SIGNED(block1min) > SIGNED(normalCoeffL)) then
           block1min <= normalCoeffL;
        end if;
     else
        if(SIGNED(block1max) < SIGNED(normalCoeffL)) then
           block1max <= normalCoeffL;
        end if;
        if(SIGNED(block1min) > SIGNED(normalCoeffU)) then
           block1min <= normalCoeffU;
        end if;
     end if;
   end if;

   if ((xtest = '0') and (ytest = '1') and (ROWorCOL3='1')) then
     if(Fwav_MaxY='0') then
        if(SIGNED(block2max) < SIGNED(normalCoeffU)) then
           block2max <= normalCoeffU;
        end if;
        if(SIGNED(block2min) > SIGNED(normalCoeffL)) then
           block2min <= normalCoeffL;
        end if;
     else
        if(SIGNED(block2max) < SIGNED(normalCoeffL)) then
           block2max <= normalCoeffL;
        end if;
        if(SIGNED(block2min) > SIGNED(normalCoeffU)) then
            block2min <= normalCoeffU;
        end if;
     end if;
   end if;

   if ((xtest = '1') and (ytest = '0') and (ROWorCOL3='1')) then
     if(Fwav_MaxY='0') then
        if(SIGNED(block3max) < SIGNED(normalCoeffU)) then
           block3max <= normalCoeffU;
        end if;
        if(SIGNED(block3min) > SIGNED(normalCoeffL)) then
           block3min <= normalCoeffL;
        end if;
     else
        if(SIGNED(block3max) < SIGNED(normalCoeffL)) then
           block3max <= normalCoeffL;
        end if;
        if(SIGNED(block3min) > SIGNED(normalCoeffU)) then
```

```
               block3min <= normalCoeffU;
            end if;
          end if;
        end if;

        if ((xtest = '1') and (ytest = '1') and (ROWorCOL3='1')) then
          if(Fwav_MaxY='0') then
             if(SIGNED(block4max) < SIGNED(normalCoeffU)) then
                block4max <= normalCoeffU;
             end if;
             if(SIGNED(block4min) > SIGNED(normalCoeffL)) then
                block4min <= normalCoeffL;
             end if;
          else
             if(SIGNED(block4max) < SIGNED(normalCoeffL)) then
                block4max <= normalCoeffL;
             end if;
             if(SIGNED(block4min) > SIGNED(normalCoeffU)) then
                block4min <= normalCoeffU;
             end if;
          end if;
        end if;

     end if;

  end if;
end process st_update;


start_end : process(ROWorCOL,
                    cntrCOL, cntrROW,
                    ROW_limit, COL_limit)
   begin
       STARTofROWx <= '0';
       ENDofROWx   <= '0';
       STARTofROWy <= '0';
       ENDofROWy   <= '0';

       if(ROWorCOL = '0') then             -- if X direction
           if (cntrCOL = "00000000") then  --    STARTofROW
               STARTofROWx <= '1';
           end if;

           if (cntrCOL = COL_limit) then   --    ENDofROW
               ENDofROWx <= '1';
           end if;
       else                                -- else if Y direction
           if (cntrROW = "000000000") then --    STARTofROW
               STARTofROWy <= '1';
           end if;

           if (cntrROW = ROW_limit) then   --    ENDofROW
               ENDofROWy <= '1';
           end if;
       end if;
   end process;


PE_MemAddr_OutReg(21 downto 18) <= (others => '0');


mem_state: process(Mem_PState, nPass,
                   PE_MemBusGrant_n,
                   ROWorCOL2, ROWorCOL3,
                   cntrROW, cntrCOL,
                   cntrROW_old3, cntrCOL_old3,
                   PE_InterruptAck_n)
  begin
       PE_InterruptReq_n   <= '1';  -- Default, do not interrupt host
       PE_MemWriteSel_n    <= '1';  -- read/write, default read
       PE_MemStrobe_n      <= '1';  -- No strobe, later
       PE_MemBusReq_n      <= '0';  -- Always request bus
       Enbl                <= '0';
       PE_MemAddr_OutReg(17 downto 0)  <= (others => 'X');

 case Mem_PState is

    when MemWaitforBus =>                   -- Wait for bus, when bus is
       if(PE_MemBusGrant_n = '0') then      -- available, fire READ in
          Mem_NState        <= MemRead001;  -- in next clock. Firing READ
       else                                 -- in same clock kills the
          Mem_NState        <= MemWaitforBus;-- timing performance...
       end if;

    when MemRead001 =>                      -- Fire READ, results of
       PE_MemStrobe_n    <= '0';            -- this will come later...
       Mem_NState        <= MemRead010;
```

```
   PE_MemAddr_OutReg(17)          <= '0';
   PE_MemAddr_OutReg(16 downto 8) <= cntrROW; -- Use cntrROW and
   PE_MemAddr_OutReg( 7 downto 0) <= cntrCOL; -- cntrCOL for READ

when MemRead010 =>                       -- Still waiting for
   Mem_NState        <= MemRead100;      -- READ results...

when MemRead100 =>                       -- Got READ results here
   Mem_NState        <= MemWrite;

when MemWrite   =>
   Enbl              <= '1';
   PE_MemWriteSel_n  <= '0';  -- for writing
   PE_MemStrobe_n    <= '0';

   PE_MemAddr_OutReg(17)          <= '0';
   PE_MemAddr_OutReg(16 downto 8) <= cntrROW_old3; --Use cntrROW_old3
   PE_MemAddr_OutReg( 7 downto 0) <= cntrCOL_old3; --cntrCOL_old3 for
                                                   --WRITE

   if((ROWorCOL3 = '1') and              -- If COL->ROW switch
      (ROWorCOL2 = '0')) then            -- Write max/min statistics
       Mem_NState    <= MemWriteMinMax1;
   else
       Mem_NState    <= MemRead001;
   end if;

-- After each stage of wave-letting, we get 4 blocks,
-- the MAX and MIN values of coefficients in each block are
-- computed for use in next stage, (dynamic quantization).
-- At the end of each stage, we write back 4 WORDs for
-- each of the 4 blocks (each word contains MAX and MIN, 15 bits each),
-- into an upper portion of memory (unused).
-- The addressing scheme is as follows:

--     0XXXXXXXXXXXXX XX XX  // normal data/coefficients

--     10000000000000 00 00  // 4 blocks from stage1
--     10000000000000 00 01
--     10000000000000 00 10
--     10000000000000 00 11
--
--     10000000000000 01 00  // 4 blocks from stage2
--     10000000000000 01 01
--     10000000000000 01 10
--     10000000000000 01 11
--
--     10000000000000 10 00  // 4 blocks from stage3
--     10000000000000 10 01
--     10000000000000 10 10
--     10000000000000 10 11


when MemWriteMinMax1   =>
   PE_MemWriteSel_n  <= '0';  -- for writing
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= MemWriteMinMax2;

   PE_MemAddr_OutReg(17 downto 4)    <= "10000000000000";
   PE_MemAddr_OutReg(3 downto 2)     <= nPass;
   PE_MemAddr_OutReg(1 downto 0)     <= "00";


when MemWriteMinMax2   =>
   PE_MemWriteSel_n  <= '0';  -- for writing
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= MemWriteMinMax3;

   PE_MemAddr_OutReg(17 downto 4)    <= "10000000000000";
   PE_MemAddr_OutReg(3 downto 2)     <= nPass;
   PE_MemAddr_OutReg(1 downto 0)     <= "01";


when MemWriteMinMax3   =>
   PE_MemWriteSel_n  <= '0';  -- for writing
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= MemWriteMinMax4;

   PE_MemAddr_OutReg(17 downto 4)    <= "10000000000000";
   PE_MemAddr_OutReg(3 downto 2)     <= nPass;
   PE_MemAddr_OutReg(1 downto 0)     <= "10";


when MemWriteMinMax4   =>
   PE_MemWriteSel_n  <= '0';  -- for writing
   PE_MemStrobe_n    <= '0';

   PE_MemAddr_OutReg(17 downto 4)    <= "10000000000000";
```

```
            PE_MemAddr_OutReg(3 downto 2)     <= nPass;
            PE_MemAddr_OutReg(1 downto 0)     <= "11";

            if((cntrROW_old3 = "000000000") and   -- Wind up after
               (cntrCOL_old3 = "00000000")  and   -- 3 passes.
               (nPass = "11")) then
                Mem_NState    <= MemInterrupt;
            else
                Mem_NState    <= MemRead001;
            end if;

         when MemInterrupt =>
            PE_MemBusReq_n    <= '1';         -- Give up bus
            PE_InterruptReq_n <= '0';         -- Interrupt host
            if(PE_InterruptAck_n = '0') then
               Mem_NState        <= MemDone;
            else
               Mem_NState        <= MemInterrupt;
            end if;

         when MemDone =>
            PE_MemBusReq_n    <= '1';         -- Give up bus, host program
            Mem_NState        <= MemDone;     -- to READ memory now...
      end case;

      end process mem_state;

      ---------------------------------------------------------------------------
      --  "Inactive" output port signal assignments
      ---------------------------------------------------------------------------
       PE_MemHoldReq_n   <= '1';           -- Disable memory hold requests

       PE_Left_OE  <= ( others => '0' );   -- Disable left port output

       PE_Right_OE <= ( others => '0' );   -- Disable right port output

       PE_FifoSelect <= "00";              -- Deselect fifo
                                           --  "00" selects none
                                           --  "01" selects External I/O Fifo
                                           --  "10" selects On-Board Fifo
                                           --  "11" selects On-Board Mailbox
       PE_Fifo_WE_n      <= '1';           -- Disable fifo write mode
       PE_FifoPtrIncr_EN <= '0';           -- Disable fifo pointer increment

end Memory_Access;
```

B.2 Stage 2 - VHDL source code

B.2.1 quantizer.vhd

```
--
--                WAVELET TRANSFORM IMPLEMENTATION
--                    Stage2 - Dynamic Quantizer
--
--  Author : sarin@ittc.ukans.edu, 06/10/2000
--
--    File : quantizer.vhd
--
--  Design : Given a stream of numbers, the stream is quantized into 16
--           levels (4 bits). The 16 quantization levels are:
--
--           [ min                    -> min +  1*(max-min+8)/16 ] => "0000"
--           [ min +  2*(max-min+8)/16 -> min +  3*(max-min+8)/16 ] => "0001"
--           [ min +  3*(max-min+8)/16 -> min +  4*(max-min+8)/16 ] => "0010"
--             ....
--           [ min + 14*(max-min+8)/16 -> min + 15*(max-min+8)/16 ] => "1110"
--           [ min + 15*(max-min+8)/16 -> max                     ] => "1111"
--
--           'min' and 'max' are not know prior and depends on the
--           input stream making it a dynamic quantizer.
--
--  Input  : A stream of 15 bit numbers on 'QUANTin'
--  Output : The quantized (4 bit) values on 'QUANTout'.
--

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

```
entity QUANT is
   port
   (  QUANTclk   : in  std_logic;
      QUANTen    : in  std_logic;
      QUANTmax   : in  std_logic_vector (15 downto 0);
      QUANTmin   : in  std_logic_vector (15 downto 0);
      QUANTin    : in  std_logic_vector (15 downto 0);
      QUANTout   : out std_logic_vector ( 3 downto 0)
   );
end QUANT;

architecture structural of QUANT is
  subtype std4  is std_logic_vector ( 3 downto 0);
  subtype std16 is std_logic_vector (15 downto 0);
  subtype std20 is std_logic_vector (19 downto 0);

  signal r      : std16;
  signal r_by_2 : std20;
  signal r_by_4 : std20;
  signal r_by_8 : std20;
  signal r_by_16: std20;

  signal in1    : std16;
  signal in2    : std16;
  signal in3    : std16;
  signal in4    : std16;

  signal cmp1   : std20;
  signal cmp2   : std20;
  signal cmp3   : std20;
  signal cmp4   : std20;

  signal level1 : std4;
  signal level2 : std4;
  signal level3 : std4;
  signal level4 : std4;

begin

      r <= (QUANTmax - QUANTmin);

  run : process(QUANTclk)
    begin

      if(rising_edge(QUANTclk)) then

        if(QUANTen = '1') then

          -- The nice thing here is that at the edges of subbands
          -- when the range changes, the subranges also changes
          -- in sync with the data.

          r_by_2  <= (r(15) & r & "000");
          r_by_4  <= (r_by_2(19) & r_by_2(19 downto 1));
          r_by_8  <= (r_by_4(19) & r_by_4(19 downto 1));
          r_by_16 <= (r_by_8(19) & r_by_8(19 downto 1));

          in4 <= in3;
          in3 <= in2;
          in2 <= in1;
          in1 <= (QUANTin - QUANTmin); -- DC shifting.

          if(SIGNED(in1) > SIGNED(r_by_2(19 downto 4))) then
             level1 <= "1000";
             cmp1   <= (r_by_2 + r_by_4);
          else
             level1 <= "0000";
             cmp1   <= (r_by_2 - r_by_4);
          end if;

          if(SIGNED(in2 & '0') > SIGNED(cmp1(19 downto 3))) then
             level2 <= (level1 or "0100");
             cmp2   <= (cmp1 + r_by_8);
          else
             level2 <= level1;
             cmp2   <= (cmp1 - r_by_8);
          end if;

          if(SIGNED(in3 & "0000") > SIGNED(cmp2(19 downto 0))) then
             level3 <= (level2 or "0010");
             cmp3   <= (cmp2 + r_by_16);
          else
             level3 <= level2;
             cmp3   <= (cmp2 - r_by_16);
          end if;
             cmp4   <= cmp3;

          if(SIGNED(in4 & "0000") > SIGNED(cmp3(19 downto 0))) then
```

```
            level4 <= (level3 or "0001");
         else
            level4 <= level3;
         end if;

       end if;       -- end if(QUANTen='1')
     end if;       -- rising_edge(clk)
   end process run;

   QUANTout <= level4;
end structural;
```

B.2.2 rle.vhd

```
--
--                WAVELET TRANSFORM IMPLEMENTATION
--                    Stage3 - Run Length Encoder (for ZEROS only)
--
--  Author : sarin@ittc.ukans.edu, 04/14/2000
--
--   File  : rle.vhd
--
--  Input  : A stream of 15 bit numbers on 'RLEin',
--           the zero threshold value on 'RLEzeroth',
--           an enable signal on 'RLEen'.
--
--  Output : Output stream of 8 bit numbers on 'RLEout',
--           other control outputs on 'RLErunning' and 'RLEspellEnd'.
--
--  Design : The input stream is compared with zero threshold
--           to decide if it should be truncated to zero.
--           Any continuous sequence of ZEROes are run length
--           encoded, and the sum is output on 'RLEout'.
--
--           The RLE works like this:
--
--               Whenever we detect a ZERO, we would assert 'RLErunning',
--               and start counting the sequence of continuous ZEROes.
--               The current sum of ZEROes is always available on 'RLEout'.
--               When ever the continuous spell of ZEROes end,
--               we unset 'RLErunning' and assert 'RLEspellEnd' for one cycle
--               ( to allow the higher block to read off the RLE count )
--               and we also reset our internal counter.
--
--               Yeah, there is look ahead problem? Before we signal the end
--               of a spell, we need to see the next value is the stream.
--               Luckily, RLE is used in conjunction with a quantizer,
--               (RLE and quantizer are connected in parallel) which is a
--               4 staged pipeline.
--
--               We may face an arbitrarily long sequence of ZEROes. From the
--               design specs we are allowed to count only upto 240 ZEROes:
--
--                    output of quantizer:  00000000  (16 quantization levels)
--                                            ...
--                                          00001111
--
--                    output of RLE:        00010000  (256-16 = 240)
--                                            ...
--                                          11111111
--
--               Thus, when we have seen 240 continuous ZEROes and still going
--               strong, 'RLEspellEnd' would be asserted for one clock cycle,
--               and we would reset our internal counter to 00010000.
--               Ofcourse 'RLErunning' would be high through out the spell.
--
--               We know that the preceeding stage may not have an output
--               on every clock, (due to memory READ/WRITE scheduling)
--               so please let us know on which all clocks we need to run,
--               by asserting 'RLEen'.
--
--
--           The higher block using RLE works (should work) like this:
--
--               if(RLErunning = 1)
--               {
--                  wait till (RLEspellEnd = 1)
--                  collect 'RLEout'.
--               }
--               else  // (RLErunning = 0)
--               {
--                  collect the output of the quantizer.
--               }
```

```
--
--           The enable signal for the next stage is as follows:
--
--              if((RLEspellEnd = 1) or  // output from RLE
--                 (RLErunning = 0))     // output from QUANT
--                   NextStageEnable = 1;
--              else
--                   NextStageEnable = 0;
--              end
--

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

entity RLE is
   port
   (  RLEclk      : in  std_logic;
      RLEreset    : in  std_logic;
      RLEen       : in  std_logic;
      RLEflush    : in  std_logic;
      RLEin       : in  std_logic_vector (15 downto 0);
      RLEzeroth   : in  std_logic_vector (15 downto 0);
      RLEout      : out std_logic_vector ( 7 downto 0);
      RLErunning  : out std_logic;
      RLEspellEnd : out std_logic
   );
end RLE;

architecture structural of RLE is

  signal z1     : std_logic;
  signal z2     : std_logic;
  signal z3     : std_logic;
  signal z4     : std_logic;
  signal z5     : std_logic;
  signal s240   : std_logic;

  signal count  : std_logic_vector ( 7 downto 0) := "00010000";

begin

  run : process(RLEreset, RLEclk)
    begin
      if(RLEreset = '1') then

            count <= "00001111";

            z1 <= '0';
            z2 <= '0';
            z3 <= '0';
            z4 <= '0';
            z5 <= '0';

     elsif(rising_edge(RLEclk)) then

       if(RLEen = '1') then

          if((SIGNED(RLEin) < SIGNED(RLEzeroth)) and
             (SIGNED(RLEin) > SIGNED(-RLEzeroth)) and
             (RLEflush = '0')) then
             z1 <= '1';
          else
             z1 <= '0';
          end if;

          z2 <= z1;
          z3 <= z2;
          z4 <= z3;
          z5 <= z4;

          s240 <= '0'; -- default assignment


          if(z4 = '0') then              -- ZERO spell broken
              count <= "00001111";
          else
              if(count = "11111110") then
                 s240 <= '1';
              end if;

              if(count = "11111111") then
                  count <= "00010000";
              else
                  count <= UNSIGNED(count) + 1;
              end if;
```

```
          end if;

        end if;     -- (RLEen = '1')

      end if;       -- rising_edge(RLEclk)
    end process run;

  RLEout      <= count;
  RLErunning  <= z5;
  RLEspellEnd <= (z5 and not(z4)) or s240;

end structural;
```

B.2.3 huffman.vhd

```
--
--                 WAVELET TRANSFORM IMPLEMENTATION
--                     Stage4 - Huffman Encoder
--
--
-- Author  : sarin@ittc.ukans.edu, 01/30/2000
--
--  Input  : A stream of 8 bit characters on 'in_stream'.
--
--  Output : Huffman tree encoded coefficients, and length.
--
--  Design : Huffman table implementation, takes about 165 CLBs.
--           8 bit input values are variable length (3-18) encoded.
--
--  Revision history:
--
--           v1.0 reversed the bits for HUFFdout, to suit honeywell
--           style of bit packing. This change along with the byte reversal
--           in shifter would do it.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

 entity HUFF is
   port
    (  HUFFclk   : in  std_logic;
       HUFFin    : in  std_logic_vector (7 downto 0);
       HUFFlout  : out std_logic_vector (4 downto 0);
       HUFFdout  : out std_logic_vector (17 downto 0)
    );
   end HUFF;

 architecture structural of HUFF is
   signal tmp : std_logic_vector(7 downto 0);

  begin

   run : process (HUFFclk)
    begin

    if(rising_edge(HUFFclk)) then

     tmp <= HUFFin;

     case tmp is
      when "00000000" => HUFFdout<="111010010XXXXXXXXX"; HUFFlout<="01001";
      when "00000001" => HUFFdout<="0110011XXXXXXXXXXX"; HUFFlout<="00111";
      when "00000010" => HUFFdout<="111000XXXXXXXXXXXX"; HUFFlout<="00110";
      when "00000011" => HUFFdout<="01101XXXXXXXXXXXXX"; HUFFlout<="00101";
      when "00000100" => HUFFdout<="0000XXXXXXXXXXXXXX"; HUFFlout<="00100";
      when "00000101" => HUFFdout<="1101XXXXXXXXXXXXXX"; HUFFlout<="00100";
      when "00000110" => HUFFdout<="100XXXXXXXXXXXXXXX"; HUFFlout<="00011";
      when "00000111" => HUFFdout<="1111XXXXXXXXXXXXXX"; HUFFlout<="00100";
      when "00001000" => HUFFdout<="010XXXXXXXXXXXXXXX"; HUFFlout<="00011";
      when "00001001" => HUFFdout<="001XXXXXXXXXXXXXXX"; HUFFlout<="00011";
      when "00001010" => HUFFdout<="0111XXXXXXXXXXXXXX"; HUFFlout<="00100";
      when "00001011" => HUFFdout<="10101XXXXXXXXXXXXX"; HUFFlout<="00101";
      when "00001100" => HUFFdout<="111011XXXXXXXXXXXX"; HUFFlout<="00110";
      when "00001101" => HUFFdout<="101001XXXXXXXXXXXX"; HUFFlout<="00110";
      when "00001110" => HUFFdout<="0001110XXXXXXXXXX"; HUFFlout<="00111";
      when "00001111" => HUFFdout<="10110111XXXXXXXXXX"; HUFFlout<="01000";
      when "00010000" => HUFFdout<="1100XXXXXXXXXXXXXX"; HUFFlout<="00100";
      when "00010001" => HUFFdout<="10111XXXXXXXXXXXXX"; HUFFlout<="00101";
      when "00010010" => HUFFdout<="00010XXXXXXXXXXXXX"; HUFFlout<="00101";
      when "00010011" => HUFFdout<="101100XXXXXXXXXXXX"; HUFFlout<="00110";
```

```
when "00010100" => HUFFdout<="000110XXXXXXXXXXXX"; HUFFlout<="00110";
when "00010101" => HUFFdout<="1110011XXXXXXXXXXX"; HUFFlout<="00111";
when "00010110" => HUFFdout<="1010000XXXXXXXXXXX"; HUFFlout<="00111";
when "00010111" => HUFFdout<="0110000XXXXXXXXXXX"; HUFFlout<="00111";
when "00011000" => HUFFdout<="11101011XXXXXXXXXX"; HUFFlout<="01000";
when "00011001" => HUFFdout<="10110100XXXXXXXXXX"; HUFFlout<="01000";
when "00011010" => HUFFdout<="01100011XXXXXXXXXX"; HUFFlout<="01000";
when "00011011" => HUFFdout<="00011111XXXXXXXXXX"; HUFFlout<="01000";
when "00011100" => HUFFdout<="111001011XXXXXXXXX"; HUFFlout<="01001";
when "00011101" => HUFFdout<="111010011XXXXXXXXX"; HUFFlout<="01001";
when "00011110" => HUFFdout<="101101101XXXXXXXXX"; HUFFlout<="01001";
when "00011111" => HUFFdout<="111001001XXXXXXXXX"; HUFFlout<="01001";
when "00100000" => HUFFdout<="101000111XXXXXXXXX"; HUFFlout<="01001";
when "00100001" => HUFFdout<="011001011XXXXXXXXX"; HUFFlout<="01001";
when "00100010" => HUFFdout<="011000101XXXXXXXXX"; HUFFlout<="01001";
when "00100011" => HUFFdout<="1110101011XXXXXXXX"; HUFFlout<="01010";
when "00100100" => HUFFdout<="000111101XXXXXXXXX"; HUFFlout<="01001";
when "00100101" => HUFFdout<="1110101001XXXXXXXX"; HUFFlout<="01010";
when "00100110" => HUFFdout<="1011010100XXXXXXXX"; HUFFlout<="01010";
when "00100111" => HUFFdout<="1011011000XXXXXXXX"; HUFFlout<="01010";
when "00101000" => HUFFdout<="1110010100XXXXXXXX"; HUFFlout<="01010";
when "00101001" => HUFFdout<="1011010111XXXXXXXX"; HUFFlout<="01010";
when "00101010" => HUFFdout<="1011010110XXXXXXXX"; HUFFlout<="01010";
when "00101011" => HUFFdout<="1110100000XXXXXXXX"; HUFFlout<="01010";
when "00101100" => HUFFdout<="1010001100XXXXXXXX"; HUFFlout<="01010";
when "00101101" => HUFFdout<="1010001101XXXXXXXX"; HUFFlout<="01010";
when "00101110" => HUFFdout<="1010001010XXXXXXXX"; HUFFlout<="01010";
when "00101111" => HUFFdout<="1110100010XXXXXXXX"; HUFFlout<="01011";
when "00110000" => HUFFdout<="0110010100XXXXXXXX"; HUFFlout<="01010";
when "00110001" => HUFFdout<="1110010001XXXXXXX"; HUFFlout<="01011";
when "00110010" => HUFFdout<="1110010101XXXXXXX"; HUFFlout<="01011";
when "00110011" => HUFFdout<="0110001001XXXXXXX"; HUFFlout<="01010";
when "00110100" => HUFFdout<="0110010001XXXXXXX"; HUFFlout<="01010";
when "00110101" => HUFFdout<="1011010101XXXXXXX"; HUFFlout<="01011";
when "00110110" => HUFFdout<="1110100001XXXXXXX"; HUFFlout<="01011";
when "00110111" => HUFFdout<="1110010000XXXXXXX"; HUFFlout<="01011";
when "00111000" => HUFFdout<="0110010101XXXXXXX"; HUFFlout<="01011";
when "00111001" => HUFFdout<="0110010101XXXXXXX"; HUFFlout<="01011";
when "00111010" => HUFFdout<="1010001000 1XXXXXX"; HUFFlout<="01011";
when "00111011" => HUFFdout<="1110010001 01XXXXX"; HUFFlout<="01100";
when "00111100" => HUFFdout<="1011011001 01XXXXX"; HUFFlout<="01100";
when "00111101" => HUFFdout<="1110100011 10XXXXX"; HUFFlout<="01100";
when "00111110" => HUFFdout<="1110101010 11XXXXX"; HUFFlout<="01100";
when "00111111" => HUFFdout<="1010001011 0XXXXXX"; HUFFlout<="01011";
when "01000000" => HUFFdout<="0110001000 1XXXXXX"; HUFFlout<="01011";
when "01000001" => HUFFdout<="1110101000 00XXXXX"; HUFFlout<="01100";
when "01000010" => HUFFdout<="0110010011 1XXXXXX"; HUFFlout<="01011";
when "01000011" => HUFFdout<="1110010101 10XXXXX"; HUFFlout<="01100";
when "01000100" => HUFFdout<="0110010000 00XXXXX"; HUFFlout<="01100";
when "01000101" => HUFFdout<="1110010101 11XXXXX"; HUFFlout<="01100";
when "01000110" => HUFFdout<="1110100001 11XXXXX"; HUFFlout<="01100";
when "01000111" => HUFFdout<="0110010010 0XXXXXX"; HUFFlout<="01011";
when "01001000" => HUFFdout<="1011011001 000XXXX"; HUFFlout<="01101";
when "01001001" => HUFFdout<="0110010010 11XXXXX"; HUFFlout<="01100";
when "01001010" => HUFFdout<="1011010101 10XXXXX"; HUFFlout<="01100";
when "01001011" => HUFFdout<="1110100011 01XXXXX"; HUFFlout<="01100";
when "01001100" => HUFFdout<="1010001000 01XXXXX"; HUFFlout<="01100";
when "01001101" => HUFFdout<="1110100011 110XXXX"; HUFFlout<="01101";
when "01001110" => HUFFdout<="1110100011 00XXXXX"; HUFFlout<="01100";
when "01001111" => HUFFdout<="1110100001 100XXXX"; HUFFlout<="01101";
when "01010000" => HUFFdout<="1110101000 101XXXX"; HUFFlout<="01101";
when "01010001" => HUFFdout<="0110010011 001XXXX"; HUFFlout<="01101";
when "01010010" => HUFFdout<="0110010010 1001XXX"; HUFFlout<="01110";
when "01010011" => HUFFdout<="0110010011 010XXXX"; HUFFlout<="01101";
when "01010100" => HUFFdout<="1110100010 111XXXX"; HUFFlout<="01101";
when "01010101" => HUFFdout<="1110100011 1110XXX"; HUFFlout<="01110";
when "01010110" => HUFFdout<="1110010001 100XXXX"; HUFFlout<="01101";
when "01010111" => HUFFdout<="1110101000 111XXXX"; HUFFlout<="01101";
when "01011000" => HUFFdout<="1011011001 101XXXX"; HUFFlout<="01101";
when "01011001" => HUFFdout<="1011011001 110XXXX"; HUFFlout<="01101";
when "01011010" => HUFFdout<="1110101010 0100XXX"; HUFFlout<="01110";
when "01011011" => HUFFdout<="1011011001 111XXXX"; HUFFlout<="01101";
when "01011100" => HUFFdout<="1010001001 111XXXX"; HUFFlout<="01101";
when "01011101" => HUFFdout<="0110010010 101XXXX"; HUFFlout<="01101";
when "01011110" => HUFFdout<="1110100010 100XXXX"; HUFFlout<="01101";
when "01011111" => HUFFdout<="0110001000 011XXXX"; HUFFlout<="01101";
when "01100000" => HUFFdout<="1010001011 100XXXX"; HUFFlout<="01101";
when "01100001" => HUFFdout<="1110100001 101XXXX"; HUFFlout<="01101";
when "01100010" => HUFFdout<="1110101010 0000XXX"; HUFFlout<="01101";
when "01100011" => HUFFdout<="1110010001 101XXXX"; HUFFlout<="01101";
when "01100100" => HUFFdout<="1110010001 1100XXX"; HUFFlout<="01110";
when "01100101" => HUFFdout<="1011010101 111XXXX"; HUFFlout<="01101";
when "01100110" => HUFFdout<="1110010001 0000XXX"; HUFFlout<="01110";
when "01100111" => HUFFdout<="1010001011 101XXXX"; HUFFlout<="01101";
when "01101000" => HUFFdout<="1010001001 000XXXX"; HUFFlout<="01101";
when "01101001" => HUFFdout<="1110101010 0110XXX"; HUFFlout<="01110";
when "01101010" => HUFFdout<="1110101010 01011XX"; HUFFlout<="01111";
when "01101011" => HUFFdout<="1110100010 101XXXX"; HUFFlout<="01101";
```

```
when "01101100" => HUFFdout<="11101010100001XXXXX"; HUFFlout<="01101";
when "01101101" => HUFFdout<="10110110010011XXXXX"; HUFFlout<="01110";
when "01101110" => HUFFdout<="10100010011101XXXXX"; HUFFlout<="01101";
when "01101111" => HUFFdout<="11101010101001XXXX"; HUFFlout<="01110";
when "01110000" => HUFFdout<="10100010010101XXXXX"; HUFFlout<="01101";
when "01110001" => HUFFdout<="11101010100111XXXX"; HUFFlout<="01110";
when "01110010" => HUFFdout<="01100010000001XXXXX"; HUFFlout<="01101";
when "01110011" => HUFFdout<="10100010000011XXXX"; HUFFlout<="01110";
when "01110100" => HUFFdout<="11101010001000XXXX"; HUFFlout<="01110";
when "01110101" => HUFFdout<="11100100010001XXXX"; HUFFlout<="01110";
when "01110110" => HUFFdout<="11101010101000XXXX"; HUFFlout<="01110";
when "01110111" => HUFFdout<="10100010010111XXXXX"; HUFFlout<="01101";
when "01111000" => HUFFdout<="01100010000000XXXXX"; HUFFlout<="01101";
when "01111001" => HUFFdout<="11101010000110XXXXX"; HUFFlout<="01101";
when "01111010" => HUFFdout<="01100100011000XXXXX"; HUFFlout<="01101";
when "01111011" => HUFFdout<="11101010001001XXXX"; HUFFlout<="01110";
when "01111100" => HUFFdout<="10110101011110XXXXX"; HUFFlout<="01101";
when "01111101" => HUFFdout<="01100100111011XXXXX"; HUFFlout<="01101";
when "01111110" => HUFFdout<="10100010011100XXXXX"; HUFFlout<="01101";
when "01111111" => HUFFdout<="11100100010011XXXXX"; HUFFlout<="01101";
when "10000000" => HUFFdout<="01100100001010XXXX"; HUFFlout<="01110";
when "10000001" => HUFFdout<="10110110011100XXXXX"; HUFFlout<="01101";
when "10000010" => HUFFdout<="10100010011101OXXX"; HUFFlout<="01111";
when "10000011" => HUFFdout<="11101010101010XXXX"; HUFFlout<="01110";
when "10000100" => HUFFdout<="11101000111110XXX"; HUFFlout<="01111";
when "10000101" => HUFFdout<="11100100011101XXXX"; HUFFlout<="01110";
when "10000110" => HUFFdout<="01100100001011XXXX"; HUFFlout<="01110";
when "10000111" => HUFFdout<="10100010011001XXXXX"; HUFFlout<="01101";
when "10001000" => HUFFdout<="11101000101100XXXX"; HUFFlout<="01110";
when "10001001" => HUFFdout<="11101010100101OOXX"; HUFFlout<="10000";
when "10001010" => HUFFdout<="11101010101011XXXX"; HUFFlout<="01110";
when "10001011" => HUFFdout<="11101000101101XXXX"; HUFFlout<="01110";
when "10001100" => HUFFdout<="10100010000000XXXXX"; HUFFlout<="01101";
when "10001101" => HUFFdout<="1110101000001XXXXXX"; HUFFlout<="01100";
when "10001110" => HUFFdout<="10100010111XXXXXX"; HUFFlout<="01100";
when "10001111" => HUFFdout<="01100010000010XXXXX"; HUFFlout<="01101";
when "10010000" => HUFFdout<="10110110010010XXX"; HUFFlout<="01111";
when "10010001" => HUFFdout<="01100100001000OXXX"; HUFFlout<="01111";
when "10010010" => HUFFdout<="11101010100101011X"; HUFFlout<="10001";
when "10010011" => HUFFdout<="01100100001101000"; HUFFlout<="10010";
when "10010100" => HUFFdout<="01100100001101001"; HUFFlout<="10010";
when "10010101" => HUFFdout<="01100100001000101O"; HUFFlout<="10010";
when "10010110" => HUFFdout<="10100010000010111X"; HUFFlout<="10001";
when "10010111" => HUFFdout<="11100100011110010X"; HUFFlout<="10001";
when "10011000" => HUFFdout<="01100100101000OXXX"; HUFFlout<="01111";
when "10011001" => HUFFdout<="11100100011110011X"; HUFFlout<="10001";
when "10011010" => HUFFdout<="01100100001000100O"; HUFFlout<="10010";
when "10011011" => HUFFdout<="10100010011101OOXX"; HUFFlout<="10000";
when "10011100" => HUFFdout<="11100100011110100X"; HUFFlout<="10001";
when "10011101" => HUFFdout<="11100100011110101X"; HUFFlout<="10001";
when "10011110" => HUFFdout<="01100100001010OOXX"; HUFFlout<="10000";
when "10011111" => HUFFdout<="01100100101000OXX"; HUFFlout<="10000";
when "10100000" => HUFFdout<="01100100001000100I"; HUFFlout<="10010";
when "10100001" => HUFFdout<="01100100001100100"; HUFFlout<="10010";
when "10100010" => HUFFdout<="10100010011101OXX"; HUFFlout<="10000";
when "10100011" => HUFFdout<="01100100001100110I"; HUFFlout<="10010";
when "10100100" => HUFFdout<="01100100101000OXX"; HUFFlout<="10000";
when "10100101" => HUFFdout<="01100100001101010"; HUFFlout<="10010";
when "10100110" => HUFFdout<="01100100001101011"; HUFFlout<="10010";
when "10100111" => HUFFdout<="01100100001101010O"; HUFFlout<="10010";
when "10101000" => HUFFdout<="11100100011111OXX"; HUFFlout<="10000";
when "10101001" => HUFFdout<="01100100000110000XX"; HUFFlout<="10000";
when "10101010" => HUFFdout<="01100100000110010XX"; HUFFlout<="10000";
when "10101011" => HUFFdout<="01100100001101010I"; HUFFlout<="10010";
when "10101100" => HUFFdout<="01100100001100000XXX"; HUFFlout<="01111";
when "10101101" => HUFFdout<="01100100000110100XX"; HUFFlout<="10000";
when "10101110" => HUFFdout<="01100100000101010O0"; HUFFlout<="10010";
when "10101111" => HUFFdout<="10100010000010000XXX"; HUFFlout<="01111";
when "10110000" => HUFFdout<="01100100000110110XX"; HUFFlout<="10000";
when "10110001" => HUFFdout<="01100100000101010I"; HUFFlout<="10010";
when "10110010" => HUFFdout<="01100100001100100XX"; HUFFlout<="10000";
when "10110011" => HUFFdout<="10100010011001100XX"; HUFFlout<="10000";
when "10110100" => HUFFdout<="01100100000101011O"; HUFFlout<="10010";
when "10110101" => HUFFdout<="01100100000101011I"; HUFFlout<="10010";
when "10110110" => HUFFdout<="01100100001101011O"; HUFFlout<="10010";
when "10110111" => HUFFdout<="01100100001101011I"; HUFFlout<="10010";
when "10111000" => HUFFdout<="01100100001000110XX"; HUFFlout<="10000";
when "10111001" => HUFFdout<="01100100001100111O"; HUFFlout<="10010";
when "10111010" => HUFFdout<="10110110010010000XX"; HUFFlout<="10000";
when "10111011" => HUFFdout<="01100100001101111O"; HUFFlout<="10010";
when "10111100" => HUFFdout<="01100100001101111I"; HUFFlout<="10010";
when "10111101" => HUFFdout<="01100100001110100"; HUFFlout<="10010";
when "10111110" => HUFFdout<="01100100001110101"; HUFFlout<="10010";
when "10111111" => HUFFdout<="01100100001110100O"; HUFFlout<="10010";
when "11000000" => HUFFdout<="01100100001100100"; HUFFlout<="10010";
when "11000001" => HUFFdout<="01100100001100101"; HUFFlout<="10010";
when "11000010" => HUFFdout<="01100100001001110"; HUFFlout<="10010";
when "11000011" => HUFFdout<="01100100001001111"; HUFFlout<="10010";
```

```
     when "11000100" => HUFFdout<="011001000001001000"; HUFFlout<="10010";
     when "11000101" => HUFFdout<="011001000001001001"; HUFFlout<="10010";
     when "11000110" => HUFFdout<="011001000001001100"; HUFFlout<="10010";
     when "11000111" => HUFFdout<="011001000001001101"; HUFFlout<="10010";
     when "11001000" => HUFFdout<="0110010000011100XX"; HUFFlout<="10000";
     when "11001001" => HUFFdout<="0110010000011101XX"; HUFFlout<="10000";
     when "11001010" => HUFFdout<="011001000010011010"; HUFFlout<="10010";
     when "11001011" => HUFFdout<="011001000010011011"; HUFFlout<="10010";
     when "11001100" => HUFFdout<="11100100011110110X"; HUFFlout<="10001";
     when "11001101" => HUFFdout<="011001000010011000"; HUFFlout<="10010";
     when "11001110" => HUFFdout<="011001000001011100"; HUFFlout<="10010";
     when "11001111" => HUFFdout<="011001000001011101"; HUFFlout<="10010";
     when "11010000" => HUFFdout<="011001000001001010"; HUFFlout<="10010";
     when "11010001" => HUFFdout<="11100100011110111X"; HUFFlout<="10001";
     when "11010010" => HUFFdout<="11100100011111110XXX"; HUFFlout<="01111";
     when "11010011" => HUFFdout<="011001000001001011"; HUFFlout<="10010";
     when "11010100" => HUFFdout<="011001000010010100"; HUFFlout<="10010";
     when "11010101" => HUFFdout<="011001000001111XXX"; HUFFlout<="01111";
     when "11010110" => HUFFdout<="011001000001000XXX"; HUFFlout<="01111";
     when "11010111" => HUFFdout<="011001000010010101"; HUFFlout<="10010";
     when "11011000" => HUFFdout<="11100100011111110X"; HUFFlout<="10001";
     when "11011001" => HUFFdout<="0110010000111101XX"; HUFFlout<="10000";
     when "11011010" => HUFFdout<="0110010000010100XX"; HUFFlout<="10000";
     when "11011011" => HUFFdout<="101000100111000XXX"; HUFFlout<="01111";
     when "11011100" => HUFFdout<="0110010000111110XX"; HUFFlout<="10000";
     when "11011101" => HUFFdout<="1011011001001001XX"; HUFFlout<="10000";
     when "11011110" => HUFFdout<="101000100000101000"; HUFFlout<="10010";
     when "11011111" => HUFFdout<="101000100000101001"; HUFFlout<="10010";
     when "11100000" => HUFFdout<="0110010000111111XX"; HUFFlout<="10000";
     when "11100001" => HUFFdout<="11100100011111111X"; HUFFlout<="10001";
     when "11100010" => HUFFdout<="101000100000101010"; HUFFlout<="10010";
     when "11100011" => HUFFdout<="101000100000101011"; HUFFlout<="10010";
     when "11100100" => HUFFdout<="11101000111111XXX"; HUFFlout<="01111";
     when "11100101" => HUFFdout<="011001000010010110"; HUFFlout<="10010";
     when "11100110" => HUFFdout<="0110010000010110XX"; HUFFlout<="10000";
     when "11100111" => HUFFdout<="0110010000100111XX"; HUFFlout<="10000";
     when "11101000" => HUFFdout<="011001000010010111"; HUFFlout<="10010";
     when "11101001" => HUFFdout<="011001000011100110"; HUFFlout<="10010";
     when "11101010" => HUFFdout<="011001000011100111"; HUFFlout<="10010";
     when "11101011" => HUFFdout<="011001000011100100"; HUFFlout<="10010";
     when "11101100" => HUFFdout<="0110010000100100XX"; HUFFlout<="10000";
     when "11101101" => HUFFdout<="011001000011100101"; HUFFlout<="10010";
     when "11101110" => HUFFdout<="011001000010011001"; HUFFlout<="10010";
     when "11101111" => HUFFdout<="0110010000111000XX"; HUFFlout<="10000";
     when "11110000" => HUFFdout<="011001000011110010"; HUFFlout<="10010";
     when "11110001" => HUFFdout<="1110010001111000XX"; HUFFlout<="10000";
     when "11110010" => HUFFdout<="011001000011110011"; HUFFlout<="10010";
     when "11110011" => HUFFdout<="101000100000101100"; HUFFlout<="10010";
     when "11110100" => HUFFdout<="101000100000101101"; HUFFlout<="10010";
     when "11110101" => HUFFdout<="011001000011110000"; HUFFlout<="10010";
     when "11110110" => HUFFdout<="011001000011001111"; HUFFlout<="10010";
     when "11110111" => HUFFdout<="011001000011110001"; HUFFlout<="10010";
     when "11111000" => HUFFdout<="011001000001011110"; HUFFlout<="10010";
     when "11111001" => HUFFdout<="10100010011100010XX"; HUFFlout<="10000";
     when "11111010" => HUFFdout<="011001000001011111"; HUFFlout<="10010";
     when "11111011" => HUFFdout<="011001000011011100"; HUFFlout<="10010";
     when "11111100" => HUFFdout<="11101010100101010X"; HUFFlout<="10001";
     when "11111101" => HUFFdout<="011001000011011101"; HUFFlout<="10010";
     when "11111110" => HUFFdout<="011001000010001011"; HUFFlout<="10010";
     when "11111111" => HUFFdout<="000111100XXXXXXXXX"; HUFFlout<="01001";
     when  others    => HUFFdout <="XXXXXXXXXXXXXXXXXX"; HUFFlout <="XXXXX";
    end case;
   end if;
   end process;
 end structural;
```

B.2.4 shifter.vhd

```
--
--                WAVELET TRANSFORM IMPLEMENTATION
--                    Stage4 - Bit packer in Huffman Encoder
--
-- Author  : sarin@ittc.ukans.edu, 04/22/2000
--
--  Input  : A stream of variable length data (length varies between 3 and 18)
--
--  Output : A stream of 32 bit WORDS (packed data), to be written to memory.
--
--  Design : The aim is to pack the variable length data (3->18 bits) into
--           32 bit WORDS. This is done by a 5 ( =ln2(32) ) stage shifter.
--           When ever we have a full load of 32 bits, we do a MEM_WRITE.
--
--           The shifter is inspired by the 'Xtetris' computer game and the
```

```
--           'Binary search' algorithm. In stage 1, (SFTRdatin -> stage1),
--           we either shift (actually rotate) by 16 or pass the data straight.
--           In the next 4 stages we shift by 8, 4, 2, 1 respectively or
--           pass on straight.
--
--           The idea is to shift the incoming variable length data into the
--           correct position over the 5 stages of the shifter.
--
--   #1     #2     #3     #4     #5     #6     #7     #8     #9    #10    #11
-- +-----+------+------+------+------+------+------+------+------+------+------
-- a.....|ba....|c.a...|dc.a..|edc.a.|fedc.a|gfedca|hgfedc|ihgf.c|...gfd|....gd
-- a.....|ba....|c.a...|dc.a..|edc.a.|fedc.a|g.edca|hg.edc|ihg..c|...g.d|....gd
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|ihg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|...g.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|..hg.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|..hg.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|g.e.ca|hg.e.c|.hg.ec|..hg.e|....ge
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|..e.ca|h..e.c|.h..ec|..h..e|.....e
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|..e.ca|h..e.c|.h..ec|..h..e|...h.e
-- a.....|ba....|c.a...|dc.a..|e.c.a.|fe.c.a|..e.ca|h..e.c|.h..ec|..h..e|...h.e
-- a.....|.a....|c.a...|dc.a..|..c.a.|f..c.a|....ca|h....c|.h..ec|..h..e|...h.e
-- a.....|.a....|c.a...|dc.a..|..c.a.|f..c.a|....ca|h....c|.h..ec|..h..e|...h.e
-- a.....|.a....|.ba...|d.ba..|.d.ba.|f.d..a|.f.d.a|h.f.d.|.h.f.d|.ih.f.|...h.f
-- a.....|.a....|.ba...|d.ba..|.d.ba.|..d..a|.f.d.a|..f.d.|...f.d|.ih.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|.ih.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|..h.f.|..ih.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|....f.|..ih.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|....f.|..ih.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|....f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|....f.|...h.f
-- ......|......|.b....|..b...|.d.b..|..d.b.|.f.d.b|..f.d.|...f.d|....f.|.....f
-- ......|......|......|......|.d....|..d.b.|.f.d.b|..f.d.|...f.d|....f.|.....f
-- ......|......|......|......|.d....|..d.b.|.f.d.b|..f.d.|...f.d|....f.|.....f
-- ------+------+------+------+------+------+------+------+------+------+------
--                                         WRITE         WRITE        WRITE
--
--           Things are not that straight forward, as you can see the problem
--           we face in cycle #8. We want to shift forward 'd' to complete the
--           32 bits but we cannot do it for all bits of d. Yeah you guessed
--           it right, part of the last stage is double buffered ...
--
--           [ To better understand the action of double buffering, see the
--             detailed transcript in the simulation file, data.shifter. ]
--
--           Since (input_flow_rate != output_flow_rate) output will not be
--           available on all clock cycles, 'SFTRoutEn' is asserted when ever
--           output is available.
--
--           As you can see, we do not have a serial->parallel and
--           parallel->serial convertor and separate clocks.
--
--           As we know the previous stages may not have data ready in every
--           clock cycle, so if they cut off our 'SFTRen', we would wait
--           till they are ready. The other trick was to AND 'SFTRen' with
--           'SFTRclk', before giving us the 'SFTRclk', but it was causing
--           excessive clock skew ...
--
--  Note:    All signal assingments which gets a latch inferred in synthesis
--           have been annotated with a prop_delay. To get identical
--           simulation and synthesis, define the clock period for
--           simulation as 2*prop_delay.
--
--  Revision history:
--
--           Bug : With RLE ahead, if shifter gets stopped
--                 (because RLE is counting) with SFTRoutEn=1 it keeps on writing.
--           Soln: Soft reset SFTRoutEn to 0 after it has been read once.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SFTR is
   port
   (  SFTRclk    : in  std_logic;
      SFTRen     : in  std_logic;
      SFTRdatin  : in  std_logic_vector (17 downto 0);
```

```
      SFTRlenIn  : in  std_logic_vector ( 4 downto 0);
      SFTRout    : out std_logic_vector (31 downto 0);
      SFTRoutEn  : out std_logic
   );
end SFTR;

architecture structural of SFTR is


  -- A custom comparator!, this comes as part of the double
  -- buffering for the last stage.

  -- We have 17 registers in which we are going to latch
  -- new values. We do not want to latch new values to any
  -- registers above the value in c32. For e.g., with c32=5,
  -- we only want to load up the first 5 registers,
  -- the rest of the 12 registers are ZEROed.

  -- The return value of this function is a mask, which is
  -- ANDed with the inputs to the registers. Thus with c32=5,
  -- the output would look like "11111000000000000".

  -- Phew, was it all worth it?
  -- A simpler way to code this up would be something like:
  --
  --    for i in 16 downto 0 loop
  --      ret(i) := (c32 > (16 - i));
  --    end loop;

  function comparator17(c32: std_logic_vector(4 downto 0))
           return std_logic_vector is
     variable ret : std_logic_vector(16 downto 0);
   begin
     ret(16) := c32(4) or c32(3) or c32(2) or c32(1) or c32(0);
     ret(15) := c32(4) or c32(3) or c32(2) or c32(1);
     ret(14) := c32(4) or c32(3) or c32(2) or (c32(1) and c32(0));
     ret(13) := c32(4) or c32(3) or c32(2);
     ret(12) := c32(4) or c32(3) or (c32(2) and (c32(1) or c32(0)));
     ret(11) := c32(4) or c32(3) or (c32(2) and c32(1));
     ret(10) := c32(4) or c32(3) or (c32(2) and c32(1) and c32(0));
     ret( 9) := c32(4) or c32(3);
     ret( 8) := c32(4) or (c32(3) and (c32(2) or c32(1) or c32(0)));
     ret( 7) := c32(4) or (c32(3) and c32(2)) or (c32(3) and c32(1));
     ret( 6) := c32(4) or (c32(3) and c32(2)) or (c32(3) and c32(1) and c32(0));
     ret( 5) := c32(4) or (c32(3) and c32(2));
     ret( 4) := c32(4) or (c32(3) and c32(2) and (c32(1) or c32(0)));
     ret( 3) := c32(4) or (c32(3) and c32(2) and c32(1));
     ret( 2) := c32(4) or (c32(3) and c32(2) and c32(1) and c32(0));
     ret( 1) := c32(4);
     ret( 0) := (c32(4) and c32(3)) or (c32(4) and (c32(2) or c32(1) or c32(0)));
     return ret;
   end function comparator17;

  constant prop_delay : time := 5 ns;
  subtype std32 is std_logic_vector (31 downto 0);

  signal tmp        : std_logic_vector(5 downto 0):="000000";
  signal stage0_len : std_logic_vector(4 downto 0):="00000";
  signal stage1_len : std_logic_vector(4 downto 0):="00000";
  signal stage2_len : std_logic_vector(4 downto 0):="00000";
  signal stage3_len : std_logic_vector(4 downto 0):="00000";
  signal stage4_len : std_logic_vector(4 downto 0):="00000";

  signal timeout    : std_logic_vector(1 downto 0):="00";

  signal write_ready1 : std_logic := '0';
  signal write_ready2 : std_logic := '0';
  signal write_ready3 : std_logic := '0';
  signal write_ready4 : std_logic := '0';
  signal write_ready5 : std_logic := '0';

  -- 5 register stages, last one is partly double buffered ...

  signal stage1  : std32 :="00000000000000000000000000000000";
  signal stage2  : std32 :="00000000000000000000000000000000";
  signal stage3  : std32 :="00000000000000000000000000000000";
  signal stage4  : std32 :="00000000000000000000000000000000";
  signal stage5  : std32 :="00000000000000000000000000000000";
  signal stage5_d: std_logic_vector(31 downto 15):="00000000000000000";

begin


  -- Catch the overflow!, we have 5 bits in 'SFTRlenIn', keep adding to
  -- 'stage0_len'. When it overflows, we know we crossed 32 bits,
  -- we are ready for a MEM_WRITE.

  tmp            <= ('0' & stage0_len) + ('0' & SFTRlenIn);
```

```
-- A soft rest for SFTRoutEn
-- SFTRoutEn lasts only for 2 cycles.

SFTRoutEn      <= write_ready5 and (timeout(1) or timeout(0));


run : process(SFTRclk)
    variable stage5_tmp : std_logic_vector (31 downto 0);
    variable mask       : std_logic_vector (31 downto 15);
    variable load_db    : std_logic;
 begin

  if(rising_edge(SFTRclk)) then
    if(SFTRen = '1') then

      timeout      <= "11"                 after prop_delay;

      write_ready1 <= tmp(5)               after prop_delay;
      write_ready2 <= write_ready1         after prop_delay;
      write_ready3 <= write_ready2         after prop_delay;
      write_ready4 <= write_ready3         after prop_delay;
      write_ready5 <= write_ready4         after prop_delay;

      stage0_len <= tmp(4 downto 0)        after prop_delay;
      stage1_len <= stage0_len             after prop_delay;
      stage2_len <= stage1_len             after prop_delay;
      stage3_len <= stage2_len             after prop_delay;
      stage4_len <= stage3_len             after prop_delay;

      -- Stage 1 (SFTRdatin -> stage1), shift by 16 or pass thru

      if(stage0_len(4) = '1') then
         stage1(31 downto 30) <= SFTRdatin(1  downto 0) after prop_delay;
         stage1(29 downto 16) <= (others => '0')        after prop_delay;
         stage1(15 downto  0) <= SFTRdatin(17 downto 2) after prop_delay;
      else
         stage1(31 downto 14) <= SFTRdatin              after prop_delay;
         stage1(13 downto  0) <= (others => '0')        after prop_delay;
      end if;


      -- Stage 2 (stage1 -> stage2), shift by 8 or pass thru

      if(stage1_len(3) = '1') then
         stage2(31 downto 24) <= stage1(7  downto 0) after prop_delay;
         stage2(23 downto  0) <= stage1(31 downto 8) after prop_delay;
      else
         stage2 <= stage1                           after prop_delay;
      end if;


      -- Stage 3 (stage2 -> stage3), shift by 4 or pass thru

      if(stage2_len(2) = '1') then
         stage3(31 downto 28) <= stage2(3  downto 0) after prop_delay;
         stage3(27 downto  0) <= stage2(31 downto 4) after prop_delay;
      else
         stage3 <= stage2                           after prop_delay;
      end if;


      -- Stage 4 (stage3 -> stage4), shift by 2 or pass thru

      if(stage3_len(1) = '1') then
         stage4(31 downto 30) <= stage3(1  downto 0) after prop_delay;
         stage4(29 downto  0) <= stage3(31 downto 2) after prop_delay;
      else
         stage4 <= stage3                           after prop_delay;
      end if;


      -- Stage 5 (stage4 -> stage5), shift by 1 or pass thru

      if(stage4_len(0) = '1') then
         stage5_tmp(31)          := stage4(0);
         stage5_tmp(30 downto  0):= stage4(31 downto 1);
      else
         stage5_tmp := stage4;
      end if;

      -- How do we detect a scenario like the one in cycle #8?
      --  ((current_offset > "00000") AND
      --   (prev_offset < "11111") AND
      --   (current_offset < prev_offset)) // i.e, it overflowed

      if( ((stage3_len(4) or stage3_len(3) or stage3_len(2) or
            stage3_len(1) or stage3_len(0)) = '1')
```

```
                          and
            ((stage2_len(4) and stage2_len(3) and stage2_len(2) and
                 stage2_len(1) and stage2_len(0)) = '0')
                          and
            (write_ready4 = '1') ) then
                 load_db := '1';
        else
                 load_db := '0';
        end if;

        mask := comparator17(stage3_len);


        -- If(load_db)
        -- {
        --   double_buffer <= overflow_of_stage5_tmp
        --   stage5        <= stage5 + (stage5_tmp - overflow_of_stage5_tmp)
        -- }
        -- else
        -- {
        --   double_buffer <= 0
        --   if(MEM_WRITE)
        --      stage5     <= stage5_tmp
        --   else
        --      stage5     <= stage5 + stage5_tmp + double_buffer
        -- }

        if(load_db = '1') then
             stage5_d  <= (mask and stage5_tmp(31 downto 15))  after prop_delay;
             stage5    <= ((stage5(31 downto 15) or
                            (not(mask) and stage5_tmp(31 downto 15))) &
                           (stage5(14 downto 0) or stage5_tmp(14 downto 0)))
                                                               after prop_delay;
        else
             stage5_d  <= (others => '0')                      after prop_delay;

             if(write_ready5 = '1') then
                 stage5    <= ((stage5_tmp(31 downto 15) or stage5_d(31 downto 15)) &
                               (stage5_tmp(14 downto 0)))      after prop_delay;
             else
                 stage5    <= ((stage5_tmp(31 downto 15) or
                                stage5(31 downto 15) or
                                stage5_d(31 downto 15)) &
                               (stage5(14 downto 0) or
                                stage5_tmp(14 downto 0)))      after prop_delay;
             end if;
        end if;

      else -- (SFTRen=0)

        timeout(1)   <=  timeout(0)   after prop_delay;
        timeout(0)   <=  '0'          after prop_delay;

      end if;     -- SFTRen

    end if;       -- rising_edge(SFTRclk)
   end process run;



 --SFTRout <= stage5;

   SFTRout( 7 downto  0) <= (stage5(24) & stage5(25) & stage5(26) & stage5(27) &
                             stage5(28) & stage5(29) & stage5(30) & stage5(31));
   SFTRout(15 downto  8) <= (stage5(16) & stage5(17) & stage5(18) & stage5(19) &
                             stage5(20) & stage5(21) & stage5(22) & stage5(23));
   SFTRout(23 downto 16) <= (stage5( 8) & stage5( 9) & stage5(10) & stage5(11) &
                             stage5(12) & stage5(13) & stage5(14) & stage5(15));
   SFTRout(31 downto 24) <= (stage5(0)  & stage5(1)  & stage5(2)  & stage5(3) &
                             stage5(4)  & stage5(5)  & stage5(6)  & stage5(7));
end structural;

configuration SFTR_default of SFTR is
   for structural
   end for;
end SFTR_default;
```

B.2.5 pe1lca.vhd (top level for stage2)

```
--
--              Top level for Wavelet based image compression
--                  - stage 2 (dynamic quantization, RLE, huffman).
--
```

```
--  File  : pe1lca.vhd
--
--  Author: sarin@ittc.ukans.edu, 06/01/2000
--
--  Description:
--
--        Reads coefficients from lower memory (lower 0.5MB),
--        Reads coeff min/max for each blocks from upper memory (upper 0.5MB),
--        Does dynamic quantizing for each block,
--        Does zero thresholding for each block, and RLEs ZEROs,
--        Entropy encodes based on a static Huffman tree,
--        Packs the bit into 32 bit words and
--        Writes it back to upper memory.
--
--        Writes total number of bytes written in upper memory at location 0.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

architecture Memory_Access of PE1_Logic_Core is

    subtype std16 is std_logic_vector (15 downto 0);

    component QUANT is
       port ( QUANTclk   : in  std_logic;
              QUANTen    : in  std_logic;
              QUANTmax   : in  std_logic_vector (15 downto 0);
              QUANTmin   : in  std_logic_vector (15 downto 0);
              QUANTin    : in  std_logic_vector (15 downto 0);
              QUANTout   : out std_logic_vector ( 3 downto 0));
    end component;

    component RLE is
       port ( RLEclk     : in  std_logic;
              RLEreset   : in  std_logic;
              RLEen      : in  std_logic;
              RLEflush   : in  std_logic;
              RLEin      : in  std_logic_vector (15 downto 0);
              RLEzeroth  : in  std_logic_vector (15 downto 0);
              RLEout     : out std_logic_vector ( 7 downto 0);
              RLErunning : out std_logic;
              RLEspellEnd: out std_logic);
    end component;

    component HUFF is
       port ( HUFFclk   : in  std_logic;
              HUFFin    : in  std_logic_vector (7 downto 0);
              HUFFlout  : out std_logic_vector (4 downto 0);
              HUFFdout  : out std_logic_vector (17 downto 0));
    end component;

    component SFTR is
       port ( SFTRclk    : in  std_logic;
              SFTRen     : in  std_logic;
              SFTRdatin  : in  std_logic_vector (17 downto 0);
              SFTRlenIn  : in  std_logic_vector ( 4 downto 0);
              SFTRout    : out std_logic_vector (31 downto 0);
              SFTRoutEn  : out std_logic);
    end component;

    -- We have the problem of (input rate != output rate)
    -- Each memory read brings in 2 coefficients from memory.
    -- when processed each of these coefficients could expand
    -- upto 18 bits, needing 2 memory writes before next read.
    --
    -- ReadBlockData_001:  fire READ
    -- ReadBlockData_010:  optional WRITE
    -- ReadBlockData_100:  32 bit READ arrives, use up upper 16 bits from READ
    -- WriteData        :  optional WRITE, use up lower 16 bits from READ

    type MemoryStates is
      ( WaitforBus,
        ReadBlock1MinMax_001,
        ReadBlock1MinMax_011,
        ReadBlock1MinMax_111,  -- got block1 min/max
        ReadBlock2MinMax_111,  -- got block2 min/max
        ReadBlock3MinMax_111,  -- got block3 min/max
        ReadBlock4MinMax_111,  -- got block4 min/max
        ReadBlock5MinMax_111,  -- got block5 min/max
        ReadBlock6MinMax_110,  -- got block6 min/max
        ReadBlock7MinMax_100,  -- got block7 min/max

        ReadBlockData_001,
        ReadBlockData_010,
        ReadBlockData_100,
```

```
      WriteData,
      WriteDataCount,
      WriteBlock12,
      WriteBlock34,
      WriteBlock56,
      WriteBlock7,
      MemInterrupt,
      MemDone
    );

  signal Mem_PState   : MemoryStates;    -- Present state
  signal Mem_NState   : MemoryStates;    -- Next state


  -- For reading coefficient data from memory,
  -- we have to read blocks 1, 2, 3, 4, 5, 6, 7.
  -- these blocks are interleaved.

  signal ReadCntrROW  : std_logic_vector(8 downto 0); -- ROW, COL address
  signal ReadCntrCOL  : std_logic_vector(7 downto 0); -- registers for READ

  signal eReadCntrROW : std_logic_vector(8 downto 0); -- effective
  signal eReadCntrCOL : std_logic_vector(7 downto 0); --

  signal ROW_limit    : std_logic_vector(8 downto 0);
  signal COL_limit    : std_logic_vector(7 downto 0);

  signal ROW_skip     : std_logic_vector(8 downto 0);
  signal COL_skip     : std_logic_vector(7 downto 0);

  signal ladj         : std_logic_vector(6 downto 0); -- latency adjust


  -- For writing back the output of this stage,
  -- we dump it to the upper memory and increment the
  -- write pointer when ever we write.

  signal WriteCntr    : std_logic_vector(16 downto 0);

  signal RLE_Count1   : std_logic_vector(15 downto 0);
  signal RLE_Count2   : std_logic_vector(15 downto 0);
  signal RLE_Count3   : std_logic_vector(15 downto 0);
  signal RLE_Count4   : std_logic_vector(15 downto 0);
  signal RLE_Count5   : std_logic_vector(15 downto 0);
  signal RLE_Count6   : std_logic_vector(15 downto 0);
  signal RLE_Count7   : std_logic_vector(15 downto 0);

  -- Coefficient data fits in the lower 0.5MB, the upper 0.5 MB is
  -- used for storing block min/max. Before we start using the
  -- upper memory, we have to retrieve these...

  signal Block1Min    : std16;
  signal Block1Max    : std16;
  signal Block2Min    : std16;
  signal Block2Max    : std16;
  signal Block3Min    : std16;
  signal Block3Max    : std16;
  signal Block4Min    : std16;
  signal Block4Max    : std16;
  signal Block5Min    : std16;
  signal Block5Max    : std16;
  signal Block6Min    : std16;
  signal Block6Max    : std16;
  signal Block7Min    : std16;
  signal Block7Max    : std16;


  -- Different blocks are zero thresholded at different levels.
  -- These are design constants. To vary the compression ratio,
  -- need to adjust these.

  -- This is for minimum compression (good quality image)

-- constant Block1Th : std16 := "0000000000000000"; --   0 x 0
-- constant Block2Th : std16 := "0000000000000000"; --  27 x 0
-- constant Block3Th : std16 := "0000000000000000"; --  39 x 0
-- constant Block4Th : std16 := "0000000000000000"; -- 104 x 0
-- constant Block6Th : std16 := "0000000000000000"; --  50 x 0
-- constant Block5Th : std16 := "0000000000000000"; --  79 x 0
-- constant Block7Th : std16 := "0000000000000000"; -- 191 x 0

  -- This is for moderate compression

-- constant Block1Th : std16 := "0000000000000000"; --   0 x 1
-- constant Block2Th : std16 := "0000000000011011"; --  27 x 1
-- constant Block3Th : std16 := "0000000000100111"; --  39 x 1
-- constant Block4Th : std16 := "0000000001101000"; -- 104 x 1
-- constant Block5Th : std16 := "0000000000110010"; --  50 x 1
```

```
-- constant Block6Th : std16 := "0000000001001111"; --  79 x 1
-- constant Block7Th : std16 := "0000000010111111"; -- 191 x 1

   -- This is maximum compression

   constant Block1Th : std16 := "0000000000000000"; --   0 x 2
   constant Block3Th : std16 := "0000000000110110"; --  27 x 2
   constant Block2Th : std16 := "0000000001001110"; --  39 x 2
   constant Block4Th : std16 := "0000000011010000"; -- 104 x 2
   constant Block6Th : std16 := "0000000001100100"; --  50 x 2
   constant Block5Th : std16 := "0000000010011110"; --  79 x 2
   constant Block7Th : std16 := "0000000101111110"; -- 191 x 2


   signal QUANTen      : std_logic;
   signal QUANTmax     : std_logic_vector(15 downto 0);
   signal QUANTmin     : std_logic_vector(15 downto 0);
   signal QUANTin      : std_logic_vector(15 downto 0);
   signal QUANTin2     : std_logic_vector(15 downto 0);
   signal QUANTout     : std_logic_vector( 3 downto 0);
   signal QUANTout2    : std_logic_vector( 3 downto 0);

   signal RLEflush     : std_logic;
   signal RLEen        : std_logic;
   signal RLEin        : std_logic_vector(15 downto 0);
   signal RLEzeroth    : std_logic_vector(15 downto 0);
   signal RLEout       : std_logic_vector( 7 downto 0);
   signal RLErunning   : std_logic;
   signal RLEspellEnd  : std_logic;
   signal RLErunning1  : std_logic;
   signal RLEspellEnd1 : std_logic;
   signal RLErunning2  : std_logic;
   signal RLEspellEnd2 : std_logic;

   signal HUFFin       : std_logic_vector( 7 downto 0);
   signal HUFFlout     : std_logic_vector( 4 downto 0);
   signal HUFFdout     : std_logic_vector(17 downto 0);

   signal SFTRen       : std_logic;
   signal SFTRdatin    : std_logic_vector(17 downto 0);
   signal SFTRlenIn    : std_logic_vector( 4 downto 0);
   signal SFTRout      : std_logic_vector(31 downto 0);
   signal SFTRoutEn    : std_logic;

   signal readComplete : std_logic;

   signal nStages      : std_logic_vector(2 downto 0); -- counts which quadrant we are in
   signal nStages1     : std_logic_vector(2 downto 0); -- delayed by one clock

   signal nStages_1    : std_logic_vector(2 downto 0); -- delayed by  4 cycles
   signal nStages_2    : std_logic_vector(2 downto 0); -- delayed by  8 cycles
   signal nStages_3    : std_logic_vector(2 downto 0); -- delayed by 12 cycles

begin

     quantizer : QUANT           -- Dynamic quantizer
        port map ( PE_Pclk,
                   QUANTen,
                   QUANTmax,
                   QUANTmin,
                   QUANTin,
                   QUANTout);

     rle : RLE
        port map ( PE_Pclk,      -- Run length encoder
                   PE_Reset,
                   RLEen,
                   RLEflush,
                   RLEin,
                   RLEzeroth,
                   RLEout,
                   RLErunning,
                   RLEspellEnd);

     huffman : HUFF
        port map ( PE_Pclk,      -- Huffman encoder
                   HUFFin,
                   HUFFlout,
                   HUFFdout);

     bitpacker : SFTR
        port map ( PE_Pclk,      -- Bit packer
                   SFTRen,
                   SFTRdatin,
                   SFTRlenIn,
                   SFTRout,
                   SFTRoutEn);
```

```
quantizer_in : process(Mem_PState, PE_MemData_InReg, QUANTin2)
 begin
     if (Mem_PState = ReadBlockData_100) then
        QUANTin  <= PE_MemData_InReg(31 downto 16);
     else
        QUANTin  <= QUANTin2;
     end if;
 end process quantizer_in;

RLEin  <= QUANTin;
RLEen  <= QUANTen;


with RLErunning select HUFFin <=     -- Input to huffman:
 RLEout               when '1',     --   from RLE, when RLE
 ("0000" & QUANTout)   when others;  --   from QUANT, else


 SFTRdatin   <= HUFFdout;
 SFTRlenIn   <= HUFFlout;

with nStages1 select QUANTmax <=
  Block1Max when "000",
  Block2Max when "001",
  Block3Max when "010",
  Block4Max when "011",
  Block5Max when "101",
  Block6Max when "110",
  Block7Max when "111",
  (others => 'X')  when others;

with nStages1 select QUANTmin <=
  Block1Min when "000",
  Block2Min when "001",
  Block3Min when "010",
  Block4Min when "011",
  Block5Min when "101",
  Block6Min when "110",
  Block7Min when "111",
  (others => 'X')  when others;

with nStages1 select RLEzeroth <=
  Block1Th when "000",
  Block2Th when "001",
  Block3Th when "010",
  Block4Th when "011",
  Block5Th when "101",
  Block6Th when "110",
  Block7Th when "111",
  (others => '0')  when others;


st_update : process (PE_Pclk, PE_Reset)
 begin

  if (PE_Reset = '1') then
     Mem_PState      <= WaitforBus;

     readComplete    <= '0';
     nStages         <= "000";
     nStages1        <= "000";
     nStages_1       <= "100";
     nStages_2       <= "100";
     nStages_3       <= "100";

     QUANTin2        <= (others => '0');
     QUANTout2       <= (others => '0');

     ReadCntrROW     <= "000000000";
     ReadCntrCOL     <= "00000000";
     eReadCntrROW    <= "000000000";
     eReadCntrCOL    <= "00000000";

     ladj            <= (others => '0');
     RLErunning1     <= '0';
     RLEspellEnd1    <= '0';
     RLErunning2     <= '0';
     RLEspellEnd2    <= '0';

     ROW_limit       <= "111111000"; -- 504  [0, 8, 16, ..., 504] = 64 cells
     COL_limit       <= "11111000";  -- 248  [0, 8, 16, ..., 248] = 32 cells

     ROW_skip        <= "000001000";  -- 8
     COL_skip        <= "00001000";   -- 8

     WriteCntr       <= "0000000000000000";
     RLE_Count1      <= "0000000000000000";
```

```
    RLE_Count2      <= "0000000000000000";
    RLE_Count3      <= "0000000000000000";
    RLE_Count4      <= "0000000000000000";
    RLE_Count5      <= "0000000000000000";
    RLE_Count6      <= "0000000000000000";
    RLE_Count7      <= "0000000000000000";

    Block1Min       <= (others => '0');
    Block1Max       <= (others => '0');
    Block2Min       <= (others => '0');
    Block2Max       <= (others => '0');
    Block3Min       <= (others => '0');
    Block3Max       <= (others => '0');
    Block4Min       <= (others => '0');
    Block4Max       <= (others => '0');
    Block5Min       <= (others => '0');
    Block5Max       <= (others => '0');
    Block6Min       <= (others => '0');
    Block6Max       <= (others => '0');
    Block7Min       <= (others => '0');
    Block7Max       <= (others => '0');

elsif (rising_edge(PE_Pclk)) then

    Mem_PState      <= Mem_NState;
    nStages1        <= nStages;

    RLErunning1     <= RLErunning;
    RLEspellEnd1    <= RLEspellEnd;

    RLErunning2     <= RLErunning1;
    RLEspellEnd2    <= RLEspellEnd1;

    if (Mem_PState = ReadBlock1MinMax_111) then
       Block1Max  <= PE_MemData_InReg(31 downto 16);
       Block1Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock2MinMax_111) then
       Block2Max  <= PE_MemData_InReg(31 downto 16);
       Block2Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock3MinMax_111) then
       Block3Max  <= PE_MemData_InReg(31 downto 16);
       Block3Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock4MinMax_111) then
       Block4Max  <= PE_MemData_InReg(31 downto 16);
       Block4Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock5MinMax_111) then
       Block5Max  <= PE_MemData_InReg(31 downto 16);
       Block5Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock6MinMax_110) then
       Block6Max  <= PE_MemData_InReg(31 downto 16);
       Block6Min  <= PE_MemData_InReg(15 downto 0);
    end if;
    if (Mem_PState = ReadBlock7MinMax_100) then
       Block7Max  <= PE_MemData_InReg(31 downto 16);
       Block7Min  <= PE_MemData_InReg(15 downto 0);
    end if;

    -- Quantizer works on 16 bit data, in each
    -- memory read we get two 16 bit data, so store
    -- one for next cycle.

    if (Mem_PState = ReadBlockData_100) then
       QUANTin2   <= PE_MemData_InReg(15 downto 0);
       QUANTout2  <= QUANTout;                       -- DEBUG
    end if;

    if (Mem_PState = ReadBlockData_001) then
       ladj(6) <= not(readComplete);
       ladj(5) <= ladj(6);
       ladj(4) <= ladj(5);
       ladj(3) <= ladj(4);
       ladj(2) <= ladj(3);
       ladj(1) <= ladj(2);
       ladj(0) <= ladj(1);
    end if;

    if (((Mem_PState = WriteData) or (Mem_PState = ReadBlockData_010)) and
        (SFTRoutEn = '1')) then
       WriteCntr  <=  WriteCntr + 1;
    end if;

    if (((Mem_PState = WriteData) or (Mem_PState = ReadBlockData_010)) and
        ((RLErunning = '0') or (RLEspellEnd = '1'))) then
```

```
   if(nStages_3="000") then
      RLE_Count1  <=  RLE_Count1 + 1;
   end if;
   if(nStages_3="001") then
      RLE_Count2  <=  RLE_Count2 + 1;
   end if;
   if(nStages_3="010") then
      RLE_Count3  <=  RLE_Count3 + 1;
   end if;
   if(nStages_3="011") then
      RLE_Count4  <=  RLE_Count4 + 1;
   end if;

   if(nStages_3="101") then
      RLE_Count5  <=  RLE_Count5 + 1;
   end if;
   if(nStages_3="110") then
      RLE_Count6  <=  RLE_Count6 + 1;
   end if;
   if(nStages_3="111") then
      RLE_Count7  <=  RLE_Count7 + 1;
   end if;

end if;


--     +----+----+---------+
--     | 0  | 2  |         |
--     |    |    |    5    |
--     +----+----|         |
--     | 1  | 3  |         |
--     |    |    |         |
--     +----+----+---------+
--     |         |         |
--     |    4    |    6    |
--     |         |         |
--     |         |         |
--     +---------+---------+

-- ROW/COL address registers for reading.
--
-- Block 0:  ROW start: 00000 0000   COL start: 0000 0000
--               inc  : 00000 1000       inc  : 0000 1000
--               end  : 11111 1000       end  : 1111 1000
-- Block 1:  ROW start: 00000 0100   COL start: 0000 0000
--               inc  : 00000 1000       inc  : 0000 1000
--               end  : 11111 1100       end  : 1111 1000
-- Block 2:  ROW start: 00000 0000   COL start: 0000 0100
--               inc  : 00000 1000       inc  : 0000 1000
--               end  : 11111 1000       end  : 1111 1100
-- Block 3:  ROW start: 00000 0100   COL start: 0000 0100
--               inc  : 00000 1000       inc  : 0000 1000
--               end  : 11111 1100       end  : 1111 1100
-- Block 4:  ROW start: 00000 0010   COL start: 0000 0000
--               inc  : 00000 0100       inc  : 0000 0100
--               end  : 11111 1110       end  : 1111 1100
-- Block 5:  ROW start: 00000 0000   COL start: 0000 0010
--               inc  : 00000 0100       inc  : 0000 0100
--               end  : 11111 1100       end  : 1111 1110
-- Block 6:  ROW start: 00000 0010   COL start: 0000 0010
--               inc  : 00000 0100       inc  : 0000 0100
--               end  : 11111 1110       end  : 1111 1110
--
-- ReadCntrCOL and ReadCntrROW are our main
-- ROW and COL address registers. We also maintain
-- a pair of effective address regs, as in some cases
-- the effective addresses would be normal address +inc/2
--

if(nStages(1) = '1') then
   eReadCntrCOL <= ReadCntrCOL + ('0' & COL_skip(7 downto 1));
else
   eReadCntrCOL <= ReadCntrCOL;
end if;

if(nStages(0) = '1') then
   eReadCntrROW <= ReadCntrROW + ('0' & ROW_skip(8 downto 1));
else
   eReadCntrROW <= ReadCntrROW;
end if;


if (Mem_PState = ReadBlockData_100) then

     nStages_1 <= nStages;
     nStages_2 <= nStages_1;
     nStages_3 <= nStages_2;
```

```
          ReadCntrCOL <= ReadCntrCOL + COL_skip;
          if (ReadCntrCOL = COL_limit) then
              ReadCntrROW <= ReadCntrROW + ROW_skip;
          end if;

          if((ReadCntrROW = ROW_limit) and      -- End of current
             (ReadCntrCOL = COL_limit)) then    -- block

              -- Update nStages as :(000 001 010 011) (101 110 111)
              -- Whenever nStages(0)=1, eRowAddr = RowAddr + RowInc/2
              -- Whenever nStages(1)=1, eColAddr = ColAddr + ColInc/2

              if (nStages = "011") then
                  nStages <= "101";
              elsif (nStages = "111") then
                  nStages <= "100";
              else
                  nStages <= nStages + 1;
              end if;

              if (nStages(1 downto 0) = "11") then
                 ROW_skip   <= ('0' & ROW_skip(8 downto 1));
                 COL_skip   <= ('0' & COL_skip(7 downto 1));

                 ROW_limit  <= ('1' & ROW_limit(8 downto 1));
                 COL_limit  <= ('1' & COL_limit(7 downto 1));
              end if;

              if (nStages = "111") then
                  readComplete <= '1';
              end if;

          end if;
      end if;
    end if;
 end process st_update;


 PE_MemAddr_OutReg(21 downto 18) <= (others => '0');

 mem_state: process(Mem_PState, ladj,
                    PE_MemBusGrant_n,
                    eReadCntrROW, eReadCntrCOL,
                    WriteCntr,
                    nStages, nStages1,
                    RLE_Count1, RLE_Count2, RLE_Count3, RLE_Count4,
                    RLE_Count5, RLE_Count6, RLE_Count7,
                    RLErunning2, RLEspellEnd2,
                    SFTRoutEn, SFTRout,
                    PE_InterruptAck_n)
   begin
        PE_InterruptReq_n   <= '1';  -- Default, do not interrupt host
        PE_MemWriteSel_n    <= '1';  -- read/write, default read
        PE_MemStrobe_n      <= '1';  -- No strobe, later
        PE_MemBusReq_n      <= '0';  -- Always request bus
        QUANTen             <= '0';  --
        SFTRen              <= '0';  --
        RLEflush            <= '0';  --
        PE_MemAddr_OutReg(17 downto 0)  <= (others => '0');
PE_MemData_OutReg(31 downto 0)  <= (others => '0');

  case Mem_PState is

     when WaitforBus =>
        if(PE_MemBusGrant_n = '0') then
           Mem_NState        <= ReadBlock1MinMax_001;
        else
           Mem_NState        <= WaitforBus;
        end if;

     when ReadBlock1MinMax_001 =>
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= ReadBlock1MinMax_011;
        PE_MemAddr_OutReg(17 downto 0) <=  "100000000000001000";

     when ReadBlock1MinMax_011 =>
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= ReadBlock1MinMax_111;
        PE_MemAddr_OutReg(17 downto 0) <=  "100000000000001001";

     when ReadBlock1MinMax_111 =>
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= ReadBlock2MinMax_111;
        PE_MemAddr_OutReg(17 downto 0) <=  "100000000000001010";

     when ReadBlock2MinMax_111 =>
        PE_MemStrobe_n    <= '0';
```

```
   Mem_NState        <= ReadBlock3MinMax_111;
   PE_MemAddr_OutReg(17 downto 0) <=  "100000000000001011";

when ReadBlock3MinMax_111 =>
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= ReadBlock4MinMax_111;
   PE_MemAddr_OutReg(17 downto 0) <=  "100000000000000101";

when ReadBlock4MinMax_111 =>
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= ReadBlock5MinMax_111;
   PE_MemAddr_OutReg(17 downto 0) <=  "100000000000000110";

when ReadBlock5MinMax_111 =>
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= ReadBlock6MinMax_110;
   PE_MemAddr_OutReg(17 downto 0) <=  "100000000000000111";

when ReadBlock6MinMax_110 =>
   Mem_NState        <= ReadBlock7MinMax_100;

when ReadBlock7MinMax_100 =>
   Mem_NState        <= ReadBlockData_001;


--
--   State Mem  QNT QNT RLE RLE HUF SFT SFT   ladj
--         RD   en  out en  out out en  out  6543 210
---------------+--------------+---+--------------------
--   R_001     |              |   |         0000 000
--   R_010     |              |   |         1000 000
--   R_100 a.b | 1       1    |   |         1000 000
--   W         | 1       1    |   |         1000 000
--   R_001     |              |   |         1000 000
--   R_010     |              |   |         1100 000 "ladj" is a shift
--   R_100 c.d | 1       1    |   |         1100 000 reg used to manage
--   W         | 1       1    |   |         1100 000 the latencies...
--   R_001     |              |   |         1100 000
--   R_010     |              |   |         1110 000 ladj is updated in
--   R_100 e.f | 1       1    |   |         1110 000 every 'R_001'.
--   W         | 1   a   1   a |   |         1110 000
--   R_001     |     b       b |   |         1110 000
--   R_010     |     b       b | a | 1       1111 000
--   R_100 g.h | 1   b   1   b | b |         1111 000
--   W         | 1   c   1   c | b | 1       1111 000
--   R_001     |     d       d | b |         1111 000 SHFTen is enabled
--   R_010     |     d       d | c | 1       1111 100 in 'R_010' and 'W'
--   R_100 i.j | 1   d   1   d | d |         1111 100 if ladj(3)=1 OR
--   W         | 1   e   1   e | d | 1       1111 100 ladj(1)=1
--   R_001     |     f       f | d |         1111 100
--   R_010     |     f       f | e | 1       0111 110
--   R_100     | 1   f   1   f | f |         0111 110
--   W         | 1   g   1   g | f | 1   a?  0111 110
--   R_001     |     h       h | f |         0111 110
--   R_010     |     h       h | g | 1   b?  0011 111
--   R_100     | 1   h   1   h | h |         0011 111
--   W         | 1   i   1   i | h | 1   c?  0011 111
--   R_001     |     j       j | h |         0011 111
--   R_010     |     j       j | i | 1   d?  0001 111
--   R_100     | 1   j   1   j | j |         0001 111
--   W         | 1       1     | j | 1   e?  0001 111
--   R_001     |               | j |         0001 111
--   R_010     |               |   | 1   f?  0000 111
--   R_100     | 1       1     |   |         0000 111
--   W         | 1       1     |   | 1   g?  0000 111
--   R_001     |               |   |         0000 111
--   R_010     |               |   | 1   h?  0000 011
--   R_100     | 1       1     |   |         0000 011
--   W         | 1       1     |   | 1   i?  0000 011 From 'W', break
--   R_001     |               |   |         0000 011 the loop
--   R_010     |               |   | 1   j?  0000 001 if ladj(6)=0 and
--   R_100     | 1       1     |   |         0000 001 ladj(1)=0
--   W         | 1       1     |   | 1       0000 001


when ReadBlockData_001 =>
   PE_MemStrobe_n    <= '0';
   Mem_NState        <= ReadBlockData_010;

   PE_MemAddr_OutReg(17)          <= '0';
   PE_MemAddr_OutReg(16 downto 8) <= eReadCntrROW;
   PE_MemAddr_OutReg( 7 downto 0) <= eReadCntrCOL;

when ReadBlockData_010 =>
   Mem_NState        <= ReadBlockData_100;
   PE_MemWriteSel_n  <= '0';  -- for writing
   SFTRen            <= ((ladj(3) or ladj(0)) and
                         (not(RLErunning2) or RLEspellEnd2));
```

```
        PE_MemStrobe_n    <= not(SFTRoutEn);
PE_MemData_OutReg(31 downto 0)  <= SFTRout;
        PE_MemAddr_OutReg(17)          <= '1';
        PE_MemAddr_OutReg(16 downto 0) <= WriteCntr;

     when ReadBlockData_100 =>
        Mem_NState        <= WriteData;
        QUANTen           <= '1';

     when WriteData   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= not(SFTRoutEn);
        QUANTen           <= '1';
        SFTRen            <= ((ladj(3) or ladj(0)) and
                              (not(RLErunning2) or RLEspellEnd2));
        if(nStages /= nStages1) then
           RLEflush       <= '1';
        end if;

        PE_MemAddr_OutReg(17)          <= '1';
        PE_MemAddr_OutReg(16 downto 0) <= WriteCntr;


        if ((ladj(6) = '0') and (ladj(0) = '0')) then
            Mem_NState    <= WriteDataCount;
        else
            Mem_NState    <= ReadBlockData_001;
        end if;

PE_MemData_OutReg(31 downto 0)  <= SFTRout;

     when WriteDataCount   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= WriteBlock12;
        PE_MemData_OutReg(31 downto 17) <= (others => '0');
        PE_MemData_OutReg(16 downto 0)  <= WriteCntr;
        PE_MemAddr_OutReg(17 downto 0)  <= "000000000000000000";

     when WriteBlock12   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= WriteBlock34;
        PE_MemData_OutReg(31 downto 16) <= RLE_Count1;
        PE_MemData_OutReg(15 downto 0)  <= RLE_Count2;
        PE_MemAddr_OutReg(17 downto 0)  <= "000000000000000001";

     when WriteBlock34   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= WriteBlock56;
        PE_MemData_OutReg(31 downto 16) <= RLE_Count3;
        PE_MemData_OutReg(15 downto 0)  <= RLE_Count4;
        PE_MemAddr_OutReg(17 downto 0)  <= "000000000000000010";

     when WriteBlock56   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= WriteBlock7;
        PE_MemData_OutReg(31 downto 16) <= RLE_Count5;
        PE_MemData_OutReg(15 downto 0)  <= RLE_Count6;
        PE_MemAddr_OutReg(17 downto 0)  <= "000000000000000011";

     when WriteBlock7   =>
        PE_MemWriteSel_n  <= '0';  -- for writing
        PE_MemStrobe_n    <= '0';
        Mem_NState        <= MemInterrupt;
        PE_MemData_OutReg(31 downto 16) <= "0000000000000000";
        PE_MemData_OutReg(15 downto 0)  <= RLE_Count7;
        PE_MemAddr_OutReg(17 downto 0)  <= "000000000000000100";


     when MemInterrupt   =>
        PE_MemBusReq_n    <= '1';         -- Give up bus
        PE_InterruptReq_n <= '0';         -- Interrupt host
        if(PE_InterruptAck_n = '0') then
           Mem_NState        <= MemDone;
        else
           Mem_NState        <= MemInterrupt;
        end if;

     when MemDone =>
        PE_MemBusReq_n    <= '1';         -- Give up bus, host program
        Mem_NState        <= MemDone;     -- to READ memory now...
  end case;

  end process mem_state;
```

```
      ------------------------------------------------------------------------
      --  "Inactive" output port signal assignments
      ------------------------------------------------------------------------
       PE_MemHoldReq_n   <= '1';           -- Disable memory hold requests

       PE_Left_OE  <= ( others => '0' );   -- Disable left port output

       PE_Right_OE <= ( others => '0' );   -- Disable right port output

       PE_FifoSelect <= "00";              -- Deselect fifo
                                           --  "00" selects none
                                           --  "01" selects External I/O Fifo
                                           --  "10" selects On-Board Fifo
                                           --  "11" selects On-Board Mailbox
       PE_Fifo_WE_n      <= '1';           -- Disable fifo write mode
       PE_FifoPtrIncr_EN <= '0';           -- Disable fifo pointer increment

end Memory_Access;
```

B.3 Control software - C source code listing

B.3.1 pgm.h

```
/*
 *                PGM image file format APIs
 *
 *  File  : pgm.h
 *
 *  Author: sarin@ittc.ukans.edu, 01/11/2000
 */

 #include<stdio.h>
 #include<stdlib.h>

 // These are the data files.
 #define LENA     "/users/sarin/wildforce/Wavelet/c.given/data/lena.pgm"
 #define GOLDHILL "/users/sarin/wildforce/Wavelet/c.given/data/goldhill.pgm"
 #define BARBARA  "/users/sarin/wildforce/Wavelet/c.given/data/barbara.pgm"

 #define DATA LENA

 #define IMG_SIZE 262146

 int load_PGM_block(char *fname, DWORD *pBuffer)
 {
     FILE *fp;
     int i, j, k;
     char p5[10];
     int num_cols,num_rows,num_bits;
     unsigned char char_buff[IMG_SIZE];
     int a, b;

     fp = fopen(fname,"rb");
     if(fp == NULL)
     {
        fprintf(stderr, "E: pgm.h, fopen(%s) failed\n", fname);
        exit -1;
     }

     /* Skip .pgm image file header */
     fscanf(fp,"%s\n %d %d\n %d\n",p5,&num_rows,&num_cols,&num_bits);
     k=fread(char_buff, 1, IMG_SIZE, fp);
     fclose(fp);
     fprintf(stderr, "N: Read %d bytes from %s\n", k, fname);

     for(i=0,j=0; i<(512*256); j+=2, ++i)
     {
       a = char_buff[j];
       b = char_buff[j+1];

       a = a << 16;

       pBuffer[i] = (a & 0xffff0000) | (0x0000ffff & b);
     }
     return 1;
 }
```

B.3.2 wlt.h

```
/*
 *                WLT file format APIs
 *
 *  File  : wlt.h
 *
 *  Author: sarin@ittc.ukans.edu, 07/10/2000
 */


int write_wlt(DWORD *pBuffer, char * fname)
{
  int i;
  FILE *outFp;

  int rle_size[7];
  int blockminval[7];
  int nBytes;

  outFp  = fopen(fname, "w");
  if(outFp == NULL)
  {
     fprintf(stderr, "E: wlt.h, fopen(%s) failed\n", fname);
     exit(-1);
  }

  nBytes = pBuffer[0]*4;

  rle_size[0] = (pBuffer[1] >> 16);
  rle_size[1] = (pBuffer[1] & 0x0000ffff);
  rle_size[2] = (pBuffer[2] >> 16);
  rle_size[3] = (pBuffer[2] & 0x0000ffff);
  rle_size[4] = (pBuffer[3] >> 16);
  rle_size[5] = (pBuffer[3] & 0x0000ffff);
  rle_size[6] = (pBuffer[4] & 0x0000ffff);

  fprintf(stderr, "N: Writing file %s\n", fname);

  fwrite(blockminval, sizeof(int)  ,7       , outFp);
  fwrite(blockmaxval, sizeof(int)  ,7       , outFp);
  fwrite(rle_size   , sizeof(int)  ,7       , outFp);
  fwrite(&nBytes    , sizeof(int)  ,1       , outFp);
  i=fwrite(pBuffer+131072, sizeof(DWORD),nBytes/4, outFp);

  fprintf(stderr, "N:   rle_size: %d %d %d %d %d %d %d\n",
                  rle_size[0], rle_size[1], rle_size[2], rle_size[3],
                  rle_size[4], rle_size[5], rle_size[6], rle_size[7]);
  fprintf(stderr, "N:   nBytes: %d (%d DWORDs)\n", nBytes, nBytes/4);
  fprintf(stderr, "N: Wrote %d DWORDs to %s\n", i, fname);


  fclose(outFp);
}
```

B.3.3 stage1.c

```
/*
 *        Wavelet Transform - WF4 host program for stage 1
 *
 *  File  : stage1.c
 *
 *  Author: sarin@ittc.ukans.edu, 01/11/2000
 *
 *  Description: - Programs the FPGA with the stage1 bit file,
 *               - Loads a PGM file to PE memory
 *               - Starts up the PE and waits for it to interrupt the
 *                 host on completion
 *               - On being interrupted, reads back the PE memory,
 *                 and dumps the output in "image.st1".
 */

 #include <stdio.h>
 #include <signal.h>
 #include <time.h>
 #include "wf4errs.h"
 #include "wf4api.h"
 #include "pgm.h"
 #include "utils.h"

 #define BIT_FILE "/users/sarin/wildforce/Wavelet/vhdl.stage1/pe1.bin"
```

```
#define MEM_IN_DWORDS  262144

#define OUT_FILE "image.st1"

WF4_BoardNum board = 0;
DWORD *pBuffer;

#define CHECK_ERROR(fn, ret) \
 if(ret != WF4_SUCCESS) \
 { \
   fprintf(stderr,"E: mem.c, %s()=>%d, %s\n",fn,ret, \
                   WF4_ErrorString(rc)); \
   WF4_DmaFree(board, pBuffer ); \
   WF4_Close(board); \
   return -1; \
 }

void printbin(DWORD i)
{
  int j;
  for(j=31; j>=0; --j)
  {
     if(i & ( 1 << j))
       fprintf(stderr, "1");
     else
       fprintf(stderr,"0");
  }
}

int main()
{
  int i, j;
  DWORD  penum = 1, timeout=10000, count;
  double time1, time2;
  FILE *outFp;
  WF4_RetCode rc;

  rc= WF4_Open(board, WF4_INIT_FIFO_FLAG | WF4_INIT_CLK_FLAG |
                      WF4_INIT_PE_EVENTS_FLAG);

  rc= WF4_ClkInitLocal( board, FALSE, 24.0f );
  CHECK_ERROR("WF4_ClkInitLocal", rc);

  rc= WF4_PeResetInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeResetInterrupts", rc);

  rc= WF4_PeDisableInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeDisableInterrupts", rc);

  rc= WF4_PeProgram(board, WF4_PE(penum), BIT_FILE);
  CHECK_ERROR("WF4_PeProgram", rc);
  fprintf(stderr, "N: PE programmed with %s\n", BIT_FILE);

  rc= WF4_PeEnableInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeDisableInterrupts", rc);

  pBuffer = (DWORD *)WF4_DmaAllocate(board, MEM_IN_DWORDS);
  if(pBuffer == NULL)
  {
    fprintf(stderr, "E: stage1.c , WF4_DmaAllocate() failed\n");
    WF4_Close(board);
    return;
  }

  load_PGM_block(DATA, pBuffer);

  outFp = fopen(OUT_FILE, "w");
  if(outFp ==NULL)
  {
     fprintf(stderr, "E: stage1.c, fopen(%s) failed\n", OUT_FILE);
     exit(-1);
  }

  rc= WF4_MemBlockPe(board, WF4_PE(penum), TRUE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  time1=timer();
  rc = WF4_DmaMemWrite( board, penum, 0, MEM_IN_DWORDS/2, pBuffer, &count, timeout);
  CHECK_ERROR("WF4_DmaMemWrite", rc);
  time2=timer();
  rc= WF4_MemBlockPe(board, WF4_PE(penum), FALSE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  fprintf(stderr, "N: Embedded memory written to in %f ms\n", (time2-time1)*1000);

  time1=timer();
  WF4_ClkSuspend(board,FALSE); // resume clock
  WF4_PeWaitInterrupt(board, penum, 10000);
  WF4_ClkSuspend(board,TRUE); // suspend clock
  time2=timer();
```

```
  fprintf(stderr, "N: PE ran for %f ms\n", (time2-time1)*1000);

  rc= WF4_MemBlockPe( board, WF4_PE(penum), TRUE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  time1=timer();
  rc= WF4_DmaMemRead( board, penum, 0, MEM_IN_DWORDS, pBuffer, &count, timeout);
  CHECK_ERROR("WF4_DmaMemRead", rc);
  time2=timer();
  rc= WF4_MemBlockPe( board, WF4_PE(penum), FALSE);
  CHECK_ERROR( "WF4_MemBlockPe", rc);
  fprintf(stderr, "N: Embedded memory read in %f ms\n", (time2-time1)*1000);

  for(i=0; i<0; ++i)
  {
    fprintf(stderr,"\n%3d:",2*i); printbin(pBuffer[i]);
  }
  fprintf(stderr,"\n");

  i=fwrite(pBuffer, 4, MEM_IN_DWORDS, outFp);
  fprintf(stderr, "N: Wrote %d DWORDs %s\n", i, OUT_FILE);

  fclose(outFp);

  WF4_DmaFree(board, pBuffer);
  WF4_Close( board );
  fprintf(stderr, "N: Board closed\n");
}
```

B.3.4 stage2.c

```
/*
 *             Wavelet Transform, WF4 host program for stage 2
 *
 *  File  : stage2.c
 *
 *  Author: sarin@ittc.ukans.edu, 06/11/2000
 *
 *  Description: - Programs the FPGA with the stage 2 bit file,
 *               - Loads the memory dump from stage1
 *               - Starts up the PE and waits for it to interrupt the
 *                 host on completion
 *               - On being interrupted, reads back the PE memory,
 *                 and dumps the output in "image.st2".
 */

 #include <stdio.h>
 #include <signal.h>
 #include <time.h>
 #include "wf4errs.h"
 #include "wf4api.h"
 #include "utils.h"
 #include "wlt.h"

   #define BIT_FILE "/users/sarin/wildforce/Wavelet/vhdl.stage2/pe1.bin.0"
// #define BIT_FILE "/users/sarin/wildforce/Wavelet/vhdl.stage2/pe1.bin.128"
// #define BIT_FILE "/users/sarin/wildforce/Wavelet/vhdl.stage2/pe1.bin.256"

 #define MEM_IN_DWORDS  262144

 #define STAGE1_OUT_FILE "image.st1"
 #define STAGE2_OUT_FILE "image.st2"

 WF4_BoardNum board = 0;
 DWORD *pBuffer;

 #define CHECK_ERROR(fn, ret) \
  if(ret != WF4_SUCCESS) \
  { \
    fprintf(stderr,"E: mem.c, %s()=>%d, %s\n",fn,ret, \
                    WF4_ErrorString(rc)); \
    WF4_DmaFree(board, pBuffer ); \
    WF4_Close(board); \
    return -1; \
  }

void printbin(DWORD i)
{
  int j;
  for(j=31; j>=0; --j)
  {
     if(i & ( 1 << j))
       fprintf(stderr, "1");
     else
```

```
      fprintf(stderr,"0");
  }
}

int main()
{
  int i, j, tmp;
  DWORD  penum = 1, count, timeout=5000;
  double time1, time2;
  char a, b;

  FILE *outFp, *inFp;
  WF4_RetCode rc;

  rc= WF4_Open(board, WF4_INIT_FIFO_FLAG | WF4_INIT_CLK_FLAG |
                      WF4_INIT_PE_EVENTS_FLAG);

  rc= WF4_ClkInitLocal( board, FALSE, 8.0f );
  CHECK_ERROR("WF4_ClkInitLocal", rc);

  rc= WF4_PeResetInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeResetInterrupts", rc);

  rc= WF4_PeDisableInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeDisableInterrupts", rc);

  rc= WF4_PeProgram(board, WF4_PE(penum), BIT_FILE);
  CHECK_ERROR("WF4_PeProgram", rc);
  fprintf(stderr, "N: PE programmed with %s\n", BIT_FILE);

  rc= WF4_PeEnableInterrupts(board, WF4_PE(penum));
  CHECK_ERROR("WF4_PeDisableInterrupts", rc);

  pBuffer = (DWORD *)WF4_DmaAllocate(board, MEM_IN_DWORDS);
  if(pBuffer == NULL)
  {
    fprintf(stderr, "E: stage2.c , WF4_DmaAllocate() failed\n");
    WF4_Close(board);
    return;
  }

  inFp  = fopen(STAGE1_OUT_FILE, "r");
  outFp = fopen(STAGE2_OUT_FILE, "w");

  if((inFp == NULL) || (outFp ==NULL))
  {
     fprintf(stderr, "E: fpga_QRH.c, fopen(%s)\n", STAGE1_OUT_FILE);
     exit(-1);
  }
  i=fread(pBuffer, 4, MEM_IN_DWORDS, inFp);
  fprintf(stderr, "N: Read %d DWORDs from %s\n", i, STAGE1_OUT_FILE);

  rc= WF4_MemBlockPe(board, WF4_PE(penum), TRUE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  time1=timer();
  rc = WF4_DmaMemWrite( board, penum, 0, MEM_IN_DWORDS, pBuffer, &count, timeout);
  CHECK_ERROR("WF4_DmaMemWrite", rc);
  time2=timer();
  rc= WF4_MemBlockPe(board, WF4_PE(penum), FALSE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  fprintf(stderr, "N: Embedded memory written to in %f ms\n", (time2-time1)*1000);

  time1=timer();
  WF4_ClkSuspend(board,FALSE); // resume clock
  WF4_PeWaitInterrupt(board, penum, 10000);
  WF4_ClkSuspend(board,TRUE); // suspend clock
  time2=timer();
  fprintf(stderr, "N: PE ran for %f ms\n", (time2-time1)*1000);

  rc= WF4_MemBlockPe( board, WF4_PE(penum), TRUE);
  CHECK_ERROR("WF4_MemBlockPe", rc);
  time1=timer();
  rc= WF4_DmaMemRead( board, penum, 0, MEM_IN_DWORDS, pBuffer, &count, timeout);
  CHECK_ERROR("WF4_DmaMemRead", rc);
  time2=timer();
  rc= WF4_MemBlockPe( board, WF4_PE(penum), FALSE);
  CHECK_ERROR( "WF4_MemBlockPe", rc);
  fprintf(stderr, "N: Embedded memory read in %f ms\n", (time2-time1)*1000);

  for(i=0; i<6; i+=2)
  {
    fprintf(stderr,"\n%7d:",i); printbin(pBuffer[i]);
    fprintf(stderr," "); printbin(pBuffer[i+1]);
  }
  fprintf(stderr,"\n");

  i = 131072;
  i=fwrite(pBuffer+i, 4, i, outFp);
```

```
  fprintf(stderr, "N: Wrote %d DWORDs to %s\n", i, STAGE2_OUT_FILE);
  fclose(outFp);

  write_wlt(pBuffer, "lena.wlt");

  WF4_DmaFree(board, pBuffer);
  WF4_Close( board );
}
```

Bibliography

[BRIAN] Brian Schoner, John Villasenor, Steve Molloy and, Rajeev Jain, *Techniques for FPGA Implementation of Video Compression Systems*, ACM/SIGBA International Symposium on Field-Programmable Gate Arrays, 1995.

[CALDERBANK] R. Calderbank and I. Daubechies and W. Sweldens and B.-L. Yeo, *Losless Image Compression using Integer to Integer Wavelet Transforms*, International Conference on Image Processing (ICIP), Vol. I, 1997.

[COHEN] A. Cohen, I. Daubechies, J. Feauveau, *Biorthogonal Bases of Compactly Supported Wavelets*, Communications of Pure Applied Math, vol 45, 1992.

[GEOFF] Geoff Davis, *Wavelet Image Compression Construction Kit Version 0.3 (1/29/97)*, http://www.cs.dartmouth.edu/ gdavis/wavelet/wavelet.html.

[SHA] Sarin Mathen, *Secure Hashing Implementation on FPGA*, ITTC Technical Report.

[STRANG] Gilbert Strang, *Wavelets and Dialation Equations: A Brief Introduction*, SIAM Review, vol 31, no. 4, December 1989, pp. 614-627.

[WILDFORCE] Annapolis Micro Systems Inc., *Wildforce Reference Mannual, 1999, revision 3.4*.

[XC4000] Xilinx 4000 series FPGAs, *The Programmable Logic Data Book*, 1996.