

# JPEG2000 Wavelet Transform on StarCore™-Based DSPs

Freescale Semiconductor's StarCore™-based DSPs have many features to perform image compression algorithms quickly and efficiently. This application note presents an example of a wavelet transform to highlight these features. It describes in detail how to implement one discrete wavelet transform that is included in the JPEG2000 image compression standard. One of the strengths of JPEG2000 is that it offers a large range of compression features with a unified framework. Among the capabilities of the JPEG2000 compression standard are:

- Carry out lossy compression at the same performance level as JPEG.
- Handle bi-level still images.
- Provide high resolution processing, that is, more than 8-bits per pixel.
- Perform lossless compression within the same unified framework.
- Provide progression by resolution from lossy images, including lossless compression.
- Allow region of interest (ROI) compression and decompression.
- Provide error resilience.

## CONTENTS

<b>1</b>	Wavelet Transform .....	2
<b>1.1</b>	Theory .....	3
<b>2</b>	Implementation on StarCore-Based DSPs .....	7
<b>2.1</b>	Lifting .....	8
<b>2.2</b>	Integer Lifting .....	10
<b>2.3</b>	StarCore Implementation in C Code .....	10
<b>2.3.1</b>	Format of a Tile .....	11
<b>2.3.2</b>	Decomposition .....	12
<b>2.3.3</b>	Memory Organization .....	12
<b>2.4</b>	StarCore Implementation in Assembly Code .....	14
<b>2.4.1</b>	Code Structure and Features .....	14
<b>2.4.2</b>	Memory Use .....	14
<b>2.4.3</b>	Parallel Features .....	16
<b>2.5</b>	StarCore Implementation in Optimized Assembler .....	17
<b>3</b>	Summary .....	17
<b>4</b>	References .....	17

## APPENDIXES:

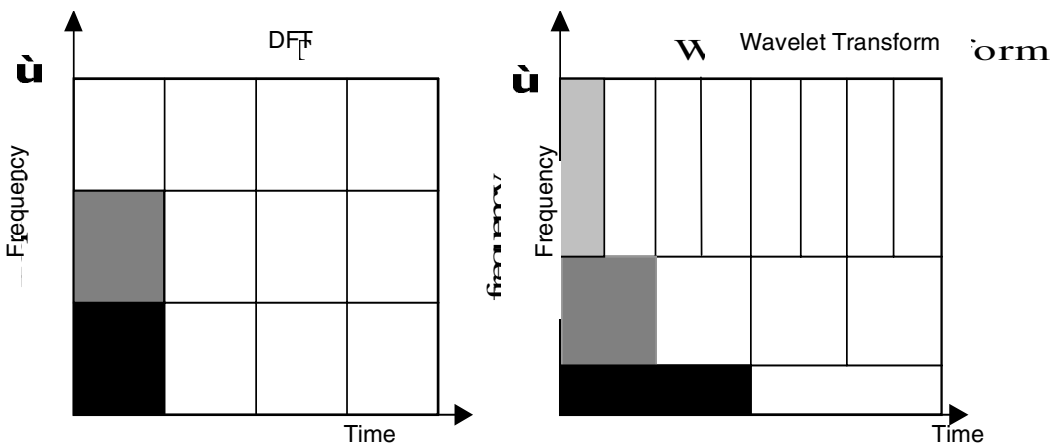
<b>Appendix A</b>	Discrete Wavelet Transform C Code .....	19
<b>Appendix B</b>	Discrete Wavelet Transform - Assembly Code .....	30
<b>Appendix C</b>	Discrete Wavelet Transform Optimized Assembly Code .....	37

# 1 Wavelet Transform

Wavelet transformation does not lead to compression itself but converts the data into a more suitable form for later compression by other components of the standard. Data transformation in image compression serves three primary purposes:

- *Decorrelate the image components.* An appropriate transform can decorrelate the image components. The information in the spatial domain is represented more efficiently in the transform domain because if the coefficients are totally decorrelated, there is no duplication of information between the coefficients. However, in practice, complete decorrelation of an input signal can be achieved only if it is a Gaussian source.
- *Quantize the image data within the transform domain.* The transform domain is usually better for quantization because the quantization errors can be distributed to improve perception by the human visual system (HVS).
- *Take advantage of fast algorithms* to perform key transforms. The availability of these fast algorithms has led to the widespread use of certain transforms such as the discrete Fourier transform (DFT) and the discrete cosine transform (DCT). Here, the term fast is in comparison carrying out a DFT without the use of the fast Fourier transform (FFT).

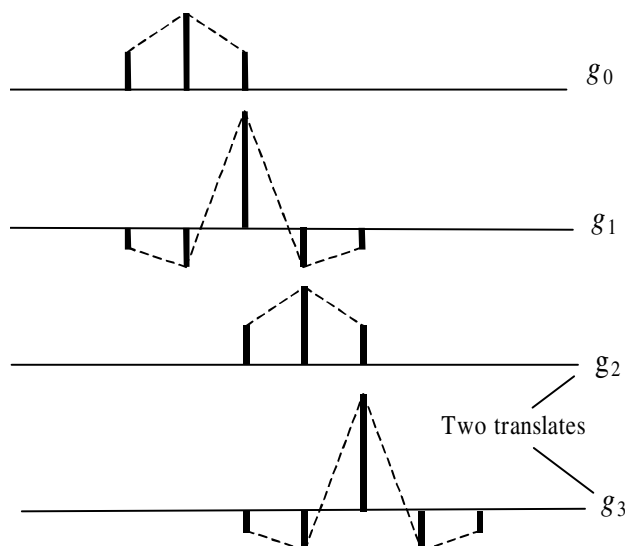
A typical image tends to have a predominance of low frequency components. Therefore, it is useful to have a detailed analysis in the low frequency band and a coarse analysis in the higher frequency bands. This requirement explains the interest in the wavelet transform. While the DFT and DCT split the frequency spectrum into even bands of frequencies during analysis, the wavelet transform splits the spectrum so that it provides good frequency resolution at low frequencies, with associated poor time resolution, and coarse frequency resolution at high frequencies, with associated fine time resolution. In other words, a large bandwidth in the frequency domain is coupled with a short window in the time domain and *vice versa*. Conversely, the windowed DFT and DCT have a fixed-length window in the time domain, which ensures that the bandwidth in the frequency domain is also fixed. **Figure 1** illustrates these different properties for both the DFT and a typical wavelet transform.



**Figure 1.** Time-Frequency Resolution of DFT and Wavelet Transform

The DFT and DCT do not allow for non-stationary signal properties. A stationary signal is one where the statistical properties of the signal remain constant over time, that is, the mean and variance of the signal do not change. The wavelet transformation has an advantage over both the DFT and DCT transforms because it does allow for non-stationary signal properties.

In JPEG, the DCT is implemented as a block-based transform to get around the fact that an image is usually not stationary. An advantage of the wavelet transform over the DCT is that, because it allows for the signal to be non-stationary, it does not have to be implemented as a block-based transform. Usually a transformation is performed because each transformed coefficient can be processed independently. A disadvantage of using the DCT for image processing is that the image must be decomposed into small contiguous blocks that are each transformed separately. Although the DCT does a fair job of decorrelating the coefficients within a single block, this is not true across different blocks because the transformation on one block is carried out independently of all other blocks. For a DCT implemented as a block transform of eight, every eight samples in the input signal are independently transformed into a new set of eight coefficients. However, a wavelet transform can be carried out across a whole image, and the decorrelation that it achieves between coefficients is extended throughout the image because, apart from a simple case called a Haar transform, all other wavelet transforms are lapped transforms. **Figure 2** shows the basis vectors for the (5,3) wavelet transform that is discussed in this application note.



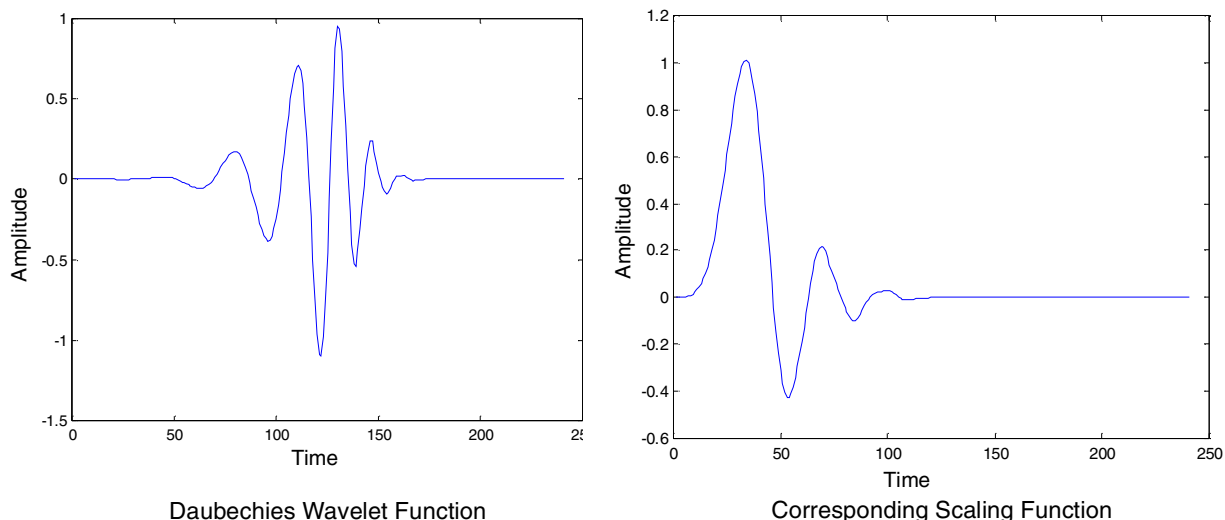
**Figure 2.** Lapped Transform Nature of the (5,3) Wavelet Transform

The translates of the primary basis vectors ( $g_0$  and  $g_1$ ) that form the next two basis vectors ( $g_2$  and  $g_3$ ) overlap the primary basis vectors. Therefore, every group of transformed coefficients has an area of overlap with the original image samples, which helps to prevent blocking artifacts. Another advantage of the wavelet transform over the DCT in image processing is that it naturally affords multi-resolution processing.

## 1.1 Theory

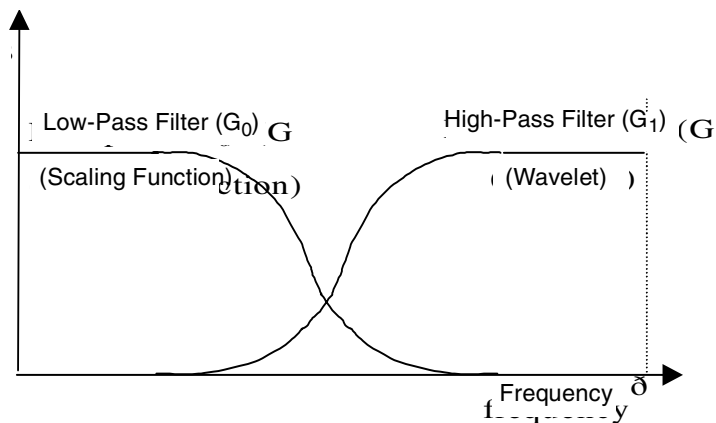
**Figure 3** shows an example of a wavelet, which is oscillatory and has an average value of zero that decays to zero after a finite period. The Fourier transform uses a series of different frequency sine and cosine waves as its basis functions. In contrast, the wavelet transform uses a series of wavelets as its basis functions. However, this collection of wavelets is derived by the dilation and translation of a single primary wavelet, as shown in **Figure 3**, which is based on one of the Daubechies series of wavelets. Therefore, unlike the Fourier transform with only one possible set of basis functions, (that is, sine and cosine), there is an infinite number of possible wavelet transforms because the basis functions can be derived from different primary wavelets. However, wavelets must have the following properties to serve as a basis for a wavelet transform:

- Be absolutely integrable and square integrable.
- Be composed only of positive frequency components.
- Have a DC component of zero.

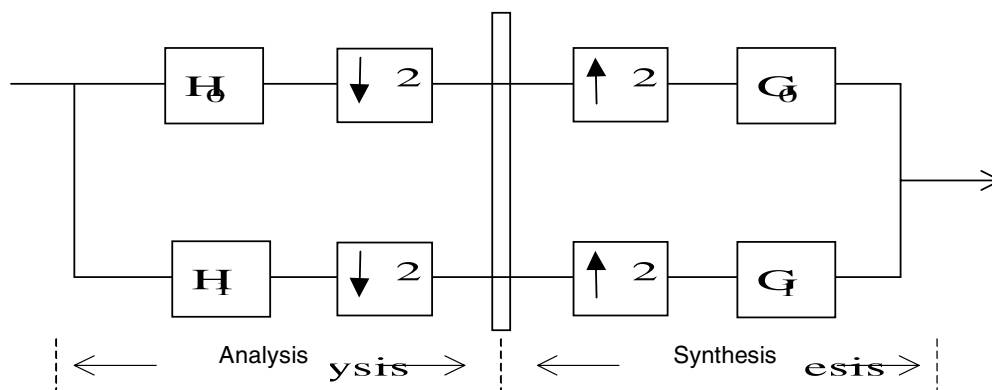


**Figure 3.** Daubechies Wavelet Function and the Corresponding Scaling Function

The wavelet transform is a form of subband coding and can be implemented using filter banks [4] [5]. We assume that there are only two filters in both the synthesis and analysis phases of the transformation. These filters take the form of a low-pass and a high-pass, which are quadrature mirror filters (QMF) with respect to one another, that is, they have frequency responses of the form shown in **Figure 4**. These filters can be used to analyze a signal and then resynthesize it as illustrated in **Figure 5**. Examination of a wavelet in the frequency domain reveals that it has the form of a band-pass filter. For the low-pass part of the QMF to be constructed, a wavelet cannot be used. Instead a so-called scaling function is used as the low-pass filter (see **Figure 4**). Indeed, one method to produce a wavelet is to develop a low-pass filter of the form of  $G_0$  by defining its frequency response. This filter can then be used as the basis from which to produce the other filters. In **Figure 5**,  $H_0$  and  $G_0$  are the low-pass filters, and  $H_1$  and  $G_1$  are the high-pass filters. The low-pass filters gives rise to the scaling functions and the high-pass filters give rise to the wavelet functions. If certain rules are followed [6], this method produces orthogonal wavelets with perfect reconstruction properties.



**Figure 4.** Quadrature Mirror Filters Used in Wavelet Transformation



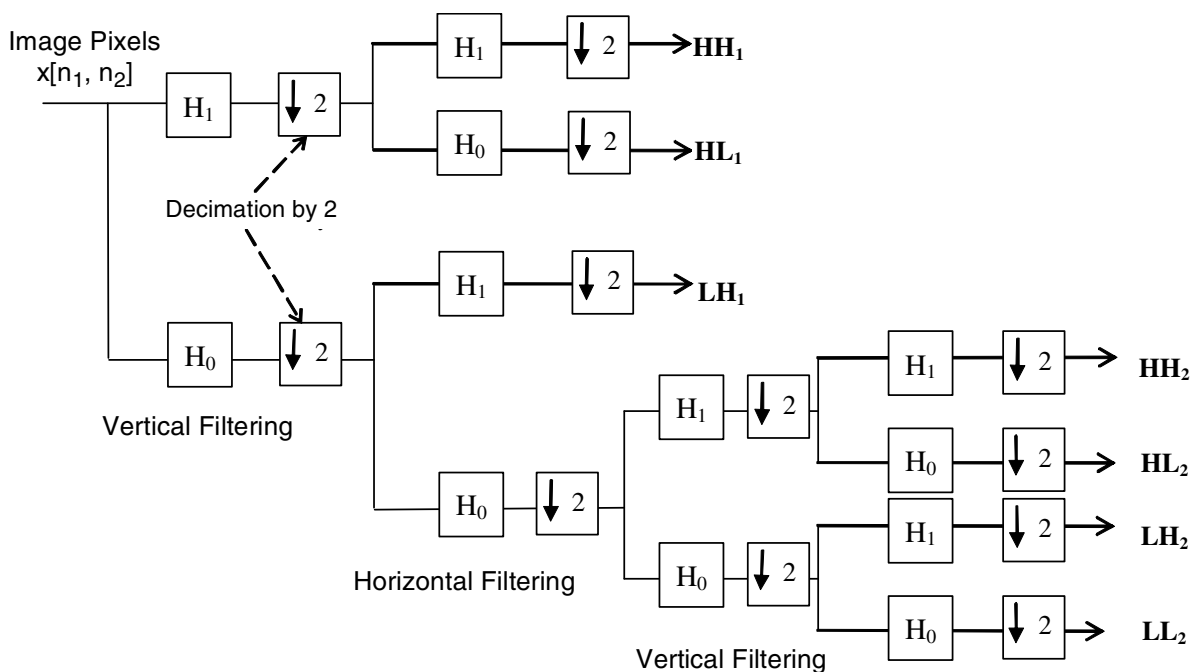
**Figure 5.** Two-Channel Subband Analysis and Synthesis Banks

In image processing it is important to have zero phase filters so that if the image is shifted the frequencies maintain their relative phase in the transform domain. If relative phase is not maintained, distortion occurs particularly to the areas of the image with sharp changes in frequency, such as edges or lines. This distortion is accentuated with color images where non-zero phase filtering can shift different colors by different amounts relative to each other. It is difficult to achieve zero phase using the design method described here, where all the synthesis and analysis filters are derived from the same low-pass synthesis filter. Therefore, bi-orthogonal wavelets are very popular in image processing because in these cases, the design method uses two wavelets instead of one. One wavelet is used for decomposition and a different one is used for reconstruction, making it easier to achieve zero-phase filtering. The condition of bi-orthogonality ensures that the forward transform is invertible. This is one of the reasons why bi-orthogonal wavelets have been selected for the JPEG2000 standard. In particular, the (5,3) bi-orthogonal wavelet is described in this application note.

Consider an example in which there are only two separable wavelet filters, one for low-pass filtering and one for high-pass filtering. The filters are separable because the horizontal rows in a two-dimensional image can be filtered separately from the vertical columns. The image can be reduced to a low resolution image (the scaling functions) and a succession of detail images (the wavelets) by repeatedly applying the filters in both the horizontal and vertical directions. **Figure 6** shows this process for a two-level decomposition. Notice how only the low-pass results from any particular level are further decomposed. As in **Figure 5**, the high-pass analysis filter is  $H_1$  and the low-pass analysis filter is  $H_0$ . Also note that **Figure 6** follows the JPEG2000 convention that vertical filtering is followed by horizontal filtering.

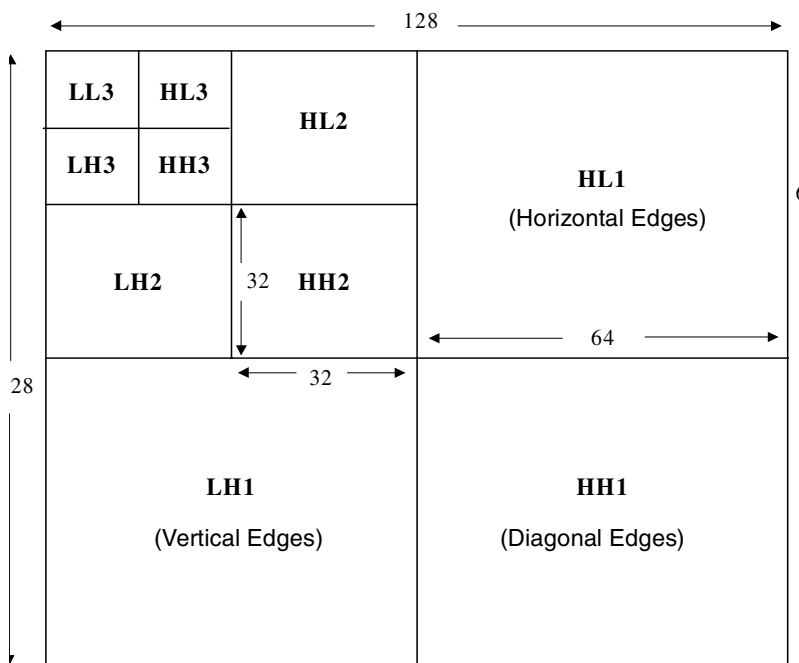
Our example employs the so-called tree structured decomposition using filter banks, in two dimensions, which Mallat first described [4]. Decimation occurs after each filtering process, which reduces the number of coefficients. Therefore, it does not matter how many levels of decomposition have taken place; the final number of coefficients equals the original number of pixels in the image. The recomposition, or decoder, is the inverse of this decomposition except that  $G_0$  replaces  $H_0$  and  $G_1$  replaces  $H_1$  and the decimators are replaced by interpolators.

**Figure 7** shows a three-level decomposition of an image. The HH1 band is formed by filtering the image once in both the horizontal and vertical directions with the high-pass analysis filter, whereas the LH1 band is formed by low-pass filtering horizontally followed by high-pass filtering vertically. Two-level decomposition is denoted as  $xx2$  (where  $x$  can be either an L or an H) and is formed by taking the LL1 band and repeating the horizontal and vertical filtering with the appropriate filters. This operation can be repeated to decompose the low-pass portion of the image further until, in theory, only 1 pixel remains in the lowest level of decomposition. These different resolution images can be used, through linear combination, to reconstruct the original image.



**Figure 6.** Two-Level Wavelet Decomposition in Two Dimensions

In fact, each subband represents the whole image even though they are different sizes. Therefore, if an error occurs in one subband, for example, through quantization, it is not detrimental because the different image representations in the other subbands help to remove the error.



**Figure 7.** Subband Passbands for Three-Level Wavelet Analysis

**Figure 8** and **Figure 9** compare the results of using JPEG2000 (with the (5,3) wavelet transform) and baseline JPEG (which uses the DCT) on a portion of the ‘Barbara’ image. Here, the image has been compressed by approximately the same amount for both JPEG and JPEG2000—that is, about 0.38-bits per pixel. To obtain these pictures, a quantization step size of 0.125 was used for JPEG2000 whereas the JPEG quality setting was twelve. It is stressed that the compression used to get these pictures is quite severe in comparison to the level of compression an image would normally be subjected to. The intention is to highlight the differences between the two techniques. The JPEG compressed image in **Figure 8** clearly shows blocking artifacts. The JPEG2000 image in **Figure 9**, on the other hand, shows only an overall degraded image rather than specific blocking effects. For details on wavelets and subband coding theory, consult the two introductory texts listed in the references [5] and [11].



**Figure 8.** Compression with JPEG –Quality 12



**Figure 9.** Compression with JPEG2000 –Step 0.125

## 2 Implementation on StarCore-Based DSPs

Wavelets are either orthogonal or non-orthogonal. For example, Daubechies’ wavelets [6] are orthogonal while Mallat’s [4] are bi-orthogonal. The Le Gall (5,3) bi-orthogonal wavelet [7] was chosen as the transform to use in all the experiments undertaken for this application note. This particular wavelet was chosen because it is one of the transforms in the JPEG2000 specification. In addition, it is relatively simple to implement this wavelet because it is so short. The implementation described here is based on the JPEG2000 standard [8] in Annex F, section F3.6, the so-called 1D-SD procedure, (which means 1-dimensional sub-band decomposition). This implementation is for a wavelet transform that produces lossless compression and uses reversible 1D filtering. Lossless means that the integer lifting technique is used to carry out the transformation [9].

For an image to be filtered, its boundaries must be extended by an amount that is dictated by the length of the filter. In this case, symmetric extension is performed so that the extra pixels are the mirror image of the actual image pixels in both horizontal and vertical directions. This implementation uses tiling to split the image into separate tiles of size  $128 \times 128$  pixels. Each tile is then transformed independently of any other tile. If the tile sizes become too small, blocking artifacts becomes a problem. However, a relatively large tile size,  $64 \times 64$  and above, should not suffer significantly from blocking artifacts. Ease of implementation outweighs blocking considerations. The wavelet transform is still a lapped transform and are therefore non-stationary within the tile boundaries [2]. This implementation was tested using the Metrowerks<sup>®</sup> CodeWarrior<sup>®</sup> software development tools.

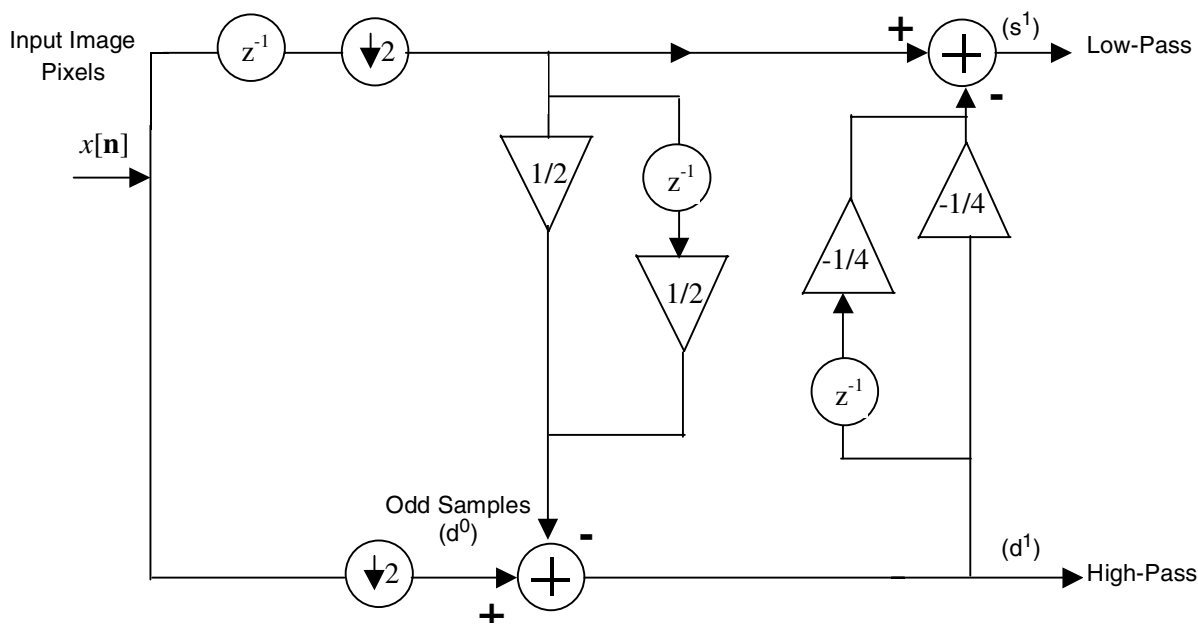
## 2.1 Lifting

Integer lifting is explained using the Le Gall (5,3) wavelet as an example. The coefficients for the analysis filters are listed in **Table 1**, where  $h_0$  is the low-pass filter and  $h_1$  is the high-pass filter.

**Table 1.** Le Gall (5,3) Analysis Filter Coefficients

Index	$h_0$	$h_1$
0	$3/4$	1
1	$1/4$	$-1/2$
2	$-1/8$	0

The conventional method for performing a wavelet transform is to do a pyramidal decomposition of the input data, as shown in **Figure 6**. However, for the lifting method, the low-pass and high-pass filter paths are not completely independent in that the decomposition is now performed as shown in **Figure 10** for a one-dimensional case. As illustrated, the input data is split into even and odd samples, the so-called lazy transform. Notice that the resulting samples are superscripted with a zero to denote that they are still at level zero of the decomposition tree. Similarly, the output samples from each of the low-pass and high-pass streams are denoted with a superscript of 1 to indicate one level of decomposition.



**Figure 10.** Dependence of Low-Pass and High-Pass Filter Paths for (5,3) Filter



After the ‘lazy’ transform, filtering takes place as shown and this process can be described by **Equation 1** and **Equation 2**.

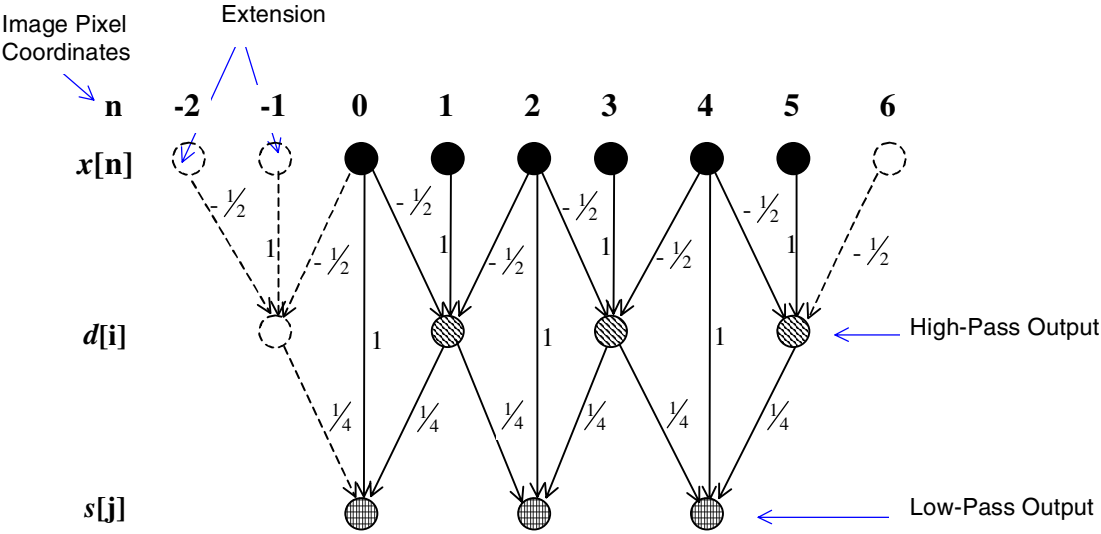
**Equation 1**

$$\text{Low-Pass: } s_n^1 = s_n^0 + \frac{1}{4}(d_{n-1}^1 + d_n^1)$$

**Equation 2**

$$\text{High-Pass: } d_n^1 = d_n^0 - \frac{1}{2}(s_n^0 + s_{n+1}^0)$$

Where,  $n$  is the sample index for the separate even and odd sample vectors. The  $s$  denotes sum and represents the low-pass output. The  $d$  denotes difference and represents the high-pass output. **Figure 11** illustrates the process for the (5,3) filter with a view to implementation. As in JPEG2000, odd symmetrical extension is used when the pixels at  $n$  equal  $-1$  and  $-2$ , and have the values for the pixels at  $n$  equal  $+1$  and  $+2$  respectively. Similarly, the value for  $n$  equals  $6$  is the same as the pixel at  $n$  equals  $4$ , in this example. In the diagram, the values for the filter multipliers are placed next to the appropriate path.



**Figure 11.** Lifting Method for the (5,3) Wavelet Filter

The lifting technique does allow for the even samples to become the low-pass output and the odd samples to be the high-pass output. However, we store the two output streams separately. The relation between the filter coefficients in **Table 1** and the multipliers shown in **Figure 11** is explained with reference to **Equation 3** and **Equation 4**.

**Equation 3**

$$d[0] = -\frac{1}{2}x[0] + x[1] - \frac{1}{2}x[2]$$

**Equation 4**

$$d[1] = -\frac{1}{2}x[2] + x[3] - \frac{1}{2}x[4]$$

The coefficients of these two high-pass filtering operations are simply the high-pass filter coefficients. **Equation 5** shows how the low-pass filtering operation is undertaken.

**Equation 5**

$$\begin{aligned}
 s[1] &= x[2] + \frac{1}{4}d[0] + \frac{1}{4}d[1] \\
 &= -\frac{1}{8}x[0] + \frac{1}{4}x[1] + x[2] - \frac{1}{4}x[2] + \frac{1}{4}x[3] - \frac{1}{8}x[4] \\
 &= -\frac{1}{8}x[0] + \frac{1}{4}x[1] + \frac{3}{4}x[2] + \frac{1}{4}x[3] - \frac{1}{8}x[4]
 \end{aligned}$$

As expressed in the last line of this equation, there is a one-to-one correspondence between the low-pass coefficients and the coefficients of the  $h_0$  filter, as required. Therefore, in the analysis phase, the lifting technique has a direct correspondence with convolution using the filter coefficients given in **Table 1** because the input vector,  $x[n]$ , is the same for both the lifting and conventional convolution methods. However, this is not the case for synthesis, which is not discussed in this document.

## 2.2 Integer Lifting

To produce integer arithmetic, **Equation 1** and **Equation 2** are rewritten as follows:

**Equation 6**

$$d_n^1 = d_n^0 - \left\lfloor \frac{1}{2}(s_n^0 + s_{n+1}^0) \right\rfloor$$

**Equation 7**

$$s_n^1 = s_n^0 + \left\lfloor \frac{1}{4}(d_{n-1}^1 + d_n^1) + \frac{1}{2} \right\rfloor$$

To maintain integer arithmetic, truncation must take place where division operations are performed on the integer numbers, as shown by the  $\lfloor \rfloor$  operators, which denote rounding down. In **Equation 7**, one-half is added to avoid bias because multiplication by one-fourth has occurred, as in **Equation 6**. Even though this rounding operation is non-linear, lifting allows reversibility because the inverse is found by reversing the order and changing the signs of the forward transform [9]:

**Equation 8**

$$d_n^0 = d_n^1 + \left\lfloor \frac{1}{2}(s_n^1 + s_{n+1}^1) \right\rfloor$$

**Equation 9**

$$s_n^0 = s_n^1 - \left\lfloor \frac{1}{4}(d_{n-1}^1 + d_n^1) + \frac{1}{2} \right\rfloor$$

Information is not lost in this process because the sum and difference of two integers are either both even or both odd, so the least significant bit is the same in both cases. The last bit of the sum can be omitted because the information is maintained by the difference value. Thus, in the inverse transform of **Equation 8** and **Equation 9**, the information that was apparently lost in the forward transform is replaced by the sum and difference operations, respectively.

## 2.3 StarCore Implementation in C Code

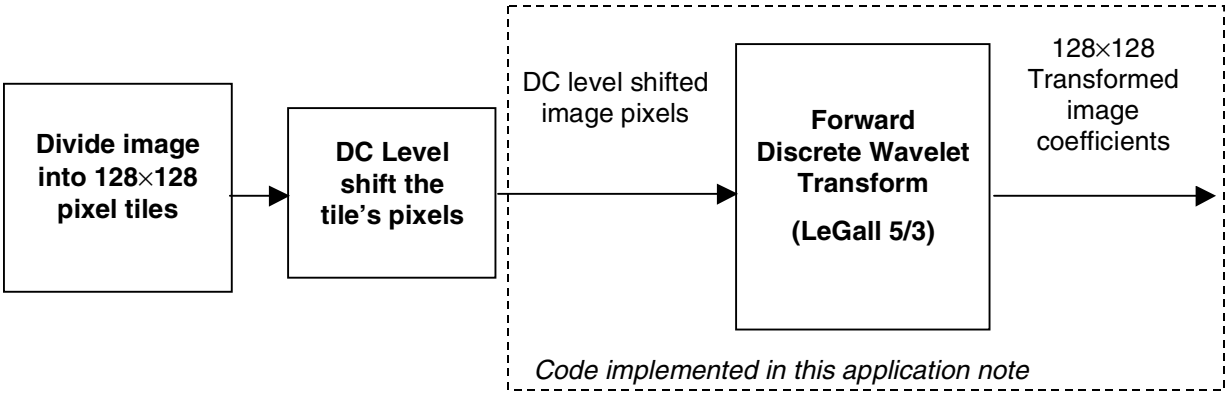
**Appendix A** provides the C code for implementation on a StarCore DSP. This section describes the design of this implementation and the actual variables used in the C program. Following are conventions for naming variables:

- Constants start with a letter  $k$ .
- Global variables start with a letter  $g$ .

Input parameters to the wavelet transform are:

- A pointer to an area of memory containing a tile of image pixels.
- The number of wavelet decomposition levels required.

The result of the wavelet transform is a set of subbands corresponding to the number of decomposition levels described by Figure 7 on page 6. **Figure 12** shows the interface to the wavelet transform code block.

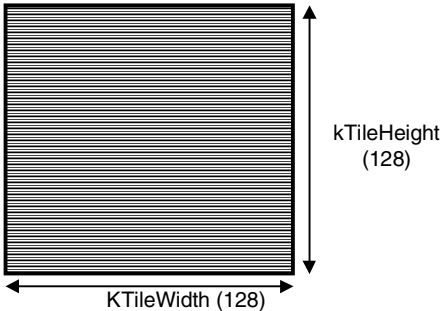


**Figure 12.** Interface to Wavelet Transform Code Block

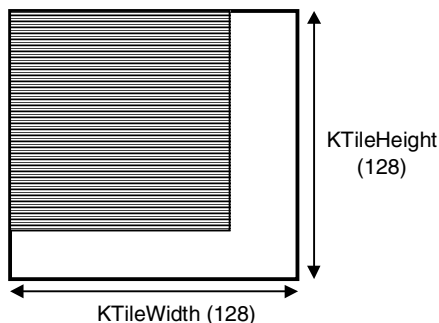
The code in this application note assumes that the image is already divided into  $128 \times 128$  pixel tiles and that each pixel is DC level shifted. We assume that each pixel is 12-bits long so that DC level shifting ensures that the input to the wavelet transform is in the range  $-2047$  to  $2048$ . As noted earlier, the transform itself does not lead to compression that transforms a tile of  $128 \times 128$  pixels into  $128 \times 128$  coefficients. Instead, the transform puts the data into a form that is better for compression by other components of the JPEG2000 standard. The number of bits in the pixel does not significantly affect the speed of operation in the wavelet transform.

### 2.3.1 Format of a Tile

In this implementation all images are split into tiles which are  $128 \times 128$  pixels. In the code, these dimensions are assigned the constant names *kTileHeight* (the rows of an image tile) and *kTileWidth* (the columns of an image tile). A tile is stored in memory as an array of coefficients,  $kTileWidth \times kTileHeight$  in length. Each tile is allocated the full array for storage. Sometimes an image is not an exact multiple of tiles in size, which results in tiles at the edges of the image that are smaller than  $128 \times 128$  pixels. Even in these instances, the program allocates the size of a full tile array. Consequently, a tile row always starts at offsets 0, *kTileWidth*,  $2 \times kTileWidth$ , and so on. At the end of each row there is undefined data for the smaller than normal tiles. Similarly, a tile that is smaller in height than normal contains undefined data in its lower rows, as illustrated in **Figure 14**. The program is written so that this undefined data is not processed.



**Figure 13.** Data Packing in Full Tile



**Figure 14.** Data Packing in Partially Full Tile

### 2.3.2 Decomposition

For each decomposition level, the data passes through the low-pass and high-pass (5,3) filters twice. The decimation by two is incorporated into the filtering action, as shown in **Figure 11**. The data is first filtered vertically into low-pass (L) and high-pass (H) subbands. The resulting subbands are then filtered horizontally into low-low (LL), low-high (LH), high-low (HL) and high-high (HH) subbands. Each of these final four subbands is one-fourth of the size of the input data. For smaller than normal tiles, the low band is the only band that is the same size as the normal tile’s subbands. The first decomposition level processes the original tile. Second and later decomposition levels reprocess the LL band, recursively. Therefore, a three-level decomposition on a  $128 \times 128$  tile would produce the subbands with the sizes shown in **Table 2**.

**Table 2.** Sizes of Subbands After Three-Level Decomposition

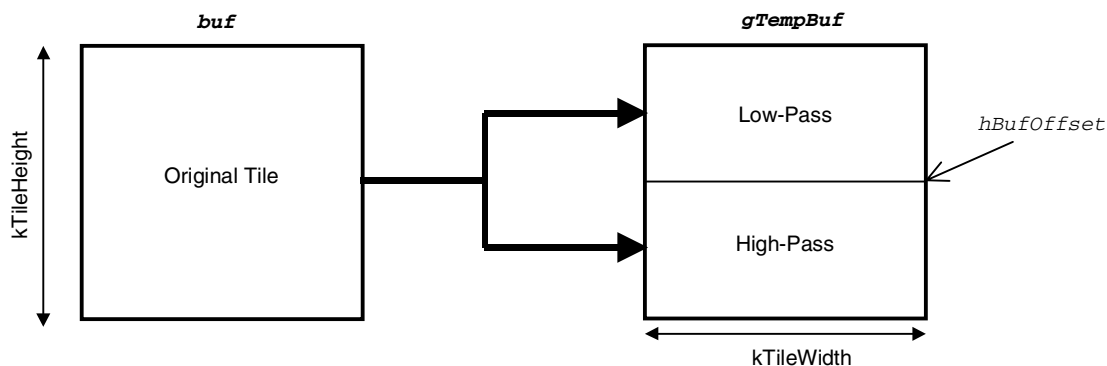
Decomposition Level	Subband	Size
1	LH1	$64 \times 64$
	HL1	
	HH1	
2	LH2	$32 \times 32$
	HL2	
	HH2	
3	LH3	$16 \times 16$
	HL3	
	HH3	
	LL3	

In JPEG2000, the subbands are transmitted with the lowest decomposition level first (three-level in **Table 2**). Notice that because of the decimation the number of output coefficients is the same as the number of input coefficients.

### 2.3.3 Memory Organization

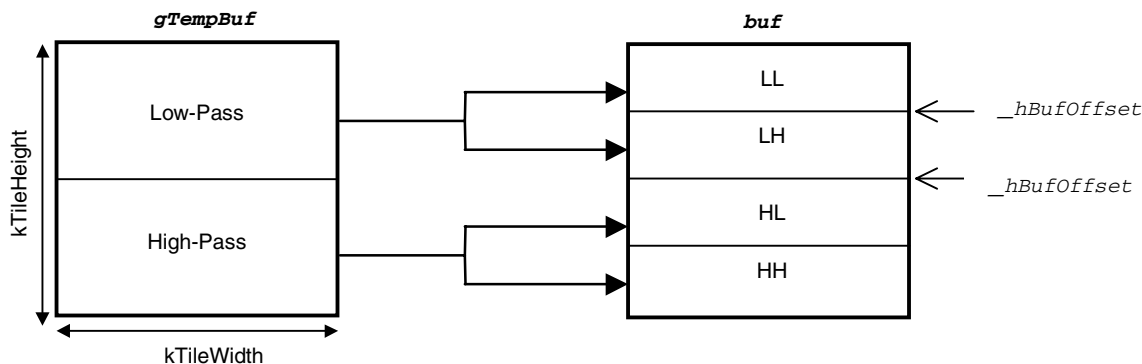
To minimize memory access and use, our implementation of the JPEG2000 algorithm performs the wavelet transform using two memory buffers the size of a normal tile. One of these buffers is the original input tile buffer, and `buf` in the code points to it. `gTempBuf` points to the other buffer. We can perform the transform using only the input buffer, but this makes the code less readable.

During vertical filtering, data is copied from the input tile buffer to the temporary tile buffer, and each column is high-pass and low-pass filtered separately. This vertically filtered data is organized so that the top half of the temporary buffer contains the low-pass data and the bottom half contains the high-pass data, as shown in **Figure 15**. The pointer, `hBufOffset`, is used to locate the high-pass vertically filtered data. The first  $kTileHeight/2$  rows are low-pass data. The bottom  $kTileHeight/2$  rows are high-pass data. Each buffer is 128 pixels wide by 64 pixels high. If there is insufficient data to fill the original buffer, the two temporary buffers are under-filled by a similar amount and aligned to the top and left. The division between the two buffers remains fixed.



**Figure 15.** Vertical Filtering of the Image Data

The two blocks of vertically filtered low-pass and high-pass data are now filtered separately in the horizontal direction. The results of these filtering operations are placed back into the original memory buffer, `buf`, where they overwrite the original image data as shown in **Figure 16**. The result of this operation is that `buf` now contains the first level of decomposition of the wavelet transform. As shown in **Figure 16**, `buf` is now divided into four buffers, each containing  $64 \times 64$  coefficients.



**Figure 16.** Horizontal Filtering of the Image Data

The process of vertical and horizontal filtering is repeated recursively with only the LL subband buffer. Again, if the input data does not completely fill the input tile, the filtered data fills each output subband buffer by a proportional amount. Subband boundaries are fixed for a tile size of  $128 \times 128$  pixels. **Table 3** shows offsets from the start of the buffer for the different levels of decomposition.

**Table 3.** Offsets Into Different Subband Buffers

Subband	Size	Offset	
LL3	$16 \times 16$	0	0
LH3		256	$1 \times 2^8$

**Table 3.** Offsets Into Different Subband Buffers (Continued)

HL3		512	$2 \times 2^8$
Subband	Size	Offset	
HH3		768	$3 \times 2^8$
LH2	$32 \times 32$	1024	$1 \times 2^{10}$
HL2		2048	$2 \times 2^{10}$
HH2		3072	$3 \times 2^{10}$
LH1	$64 \times 64$	4096	$1 \times 2^{12}$
HL1		8192	$2 \times 2^{12}$
HH1		12288	$3 \times 2^{12}$

## 2.4 StarCore Implementation in Assembly Code

This section considers how specific StarCore features help to implement the wavelet transform. Assembly code for the core part of the (5,3) wavelet transform is given in **Appendix B** for one-level decomposition. Each image pixel is assumed to be 8-bits long.

### 2.4.1 Code Structure and Features

The example code in **Appendix B** includes three nested loops called VERT\_LOOP, HORIZ\_LOOP and LEV\_LOOP. The HORIZ\_LOOP filters a single row of data while VERT\_LOOP moves through every row in the image. These two loops execute the low-pass and high-pass filtering in one direction—that is, either horizontally or vertically. The HORIZ\_LOOP constitutes the one-dimensional wavelet transform kernel and takes ten instruction cycles. The loop, LEV\_LOOP, exists to carry out one complete level of decomposition by ensuring that the whole process is executed, as shown in **Figure 17**. The horizontal and vertical filtering of both high-pass and low-pass outputs from the original image data are completed by this third outermost loop.

### 2.4.2 Memory Use

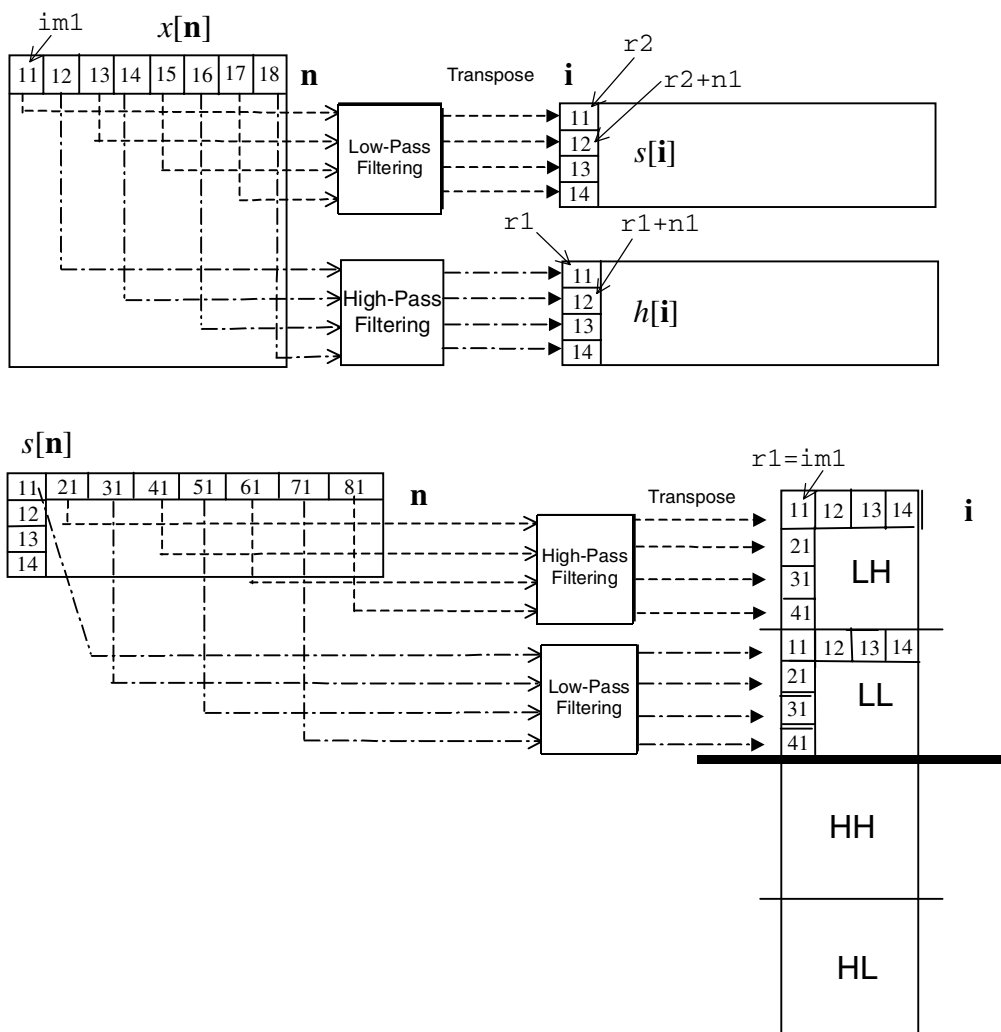
The discrete wavelet transform, high-pass filtering may cause an increase of up to 1-bit in the length of the output coefficients during each level of decomposition. (In particular, this occurs when there is a sequence of alternating maximum and minimum values). For example, the input of an 8-bit pixel can produce the output of up to 11-bits in length with three levels of decomposition on the high-pass path. The low-pass filtering does not cause any increase in coefficient length. This high-pass phenomenon has led to the use of word arithmetic rather than byte arithmetic to allow for possible growth in coefficient length. Therefore, in the example given in assembly code, the image pixels are assumed to be loaded into memory as words rather than bytes. Therefore, this assembly code can be used to process images with pixels up to 13-bits long, which is useful because many camera manufacturers offer 12-bit pixel length in their products.

In this example, the horizontal filtering is carried out before the vertical filtering. A key feature of this assembly code version of the transform is the use of register offsets so that the output from the horizontal low-pass and high-pass filters is stored in transposed form. Subsequent vertical low-pass and high-pass filtering is carried out with the same assembly code as the horizontal filtering, as illustrated in **Figure 17**. In this figure, a block of  $8 \times 8$  image data undergoes one level of decomposition. There are two main stages to the process:

- *Horizontal filtering.* In this stage, the image data is high-pass and low-pass filtered before being stored into two separate areas of memory. In the assembler, the output areas are pointed to by **r1** for the high-pass coefficients, and **r2** for the low-pass coefficients. Because the filtering operation includes

decimation by two, the output from each of the horizontal filtering operations occupies only one-half of the area of the original image. In **Figure 17**, only the first row of image data is shown to be filtered, by way of example. Each output of both the low-pass and high-pass filters is displaced by the number of rows in the image data from one another (eight bytes in this example). This transposes the output so that the coefficients corresponding to a horizontal row of input data are now effectively arranged in memory so that they are vertical, as shown. This is repeated for all rows of image data.

- Vertical Filtering.** After all the rows of the image data are horizontally filtered, the vertical filtering of the low-pass and high-pass coefficients occurs separately. Because the original image data (stored at the label location 'im1' in the assembler code) is no longer needed, the output from the vertical filtering operation can overwrite the original image data. The most memory required at any one time is twice the tile size ( $2 \times 8 \times 8$  bytes in this case). Since a transposition of the data has taken place, the vertical filtering can use the same assembly code that was used for the horizontal filtering. In **Figure 17**, the first memory area, denoted by im1, is conveniently reshaped, and the output is lined up so that a row corresponds to the original horizontal row of data (but decimated by two). Of course, in reality, the memory is actually one continuous collection of bytes.



**Figure 17.** Transposing Horizontal Output Prior to Vertical Filtering

### 2.4.3 Parallel Features

In the assembler, the MAC instruction is not used for the wavelet transform, which is the main difference between the lifting algorithm and its convolution counterpart. A great deal of parallelism can be obtained without the MAC units. The two address arithmetic units can simultaneously perform two parallel address calculations. Additionally, the four MAC and Bit Field Units (BFUs) can perform simple addition, subtraction, and shifting operations in parallel with the address calculations. The two separate pairs of memory data and address buses can read or write up to two words of data to memory in the same instruction cycle. The following code line shows the parallel set up of two data registers with memory.:

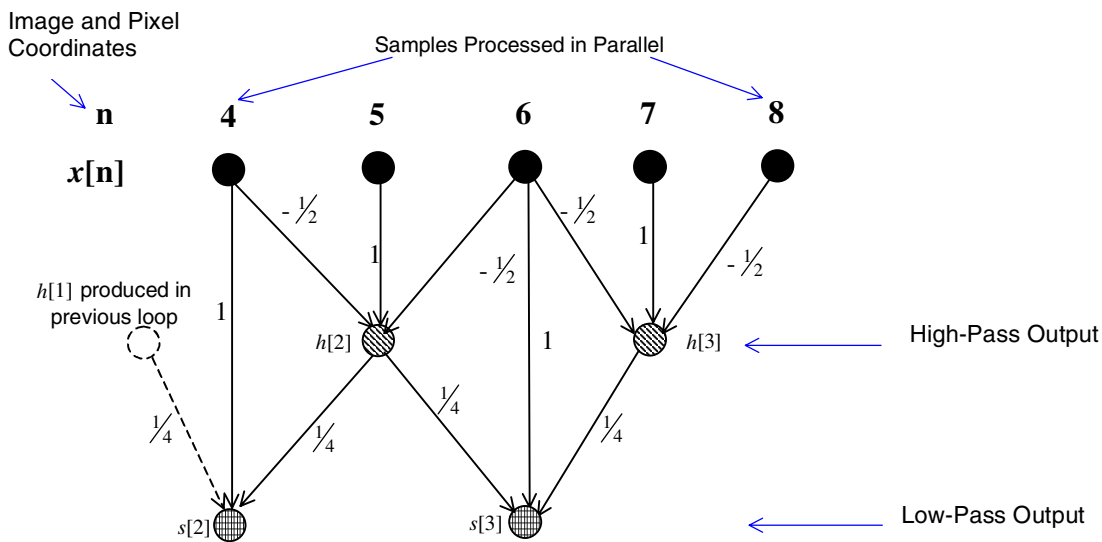
```
move.w (r0)-,d0           move.w (r6)-,d11
```

In the outer loop (LEV\_LOOP), the T bit in the status register (SR) is used for conditional execution of sets to determine whether to carry out the vertical filtering stage on the previously high-pass filtered data or on the previously low-pass filtered data. The following example, taken from the code, shows how such a decision can be made and provides an example of parallelism within the set. A move instruction is performed simultaneously with an addition to one of the address registers.

```
ift move.w #H_OUT,r0adda #HALF_PIXEL,r1,r1
```

The numerous registers are a key feature of a StarCore processor, which provide a great amount of flexibility. Therefore, there is no need to store and transfer memory addresses repeatedly to address registers because there are sufficient numbers (up to sixteen) of them to allow one register to be dedicated to a particular area of memory. Notice that there are sixteen available in this example because modulo addressing is not used. In addition to the address registers, there are sixteen data registers, which give further flexibility to the program.

Lastly, the core part of the algorithm is optimized and comprises the inner loop. In this loop, although only ten instruction cycles are used, four separate output coefficients are calculated in parallel and written to memory. These output coefficients include two low-pass filter outputs and two high-pass filter outputs. This is achieved once again by making use of the many address and data registers available. As shown in **Figure 18**, the samples are processed either wholly or partially up to five at a time creating a very efficient way of processing the input because it does not require repeatedly storing and retrieving the samples from memory. The even samples in particular, are involved directly in the production of two high-pass outputs and a low-pass output.



**Figure 18.** Samples Processed in Parallel to Produce Four Output Coefficients



## 2.5 StarCore Implementation in Optimized Assembler

**Appendix C** presents a version of the wavelet transform in highly-optimized assembler code. In this version, the inner loop is unrolled once more so that eight samples are dealt with at once. Additionally, the middle loop code for handling the symmetric extension is unrolled to handle eight samples at the start of a row and eight samples at the end of the row. This version of the transform further improves processing speed by making even more use of the many data and address registers. As few as 15 cycles are required to process 8 samples and produce 4 low and 4 high pass output coefficients. However, the image tile size must be divisible by 16. The wavelet code takes approximately 62,000 clock cycles to transform a  $128 \times 128$  pixel tile. Processing time is 0.026s to complete a color 1M pixel image with YUV:422 format. This is just over 8 times as fast as the C version of the code.

**Note:** The `skipls` function must be inserted before the inner loop if the number of decomposition levels reduce the size of the subband to as little as  $16 \times 16$ . The inner loop is not required and the `skipls` operator ensures that it is not executed. However, a solution is to repeat the outer and middle loop program code without the inner loop after the standard code (given in the **Appendix C**).

## 3 Summary

The wavelet transform C code requires approximately 510,000 cycles to process one  $128 \times 128$  pixel tile. Processing time is about 0.2 seconds to complete a 1 M color-pixel image. These measurements assume that the pixels are all 12 bits. The StarCore processor operates at a clock speed of 300 MHz. The C code in **Appendix A** is ANSI C code, and it was fully optimized by the Metrowerks CodeWarrior compiler in speed and size to produce these performance figures. Modifying the code to work with any image size is fairly straightforward. In addition to the C code, two different versions of the assembler code for the wavelet transform are included to highlight the benefits of StarCore's parallel architecture. In particular, the core wavelet algorithm is optimized so that only 15 instruction cycles are required to produce eight output coefficients. We hope this application note provides sufficient explanation to implement the Le Gall (5,3) reversible discrete wavelet transform, which is included in the JPEG2000 standard. Helpful textbooks dealing with wavelet transforms are cited in references [5], [10], and [11].

## 4 References

- [1] *JPEG2000 – The New Wave in Image Compression*, P. Ogunbona, I. Kharitonenko, and P. John, (CCTV Magazine, pp. 36–38, Jan. 2000).
- [2] *Lecture Notes on Wavelet Transforms*, Ambikairajah E., (University of New South Wales, 2000).
- [3] *Lecture notes on Digital Image Processing*, D. Taubman, (University of New South Wales, 2000).
- [4] “A Theory for Multiresolution Signal Decomposition: The Wavelet Representation,” S. Mallat, *IEEE Trans. Pattern Recognition And Machine Intelligence*, Vol.11, No.7, pp. 674–693 (Jul., 1989).
- [5] *Wavelets and Subband Coding*, M. Vetterli and J. Kovačević, (Prentice Hall: 1995).
- [6] “Orthonormal Bases of Compactly Supported Wavelets,” I. Daubechies, *Communications on Pure and Applied Mathematics*, Vol. XL1, pp. 090-996, 1988.
- [7] “Subband Coding of Digital Images Using Symmetric Short Kernel Filters and Arithmetic Coding Techniques,” D. Le Gall, and A. Tabatabai, *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 761–765, (New York: NY, 1988).
- [8] *JPEG 2000 Image Coding System*, JPEG2000 Final Committee Draft Version 1.0, 16<sup>th</sup> Mar., 2000.

- [9] “The Lifting Scheme: a Custom-Design Construction of Biorthogonal Wavelets,” W. Sweldens, *Applied Computing Harmonic Analysis*, vol. 3, no. 2, pp. 186–200, 1996.
- [10] “Wavelets in Image Communication: Advances in Image Communication,” Vol. 5, Chapter 3, *Wavelet Transform and Image Coding*, Ed. Barland M., (Elsevier: 1994).
- [11] *Wavelets and Filter Banks*, G. Strang and T. Nguyen, (Wellesley-Cambridge Press: 1996).

# Appendix A

## Discrete Wavelet Transform C Code

This program carries out the separable 2-D transform of image tile components using the LeGall (5,3) wavelet transform.

### Code Listing 1. Discrete Wavelet Transform C Code

```

/*****
Copyright (C) 2000, Freescale Semiconductor, Inc., All Rights Reserved
This copyright notice does not imply publication.

FILENAME: dwt.c

DESCRIPTION: Discrete Wavelet Transform
This program carries out the separable 2-D transform
of image tile components using the LeGall (5,3) wavelet
transform.

It Implements Annex E of the Final Committee Draft of the
JPEG 2000 standard, Part 1, 16 March 2000.

It assumes a standard tile size of 128x128 pixels. In this
particular example, the data from image.h is used for
processing.

*****/

#ifndef __DWT_H
#include "dwt.h"
#endif

/*-----*/
/* Function prototypes */
/*-----*/
/* Transform */

static void
TileTransformInPlace (
    Coeff buf [],          /* Input buffer - must be kTileWidth by */
                        /* kTileHeight in size */
    const Uint rows,     /* Rows in input buffer */
    const Uint cols,     /* Columns in input buffer */
    const Uint numLevels /* Number of levels of decomposition */
);

/* Wavelet */

static void
FilterBlock_Rows (
    const Coeff src [],   /* Source block */
    const Uint rows,     /* Number of rows in a block */
    const Uint cols,     /* Number of columns in a block */
    const Uint srcRowStep, /* Step width between rows */
    Coeff dstLow [],     /* Destination block - low pass */
    Coeff dstHigh []     /* Destination block - high pass */
);

```

```

static void
FilterBlock_Cols (
    const Coeff src [], /* Source block. */
    const Uint rows, /* Number of rows in block.*/
    const Uint cols, /* Number of columns in block. */
    const Uint rowStep, /* Step width between rows. */
    Coeff dstLow [], /* Destination block - low pass. */
    Coeff dstHigh [] /* Destination block - high pass. */
);

static void
Filter_5_3_Stepped (
    const Coeff src [], /* Incoming array for filtering.*/
    const Uint n, /* Number of elements in src */
    const Uint srcStep,
        /* Number of array elements between successive source values.*/
    Coeff dstLow [], /* Destination array for low-pass coefficients */
    Coeff dstHigh [], /* Destination array for high-pass coefficients*/
    const Uint dstStep /* Number of array elements between successive */
        /*destination values. Must be equal for both destination arrays.*/
);

/*-----*/
/* global variables */
/*-----*/

Coeff gTempBuf [kTileWidth * kTileHeight];

/*****

FUNCTION NAME: Main

VALUE RETURNED: none

FUNCTION(S) CALLED: TileTransformInPlace.

DESCRIPTION
    This function calls TileTransformInPlace
    The code has been hardcoded to provide 3 levels of decomposition
    for this application note.

*****/

void main(void)
{
    TileTransformInPlace (gBuf, kRows, kCols, LEVELS);
}

/*-----*/
/* In place transform */
/*-----*/

/*****

FUNCTION NAME: TileTransformInPlace

```

INPUTS:

```
Coeff buf[]      Input buffer (Must be kTileWidth by kTileHeight in size.)
const Uint rows  Rows in input buffer.
const Uint cols  Columns in input buffer.
const Uint numLevels  Number of levels of decomposition.
                If numLevels is 0 then element 0 of the subbands (LL) will contain the contents
                of the input tile.
```

DESCRIPTION:

Transform tile into sub-bands, in place.

Initialising bands

The array of subbands is  $1 + 3 * (\text{numLevels} - 1)$  elements long. Element 0 is the LL band, resolution 0. Elements 1-3 are HL, LH and HH of resolution 1, in that order. Elements 4-6 are HL, LH and HH of resolution 2. And so on. Resolutions correspond to numLevels.

Each array of coefficients for each band must be initialised. The array memory must exist, but values are don't care. Elements corresponding to the highest resolution (resolution numLevels) must be exactly  $k\text{TileWidth}/2$  wide and  $k\text{TileHeight}/2$  high. Elements for the second highest resolution must be exactly half as big. And so on. Elements for resolution 0 are the same size as resolution 1.

On completion, the input buffer is partitioned into sub-bands. Resolution numLevels partitions buf into quarters, each  $k\text{TileWidth}/2$  by  $k\text{TileHeight}/2$ . The 2nd, 3rd and 4th quarters contain the LH, HL and HH sub-bands. The first quarter is partitioned again into quarters. This continues until resolution 1, where the 1st quarter contains the LL sub-band (resolution 0).

If the rows or cols (or both) is less than kTileWidth or kTileHeight, data will be packed to the top and left boundaries of each quarter.

\*\*\*\*\*/

```
static void
TileTransformInPlace (
    Coeff buf [],
    const Uint rows,
    const Uint cols,
    const Uint numLevels
)
{
    /* counters */
    Uint i;

    /* Real variables */

    const Uint  maxWidth = kTileWidth >> 1;

    Uint      width = kTileWidth;

    Uint      hBufOffset = maxWidth * kTileHeight;
    Uint      _hBufOffset = hBufOffset >> 1;

    Uint      lrows, hrows;
    Uint      lcols, hcols;
```

```

lrows = rows;
lcols = cols;
hcols = cols;

for (i = numLevels; i > 0; i--)
{
    Coeff * hBuf;
    Coeff * lhBuf;
    Coeff * hlBuf;
    Coeff * hhBuf;

    /* Do High / Low filtering */
    hBuf = gTempBuf + hBufOffset;
FilterBlock_Cols (
    buf,
    lrows,
    lcols,
    width,
    gTempBuf,
    hBuf
);

    /* Update values */
    hrows = lrows >> 1;
    lrows -= hrows;

    /* Do L -> LL / LH filtering */
    lhBuf = buf + _hBufOffset;

    FilterBlock_Rows (
        gTempBuf,
        lrows,
        lcols,
        width,
        buf,
        lhBuf
    );

    /* Do H -> HL / HH filtering */
    hlBuf = buf + hBufOffset;
    hhBuf = buf + hBufOffset + _hBufOffset;

    FilterBlock_Rows (
        hBuf,
        hrows,
        hcols,
        width,
        hlBuf,
        hhBuf
    );

    /* Update values */
    width >>= 1;
    hcols = lcols >> 1;
    lcols -= hcols;

    /* Clean up buffer offsets */
    hBufOffset >>= 2;

```

```

        _hBufOffset >>= 2;
    }
}

/*****

FUNCTION NAME: filter_5_3_odd

INPUTS:

    int x   Central input sample. X(i), where i is odd.
    int xp  Previous input sample, X(i-1).
    int xn  Next input sample, X(i+1).
    (note that xp and xn may be exchanged without error, which
     is useful for symmetric extension)

VALUE RETURNED: int, resulting filtered value.

DESCRIPTION:
    Compute the result for the odd sample of the input, i.e.
    high pass filter.
*****/
INLINE
int filter_5_3_odd (int x, int xp, int xn);

INLINE
int filter_5_3_odd (int x, int xp, int xn)
{
    return x - ((xp + xn) >> 1);

    /* (xp + xn - 1) / 2, round up, is given by adding +1 before dividing. */
}

/*****

FUNCTION NAME: filter_5_3_even

INPUTS:

    int x   Central input sample. X(i), where i is even.
    int yp  Previous output sample of odd filter, Y(i-1).
    int yn  Next output sample of odd filter, Y(i+1).
    (note that yp and yn may be exchanged without error which
     is useful for symmetric extension)

VALUE RETURNED: int, resulting filtered value.
DESCRIPTION:
    low pass filter of input even samples
*****/
INLINE
int filter_5_3_even (int x, int yp, int yn);

INLINE
int filter_5_3_even (int x, int yp, int yn)
{
    return x + ((yp + yn + 2) >> 2);

    /* (yp + yn + 2) / 4, round down, is correct since >> rounds down. */
}

```

```

}

/*****

FUNCTION NAME: filter_5_3_stepped

INPUTS:
    const Coeff src    Incoming array for filtering.
    const Uint n      Number of elements in src.
    const Uint srcStep Number of array elements between successive source values.
    Coeff dst_low[]   Destination array for high-pass coefficients.
                    Minimum length n/2 (round up).
    Coeff dst_high[]  Destination array for high-pass coefficients.
                    Minimum length n/2 (round down).
    const Uint dstStep Number of array elements between successive destination values.
                    Must be equal for both destination arrays.

DESCRIPTION:
    Apply a (5,3) lifting filter to an array.
    This function works with non-contiguous arrays.
*****/

static void
Filter_5_3_Stepped (
    const Coeff src [],
    const Uint n,
    const Uint srcStep,
    Coeff dst_low [],
    Coeff dst_high [],
    const Uint dstStep
)
{
    /* Use rolling pass to avoid memory accesses. */
    /*
    Loop is centred around odd values. Current odd value is i.
    X is input array.
    Y is interleaved output array.
    Y(i), Y(i+2), ... are odd values (high coefficients).
    Y(i-1), Y(i+1), ... are even values (low coefficients).
    In practice, Y is stored to separate data arrays.
    */

    int x_odd;    /* X(i) */
    int x_even_a; /* X(i-1) */
    int x_even_b; /* X(i+1) */
    int hi_a;     /* Y(i-2) */
    int hi_b;     /* Y(i) */
    int lo;       /* Y(i-1) */

    /* Initial pass using symmetry. */
    const Uint src2Step = srcStep << 1;

    x_even_b = src [src2Step];
                /* this will be x_even_b in pass 1 also. */

    x_odd = src [srcStep];
                /* this will be x_odd in pass 1 also. */

    x_even_a = src [0];

```



```

hi_b      = filter_5_3_odd (x_odd, x_even_a, x_even_b);
           /* for pass 1, this is also hi_a. */

hi_a      = hi_b;

/* Main body */
{
    /*
Main pass
           At start of loop, we have Y(i-2) and Y(i) (odd).
           Store Y(i).
           We calculate Y(i-1) (even) and store.
           Increment i (by 2). Current Y(i) is undefined.
           Load X(i), X(i+1) and calculate Y(i).
           Re-loop.
    */

    Uint i = 1;
    Uint srcOffset = srcStep; /* i * srcStep; */
    Uint dstOffset = 0;      /* (i * dstStep)/2 */

    while (1)
    {
        /* Store */
        /* dst_high [i >> 1] = hi_b; */
        dst_high [dstOffset] = hi_b;

        /* Calculate Y(i-1) */
        lo      = filter_5_3_even (x_even_a, hi_a, hi_b);
        /* dst_low [i >> 1] = lo; */
        dst_low [dstOffset] = lo;

        /* Increment and check termination. */
        i += 2;
        srcOffset += src2Step;
        dstOffset += dstStep;
        if (i >= (n-1))
            break;

    /* Shuffle and reload */
        hi_a      = hi_b;
        /* x_odd   = src [i]; */
        x_odd     = src [srcOffset];
        x_even_a  = x_even_b;
        /* x_even_b = src [i+1]; */
        x_even_b  = src [srcOffset + srcStep];
        hi_b     = filter_5_3_odd (x_odd, x_even_a, x_even_b);
    }

    /* final pass using symmetry */
    if (n & 0x01) /* odd n */
    {
        /*
        i == n
        Useful values at termination:
        x_odd      == X(i-2) == X(n-2)
        x_even_a   == X(i-3) == X(n-3)
        x_even_b   == X(i-1) == X(n-1)
        */
    }
}
    
```

```

    hi_b      == Y(i-2) == Y(n-2)
Using symmetry:
X(n-1)      == X(n-1)
X(n)        == X(n-2)
X(n+1)      == X(n-3)

We need:
For Y(i-1): Y(i-2), Y(i), X(i-1).
For Y(i):   X(i), X(i-1), X(i+1).
            X(n), X(n-1), X(n+1)

Note however:
For Y(i-2): X(i-2), X(i-3), X(i-1).
            X(n-2), X(n-3), X(n-1).
            X(n), X(n+1), X(n-1).
So Y(i) = Y(i-2).
*/

/* Calculate Y(i-1) and store. */
lo      = filter_5_3_even (x_even_b, hi_b, hi_b);
/* dst_low [i >> 1] = lo; */
dst_low [dstOffset] = lo;
}
else    /* even n */
{
    /*
    i == n-1
    Useful values at termination:
    x_odd      == X(i-2) == X(n-3)
    x_even_a   == X(i-3) == X(n-4)
    x_even_b   == X(i-1) == X(n-2)
    hi_b      == Y(i-2) == Y(n-3)
    Using symmetry:
    X(n-1)    == X(n-1)
    X(n)      == X(n-2)
    X(n+1)    == X(n-3)

    We need:
    For Y(i-1): Y(i-2), Y(i), X(i-1).
    For Y(i):   X(i), X(i-1), X(i+1).
    All we are missing is X(i) == X(n-1).
    We don't need X(i+1) == X(n) == X(n-2), since we have this already.
    */

    /* Load X(i) */
    /* x_odd      = src [i]; */
    x_odd      = src [srcOffset];

    /* Calculate and Store Y(i) */
    hi_a      = filter_5_3_odd (x_odd, x_even_b, x_even_b);
    /* dst_high [i >> 1] = hi_a; */
    dst_high [dstOffset] = hi_a;
/* Calculate Y(i-1) and store. */
lo      = filter_5_3_even (x_even_b, hi_b, hi_a);
/* dst_low [i >> 1] = lo; */
dst_low [dstOffset] = lo;
}
}
}
}

```

```

/*****

```

```

FUNCTION NAME: FilterBlock_Rows

```

```

INPUTS:

```

```

    const Coeff src    Source block.
    const Uint rows   Number of rows in block.
    const Uint cols   Number of columns in block.
    const Uint srcRowStep  Step width between rows.
    Coeff dstLow[]    Destination block - low pass.
    Coeff dstHigh[]   Destination block - high pass.

```

```

DESCRIPTION:

```

```

    Filter the rows of a block.

```

```

    src is treated as a two dimensional array with rows starting every
    srcRowStep elements. Element 0 is row 0, column 0.

```

```

    The filter is applied over elements 0..(col-1) for each of
    rows.

```

```

    Destination blocks are considered to be in the same format, but with
    each row being only half as wide (srcRowStep/2).

```

```

    srcRowStep should be divisible by 2.

```

```

*****/

```

```

static void
FilterBlock_Rows (
    const Coeff src [],
    const Uint rows,
    const Uint cols,
    const Uint srcRowStep,
    Coeff dstLow [],
    Coeff dstHigh []
)
{
    const Uint dstRowStep = srcRowStep >> 1;
    Uint srcStart = 0;
    Uint dstStart = 0;
    Uint i;

    for (i = 0; i < rows; i++)
    {
        Filter_5_3_Stepped (
            src + srcStart,
            cols,
            1,
            dstLow + dstStart,
            dstHigh + dstStart,
            1
        );

        srcStart += srcRowStep;
        dstStart += dstRowStep;
    }
}

```

```

/*****

```

```

FUNCTION NAME: FilterBlock_Cols

```

INPUTS:

```

const Coeff src    Source block.
const Uint rows   Number of rows in block.
const Uint cols   Number of columns in block.
const Uint rowStep Step width between rows.
Coeff dstLow[]    Destination block - low pass.
Coeff dstHigh[]   Destination block - high pass.

```

DESCRIPTION:

Filter the columns of a block.

src is treated as a two dimensional array with rows starting every rowStep elements. Element 0 is row 0, column 0.

The filter is applied over elements 0..(row-1) for each of columns.

Destination blocks are considered to be in the same format as src. Destination blocks will have only half as many rows as source blocks.

\*\*\*\*\*/

```

static void
FilterBlock_Cols (
    const Coeff src [],
    const Uint rows,
    const Uint cols,
    const Uint rowStep,
    Coeff dstLow [],
    Coeff dstHigh []
)
{
    Uint i;

    for (i = 0; i < cols; i++)
    {
        Filter_5_3_Stepped (
            src + i,
            rows,
            rowStep,
            dstLow + i,
            dstHigh + i,
            rowStep
        );
    }
}

```

```

/*****
Copyright (C) 2000, MARC, Freescale, Semiconductor, Inc., All Rights Reserved
This copyright notice does not imply publication.

```

FILENAME: dwt.h

DESCRIPTION:

Contains definitions and types for discrete wavelet transform.

\*\*\*\*\*/

```

#ifndef __DWT_H
#define __DWT_H

```

```

#endif

#define INLINE static
#define LEVELS 3

/*-----*/
/* Standard Types */
/*-----*/

typedef unsigned int Uint;

typedef short int Coeff;

/*-----*/
/* Standard Constants */
/*-----*/

enum
{
    kTileWidth = 128,
    kTileHeight = 128,
    kRows = 128,
    kCols = 128
};

/*-----*/
/* Include files */
/*-----*/

#include <stdio.h>

/* DC shifted 12 bits image tile of size 128x128 */
#ifndef __IMAGE_H
#include "image.h"
#endif

/*****
    Copyright (C) 2000, MARC, Freescale Semiconductor, Inc., All Rights Reserved
    This copyright notice does not imply publication.

    FILENAME: image.h

    DESCRIPTION:
    Contains image data for testing dwt.c
*****/
#ifndef __IMAGE_H
#define __IMAGE_H
#endif

Coeff gBuf[kRows*kCols] =
{
    insert image data.
};

```

# Appendix B

## Discrete Wavelet Transform - Assembly Code

**Code Listing 2.** Discrete Wavelet Transform Assembly Code

```

;*****
; Star*Core benchmarks
;*****
; FILENAME      :    wave_5_3.asm
; AUTHOR        :                Sue Twelves
;*****
; ALGORITHM     :    1 level decomposition of the 5,3 wavelet transform
;
;*****
***
;
; Constants Used:
;
;   H_OUT      :    $500    initial address of high pass output.
;   L_OUT      :    $650    initial address of low pass output.
;
;   COLUMN_DIV :    2       number of samples in a row (number of columns) divided by four
;                           less the first 4 and last 4 for symmetric extension. The division
;                           by 4 is due to the parallel nature of the algorithm where four
;                           samples are dealt with simultaneously.
;   ROW        :    16      number of rows in image
;
; Input/Output location:
;   high output initial address:  H_OUT
;   low output initial address:   L_OUT
;
; registers used:
;   d0 through d9, r0,r1,r2,r3,r4,r5,r6
;
;   r0 points to the input x(n)
;   r1 points to the high pass output h(i), it is updated by an offset with the number of rows
;   r2 points to the low pass output s(i), it is updated by an offset with the number of rows
;   r3 points to the next contiguous high pass output memory location
;   r4 points to the next contiguous low pass output memory location,
;   r6 points to the next input x(n+4) for parallel processing
;   n0 contains 2, due to the decimation by 2 after low and high pass filtering
;   n1 contains the number of rows and provides the offset for both r1 and r2
;
; assumptions:
;   Number of columns and rows are powers of 2.
;   Only one level of decomposition is carried out.
;*****
***
COLUMN_DIV      equ          2
ROW             equ          16
HALF_PIXEL     equ          256
QUART_PIXEL    equ          128
SAMP_BLOCK     equ           8
H_OUT          equ         $500
L_OUT          equ         $650

; Input data

```

```

org p:$0
jmp $1000

org p:$200
im1:      dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00
          dcw  $02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$00,$00,$00,$00
          dcw  $09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$00,$00,$00,$00

          org p:$1000

init:
;   r0 -> input x(n),
;   r6 -> x(n+4)
;   r1 -> output h(i) (high pass)
;   r2 -> output, s(i).
;   point r3 -> h(i) too
;   point r4 -> s(i) too
;
;   r5 contains the number of rows for processing - it controls the number of times
;   the middle loop is carried out - this changes depending on which stage of the
;   decomposition the algorithm is at. Every time a low or high pass filter has
;   completed it's task, the number of outputs is halved (decimation), therefore,
;   the number of subsequent samples for a filtering operation is also halved.
;
;   set up the vertical and horizontal filter loops and memory pointers

move.l #im1,r0
move.w #H_OUT,r1
move.w #L_OUT,r2      dosetup0 LEV_LOOP
move.w #1,d8
move.w #ROW,r5        dosetup1 VERT_LOOP
move.w #2,n0
move.w #2,d9          dosetup2 HORIZ_LOOP
move.w #ROW,n2        doen0 #3
move.w #2,d7
move.w #3,n3
move.w #5,n1

; This loop ensures that one level of wavelet decomposition is undertaken.
;   skips test
LEV_LOOP
kernel:      loopstart0
            tfra r2,r4
            doen1 r5
            [   tfra r0,r6      tfra r1,r3
              ]
            adda #SAMP_BLOCK,r6

```

```

; need to process every row in the image
VERT_LOOP

; This is the outer loop to go around all the rows in the image.
; Need to do symmetrically extended bits first

loopstart1

; only go around the inner loop (n/4) - (2x4) because 1st and last 4 values
; in a row is dealt with separately because of symmetric extension.
; point r3 to h(i+1) and r4 to s(i+1)
; d11 = x(n+4)      d1 = x(n+2)      d10 = x(n+3)

[      adda #2,r3      move.w (r0+n0),d1
]
[      adda #2,r4      move.w (r6)-,d11
]

; r6 -> x(n+8)
; put x(n) into d0
; put x(n+2) into d2
; add x(n+2)+x(n+4)

[      doen2 #COLUMN_DIV  move.w (r6)+n1,d10
]
[      move.w          (r0)+,d0      iadd d1,d11
]

; calculate x(n) + x(n+2)
; calc 0,5(x(n+2) + x(n+4))
; d3 = x(n+1)
; r0 -> x(n+4)

[      tfr d1,d2          iadd d0,d1
      asr d11,d11        move.w (r0)+n3,d3
]

; calc 0.5(x(n) + x(n+2))
; calc x(n+3) - 0.5(x(n+2)+x(n+4))
; d10 = second high pass output h[i+1]

[      asr   d1,d1      sub d11,d10,d10
      tfr d9,d12      tfr d8,d4
]

; d3 now holds the high pass output, h[i]

sub d1,d3,d3

; calc x(n+1) - 0.5(x(n)+x(n+2))
; in first case, h[-1] = h[0] (symmetric extension), therefore, just divide by 2
; to calculate the low pass output, but first add 1 to prevent bias
; d3 = high pass output h[-1] = h[0]
; d4 = h[0] + 1
; add 2 for second low pass output to prevent bias (because need to divide by 4)
; d10 = h(i)+h(i+1)+2
; need to store these in the buffer high pass output area of memory.

```



```

; update pointer to high pass memory so that it is offset by the number
; of rows to ensure that the horizontal output is transposed to become
; vertical ready for the vertical filtering operations.

[      iadd d3,d4          move.w d3,(r1)+n2
      iadd d10,d3
]
[      asr d4,d4          move.w d10,(r1)+n2
      iadd d3,d12
]

; calc x(0) + 0.25(h[-1]+h[0])+0.5, i.e. s(0) and put into d4
; calc 0.25(h[0]+h[1])+0.5, put into d12
; get ready for next inner loop:
; d3 updated to h(i+1)
; d1 updated to x(n+4)

[      iadd d0,d4          tfr d10,d3
      move.w (r0)+n0,d1    asrr #2,d12
]

; calc x(n+2) + 0.25(h[i]+h[i+1])+0.5, put into d12
; need to store these in low pass output areas of memory.
; update pointer to low pass memory so that it is offset by the number
; of rows, see comment above

[      move.w d4,(r2)+n2
      iadd d2,d12
]

; d2 updated to x(n+4) too

[      move.w d12,(r2)+n2          tfr d1,d2
]

; Situation before inner loop, now got:
; x(n+4) in d1 - need for calculation of h[i+2]
; h[i+1] in d3 - need for calculation of s[i+2]
; r0 points to x(n+4)
; r6 points to x(n+8)

; for every row in the image low and high pass filter the samples
      skipls          test2
HORIZ_LOOP
      loopstart2
          ; d0 = x[n+2]

          [
              move.w (r0)-,d0          move.w (r6)-,d11
          ]

          ; add x[n] to x[n+2] and multiply by a half
          ; set d4 = x[n+1]

          [      iadd d0,d1          iadd d0,d11
              move.w (r6)+n1,d10    move.w (r0)+n3,d4
          ]

          [      asr d1,d1          asr d11,d11
          ]

```

```

; calc x03 - 1/2(x(n)+x(n+2))

[
sub d1,d4,d4                sub d11,d10,d10
]

; update d1 to x(n+4) for the next loop          as x(n)
; d4 now holds h(n) i.e. current high pass output
; calc 1/4(h(n-1)+h(n)) + 1/2

[
iadd d4,d3                move.w d4,(r1)+n2
                        move.w (r0)+n0,d1
]

; add 2 inside the loop to get addition of a half when divided by 4.

[
iadd d9,d3                iadd d10,d4
move.w d10,(r1)+n2
]

; divide by 1/4 - truncation gives desired result

[
asrr #2,d3                iadd d9,d4
]

; calc s(n) by adding x(n) + 1/4(h(i-1) + h(i) +2), d3 = s(n)
; d4 = 0.25(2 + h(i) + h(i+1))

[
iadd d2,d3                tfr d1,d2
asrr #2,d4
]

; move h(i) and s(i) to respective outputs
; add offset of number of rows to low pass output pointer, r2
; update d3 to h(n) for next time around the loop as h(n-1)

[
move.w d3,(r2)+n2        iadd d0,d4
tfr d10,d3
]

move.w d4,(r2)+n2

```

loopend2

```

; now deal with last value for high pass and low pass using odd symmetric
; extension - the value of x(n+2) now equals x(n) assuming even number of
; samples in a row.
; h(i-1) is in d3
; r0 -> x[n]
; d1 = x[n] and x[n+2] = x[n], symmetric extension
; h[i] = x[n-1] - x[n] = x[n+1] - 1/2(x[n] + x[n+2])
; set d4 = x[n+1]

```

test2:

```

[
move.w (r0)-,d0          suba #2,r6
]

[
iadd d0,d1                move.w (r0)+n3,d4
move.w (r6)+n1,d10
]

```

```

[      asr d1,d1      sub d0,d10,d10
]

sub d1,d4,d4

[      iadd d4,d3      move.w d4,(r1)+n2
      iadd d10,d4
]

[      iadd d9,d3      iadd d9,d4
      move.w d10,(r1)
]

;calc 1/4(h(i-1) + h(i) + 2)
;set r1 -> the next contiguous high pass area of memory

[      asrr #2,d3
      tfra r3,r1      asrr #2,d4
]

;calc s(i) by adding x[n]

[      iadd d2,d3      iadd d0,d4
]

; move s(i) to output
;set r2 -> next contiguous low pass area of memory

move.w d3,(r2)+n2
[      move.w d4,(r2)
      tfra r4,r2
]
loopend1

; clear the test bit in the status register: allows a conditional jump to be made
; when d7 is decremented to zero - this allows the outer loop to be performed three
; times - first on the input image and then separately on the resulting low and high
; pass output coefficients.
; In this case, the low pass filtering always precedes the high pass filtering,
; therefore, the reduction in loop counts is carried out in low pass set up line.

bmclr #0001,sr.l
move.l #im1,r1
nop
deceq d7
nop

; r0 -> the previous high pass output (i.e. new input for filtering)
; r1 -> midway through the original image memory area (i.e. new output)
; r1 will fill the HH subband
; r2 -> 3/4 way through the original image memory area
; r2 will fill the HL subband

ift [      move.w #H_OUT,r0      adda #HALF_PIXEL,r1,r1
]

; set up for low pass band filtering
; r0 -> the previous low pass output (i.e. new input for filtering)

```

```
; r1 -> the start of the original image memory area (i.e. new output)
; r1 will fill the LH subband
; r2 -> 1/4 way through the original image memory area
; r2 will fill the LL subband

iff move.w #L_OUT,r0      asra r5
iff asra n2

adda #QUART_PIXEL,r1,r2

loopend0

test:      stop
out
```

# Appendix C

## Discrete Wavelet Transform Optimized Assembly Code

**Code Listing 3.** Discrete Wavelet Transform - Optimized Assembly Code

```

;*****
; Star*Core benchmarks
;*****
; FILENAME      :   wave_5_3_opt.asm
; AUTHOR        :           Sue Twelves
; ALGORITHM     :   1 level decomposition of the 5,3 wavelet transform
;
;*****
;
; Constants Used:
;
;   H_OUT:      $500  initial address of high pass output.
;   L_OUT:      $650  initial address of low pass output.
;
;   COLUMN_DIV:  2   number of samples in a row (number of columns) divided by 8 less the
;                   first 8 and last 8 for symmetric extension. The division by 8 is
;                   due to the parallel nature of the algorithm where loop unrolling
;                   has allowed 8 samples to be dealt with simultaneously.
;   ROW:         32   number of rows in image
;   HALF_PIXEL  :   half the number of pixels in the image tile
;   QUART_PIXEL :   quarter the number of pixels in the image tile
;   SAMP_BLOCK:   number of samples dealt with in parallel with the loop unrolling
;
; Input/Output location:
;   high output initial address:  H_OUT
;   low output initial address:   L_OUT
;
; registers used:
;   d0 through d15, r0,r1,r2,r3,r4,r5,r6, n0,n1,n2,n3
;
;   r0 points to the input x(n)
;   r1 points to the high pass output h(i), it is updated by an offset with the number of rows
;   r2 points to the low pass output s(i), it is updated by an offset with the number of rows
;   r3 points to the next contiguous high pass output memory location
;   r4 points to the next contiguous low pass output memory location,
;   r5 contains the number of rows for processing - it controls the number of times
;       the middle loop is carried out
;   r6 points to the next input x(n+4) for parallel processing
;   n0 contains 2, due to the decimation by 2 after low and high pass filtering
;   n2 contains the number of rows and provides the offset for both r1 and r2
;   n1 and n3 contain the offsets needed for updating registers used during the
;       loop unrolled stages.
;
; assumptions:
;   Number of columns and rows are powers of 2.
;   The number of samples in a row/column is divisible by 16.
;   Only one level of decomposition is carried out.
;
;*****
COLUMN_DIV    equ        2
ROW           equ        32
HALF_PIXEL    equ        1024
QUART_PIXEL   equ        512
SAMP_BLOCK    equ        8

```



```

dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02
dcw
$02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$02,$03,$04,$05,$06,$07,$08,$09,$00,$00,$00,$00,$02,$
03,$04,$05,$06,$07,$08,$09
dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02
dcw
$02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$02,$03,$04,$05,$06,$07,$08,$09,$00,$00,$00,$00,$02,$
03,$04,$05,$06,$07,$08,$09
dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02
dcw
$02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$02,$03,$04,$05,$06,$07,$08,$09,$00,$00,$00,$00,$02,$
03,$04,$05,$06,$07,$08,$09
dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02
dcw
$02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$02,$03,$04,$05,$06,$07,$08,$09,$00,$00,$00,$00,$02,$
03,$04,$05,$06,$07,$08,$09
dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02
dcw
$02,$03,$04,$05,$06,$07,$08,$09,$08,$00,$00,$00,$02,$03,$04,$05,$06,$07,$08,$09,$00,$00,$00,$00,$02,$
03,$04,$05,$06,$07,$08,$09
dcw
$09,$08,$07,$06,$05,$04,$03,$02,$03,$00,$00,$00,$09,$08,$07,$06,$05,$04,$03,$02,$00,$00,$00,$00,$09,$
08,$07,$06,$05,$04,$03,$02

```

```
org p:$1000
```

```
init:
```

```

;   r0 -> input x(n),
;   r6 -> x(n+4)
;   r1 -> output h(i) (high pass)
;   r2 -> output, s(i).
;   point r3 -> h(i) too
;   point r4 -> s(i) too
;   r5 contains the number of rows for processing - it controls the number of times
;       the middle loop is carried out - this changes depending on which stage of the
;       decomposition the algorithm is at. Every time a low or high pass filter has completed
;       its task, the number of outputs is halved (decimation), therefore, the number of

```

```

;      subsequent samples for a filtering operation is also halved.

;      set up the vertical and horizontal filter loops and memory pointers

        move.l #im1,r0
        move.w #H_OUT,r1
        move.w #L_OUT,r2      dosetup0 LEV_LOOP
        move.w #1,d8
        move.w #ROW,r5        dosetup1 VERT_LOOP
        move.w #2,n0
        move.w #2,d9          dosetup2 HORIZ_LOOP
        move.w #ROW,n2        doen0 #3
        move.w #2,d7
        move.w #3,n3
        move.w #5,n1

; This outer loop ensures that one level of wavelet decomposition is undertaken.
; It allows the middle and inner loops to be used 3 times, once for horizontal high and
; low pass filtering on the whole image and then again for vertical high and low pass filtering on the
; resultant low and high pass horizontal filtered portions separately.
        skipls test
LEV_LOOP
kernel:  loopstart0
        tfra r2,r4
        doen1 r5
        [      tfra r0,r6      tfra r1,r3
        ]
        adda #SAMP_BLOCK,r6

; need to process every row in the image
; This is the middle loop to go around all the rows in the image.
; Need to do symmetrically extended bits first
; In this case the loop has been unrolled once so that 8 initial samples
; in a row are dealt with before the inner loop and the 8 last samples in a row
; are dealt with outside the loop. Therefore, this code assumes that the number
; of samples in a row are divisible by 16.

VERT_LOOP

        loopstart1
        [      adda #2,r3      move.w (r0+n0),d1
        ]
        [      adda #2,r4      move.w (r6)-,d11
        ]
        [      doen2 #COLUMN_DIV      move.w (r6)+n1,d10
        ]
        [
        move.w      (r0)+,d0      iadd d1,d11
        ]
        [      tfr d1,d2      iadd d0,d1
        asr d11,d11      move.w (r0)+n3,d3
        move.w #1,d8
        ]
        [      asr d1,d1      sub d11,d10,d10
        tfr d9,d12      tfr d8,d4
        move.w (r0)+n0,d8      move.w (r6)-,d13
        ]
        [      sub d1,d3,d3      move.w (r0)-,d6
        move.w (r6)+n1,d14      tfr d8,d11
        ]

```



```

[      iadd d3,d4          move.w d3,(r1)+n2
  iadd d10,d3            move.w (r0)+n3,d15
  iadd d6,d8            iadd d6,d13
]
[      asr d4,d4          move.w d10,(r1)+n2
  iadd d3,d12          asr d8,d8
  asr d13,d13
]
[      iadd d0,d4          tfr d10,d3
  asrr #2,d12          sub d8,d15,d15
]
[      move.w d4,(r2)+n2      sub d13,d14,d14
  iadd d2,d12          iadd d15,d3
  move.w d15,(r1)+n2
]
[      move.w d12,(r2)+n2      move.w (r0)+n0,d1
  iadd d9,d3          iadd d14,d15
]
[      asrr #2,d3          iadd d9,d15
  move.w d14,(r1)+n2      tfr d1,d2
]
[      iadd d11,d3
  asrr #2,d15
]
[      move.w d3,(r2)+n2      iadd d6,d15
  tfr d14,d3
]
      move.w d15,(r2)+n2

```

; Note that if the same code is needed to go down to the second level of decomposition, `skipls` is needed here because with a 16x16 tile, the inner loop is not required as the symmetric extension parts of the code deal with 8+8 samples at either end of a row. If the `skipls` command is not used, the inner loop will execute once. It is possible to get around this requirement by just including a sequence of code after this program which does not include the inner loop, in other words it will only deal with the 16x16 tile size.

; `skipls test2`

; for every row in the image low and high pass filter the samples  
; only go around the inner loop (samples/8) - (2x8) because 1st and last 8 values  
; in a row is dealt with separately because of symmetric extension.

HORIZ\_LOOP

```

      loopstart2

[
  move.w (r0)-,d0          move.w (r6)-,d11
]
[
  iadd d0,d1              iadd d0,d11
  move.w (r6)+n1,d10      move.w (r0)+n3,d4
]
[
  asr d1,d1          asr d11,d11
  move.w (r0)+n0,d8      move.w (r6)-,d13
]
[
  sub d1,d4,d4          sub d11,d10,d10
  move.w (r0)-,d6          move.w (r6)+n1,d14
  tfr d8,d12
]
[
  iadd d4,d3          move.w d4,(r1)+n2
  iadd d6,d13          move.w (r0)+n3,d15
  iadd d6,d8
]

```

```

]
[      iadd d9,d3                      iadd d10,d4
  move.w d10,(r1)+n2                  asr d13,d13
  move.w (r0)+n0,d1                   asr d8,d8
]
[      asrr #2,d3                      iadd d9,d4
  sub d13,d14,d14                    sub d8,d15,d15
]
]
[      iadd d2,d3
  asrr #2,d4
]
]
[      move.w d3,(r2)+n2              iadd d0,d4
  tfr d10,d3                          move.w d15,(r1)+n2
]
]
[      move.w d4,(r2)+n2              move.w d14,(r1)+n2
  iadd d15,d3                          iadd d14,d15
]
]
[      iadd d9,d3                      iadd d9,d15
  tfr d1,d2
]
]
[      asrr #2,d3                      asrr #2,d15
]
]
[      iadd d12,d3
  iadd d6,d15
]
]
[      move.w d3,(r2)+n2
  tfr d14,d3
]
]
      move.w d15,(r2)+n2

```

loopend2

; now deal with last value for high pass and low pass using odd symmetric  
; extension - the value of x(n+2) now equals x(n) assuming even number of  
; samples in a row.

test2:

```

[      move.w (r0)-,d0                  move.w (r6)-,d11
]
]
[      iadd d0,d1                      iadd d0,d11
  move.w (r6)+n1,d10                  move.w (r0)+n3,d4
]
]
[      asr d1,d1                        asr d11,d11
  move.w (r0)+n0,d8                    suba #2,r6
]
]
[      sub d1,d4,d4                      sub d11,d10,d10
  tfr d8,d11                          move.w (r0)-,d6
  move.w (r6)+n1,d14
]
]
[      iadd d4,d3                        move.w d4,(r1)+n2
  iadd d6,d8                          move.w (r0)+n3,d15
  sub d6,d14,d14                       tfr d10,d13
]
]
[      iadd d9,d3                      iadd d10,d4
  move.w d10,(r1)+n2                  asr d8,d8
]
]
[      asrr #2,d3                      iadd d9,d4
  sub d8,d15,d15
]

```

```

]
[   iadd d2,d3           iadd d15,d13
  asrr #2,d4           move.w d15,(r1)+n2
  iadd d14,d15
]
[   move.w d3,(r2)+n2   iadd d0,d4
  iadd d9,d13           iadd d9,d15
  move.w d14,(r1)
]
[   move.w d4,(r2)+n2   asrr #2,d13
  tfra r3,r1           asrr #2,d15
]
[   iadd d11,d13   iadd d6,d15
]
]
move.w d13,(r2)+n2
[   move.w d15,(r2)
  tfra r4,r2
]
]
loopend1
test3:
; clear the test bit in the status register: allows a conditional jump to be made
; when d7 is decremented to zero - this allows the outer loop to be performed three
; times - first on the input image and then separately on the resulting low and high
; pass output coefficients.
; In this case, the low pass filtering always precedes the high pass filtering,
; therefore, the reduction in loop counts is carried out in low pass set up line.

bmclr #0001,sr.1
move.l #im1,r1
nop
deceq d7
nop
ift   [   move.w #H_OUT,r0       adda #HALF_PIXEL,r1,r1
]

; set up for low pass band filtering
; r0 -> the previous low pass output (i.e. new input for filtering)
; r1 -> the start of the original image memory area (i.e. new output)
; r1 will fill the LH subband
; r2 -> 1/4 way through the original image memory area
; r2 will fill the LL subband

iff [   move.w #L_OUT,r0       asra r5
]
iff asra n2
adda #QUART_PIXEL,r1,r2
loopend0
test:
out

```

## **How to Reach Us:**

### **Home Page:**

[www.freescale.com](http://www.freescale.com)

### **E-mail:**

[support@freescale.com](mailto:support@freescale.com)

### **USA/Europe or Locations not listed:**

Freescale Semiconductor  
Technical Information Center, CH370  
1300 N. Alma School Road  
Chandler, Arizona 85224  
+1-800-521-6274 or +1-480-768-2130  
[support@freescale.com](mailto:support@freescale.com)

### **Europe, Middle East, and Africa:**

Freescale Halbleiter Deutschland GMBH  
Technical Information Center  
Schatzbogen 7  
81829 München, Germany  
+44 1296 380 456 (English)  
+46 8 52200080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
[support@freescale.com](mailto:support@freescale.com)

### **Japan:**

Freescale Semiconductor Japan Ltd.  
Headquarters  
ARCO Tower 15F  
1-8-1, Shimo-Meguro, Meguro-ku,  
Tokyo 153-0064, Japan  
0120 191014 or +81 3 5437 9125  
[support.japan@freescale.com](mailto:support.japan@freescale.com)

### **Asia/Pacific:**

Freescale Semiconductor Hong Kong Ltd.  
Technical Information Center  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T. Hong Kong  
+800 2666 8080

### **For Literature Requests Only:**

Freescale Semiconductor Literature Distribution Center  
P.O. Box 5405  
Denver, Colorado 80217  
1-800-441-2447 or 303-675-2140  
Fax: 303-675-2150  
[LDCForFreescaleSemiconductor@hibbertgroup.com](mailto:LDCForFreescaleSemiconductor@hibbertgroup.com)

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. Metrowerks and CodeWarrior are registered trademarks of Metrowerks Corp. in the U.S. and/or other countries. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2000–2004.