Discrete Wavelet Transforms: Theory and Implementation

Tim Edwards (`tim@sinh.stanford.edu`)
Stanford University, September 1991

Abstract

Section 2 of this paper is a brief introduction to wavelets in general and the discrete wavelet transform in particular, covering a number of implementation issues that are often missed in the literature; examples of transforms are provided for clarity. The hardware implementation of a discrete wavelet transform on a commercially available DSP system is described in Section 3, with a discussion on many resultant issues. A program listing is provided to show how such an implementation can be simulated, and results of the hardware transform are presented. The hardware transformer has been successfully run at a voice-band rate of 10kHz, using various wavelet functions and signal compression.

1 Introduction

The field of Discrete Wavelet Transforms (DWTs) is an amazingly recent one. The basic principles of wavelet theory were put forth in a paper by Gabor in 1945 [4], but all of the definitive papers on discrete wavelets, an extention of Gabor's theories involving functions with compact support, have been published in the past three years. Although the Discrete Wavelet Transform is merely one more tool added to the toolbox of digital signal processing, it is a very important concept for data compression. Its utility in image compression has been effectively demonstrated. This paper discusses the DWT and demonstrates one way in which it can be implemented as a real-time signal processing system. Although this paper will attempt to describe a very general implementation, the actual project used the STAR Semiconductor SPROClab digital signal processing system.[1] A complete wavelet transform system as described herein is available from STAR Semiconductor [7].

2 A Brief Discussion of Wavelets

2.1 What a Wavelet Is

The following discussion on wavelets is based on a presentation by the mathematician Gilbert Strang [8], whose paper provides a good foundation for understanding wavelets, and includes a number of derivations that are not given here.

A wavelet, in the sense of the Discrete Wavelet Transform (or DWT), is an orthogonal function which can be applied to a finite group of data. Functionally, it is very much like the Discrete Fourier Transform, in that the transforming function is orthogonal, a signal passed twice through the transformation is unchanged, and the input signal is assumed to be a set of discrete-time samples. Both transforms are convolutions.

Whereas the basis function of the Fourier transform is a sinusoid, the wavelet basis is a

[1] This research was carried out at Stanford University and supported by STAR Semiconductor, Inc. of Warren, New Jersey; and Apple Computer Inc. of Cupertino, California.

set of functions which are defined by a recursive difference equation

$$\phi(x) = \sum_{k=0}^{M-1} c_k \phi(2x - k), \tag{1}$$

where the range of the summation is determined by the specified number of nonzero coefficients M. The number of nonzero coefficients is arbitrary, and will be referred to as the *order* of the wavelet. The value of the coefficients is, of course, not arbitrary, but is determined by constraints of orthogonality and normalization. Generally, the area under the wavelet "curve" over all space should be unity, which requires that

$$\sum_k c_k = 2. \tag{2}$$

Equation (1) is orthogonal to its translations; i.e., $\int \phi(x)\phi(x-k)dx = 0$. What is also desired is an equation which is orthogonal to its dilations, or scales; i.e., $\int \psi(x)\psi(2x-k)dx = 0$. Such a function ψ does exist, and is given by

$$\psi(x) = \sum_k (-1)^k c_{1-k} \phi(2x - k), \tag{3}$$

which is dependent upon the solution of $\phi(x)$. Normalization requires that

$$\sum_k c_k c_{k-2m} = 2\delta_{0m} \tag{4}$$

which means that the above sum is zero for all m not equal to zero, and that the sum of the squares of all coefficients is two. Another important equation which can be derived from the above conditions and equations is

$$\sum_k (-1)^k c_{1-k} c_{k-2m} = 0. \tag{5}$$

A good way to solve for values of equation (1) is to construct a matrix of coefficient values. This is a square $M \times M$ matrix where M is the number of nonzero coefficients. The matrix is designated L, with entries $L_{ij} = c_{2i-j}$. This matrix always has an eigenvalue equal to 1, and its corresponding (normalized) eigenvector contains, as its components, the value of the ϕ function at integer values of x. Once these values are known, all other values of the function $\phi(x)$ can be generated by applying the recursion equation to get values at half-integer x, quarter-integer x, and so on down to the desired dilation. This effectively determines the accuracy of the function approximation.

This class of wavelet functions is constrained, by definition, to be zero outside of a small interval. This is what makes the wavelet transform able to operate on a finite set of data, a property which is formally called "compact support." Most wavelet functions, when plotted, appear to be extremely irregular. This is due to the fact that the recursion equation assures that a wavelet ϕ function is non-differentiable *everywhere*. The functions which are normally used for performing transforms consist of a few sets of well-chosen coefficients resulting in a function which has a discernible shape. Two of these functions are shown in Figure 1; the first is the Haar basis function, chosen because of its simplicity for the following discussion of wavelets, and the second is the Daubechies-4 wavelet, chosen for its usefulness

Wavelet	c_0	c_1	c_2	c_3	c_4	c_5
Haar	1.0	1.0				
Daubechies-4	$\frac{1}{4}(1+\sqrt{3})$	$\frac{1}{4}(3+\sqrt{3})$	$\frac{1}{4}(3-\sqrt{3})$	$\frac{1}{4}(1-\sqrt{3})$		
Daubechies-6	0.332671	0.806891	0.459877	-0.135011	-0.085441	0.035226

Table 1: Coefficients for three named wavelet functions.

in data compression. They are named for pioneers in wavelet theory [3, 5].[2] The nonzero coefficients c_k which determine these functions are summarized in Table 1. Coefficients for the Daubechies-6 wavelet, one used in the discussion of the wavelet transformer hardware implementation, are also given in Table 1.

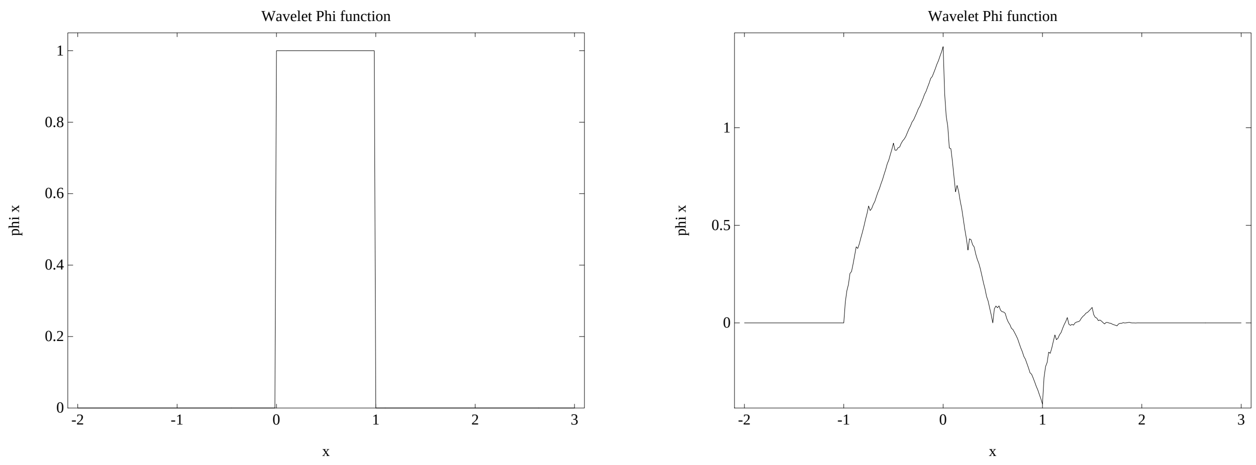

Figure 1: The Haar and Daubechies-4 wavelet basis functions.

The Mallat "pyramid" algorithm [6] is a computationally efficient method of implementing the wavelet transform, and this is the one used as the basis of the hardware implementation described in Section 3. The lattice filter is equivalent to the pyramid algorithm except that a different approach is taken for the convolution, resulting in a different set of coefficients, related to the usual wavelet coefficients c_k by a set of transformations. A proof of this relation is given in Appendix A, using results from [1].

2.2 The Pyramid Algorithm

The pyramid algorithm operates on a finite set of N input data, where N is a power of two; this value will be referred to as the *input block size*. These data are passed through two convolution functions, each of which creates an output stream that is half the length of the original input. These convolution functions are filters; one half of the output is produced by

[2] A Daubechies-n wavelet is one of a family of wavelets derived from certain solutions to the wavelet equations that cause these functions to make a best fit of data to a polynomial of degree n. The Haar wavelet makes a best fit of data to a constant value.

the "low-pass" filter function, related to equation (1):

$$a_i = \frac{1}{2}\sum_{j=1}^{N} c_{2i-j+1} f_j, \qquad i = 1, \ldots, \frac{N}{2}, \tag{6}$$

and the other half is produced by the "high-pass" filter function, related to equation (3):

$$b_i = \frac{1}{2}\sum_{j=1}^{N} (-1)^{j+1} c_{j+2-2i} f_j, \qquad i = 1, \ldots, \frac{N}{2}. \tag{7}$$

where N is the input block size, c are the coefficients, f is the input function, and a and b are the output functions. (In the case of the lattice filter, the low- and high-pass outputs are usually referred to as the odd and even outputs, respectively.) The derivation of these equations from the original ϕ and ψ equations can be found in [3]. In many situations, the odd, or low-pass output contains most of the "information content" of the original input signal. The even, or high-pass output contains the difference between the true input and the value of the reconstructed input if it were to be reconstructed from only the information given in the odd output. In general, higher-order wavelets (i.e., those with more non-zero coefficients) tend to put more information into the odd output, and less into the even output. If the average amplitude of the even output is low enough, then the even half of the signal may be discarded without greatly affecting the quality of the reconstructed signal. An important step in wavelet-based data compression is finding wavelet functions which causes the even terms to be nearly zero.

The Haar wavelet is useful for explanations because it represents a simple interpolation scheme. For example, a sampled sine wave of sixteen data points (note that this is a power of two, as required) is shown in Figure 2. After passing these data through the filter functions,

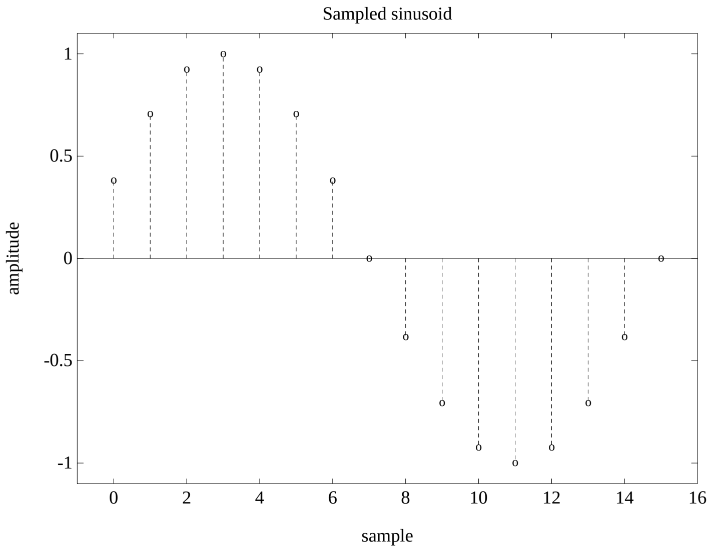

Figure 2: Sampled sinusoid.

the output of the low-pass filter consists of the *average* of every two samples, and the output of the high-pass filter consists of the *difference* of every two samples (see Figure 3). The

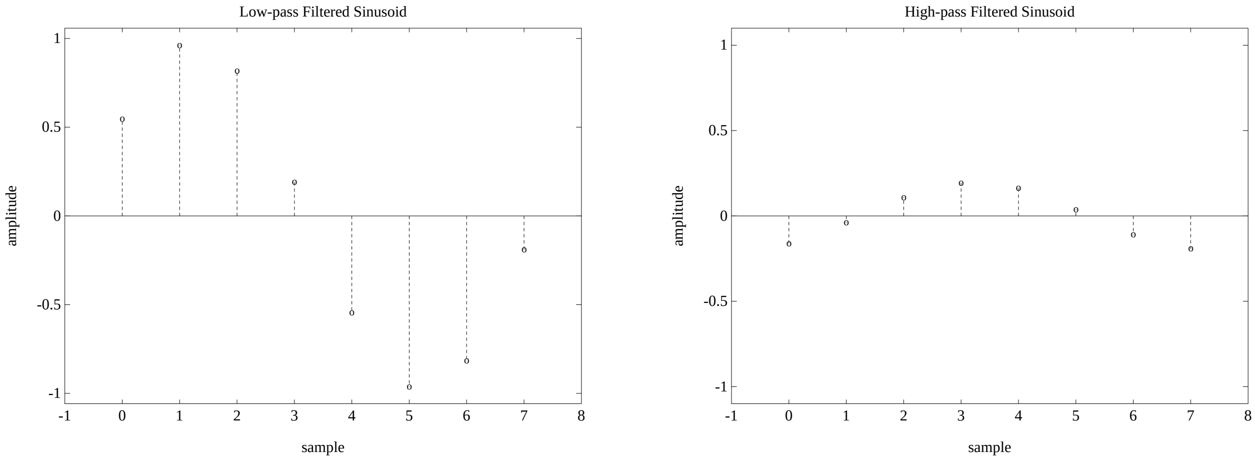

Figure 3: Low- and high-pass outputs from wavelet decomposition of the sampled sinusoid.

high-pass filter obviously contains less information than the low-pass output. If the signal is reconstructed by an inverse low-pass filter of the form

$$f_j^L = \sum_{i=1}^{N/2} c_{2i-j} a_i, \qquad j = 1, \ldots, N, \tag{8}$$

then the result is a duplication of each entry from the low-pass filter output. This is a wavelet reconstruction with $2\times$ data compression. Since the perfect reconstruction is a sum of the inverse low-pass and inverse high-pass filters, the output of the inverse high-pass filter can be calculated; it looks like that shown in Figure 4. This is the result of the inverse high-pass

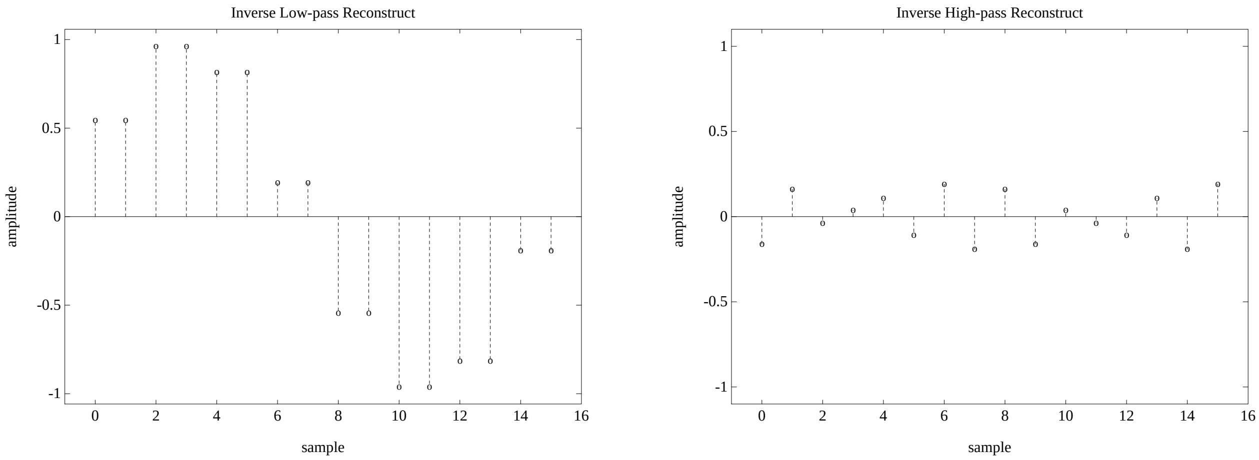

Figure 4: Inverse low- and high-pass filter outputs in sampled sinusoid reconstruction.

filter function

$$f_j^H = \sum_{j=1}^{N/2} (-1)^{j+1} c_{j+1-2i} b_i, \qquad j = 1, \ldots, N. \tag{9}$$

The perfectly reconstructed signal is

$$f = f^L + f^H, \tag{10}$$

where each f is the vector with elements f_j. Using other coefficients and other orders of wavelets yields similar results, except that the outputs are not exactly averages and differences, as in the case using the Haar coefficients.

2.3 Dilation

Since most of the information exists in the low-pass filter output, one can imagine taking this filter output and transforming it again, to get two new sets of data, each one quarter the size of the original input. If, again, little information is carried by the high-pass output, then

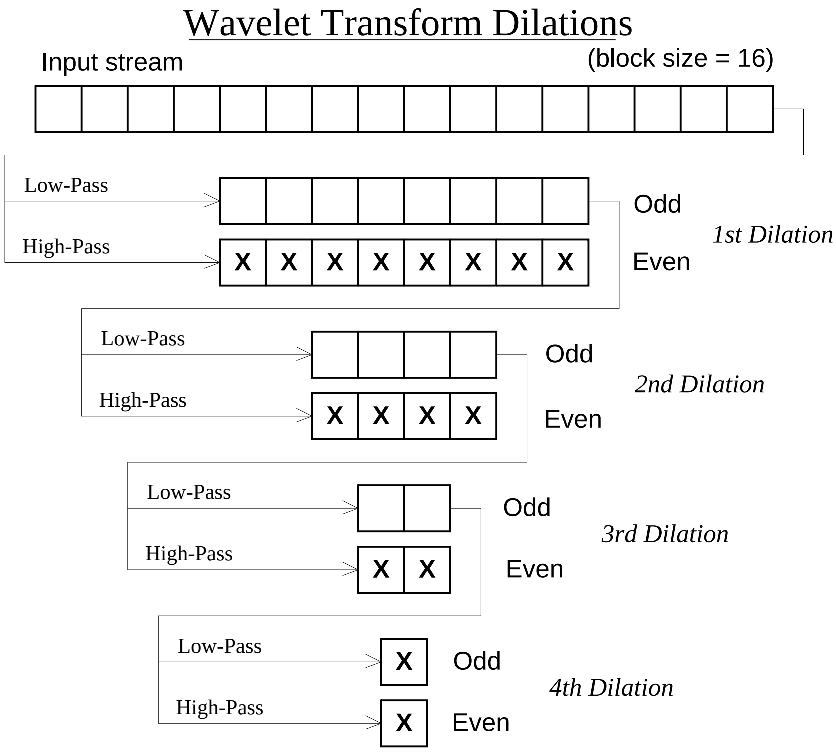

Figure 5: Dilations of a sixteen sample block of data.

it can be discarded to yield $4\times$ data compression. Each step of retransforming the low-pass output is called a *dilation*, and if the number of input samples is $N = 2^D$ then a maximum of D dilations can be performed, the last dilation resulting in a single low-pass value and single high-pass value. This process is shown in Figure 5, where each '**x**' is an actual system output. In essence, the different dilations can be thought of as representing a frequency decomposition of the input. This decomposition is on a logarithmic frequency scale as opposed to the linear scale of the Fourier transform; however, the frequency decomposition does not have the same interpretation as that resulting from Fourier techniques. The frequency dependence can be seen by looking at low-pass outputs of two different waveforms through all dilations, and their reconstruction by Haar interpolation at each stage if all high-pass filter information up to that stage is discarded. The graphs of Figures 6 and 7 were created using only low-pass and inverse low-pass filtering, but no high-pass filtering. Clearly, the low-frequency content of the first signal is maintained after many dilations, whereas the high-frequency content of the other is lost immediately. Note that this is a "frequency" decomposition of the block of fixed

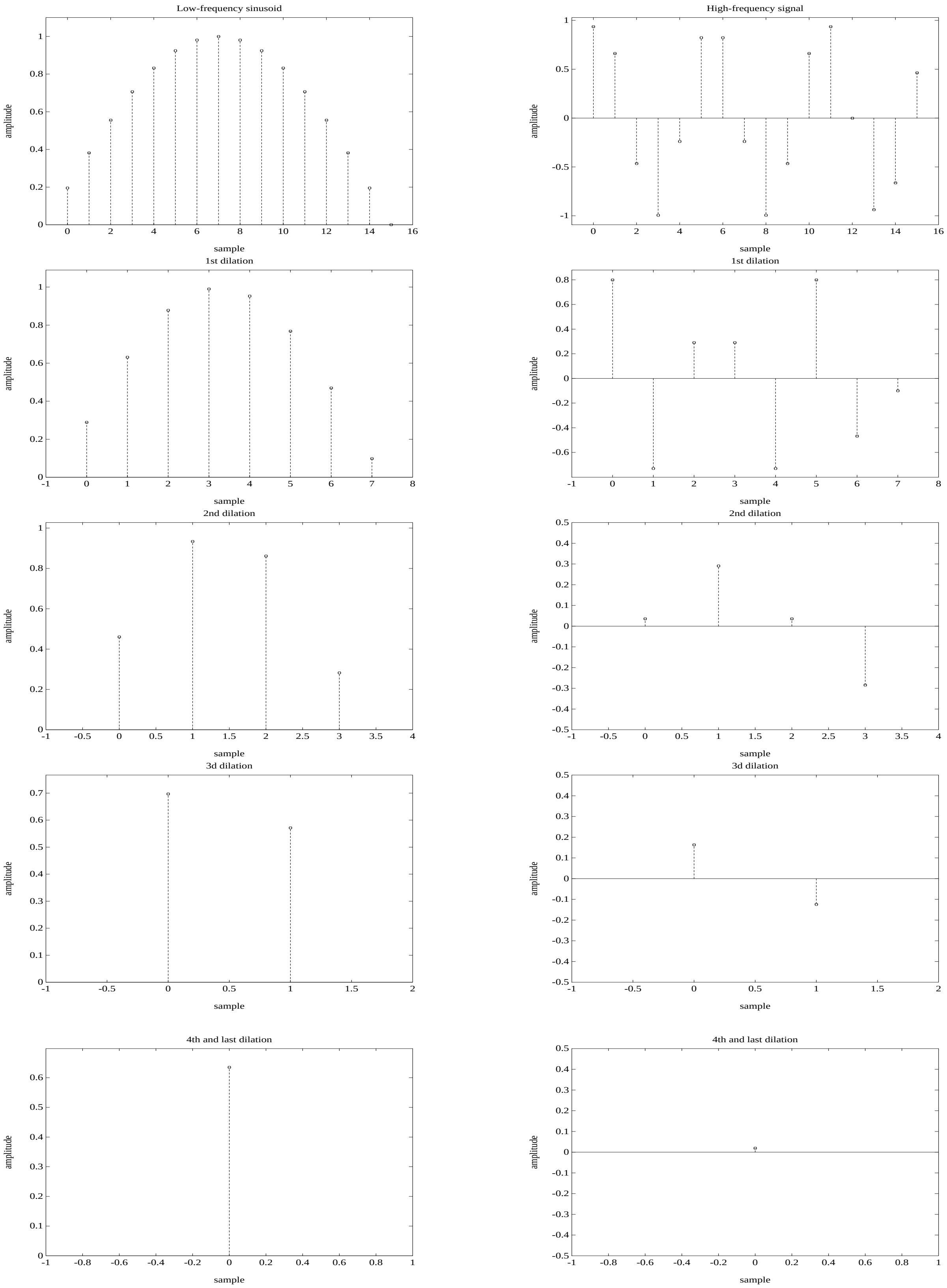

Figure 6: Decompositions of a sixteen sample block of data, using Haar wavelet coefficients.

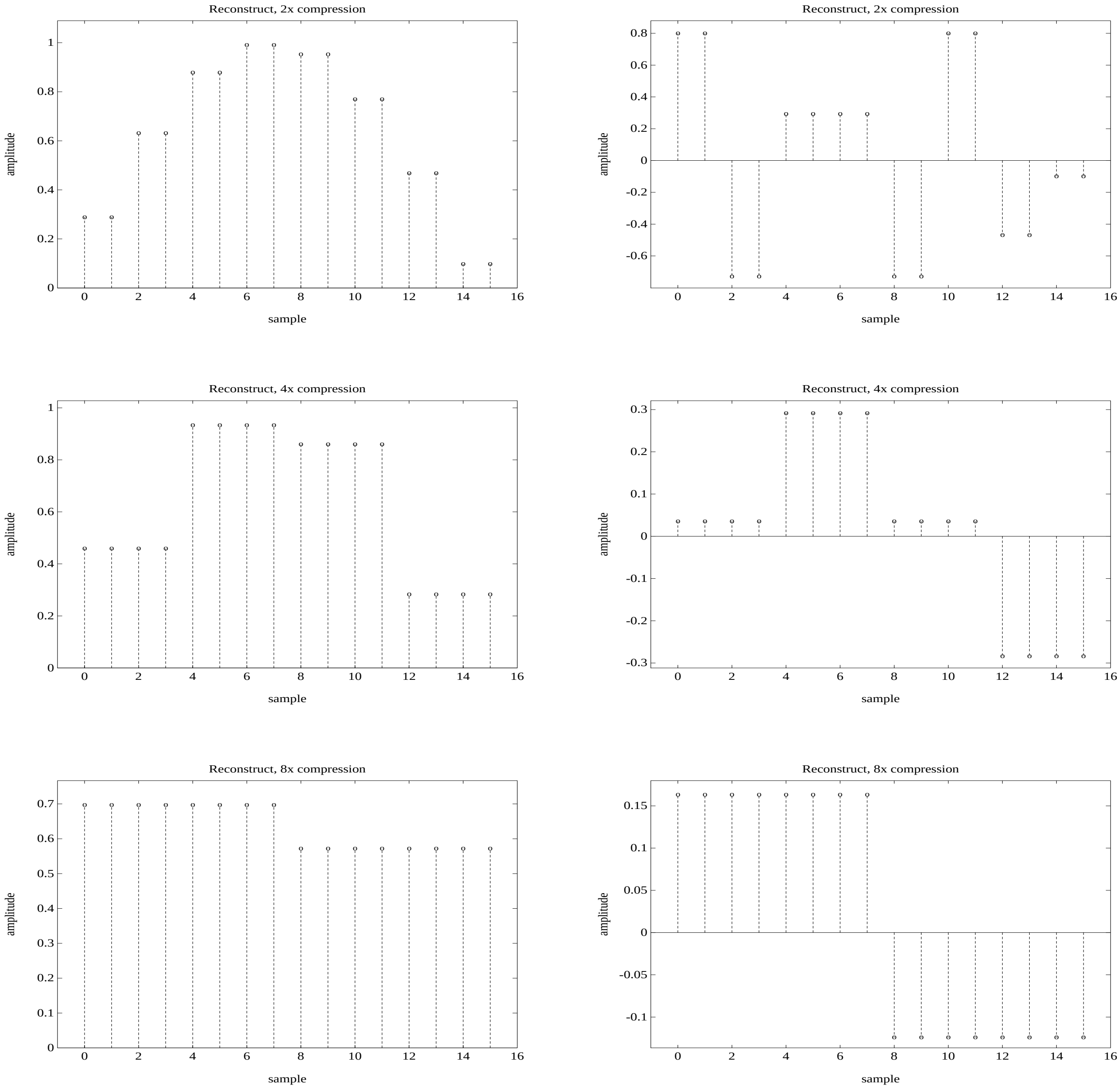

Figure 7: Reconstructions of a sixteen sample block of data, using Haar interpolation and data compression.

length N, so the lowest possible frequency which can be represented by the decomposition is clearly limited by the number of samples in the block, as opposed to the Fourier treatment in which the decomposition includes all frequencies down to zero due to its infinite support. The rule, therefore, is that successive dilations represent lower and lower frequency content by halves. It is also clear that high rates of compression may require large block sizes of the input, so that more dilations can be made, and so that lower frequencies can be represented in the decomposition.

One should note that careful handling of mean values and low frequencies is required for effective compression and reconstruction. This is particularly true for image data, where small differences in mean values between adjacent blocks can give a reconstructed image a "mottled" appearance.

3 Implementing The Wavelet Transformer

The design of the Wavelet Transformer was carried out in six stages, as follows:

1. Simulate the wavelet lattice filters in MATLAB to understand the basic operation of the wavelet transformation, and write C code to determine the proper implementation for a multi-dilation wavelet transformer. Ideally, the basic block-diagram structure of the transformer should remain the same for varying orders of wavelet functions, varying wavelet coefficients, varying input block sizes, and should also maintain the same structure between the decomposition and recomposition filters.

2. Design a simple lattice filter and confirm its function.

3. Create modules or subroutines to perform the operations of the complete wavelet transformer.

4. Create the complete transformer and confirm its function. A possible simple implementation is to have one filter to decompose an input into its wavelet representation, with its output feeding directly into another filter in order to recompose the signal back into its original form. The output can be checked against the input for various signals such as sine waves, square waves, and speech.

5. Optimize the wavelet transformer routines such that the total computation time per input sample-time is as small as possible.

6. Write software to automatically design lattice filters, given certain parameters such as wavelet coefficients, input block size, and number of dilations to decompose the input.

3.1 Simulation

The first stage involved understanding the computation involved in a multi-dilation wavelet transform, and to determine the best structure for the SPROC chip, a digital signal processing chip utilizing parallel processing and pipelining for efficiency. The SPROC chip is basically a RISC processor with an instruction set geared toward DSP applications. MATLAB and C were chosen as simulation environments.

Although it seemed fairly certain that the final version of the wavelet transformer would be a lattice filter, matrix methods (as found in Strang [8]) were studied in order to gain a basic understanding of wavelets, the results of which are presented in the discussion of Section 2. A number of MATLAB programs were available which perform lattice filter functions, some of this code being directly related to the VLSI wavelet processor which has been implemented by Aware, Inc. [2]. These contained ideas about how to optimize wavelet computation for hardware; the mathematical treatments of wavelets tend not to address hardware issues.

The second stage involved compiling routines for lattice filter structures. A basic wavelet lattice filter is shown below; it implements 6^{th}-order wavelet transforms.[3] This corresponds to six coefficients as they appear in the pyramid algorithm described in Section 2.2, but the nature of the lattice filter is such that the six coefficients of the wavelet description appear as three gain factors γ in the lattice and one scaling gain β at the end. There are always half as many "rungs" in the lattice filter as there are coefficients in the pyramid algorithm description. One rung is associated with each γ coefficient, and the placement of the γ gain alternates at each successive rung, as shown in Figure 8. An 8^{th}-order filter would have four

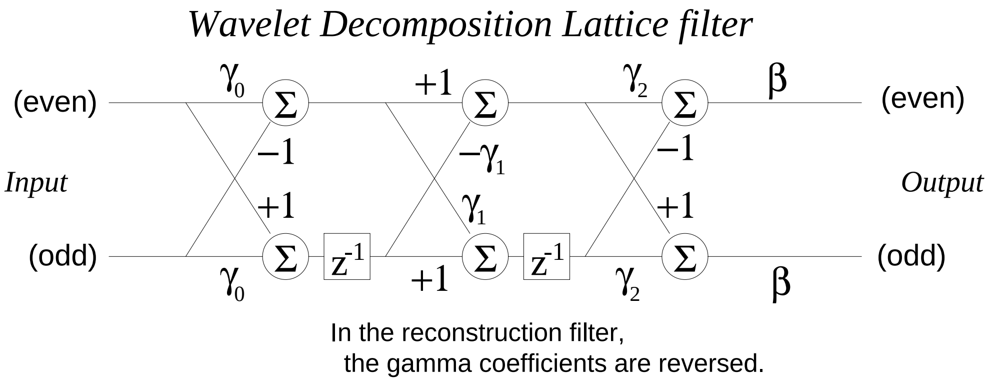

Figure 8: The Lattice Filter block diagram.

rungs, with the structure of γ_1 repeated for the new coefficient γ_3. The filter would have three delay stages.

A major difficulty in realizing this structure within the context of a complete wavelet transformer involves the fact that the lattice filter is an inherently digital structure. It should be ideally suited to a DSP system because a wavelet filter should be able to take analog signals from the outside world, digitize and decompose them, then reconstruct them and send them back to the outside world (presumably with some sort of compression or signal processing performed upon the digital data in the middle of the process). However, such a structure cannot be built efficiently by using simple building blocks. It is necessary to handle tasks such as splitting and recombining the input: If the input is a continuous stream of sampled data, even numbered samples go to the "even" input of the lattice filter, and odd numbered samples go to the "odd" input. Each set of two inputs must be processed by the filter at the same time, which means that the process which implements the filter must operate at half the rate of the A/D and D/A converters which supply the input to the DSP system.

[3]Most of the discussion of Section 3 is based on the 6^{th}-order wavelet but can be easily generalized to other orders.

3.2 Data Handling

Splitting the input stream is only one of a number of data-handling tasks needed to perform a wavelet decomposition. The other tasks are described below.

There are three basic parameters which describe a wavelet transformer, all (essentially) independent of one another. One is the filter length, which reflects the number of coefficients describing the wavelet functions. This affects the number of butterfly filter stages required in the lattice filter, and the number of delays involved. The second is the block size of the input data to be transformed, which must be a power of two. The third is the number of nested levels, or dilations, of transformation; each pass of a data stream through the lattice filter produces two sets of transformed data points, each half as long as the original. To transform one level deeper the odd-numbered outputs, which represent the low-frequency data of the input, are processed through another filter which again produces half-length outputs. If the original input stream is segmented into blocks of size 2^D, then the result of the D^{th} level of transformations is a single data point, and further splitting of the data is impossible. Usually, a full decomposition of this kind is not desired, but for good data compression, a number of dilations will be needed. For thoroughness, the original SPROC implementation was required to perform all possible dilations.

The unfortunate property of wavelet reconstruction is that the deepest nested dilation calculated in the decomposition must be the first to be reconstructed. This means that all transformed data must be saved in memory and output in the reverse order in which they were originally calculated (level-wise, that is; within a dilation, the data must be output in the order in which they were calculated). So the size of the input blocks and the resolution of the wavelet decomposition are memory-limited. Most of the work done by the wavelet transformer is scheduling the filter(s) and managing the inputs and outputs.

The above story gets even more complicated. There is a fundamental problem with lattice filters: Unlike the convolution equation which is the best mathematical description of wavelets, the lattice filter doesn't really produce even and odd output streams that are half the length of the input; the use of delay stages results in output streams which are half the length of the input *plus* the number of delay stages. It is not desirable to output all these extra numbers, because that would require an increasingly greater rate of processing for the output of each decomposition level; additionally, it is not desirable because the convolution description of wavelets assures that a full decomposition of the wavelet *can* be performed which always produces the same number of output data as input data. There are two ways to resolve this problem; one is to truncate the output and discard the extra endpoint values. Done correctly, this gives a reconstruction which is very close to the original, but not quite exact. The other method involves assuming a "circular" definition of the input, in which case the input stream is wrapped around and fed back through the filter, and the new outputs replace the first outputs (which were calculated on the same data but before all the delay stages were filled with nonzero values). This method yields a perfect reconstruction, but makes scheduling and data routing even more of a headache.

It should be noted that recirculation of the input causes the transform to "see" high-frequency kinks at the boundary between the endpoints of the data block. This prevents the even output response from being perfectly flat and so interferes with data compression. However, the number of transform output values affected by the recirculation is equal to the number of delays in the lattice filter. Good data compression can still be achieved by not discarding the affected values, and the proportion of these values to the total number of even outputs becomes negligible for large block sizes; i.e., for a Daubechies-6 wavelet (which has two delay elements) operating on a block of size 16 and yeilding an even output of size eight, six of these values may be discarded for $16/(8+2) = 1.6$ times compression for that dilation instead of the usual $2\times$ compression. However, if the block size is 256 and the even output is size 128, then 126 of these values may be discarded for $256/(128+2) \approx 1.97$ times data compression for that dilation.

3.3 Data Pointers and Block I/O Scheduling

In order to perform the proper handling of the data as described above, a system was needed to transfer the proper values back and forth to the lattice filter. The circular-definition problem necessitated the use of two filters in parallel in order that the output could keep up with the input for the particular set of parameters chosen for this implementation. Formally, the number of filters required is

$$\#\text{filters} = \lceil \frac{1}{2^N} \sum_{d=1}^{D} (2^{N-d} + \Delta) \rceil,$$

where N is the input block size, D is the number of dilations to be performed, and Δ is the number of delay stages per filter.

The delay stages in the lattice filter were discarded and replaced with general inputs and outputs due to the realization that *all* stored values should be handled in the same manner. Since every two inputs are fed to the lattice filter, the first dilation only makes 50% utilization of the filter. In the intervening sample times, the filter can perform other dilations on previous input blocks, but the delay from one calculation must appear at the next filter calculation of *its dilation*, and must skip over the intervening calculation. Accordingly, handling the delay stages becomes as complicated as handling the filter inputs and outputs, so the decision to incorporate both into the same structure is a natural one.

The structure chosen for the wavelet transformer is shown in block diagram form in Figure 9. It makes use of a block of shared memory, which is allocated in the (temporally) first module to use it, which is called the "shift register." By use of the shift register memory space, the data handling works like an assembly line, in which inputs are placed at one end, outputs are taken off the other end, and spaces in between are used to store and retrieve intermediate results. Two modules, called the input and output schedulers, are responsible for determining the location of the proper values in the shift register and transferring data from the memory to the filter inputs and from the filter outputs to the memory, respectively. The proper memory location is found by means of a lookup table. Since a "delay" can be accomplished by placing a value in the shift register on one turn and picking it up further down the register one or more sample-times later, the delays are also handled by the input and output schedulers, and requires a simplified lattice filter as shown in Figure 10.

The block-diagram structure of the wavelet transformer is independent of the block size of the input data. There must be one lookup-table entry for each input and output for each sample time up to the total number of samples in one input block. Larger blocks take more space with the lookup tables and shared memory. The sample time is counted out by the module called the counter. Because the shift register memory is shared, it remains in its allocated space, and its starting address is passed as a pointer from module to module.

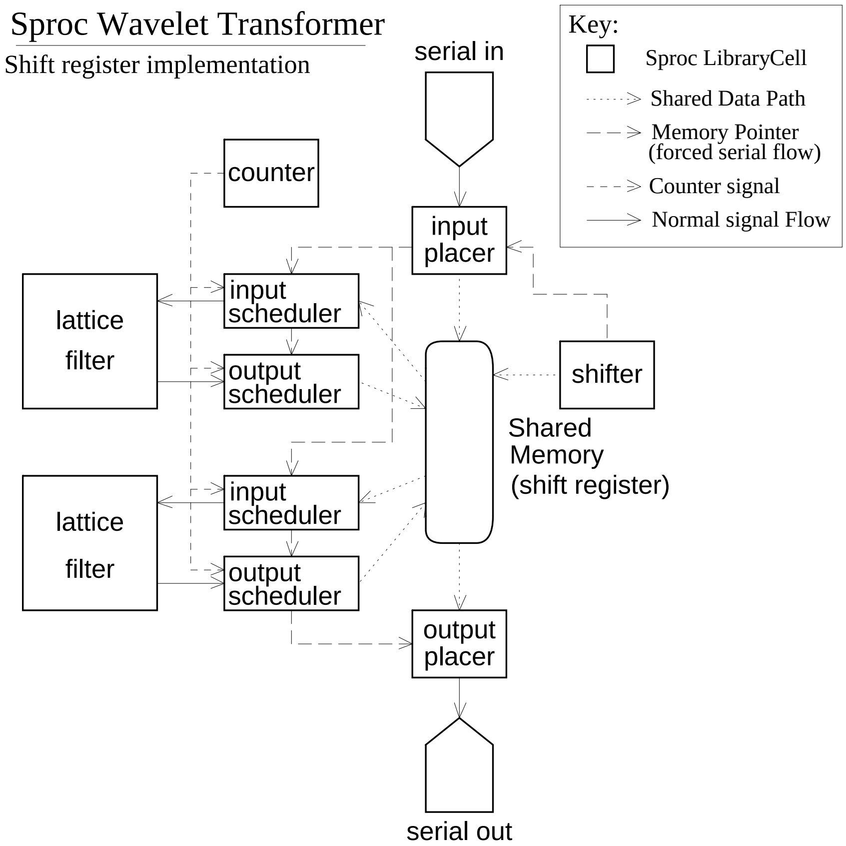

Figure 9: Block diagram of the SPROC Wavelet Transformer

3.4 Creating the Lookup Tables

This form of the wavelet transformer has put all of the design complexity into the process of generating the lookup tables for the input and output schedulers. This is a routing problem, and looks very much like the problem encountered when trying to place wires in a gate array. Heuristics such as those used for programming gate arrays can be applied in order to automate the process of creating the tables. The diagrams in Figures 11 and 12 and Table 2 outline this process. Figure 11 shows the creation of the reconstruction filter. Arbitrary numbers are assigned to label all inputs and outputs of two lattice filters and those of the reconstruction filter as a whole. This is done for each of the dilations in turn, always using the first available lattice filter after the inputs for each calculation become available. Note that the first two outputs of each level (dilation) are not assigned. This is due to the circular-input-data effect; the first two inputs are always re-routed through the lattice filter after the other inputs have been taken care of, while the first two outputs are discarded. The unnumbered lines in the table are cycles in which the filter is not used. Note that the first output appears twelve cycles after the first input. This, added to the hardware delay, gives the total throughput delay of the wavelet reconstruction filter.

Figure 12 shows one solution to the routing problem. This is accomplished with a modified "left-edge" algorithm, which works in the following manner: First, the output nets are placed; all of these nets have one end fixed at position zero. Next, the shift register size is made just

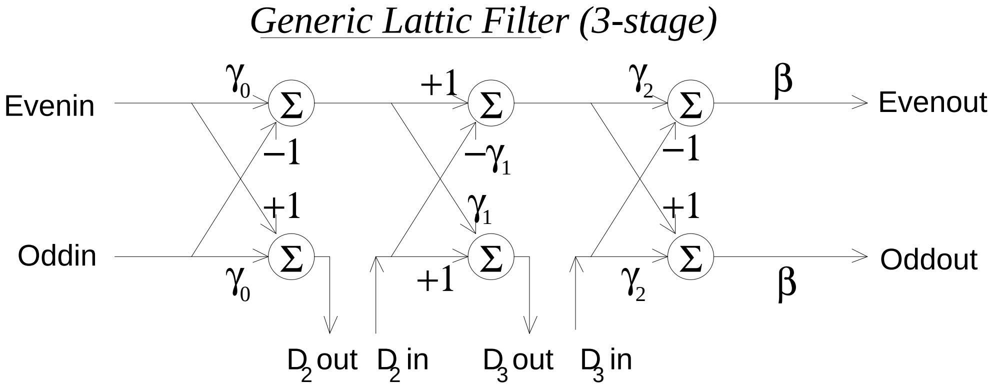

Figure 10: Generic Lattice Filter

large enough to accomodate the output nets, and the input nets are placed; these have one end fixed at the top of the shift register. If a collision occurs between an input and output net, the shift register top is raised by one location, and the process continues until all of the inputs and outputs fit into the shift register. Finally, all the other nets are placed in a similar manner to the input nets, starting with those nets which begin at step zero. These nets are placed in the first available spot counting from the top of the shift register. If no space is available, the shift register size is again increased by one and the process is repeated. Note that since the input scheduler module runs before the output scheduler, new output values can be placed in the same memory location as input values which were used for the last time on the same "step."

Table 2 is made directly from the numbers in Figures 11 and 12, to match each of the filter inputs and outputs with its exact memory location at any given moment in sample-time. The delays are determined in a similar manner; part of this process is not shown in the diagrams. The shift register actually has two more locations than shown in Figure 12; the extra two are placed at the top of the shift register, one to hold the value zero for resetting the lattice filters, and the other as a "discard pile" for calculated results which are not used.

As an example of how values are placed in these diagrams, look at the tenth input in Figure 11. This input has been given the label "9." According to the schedule, this input appears at step 9, and is used at step 11 (as counted by the module `counter`), where it is the odd input to the second lattice filter. Because it is part of the second set of inputs to the topmost dilation of the reconstruction, it must be recirculated through the lattice filter at the end of the calculations for that dilation. The schedule shows that this occurs at the second filter on step 3. Figure 12 shows the path of this net, which starts as an input at step 9, goes to step 11, wraps back around to step 0, and ends at step 3. It starts at memory location 23 and ends at location 13, having been shifted by this amount. Finally, this network is logged in the lookup table of Table 2. At step 11, the net is used as the *oddin* value for filter 2, and the network is found at memory location 21 as seen from Table 12. This memory location is the value put in the lookup table. The same procedure is needed for the *oddin* value for filter 2 at step 3. The input is automatically taken care of, since the input is always placed at the top of the memory block (actually two places below the top; see above), or at memory location 23.

Initially, it was planned to create one transformer, using the Daubechies-6 wavelet, a data block size of sixteen, and performing all four possible dilations of the input. A *C* program

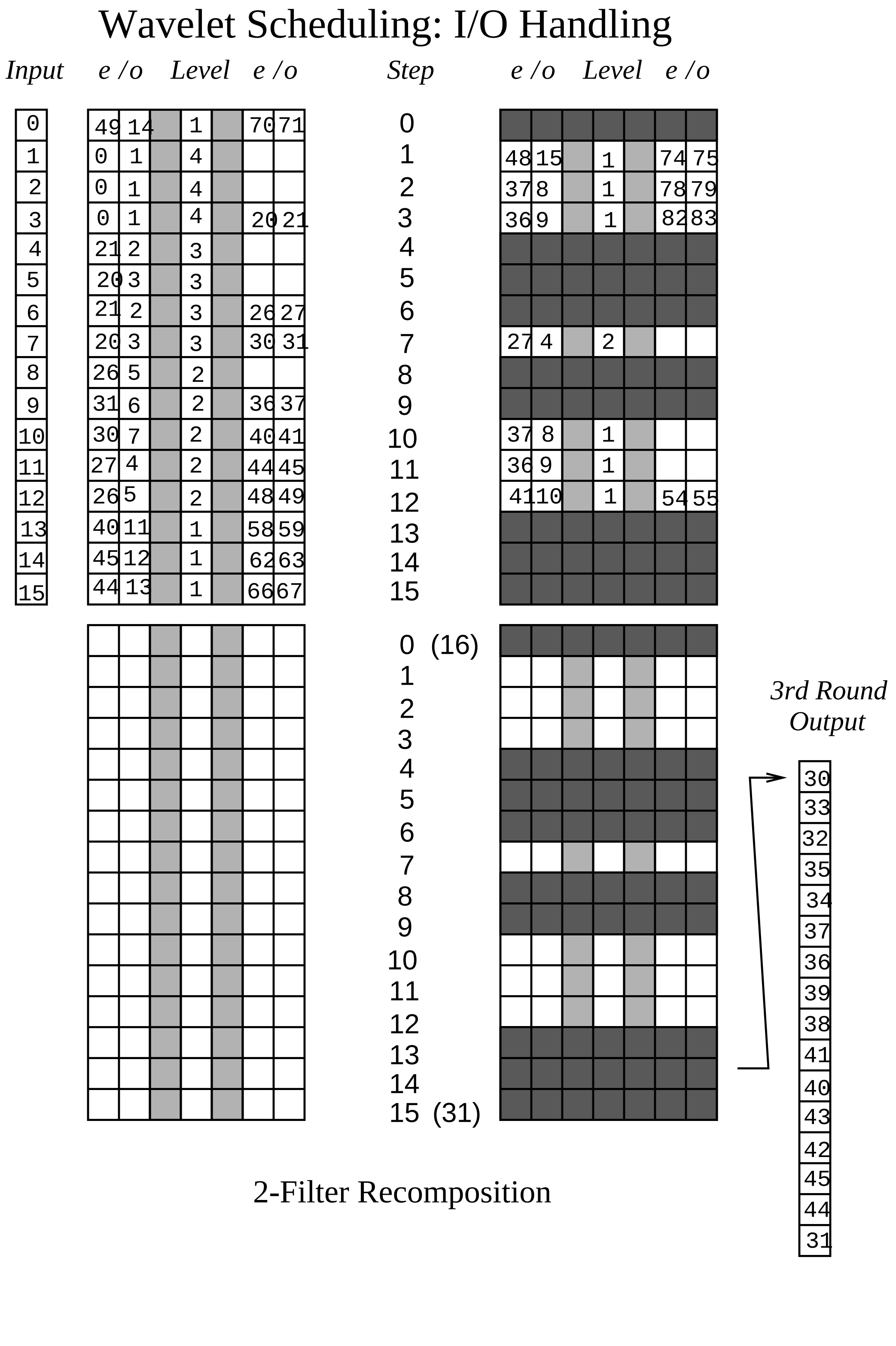

Figure 11: Solving the scheduling problem by hand: I/O handling. The two columns represent two lattice filters operating in parallel.

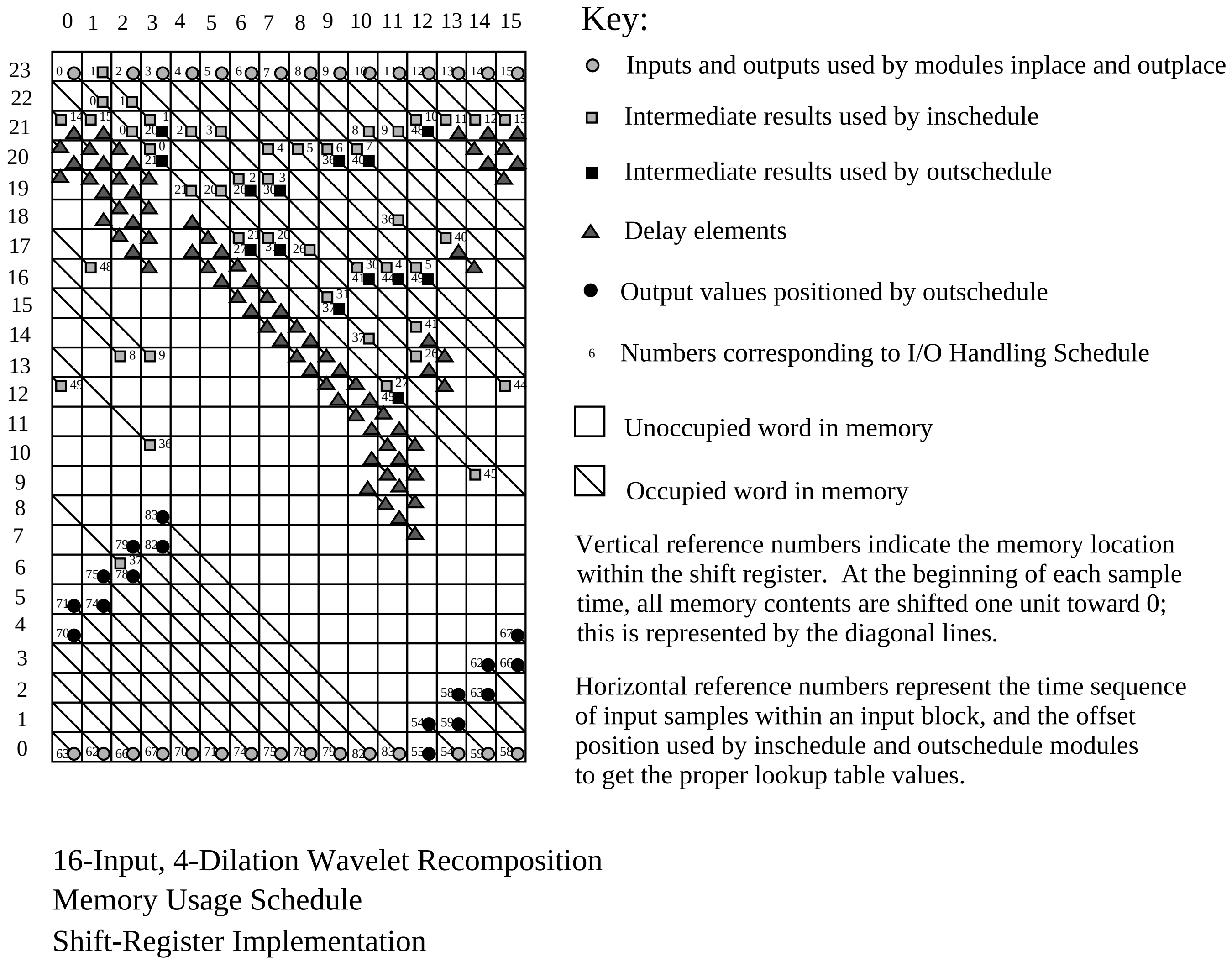

Figure 12: Solving the scheduling problem by hand: Filling memory.

Step:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data for file "`recin1.dat`" (Filt1) and "`recin2.dat`" (Filt2)																
	Evenin															
Filt1	21	20	19	19	17	17	17	15	16	12	13	17	9	12	12	22
Filt2	6	10	25	25	25	16	25	25	21	21	21	25	25	25	25	21
	Oddin															
Filt1	22	21	21	21	19	19	20	20	20	16	16	21	21	21	21	23
Filt2	13	13	25	25	25	20	25	25	21	21	21	25	25	25	25	21
	Delay2in															
Filt1	20	19	24	17	16	15	14	13	12	11	10	13	20	20	20	24
Filt2	18	17	25	25	25	24	25	25	24	9	8	25	25	25	25	20
	Delay3in															
Filt1	19	18	24	16	15	14	13	12	11	10	9	12	16	19	19	24
Filt2	17	16	25	25	25	24	25	25	24	8	7	25	25	25	25	19
Data for file "`recout1.dat`" (Filt1) and "`recout2.dat`" (Filt2)																
	Evenout															
Filt1	25	21	25	25	19	19	25	20	20	16	21	2	3	4	5	25
Filt2	7	8	25	25	25	25	25	25	25	25	1	25	25	25	25	6
	Oddout															
Filt1	25	20	25	25	17	17	25	15	16	12	16	1	2	3	4	25
Filt2	6	7	25	25	25	25	25	25	25	25	0	25	25	25	25	5
	Delay2out															
Filt1	20	25	18	17	16	25	14	13	12	11	25	21	21	21	21	21
Filt2	18	25	25	25	25	15	25	25	10	9	14	25	25	25	25	19
	Delay3out															
Filt1	19	25	17	16	15	25	13	12	11	10	25	17	20	20	20	20
Filt2	17	25	25	25	25	14	25	25	9	8	13	25	25	25	25	18

Table 2: Solving the scheduling problem by hand: Making lookup tables

implementing this system is listed in Appendix B. By using a program which automates the process of creating lookup tables, changes to the block size or number of dilations performed require little work. New wavelet coefficients can be generated by a number of methods found in the literature, and installed in the existing structure. (Note Figure 13, where the lattice coefficients are entered as parameters of each lattice filter module.) Higher-order wavelets such as the Daubechies-8 require new routines for the lattice filter having extra "rungs," or else a lattice filter routine which uses the wavelet order as one of its parameters, and generates the proper lattice structure automatically. All of the scheduling for new sizes of the lattice filter changes due to the change in the number of delay elements.

The final program revisions made the size of the code independent of the block size of the wavelet transform. Before the revisions, the implementation called for a shift register, the size of which was roughly proportional to the block size of the input data to be transformed. By having a loop which moved the data one place ahead in memory, the size of that segment of code, when executing, was proportional to the size of the shift register. This problem was eliminated by using a pointer to indicate the beginning of the register, and shifting the pointer instead of the data. The only tradeoff was that each read/write of the shift register must take into account the fact that the given address may exceed the allocated memory space of the shift register, and therefore must wrap around to the beginning of the register. The result of this was that more code locations were needed to do the extra wraparound calculations, but the time saved over shifting all the register data, even for fairly small registers, was quite significant.

3.5 Running the Transformer

The wavelet transformer as it was finally implemented corresponds to Figure 9 except that the structure was duplicated to have both the decomposition and reconstruction parts in the same system, with the output of the decomposition section becoming the input to the reconstruction section, and the output of the reconstruction going to the serial output module. The schematic of this system as built with SPROClab is shown in Figure 13. In reference to this schematic, `altinsch` is an input scheduler, `altoutsc` is the output scheduler, `altinpl` handles the system input, and `altoutpl` handles the system output. The `plug` module is an assembler routine with zero lines of code; it terminates an output which is not needed. The "`spec=`" parameter on the input and output scheduler modules indicates an *ASCII* file containing the lookup table values, in the order found in Table 2. The "`memsize=`" parameter indicates the length of the shift register. There are two shift registers in this schematic: One is for the decomposition filter and one is for the reconstruction filter. Note that `gamma` coefficients in the diagram are half the value of those listed above. This is due to numerical limits of the hardware, and this division by two is compensated for within the lattice filter module.

The system is run by connecting any signal source to the serial input port, and comparing it to the reconstructed output from the serial output port. The decomposed wavelet outputs can be viewed by probing the output of the decomposition filter. This signal is the concatenation of all four dilations, which unfortunately are not easily broken up into components for viewing on an oscilloscope, but one can get a fairly good idea of what is occurring at each dilation from the signal as it appears. The wavelet output is not static, but constantly changes as long as the frequency of the input signal does not equal the rate of the A/D converters divided by the input block size (e.g., sixteen). This demonstrates the difference between the Fourier and the wavelet concepts of frequency distribution; for a continuous input signal from a function generator, the frequency distribution is static over infinite time, but the frequency distribution over a short period of time is not, unless the period of time happens to cover exactly an integral number of cycles of the function.

A number of properties of discrete wavelets can be immediately confirmed by the transformer. Changing the dc bias voltage of the incoming signal affects the low-pass output of the last dilation; in the case that all possible dilations are performed, only a single value in the dilation changes. When the frequency of the input is very low, the outputs of the first one or two dilations are nearly zero, but increase in amplitude with increasing frequency. The Daubechies-6 function produces a much flatter response over a larger frequency range than the Haar function for most input waveforms. When the Haar wavelet is used to decompose a low frequency square wave, the resultant wavelet outputs are all zero except for the low-pass output of the last dilation. This is expected due to the square wave shape of the Haar function.

References

[1] Aszkenasy, R. "A Cascadable Wavelet Chip for Image Compression: Theory, Design, and Practice," (unpublished note), 1991.

[2] Aware, Inc. "Aware Wavelet Transform Processor (Preliminary)," 1991.

[3] Daubechies, I. "Orthonormal Bases of Compactly Supported Wavelets," *Comm. Pure Applied Mathematics*, vol. 41, 1988, pp. 909–996.

[4] Gabor, D. "Theory of Communication," *J. Inst. of Electr. Eng.*, vol. 93, 1946, pp. 429–457.

[5] Haar, A. "Zur Theorie der Orthogonalen Funktionensysteme," *Mathematics Annal.*, vol. 69, 1910, pp. 331–371.

[6] Mallat, S. "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 11, 1989, pp. 674–693.

[7] STAR Semiconductor Corp. *SPROClab Development System User's Guide*, STAR Semiconductor, Warren, NJ, 1990.

[8] Strang, G. "Wavelets and Dilation Equations: A Brief Introduction," *SIAM Review*, vol. 31, no. 4, December 1989, pp. 614–627.

[9] Vaidyanathan, P., "Lattice Structures for Optimal Design and Robust Implementation of Two-Channel Perfect Reconstruction QMF Banks," *IEEE Trans. ASSP*, vol. 36, No. 1, January 1988, pp.81–86.

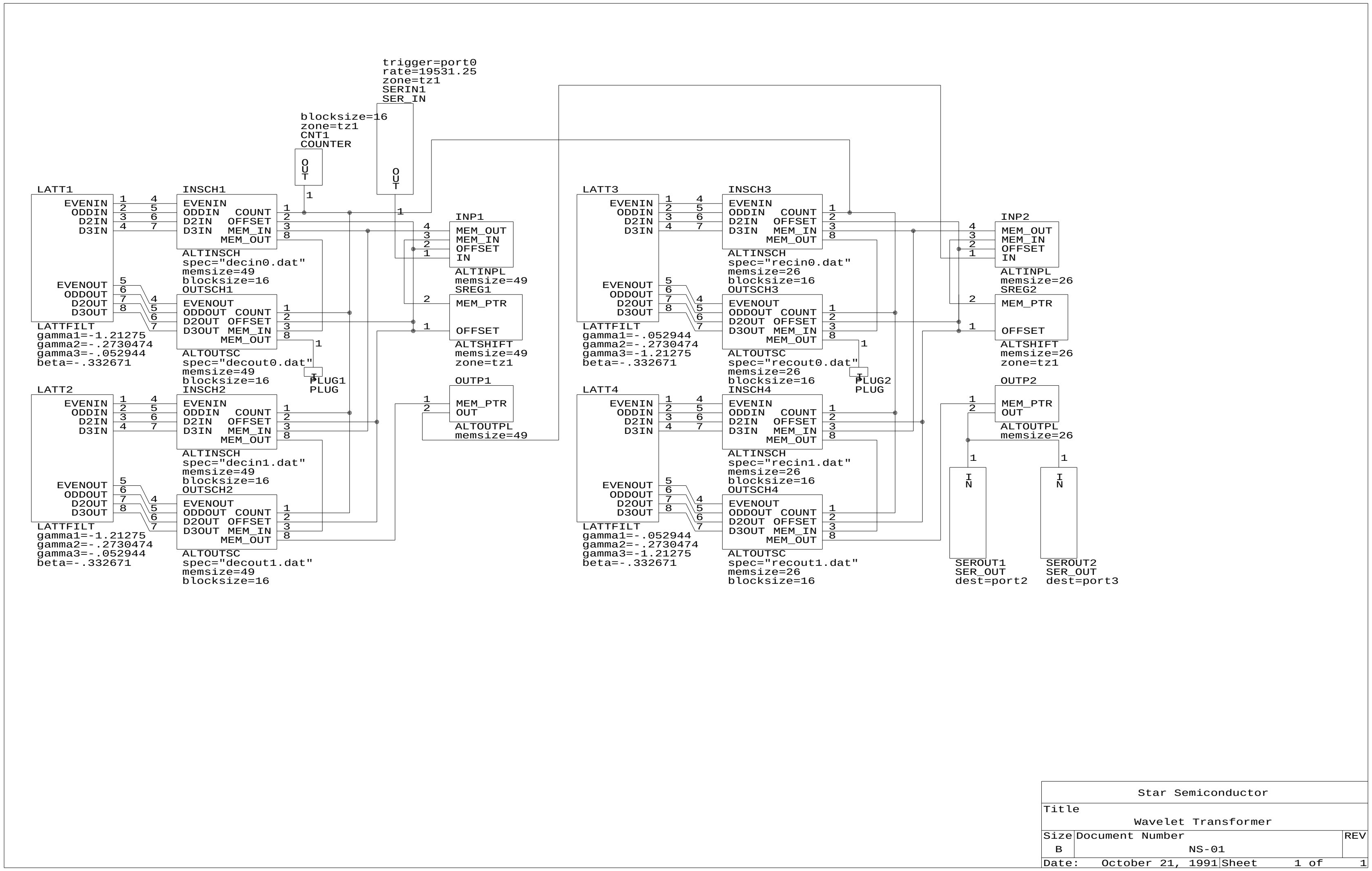

Figure 13: *OrCAD* schematic of the wavelet transformer.

Appendices

A Pyramid Algorithm to Lattice Filter: A Derivation

A.1 Deriving the Lattice Structure

Deriving the lattice filter structure is not a trivial task. This appendix will show the necessary steps, starting with the Mallat pyramid algorithm as presented by Strang [8], needed to achieve a form from which Vaidyanathan [9] has shown that the lattice filter can be constructed.

We begin with the four filter functions, the two decomposition filters (6) and (7) and the two reconstruction filters (8) and (9). From here, we take a direct z-transform, and manipulate it in a manner similar to that used for a proof of the convolution property of the z-transform. For the low-pass filter,

$$\begin{aligned} a(z) &= \sum_{n=-\infty}^{\infty} \frac{1}{2} \sum_{j=0}^{M-1} c_{2n-j} f_j z^{-n}, \qquad (11)\\ &= \frac{1}{2} \sum_{j=0}^{M-1} f_j \sum_{n=-\infty}^{\infty} c_{2n-j} z^{-n}, \\ &= \frac{1}{2} \sum_{j=0}^{M-1} f_j \left\{ \sum_{m=-\infty}^{\infty} c_m z^{-\frac{m}{2}} \right\} z^{-\frac{j}{2}}, \\ a(z^2) &= \frac{1}{2} \sum_{n=-\infty}^{\infty} c_n z^{-n} f(z). \end{aligned}$$

The z^2 term implies that a is half the length of f. M is the order of the filter. Note that all arrays are indexed from zero for simplicity. The above equation can be rewritten as a low-pass operator,

$$a(z^2) = L(z) f(z)$$

where

$$L(z) = \frac{1}{2} \sum_{n=0}^{M-1} c_n z^{-n}. \qquad (12)$$

Here, we have changed the limits in order to sum over the nonzero coefficients only. The same steps can be applied to the high-pass filter, and this is presented below:

$$\begin{aligned} b(z) &= \sum_{n=-\infty}^{\infty} \frac{1}{2} \sum_{j=0}^{M-1} (-1)^j c_{j+1-2n} f_j z^{-n}, \qquad (13)\\ &= \frac{1}{2} \sum_{j=0}^{M-1} f_j \sum_{n=-\infty}^{\infty} (-1)^j c_{j+1-2n} z^{-n}, \\ &= \frac{1}{2} \sum_{j=0}^{M-1} f_j \left\{ \sum_{m=-\infty}^{\infty} (-1)^{2n-m-1} c_{-m} z^{-\frac{m}{2}} \right\} z^{-\frac{j}{2}} z^{-\frac{1}{2}}, \\ b(z^2) &= \frac{1}{2} \sum_{n=-\infty}^{\infty} (-1)^n c_{1-n} z^{-n} f(z). \\ b(z^2) &= H(z) f(z) \end{aligned}$$

where

$$H(z) = \frac{1}{2}\sum_{n=0}^{M-1}(-1)^n c_{1-n} z^{-n}. \tag{14}$$

This procedure can also be applied to the inverse filter functions to get

$$\begin{aligned} L^*(z) &= \sum_{n=0}^{M-1} c_n z^n, \\ &= L(z^{-1}), \end{aligned} \tag{15}$$

$$\begin{aligned} H^*(z) &= \sum_{n=0}^{M-1}(-1)^n c_{1-n} z^n, \\ &= H(z^{-1}). \end{aligned} \tag{16}$$

Two proofs are needed to show that these transforms represent quadrature mirror filters. The first invokes the condition (4) and is:

$$\begin{aligned} L^*L + H^*H &= L(z)L(z^{-1}) + H(z)H(z^{-1}) \\ &= \left[\frac{1}{2}\sum_{n=0}^{M-1} c_n z^{-n}\right]\left[\sum_{n=0}^{M-1} c_n z^n\right] \\ &\quad + \left[\frac{1}{2}\sum_{n=0}^{M-1}(-1)^n c_{1-n} z^{-n}\right]\left[\sum_{n=0}^{M-1}(-1)^n c_{1-n} z^n\right] \\ &= \frac{1}{2}\sum_{n=0}^{M-1} c_n^2 + \frac{1}{2}\sum_{n=0}^{M-1} c_{1-n}^2 \\ &= \frac{1}{2}(2) + \frac{1}{2}(2) = 2 = \text{constant}. \end{aligned} \tag{17}$$

The second relates $L(z)$ to $H(z)$:

$$\begin{aligned} L(z) &= \frac{1}{2}\sum_{n=-\infty}^{\infty} c_n z^{-n} \\ L(z^{-1}) &= \frac{1}{2}\sum_{n=-\infty}^{\infty} c_{-n} z^{-n} \\ z^{-(M-1)}L(z^{-1}) &= \frac{1}{2}\sum_{-\infty}^{\infty} c_{-n-(M-1)} \\ z^{-(M-1)}L(-z^{-1}) &= \frac{1}{2}\sum_{n=0}^{M-1}(-1)^n c_{1-n} \\ z^{-(M-1)}L(-z^{-1}) &= H(z). \end{aligned} \tag{18}$$

Now the equations (17) and (18) match the requirements cited in [9] for a quadrature mirror filter, or lattice structure. Reference the Vaidyanathan paper [9] for the remainder of the proof.

If there appears to be a subscript problem in the above equations, it is because the Mallat pyramid algorithm works *only* if a circular definition of coefficients is assumed; i.e., in the case of the high-pass filter, the coefficients of subscript $(1-n)$ will never match up with the input function f_n unless either the coefficient space or the input function space wraps around on itself. In this case we have defined $c_n = c_{n+M}$. This fact is often passed over in the literature.

A.2 Confirming the Daubechies-6 Lattice Structure

Although the derivation of the lattice structure is difficult, a proof that the 6th-order lattice structure is equivalent to the Mallat pyramid algorithm is fairly straightforward, if somewhat messy. By directly working out the equations of the lattice filter outputs as a function of the inputs, working in z-transform space with each delay unit contributing z^{-1}, we get the following for decomposition:

$$\begin{aligned} e_o &= \Big[o_i\gamma_0 z^{-2} + o_i\gamma_1 z^{-1} - o_i\gamma_0\gamma_1\gamma_2 z^{-1} - o_i\gamma_2 \\ &\quad - e_i\gamma_0\gamma_2 z^0 - e_i\gamma_0\gamma_1 z^{-1} - e_i\gamma_1\gamma_2 z^{-1} + e_i z^{-2}\Big]\beta, \qquad (19) \\ o_o &= \Big[o_i\gamma_0\gamma_2 z^{-2} - o_i\gamma_0\gamma_1 z^{-1} - o_i\gamma_1\gamma_2 z^{-1} - o_i \\ &\quad + e_i\gamma_0 + e_i\gamma_0\gamma_1\gamma_2 z^{-1} - e_i\gamma_1 z^{-1} + e_i\gamma_2 z^{-2}\Big]\beta, \qquad (20) \end{aligned}$$

and two similar equations for reconstruction, except that the γ coefficients are reversed ($\gamma_0 \rightleftharpoons \gamma_2$) due to differences in structure of the reconstruction filter. Here e_i and o_i stand for the even and odd inputs, respectively, and e_o and o_o stand for the even and odd outputs, respectively (reference Figure 8).

When performing the inverse z-transform of the decomposition filter, we take into account the fact that the input is split into even and odd components. To an input function $f(n)$, each term o_i contributes $(n-0)$ and each term e_i contributes $(n-1)$. Each z^{-1} contributes $(n-2)$. The even output e_o becomes the high-pass filter output $a(n)$, and the odd output o_o becomes the low-pass filter output $b(n)$. The equations are:

$$\begin{aligned} b_{n/2} &= (\beta)f_{n-5} + (\gamma_0\beta)f_{n-4} + (\gamma_2\gamma_1\beta + \gamma_1\gamma_0\beta)f_{n-3} \\ &\quad +(\gamma_1\beta - \gamma_0\gamma_1\gamma_2\beta)f_{n-2} + (\gamma_0\gamma_2\beta)f_{n-1} - (\gamma_2\beta)f_n, \qquad (21) \\ a_{n/2} &= (\gamma_2\beta)f_{n-5} - (\gamma_0\gamma_2\beta)f_{n-4} + (\gamma_0\gamma_1\gamma_2\beta - \gamma_1\beta)f_{n-3} \\ &\quad -(\gamma_0\gamma_1\beta + \gamma_1\gamma_2\beta)f_{n-2} + (\gamma_0\beta)f_{n-1} - (\beta)f_n. \qquad (22) \end{aligned}$$

These equations are evaluated at even n only. Similar equations can be made from the reconstruction filter equations, assuming that o_i becomes $b(n)$ and that e_i becomes $a(n)$; each z^{-1} contributes $(n-1)$. The even outputs e_o become $f(n)$ for even n only, and the odd outputs o_o become $f(n)$ for odd n only.

The last necessary step of the proof is to show that while equation (22) can be written

$$a_n = \frac{1}{2}\sum_{j=1}^{N} c_{2n-j+1} f_j, \qquad n = 1, \ldots, \frac{N}{2},$$

there are only as many sum terms as there are nonzero coefficients, so this can be validly rewritten as a sum over the nonzero coefficient space, as given in the pyramid algorithm equations. If this is done, then the coefficients in (22) correspond directly to the pyramid algorithm coefficients (except for a scaling factor), and the two forms are equivalent. The purpose of the scaling factor mentioned above is to make the overall scaling the same for the reconstruction and decomposition filters. Note that in the Mallat pyramid algorithm discussion, decomposition filter outputs were scaled by 2, and the recomposition outputs by 1. For more regularity, this factor has been evenly distributed between the decomposition and reconstruction as the constant $1/\sqrt{2}$.

The above equations can be easily solved for a one-rung lattice filter in order to find the lattice coefficients for the Haar wavelet, which cannot be computed by some methods currently in use. These coefficients are shown below, along with the solution to the above equations for the 6$^{\text{th}}$-order Daubechies wavelet.

Wavelet	γ_0	γ_1	γ_2	β
Haar	1.0	0.0	0.0	$1/\sqrt{2}$
Daubechies-6	-2.425492	-0.546095	-0.105888	-0.332671

Table 3: Lattice Coefficients for two wavelet functions.

B Wavelet Transform Program

```
/* Daub6ds.c -- a wavelet decomposition program for the following */
/* parameters:                                                     */
/*     Input block size    = 16                                    */
/*     Number of dilations = 4                                     */
/*     Wavelet type        = Daubechies 6th-order                  */
/* Written by Tim Edwards, June 1991 - October 1991                */
/* Note:  for a reconstruction program, SRSIZE must be redefined, */
/*        the gamma coefficients reversed, and new values entered */
/*        in the lookup tables.                                    */

#include <stdio.h>
#define SRSIZE  49   /* size of the decomposition xform shift register */
#define BLKSIZE 16   /* input block size--must be a power of two      */
#define NUMFILT 2    /* number of lattice filters required            */
#define HALFORD 3    /* one half of the order of the filter           */

main() {

   static float gamma[HALFORD] = {-2.42549, -.5460948, -0.105889};
   static float beta = -.332671;
   int   i, j, k, l;
   float memory[SRSIZE];
   float evenin[NUMFILT], oddin[NUMFILT], e[NUMFILT][HALFORD],
         din[NUMFILT - 1][HALFORD], dout[NUMFILT][HALFORD],
         evenout[NUMFILT], oddout[NUMFILT];

   /* lookup tables for the input and output schedulers */

   static int evenintable[NUMFILT][BLKSIZE] = {
   30, 45, 43, 45, 34, 45, 39, 45, 43, 45, 41, 45, 43, 45, 41, 45,
   48, 31, 48, 31, 48, 30, 48, 27, 48, 30, 48, 28, 48, 42, 48, 48 };

   static int oddintable[NUMFILT][BLKSIZE] = {
   31, 46, 40, 46, 33, 46, 36, 46, 35, 46, 33, 46, 33, 46, 31, 46,
   48, 32, 48, 33, 48, 16, 48, 29, 48, 29, 48, 27, 48, 32, 48, 48 };

   static int evenouttable[NUMFILT][BLKSIZE] = {
   22, 48, 48, 48, 16, 35, 12, 34, 48, 33, 6, 32, 48, 31, 1, 30,
   48, 22, 48, 48, 48, 16, 48, 12, 48, 48, 48, 6, 48, 48, 48, 48 };

   static int oddouttable[NUMFILT][BLKSIZE] = {
   45, 48, 48, 48, 35, 45, 45, 45, 48, 45, 45, 42, 48, 38, 0, 22,
   48, 41, 48, 48, 48, 33, 48, 36, 48, 48, 48, 34, 48, 48, 48, 48 };
```

```
static int dintable[HALFORD-1][NUMFILT][BLKSIZE] = {
44, 47, 47, 41, 37, 39, 35, 37, 47, 36, 34, 42, 47, 43, 42, 43,
48, 43, 48, 38, 48, 36, 48, 38, 48, 44, 48, 40, 48, 44, 48, 48,

43, 47, 47, 40, 36, 38, 31, 36, 47, 35, 27, 34, 47, 38, 25, 42,
48, 42, 48, 37, 48, 35, 48, 35, 48, 33, 48, 33, 48, 31, 48, 48 };

static int douttable[HALFORD-1][NUMFILT][BLKSIZE] = {
44, 43, 39, 41, 37, 39, 39, 38, 45, 44, 41, 45, 45, 45, 48, 45,
48, 48, 48, 38, 48, 36, 48, 48, 48, 35, 48, 48, 48, 43, 48, 48,

43, 42, 38, 40, 36, 38, 36, 37, 34, 36, 34, 40, 32, 44, 48, 44,
48, 48, 48, 37, 48, 32, 48, 48, 48, 28, 48, 48, 48, 26, 48, 48 };

printf (" Circular wavelet decomposition program\n");
printf ("\n Output wavelet data:\n");

/* initialize shift register memory and counter */

for (j = 0; j < SRSIZE; j++) memory[j] = 0.0;
i = 0;

/* input placement */

while (scanf ("%f", &memory[SRSIZE - 3]) != EOF) {

   /* loop over all filters */

   for (k = 0; k < NUMFILT; k++) {

      /* resolve all inputs (input schedulers) */

      evenin[k] = memory[evenintable[k][i]];
      oddin[k] = memory[oddintable[k][i]];
      for (l = 0; l < HALFORD - 1; l++)
         din[k][l] = memory[dintable[l][k][i]];

      /* N-stage lattice filters */

      for (l = 0; l < HALFORD; l++) {
         if (l == 0) {              /* first rung (normal) */
            dout[k][0] = evenin[k] + gamma[0] * oddin[k];
            e[k][0]    = evenin[k] * gamma[0] - oddin[k];
         }
         else if ((l & 1) == 0) {   /* normal rung */
```

```
            dout[k][l] = e[k][l - 1] + gamma[l] * din[k][l - 1];
            e[k][l]    = e[k][l - 1] * gamma[l] - din[k][l - 1];
         }
         else {                        /* inverted rung */
            dout[k][l] = e[k][l - 1] * gamma[l] + din[k][l - 1];
            e[k][l]    = e[k][l - 1] - gamma[l] * din[k][l - 1];
         }
      }
      evenout[k] = beta *    e[k][HALFORD - 1];
      oddout[k]  = beta * dout[k][HALFORD - 1];
   }

   /* resolve all outputs (output schedulers) */

   for (k = 0; k < NUMFILT; k++) {
      memory[evenouttable[k][i]] = evenout[k];
      memory[oddouttable[k][i]] = oddout[k];
      for (l = 0; l < HALFORD - 1; l++)
         memory[douttable[l][k][i]] = dout[k][l];
   }

   /* output "placement" */

   printf ("%7.4f  ", memory[0]);
   if ((i % 8) == 7) printf("\n");

   /* shift the register contents */

   for (j = 1; j < SRSIZE; j++) memory[j - 1] = memory[j];
   memory[SRSIZE - 2] = 0.0;

   i++;      /* i is a continual loop from 0 to BLKSIZE - 1. */
   i &= (BLKSIZE - 1);
  }
}
```