Architectures for Wavelet Transforms: A Survey

Chaitali Chakrabarti
Dept. of Electrical Engg.,
Arizona State University,
Tempe, AZ 85287-5706.
chaitali@asu.edu

Mohan Vishwanath
Computer Science Lab,
Xerox Palo Alto Research Center,
Palo Alto Ca 94304.
mohan@parc.xerox.com

Robert M. Owens
Dept. of Computer Sci. and Engg.,
The Pennsylvania State Univ.,
Univ. Park, Pa 16802.

Abstract

Wavelet transforms have proven to be useful tools for several applications, including signal analysis, signal compression and numerical analysis. This paper surveys the VLSI architectures that have been proposed for computing the Discrete and Continuous Wavelet Transforms for 1-D and 2-D signals. The architectures are based upon on-line versions of the wavelet transform algorithms. These architectures support single chip implementations and are optimal with respect to both area and time under the word-serial model.

1 Introduction

Interest in wavelet transformations has greatly increased, as their applications have become more numerous. Such applications include signal analysis, signal compression, and numerical analysis. The wavelet transform is the most likely candidate for the transform part of future image/video compression standards. In applications such as video compression the computational burden on the transform part of the pipeline is very high. To meet this burden in real time, the design and implementation of efficient signal processing systems which can perform these transforms has itself taken on much importance.

In this paper we present a survey of efficient VLSI algorithms and architectures for computing the Discrete Wavelet Transform, DWT, (both one and two dimensional), the Continous Wavelet Transform, CWT, (again both one and two dimensional), and the Inverse Discrete Wavelet Transform, IDWT. These architectures are independent of the nature and, for the most part, size of the wavelet transformation being computed. In other words, most of the architectures are independent of the type and size of the wavelet filters. These architectures support single chip implementations in VLSI and are optimal under various models with respect to both area and time [16] [20].

A single-chip implementation is the most important criterion in the design of the architectures presented in this paper. This implies that the wavelet transforms have to be computed in an on-line/running fashion. The approach taken in this paper is to first describe the basic structure of the DWT, CWT and IDWT, then to describe on-line algorithms that permit cheap implementations and finally to describe the architectures that implement these algorithms. These architectures are based upon a number of implementations of the filters, like, systolic, parallel and digit-serial. Similarly a number of storage unit designs are also explored, like systolic, MUX-based and register minimized. In this way, the "best fit" for a given application can be chosen. Also the generic architectures we present can in many cases be easily modified to form a whole family of implementations.

The rest of the paper is organized as follows. Sections 2 and 3 describe the architectures for 1-D and 2-D DWT, Section 4 describes the architectures for 1-D IDWT, and Sections 5 and 6 describe the architectures for 1-D and 2-D CWT.

2 Architectures for the 1-D Discrete Wavelet Transform

2.1 Introduction

The 1-D Discrete Wavelet Transform can be implemented by the Pyramid Algorithm (PA) developed by Mallat [9][10]. This algorithm can be represented as follows

`begin{Direct Pyramid Algorithm}`

`for(j=1 to` J`)`

`for(n=1 to` 2^{J-j}`)`

$X[j,n] = \sum_{m=0}^{K-1} X[j-1, 2n-m]w[m]$ `Low Pass`

$Y[j,n] = \sum_{m=0}^{K-1} X[j-1, 2n-m]h[m]$ `High Pass`

`end{Direct Pyramid Algorithm}`

where, $X[0,n] = x[n]$, J is the number of octaves, $N = 2^J$ is the length of the sequence $x[]$, K is the length of the wavelet filter, and $w[]$ and $h[]$ are the quadrature mirror filters derived from the wavelet. Note that $Y[j,]$ contains the outputs for the j'th octave. The corresponding filter bank structure is given in Figure 1.

The pyramid algorithm can be implemented on general purpose, single processor computers as well as in a multiprocessor environment. If J be the number of octaves, then in a J-processor architecture, each processor computes the low pass and the high pass outputs of a particular octave. The drawback of this realization is that the computation load is not evenly distributed among the processors. This is because the processor which computes the jth octave outputs is active $\frac{1}{2^{j-1}}$ of the time compared to the processor which computes the first octave outputs. The high pass and low pass filters can also be implemented by Brent and Kung systolic arrays (instead of a single processor) of size K [1]. The computation load is uneven in this case too.

The processor utilization can be increased by folding the computations onto a single high/low filter pair. In the following subsections, we describe two categories of folded architectures: serial architectures and parallel architectures. In serial architectures, the inputs needed to compute a single output are supplied to the filters in a serial (ie. one after the other) manner. In parallel architectures the inputs needed to compute a single output are supplied to the filters in a parallel (ie. all at once) manner.

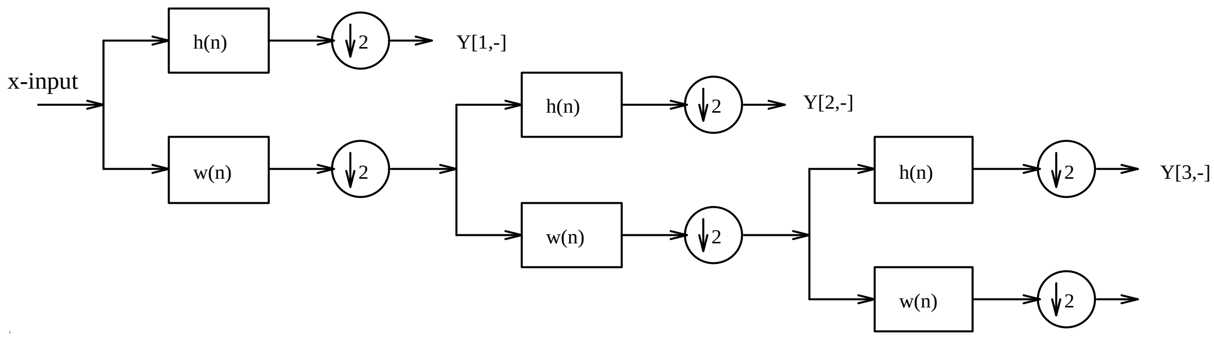

Figure 1: **The 1-D DWT Filter Bank Structure for $J = 3$ octaves.**

2.2 Folded Architectures: Serial Filters

Figure 2 describes a folded architecture where a single filter pair is serially reused to compute each of the J octaves. In this case, filter costs are reduced by a factor of J while performance degrades by only a factor of 2. Unfortunately, while serially reused filter architecture makes (or can be made) hardware efficient, it has several undesirable properties. Firstly, it requires a storage of size N (the

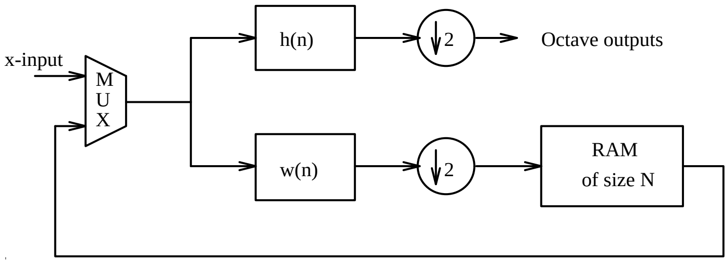

Figure 2: **Folded Architecture for 1-D DWT: Direct Implementation**

size of the input data). Secondly, it has a large latency. This is because all the outputs of one octave are generated before the outputs of the next octave.

In an attempt to reduce the storage and latency, a number of architectures have been proposed which compute the different octaves in an interlaced manner. Interlacing is achieved by scheduling each output at the 'earliest' instance that it can be scheduled. The 'earliest' instance is decided based on a strict precedence relation, ie., if the 'earliest' output of the i^{th} octave clashes with that of the $(i+1)^{th}$ octave, then the i^{th} octave output is scheduled. Finally, the outputs of the first octave are scheduled at every other cycle. The Recursive Pyramid Algorithm (RPA) [6][19], is one such scheduling algorithm which generates it's outputs in an interleaved manner. The RPA was independently developed and presented in [6] and [19]. It is presented below,

```
begin{Recursive Pyramid Algorithm}
    for(i=1 to N)
        rdwt(i,1)
end{Recursive Pyramid Algorithm}

rdwt(i,j)
begin{rdwt}
    if(i is odd)
        k=(i+1)/2
        X[j,k] = sum_{m=0}^{J-1} X[j-1,i-m]w[m]     Low Pass
        Y[j,k] = sum_{m=0}^{J-1} X[j-1,i-m]h[m]     High Pass
    else
        rdwt(i/2,j+1)
end{rdwt}
```

In the listing above the two assignments are printed as $X[j,k] = \sum_{m=0}^{J-1} X[j-1, i-m]w[m]$ (Low Pass) and $Y[j,k] = \sum_{m=0}^{J-1} X[j-1, i-m]h[m]$ (High Pass).

This algorithm assumes that a single multiply/accumulate can be performed in a single cycle. In other words, the latency of a single multiply/accumulate is one. Like the Direct Pyramid Algorithm, the Recursive Pyramid Algorithm can be implemented on general purpose, single processor computers, and on a class of architectures which share the structure described in Figure 3. The architecture in Figure 3 consists of a group of multiply-accumulate (MAC) cells to implement the low pass filter and another group of MAC cells to implement the high pass filter. Each group can be implemented by a systolic or a semi-systolic array. If K be the size of the filter, then these arrays are of size K. The routing network stores the low pass outputs and feeds them to the low pass and high pass filters for higher octave computations. In the next subsection we discuss several

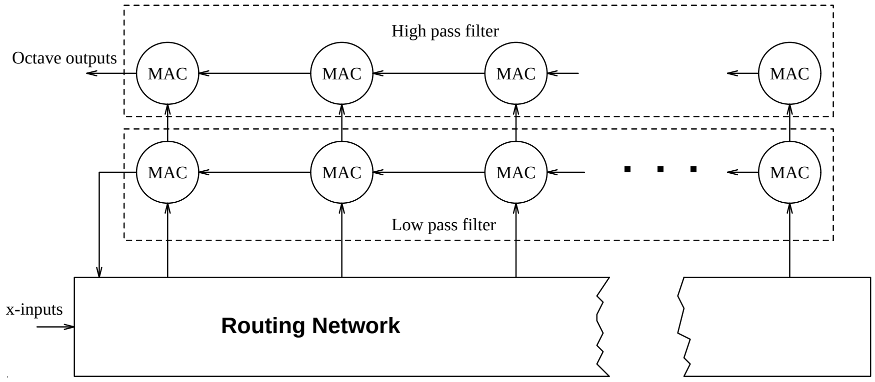

Figure 3: **Overall Architecture for 1-D DWT**

serial architectures which differ principally in how they implement the routing network.

2.2.1 Routing Network Implementations For Serial Filters

In this section we will discuss a number of architectures to implement the routing network for serial filters. The order in which we present these architectures does not implies a chronological order of publication; nor, does it imply some type of actual pedigree or lineage. The order was chosen so as to make comparisons between the architectures easier.

Figure 4 illustrates the first architecture [6]. Each of the squares in this figure represent a single storage cell. While this figure appears different than the one in the cited reference, it is actually the same. This architecture uses a number of serial-in-parallel-out queues to store each of the octaves and a set of multiplexors to deliver the correct values to the filters. Note that each queue appears as a row of storage elements in Figure 4. While this architecture has several weaknesses, it has the advantage of being regular.

The second architecture [16][19] is illustrated in Figure 5. Here each row of the network consists

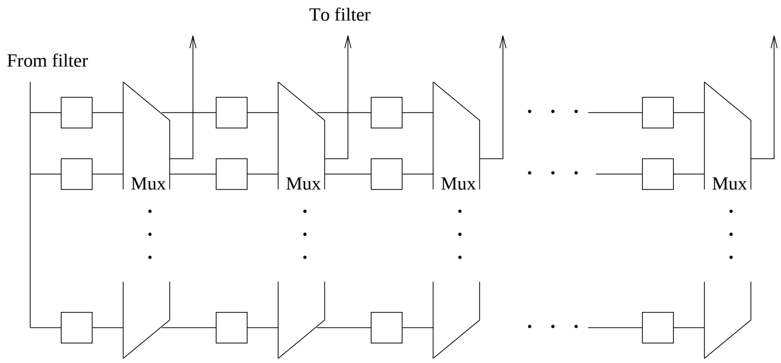

Figure 4: **Mux Based Routing Network**

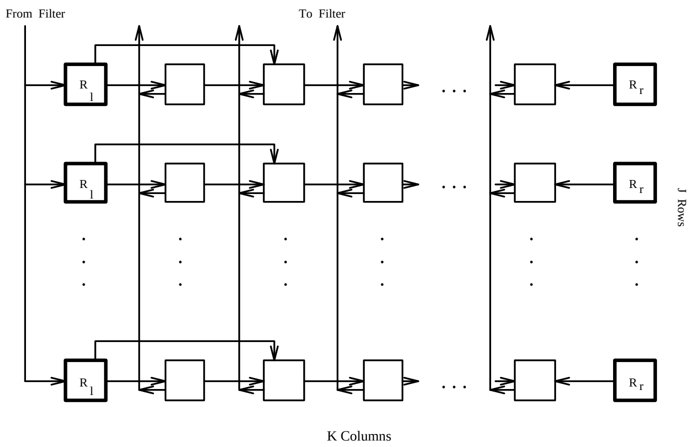

Figure 5: **Semisystolic Array Routing Network**

of K storage cells, an R_l cell to route the input and an R_r cell to control the clocking. Note that in this architecture, the multiplexors of the previous architecture are replaced by a combination of busses and tristate buffers. The advantage this grains is that more octaves can be added without the need of changing the size of the multiplexors or the preexisting interconnections. Hence, this architecture is more extensible than the prior architecture and regularity is not lost. However, the architecture still has the disadvantage of having long communication paths (e.g. the busses).

Figure 6 illustrates the third architecture [16][19]. This architecture can be viewed as is a fully

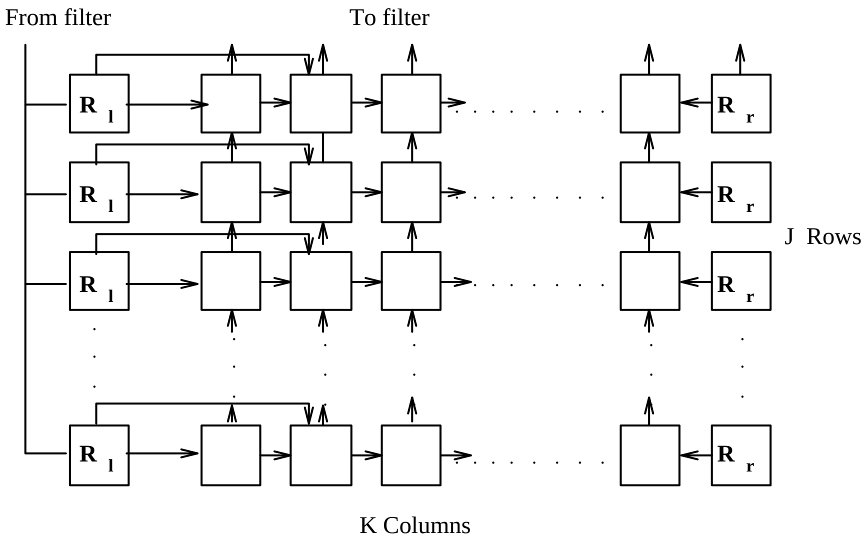

Figure 6: **Systolic Array Routing Network**

systolic version of the previous semi-systolic architecture. This gains the advantage of having only short, nearest neighbor interconnects. Also, if anything, extensibility and regularity have increased. The disadvantage is some increase in the complexity of the storage cells.

Figure 7 illustrates the fourth architecture [16][19]. In this architecture each column of storage cells in the first two architectures is stored in a single RAM. In this case selection is performed by addressing instead of multiplexors or busses. The advantage gained is in the ability to use a more compact structure (e.g. a RAM) to store data. However, while regularity is maintained, extensibility has been lost since changingthe number octaves requires the address decoders and the RAM size to be changed. This realization also requires an additional address generator. The address generator of a semi-systolic architecture is simpler than that of a systolic architecture.

Figure 8 illustrates the fifth architecture [13]. This architecture implements the routing network using the minimum number of storage cells. It uses the forward-backward allocation scheme to route data through the network. One disadvantage is that the control of the routing network appears

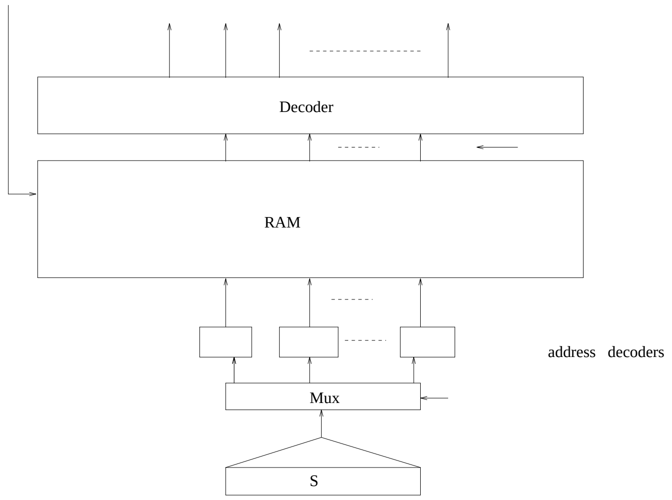

Figure 7: **RAM Based Routing Network**

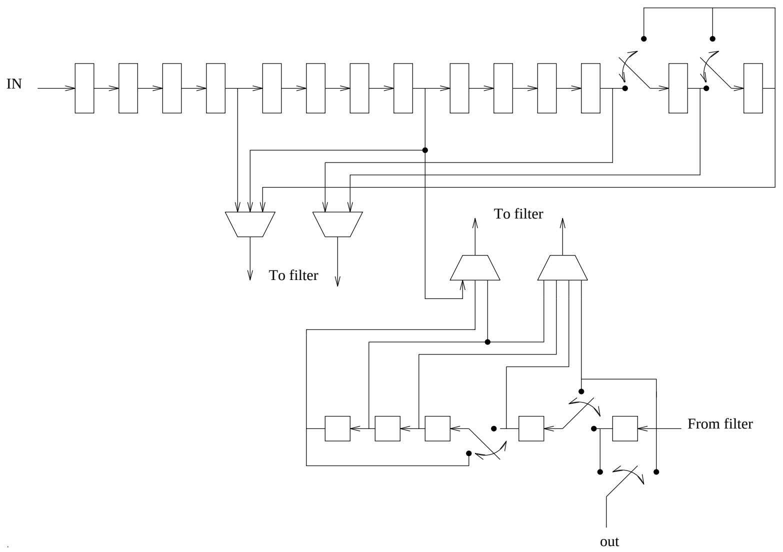

Figure 8: **Reduced Storage Routing Network**

far more complex than the control of the other alternative implementations. The more severe disadvantage is the loss of regularity and, hence, extensibility. The loss of regularity could cause any decrease in the VLSI area devoted to storage cells to be more than offset by the increase in area devoted to interconnect (wires).

The final architecture implements the routing network in a distributed manner [5],[7]. If the number of processors is K (the filter size), then each processor has a storage of size J (the number of octaves). The advantage of this implementation is that it is suitable for parallel computers, where each processor has local memory attached to it. The disadvantage of this architecture is that each processor has to have additional circuitry (multiplexers and demultiplexers) to route the data between storage units of neighboring processors. This data movement is necessary since each storage unit data is required for the computation of $\lceil K/2 \rceil$ outputs, and for each output, the data has to be multiplied by a different filter coefficient (which resides in a different processor). In contrast, data movements are handled very easily in the systolic and semi-systolic routing networks, since the networks support horizontal shifts.

2.3 Architectural Variations of Serial Filters

2.3.1 Digit Pipelining

The first architectural variation we will consider involves the use of digit pipelined arithmetic components. In digit pipelining, the digits of each input and output are consumed or generated in a digit serial manner (one after the other). Digit pipelining provides for smaller (but slower) arithmetic components. There are several advantages of digit pipelined structures. Firstly, the hardware utilization is high. Secondly, the routing and interconnection overhead are reduced. Digit pipelined variations of each of the architectures given in the previous section are possible. Of particular interest, however, are the digit pipeline version of the systolic array architecture [12] and the reduced storage architecture [13].

Figure 9 describes the digit serial version of the systolic array architecture of [20]. The squares in this figure represent digit storage and processing units as opposed to word storage and processing units of the corresponding digit parallel architecture (described earlier). Note that regularity, easy extensibility and nearest neighbor interconnections of the digit parallel systolic architecture have been maintained here.

The digit serial architecture of [13] implements the filter bank structure of Figure 1. The architecture consists of J stages, with the digit size in the jth stage being $\frac{W}{2^j}$ (W is the original

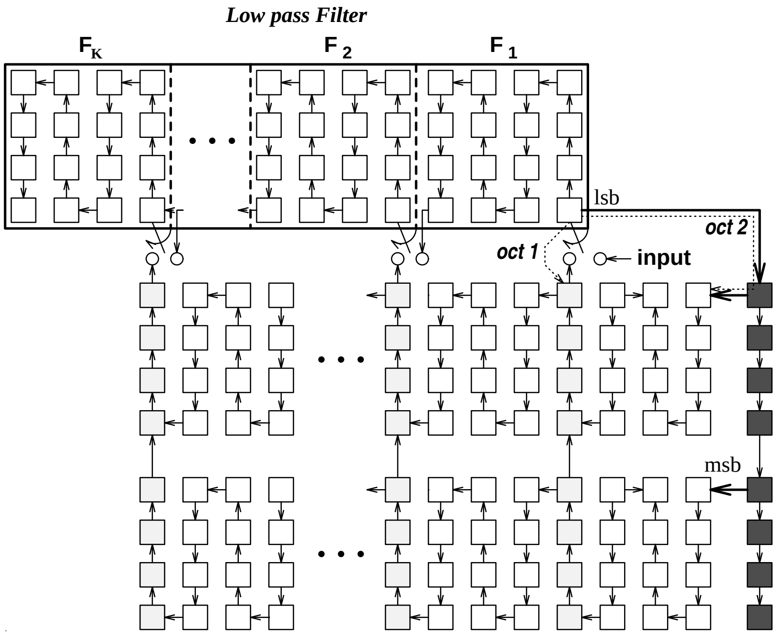

Figure 9: **Digit Serial Systolic Array Architecture**

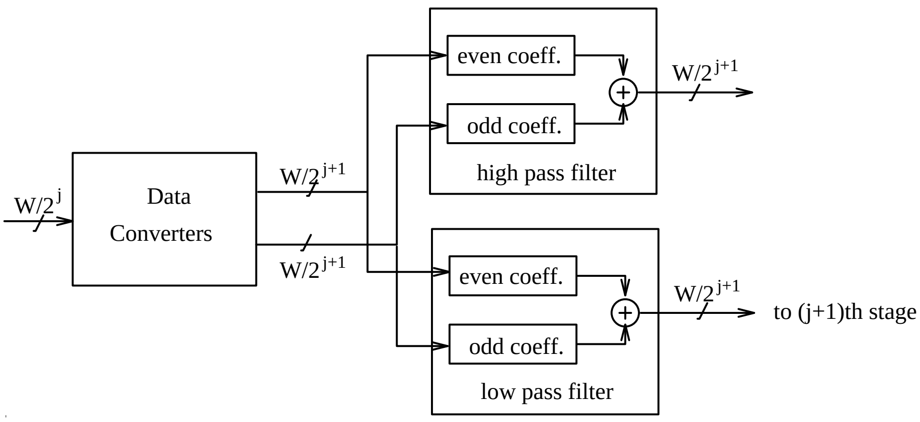

Figure 10: **Block diagram of the jth stage of the digit serial architecture**

word length). Each stage consists of a data converter and two filter units (low pass and high pass) as shown in Figure 10. The data converter in the jth stage converts one $\frac{W}{2^j}$ word into two (parallel) $\frac{W}{2^{j+1}}$ words. To match this, the output computation is broken into two components (even and odd) with each component containing half of the filter coefficients. The result of the two components are then added to get the output. While this architecture achieves high hardware utilization, it suffers from large latency. The latency is due to the delay in the data converters in each stage.

2.3.2 Short-length algorithms

Increasingly, DSP chips have a substantial amount of on-chip storage, this means that if we are able to use the RPA to compute the DWT then we will be able to use only the on-chip storage for computation of the DWT, thus saving a lot of off-chip accesses. Will this be offset by the fact that RPA does not try to reduce the number of MACs compared to an FFT based implementation of the DWT [17] ? This section attempts to study this trade-off.

In this section we will consider combining the RPA with efficient short-length filtering algorithms [16][17]. Most computationally efficient filtering algorithms exploit the redundancy in the computation of a block of outputs to reduce the total number of operations. This implies that a block of input and/or output has to be buffered to utilize these algorithms. Also, these algorithms do not retain the natural 'running' multiply-accumulate structure of FIR filtering [11]. The short-length FIR filtering algorithms [11] were introduced to avoid this extra storage requirement and to retain (at least partially) the multiply-accumulate structure. These algorithms are important because they reduce the number of operations per output and because they allow the running nature of the RPA to be retained.

We consider a specific case of the short-length filtering algorithm, the F(2,2), as described in [11]. This consists of converting the problem of computing 2 outputs of a length L filter into that of computing 1 output of three length $L/2$ filters, at the cost of 4 adds per block of 2 outputs. Such a filter taken from [11] is shown in Fig.4. The number of multiplications and additions per output point are $\frac{3}{4}L$ and $\frac{3}{4}L + \frac{1}{2}$, respectively, compared to the L multiplications and L additions which would normally be required.

The scheduling of the outputs is complicated by the input/output characteristics of such short-length filters. For example, in the case of the F(2,2) filter, each time the i^{th} octave is scheduled, 2 outputs are produced, hence 4 outputs of the $(i-1)^{th}$ octave need to be available. Fortunately, scheduling is possible and reductions in arithmetic complexity are obtainable. For example a

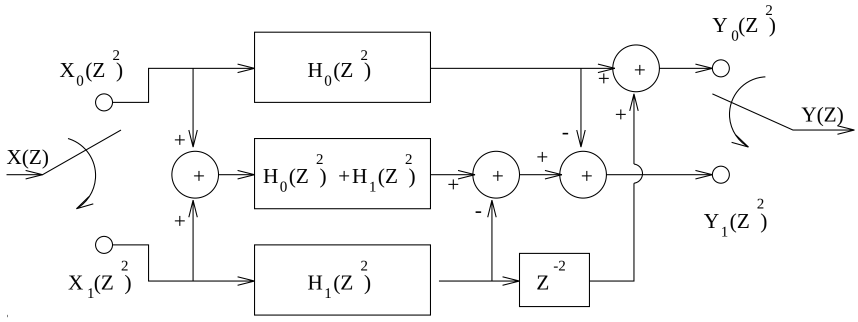

Figure 11: **The short-length filter algorithm, F(2,2).**

Table 1: **Comparison of the RPA and the PA based on operation counts. N=1024 and the short-length FIR filter used is F(2,2)**

Filter Length	**PA and RPA: Direct**		**PA: Short-Length**		**RPA: Short-Length**	
	Mults	Adds	Mults	Adds	Mults	Adds
8	8	6	6.00	7.00	6.00	7.00
12	12	10	8.00	9.00	8.00	9.00
16	16	14	12.00	13.00	12.00	13.00
20	20	18	14.00	15.00	14.00	15.00
24	24	22	18.00	19.00	18.00	19.00
32	32	30	24.00	25.00	24.00	25.00
40	40	38	30.00	31.00	30.00	31.00
48	48	46	36.00	37.00	36.00	37.00
60	60	58	44.00	45.00	44.00	45.00
64	64	62	48.00	49.00	48.00	49.00

comparison of the number of operations required by the RPA and the PA is shown in Table 1. For the sake of comparisons, only the F(2,2) algorithm is considered for all the filter lengths. It can be seen from this table that while the RPA considerably reduces storage requirements compared to the PA, it is still able to do as well as the PA in terms of the number of MACs.

2.3.3 QMF Lattice Structure

The final architectural variation that we consider involves the use of quadrature mirror filter (QMF) lattice structures [3]. The folded architectures and digit-serial architectures that have been developed for these structures is very similar to the architectures that have been developed earlier for the direct form filter implementation in [13]. An important variation is that for large filter sizes, the number of multipliers and adders that are required here is close to half the number that is required for the systems that are based on the direct form filter implementation.

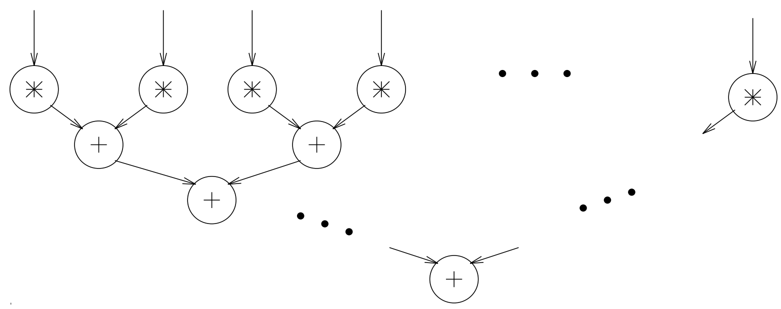

Figure 12: **Parallel Filter. The tree of adders add the products generated by the multipliers.**

2.4 Folded Architectures: Parallel

In this section we present parallel (folded) architectures for the 1-D DWT. The computation unit of these architectures consists of a parallel filter which can be implemented with K multipliers and a balanced binary tree of $(K-1)$ adders to add the products (see Figure 12). The delay in the parallel filter is $T_m + T_a \log K$, where T_m is the time taken to do a multiplication, and T_a is the time taken to do an addition. Consequently, the parallel filter can be clocked at $t_c = T_m + T_a \log K$ (in comparison, the systolic filter is clocked at $T_m + T_a$). If the parallel filter is indeed clocked at t_c, it is said to have a latency of 1 cycle, and the RPA scheduling algorithm can be used. If, however, the parallel filter is pipelined to q levels, the latency increases to q cycles and a modified version of the RPA or the MRPA [11] scheduling algorithm has to be used.

The MRPA schedule is similar to the RPA schedule in that the jth octave outputs are computed every 2^j cycles, $0 \leq j < J$. However, the MRPA schedule is a shifted version of the RPA schedule, where the shift is constant for all outputs of an octave. This shift must be enough to account for the latency of the parallel filter and to avoid conflicts with the outputs of any lower octaves. We explain this with an example where the latency $q = 4$. The inputs for the ith output of the first octave are supplied at time $2i$, $0 \leq i < N/2$. If the latency were one (1), then the inputs for the kth output of the second octave would be supplied at time $4k+1$, $0 \leq k < N/4$. Since the latency is four (4), the inputs for the kth output cannot be supplied before time $4k+4$. If a shift of $4k+4-4k+1 = 3$ is used, there would be a schedule conflict with a first output octave. Hence the earliest time at which the inputs for the kth output of the second octave can be scheduled is time $4k+5$.

Once the scheduling is determined, the design of the parallel architecture is straightforward [11]. Like the serial architecture, it consists of two filter units to compute the high pass and the low pass outputs and a routing network. This is to be expected since the parallel filter is nothing

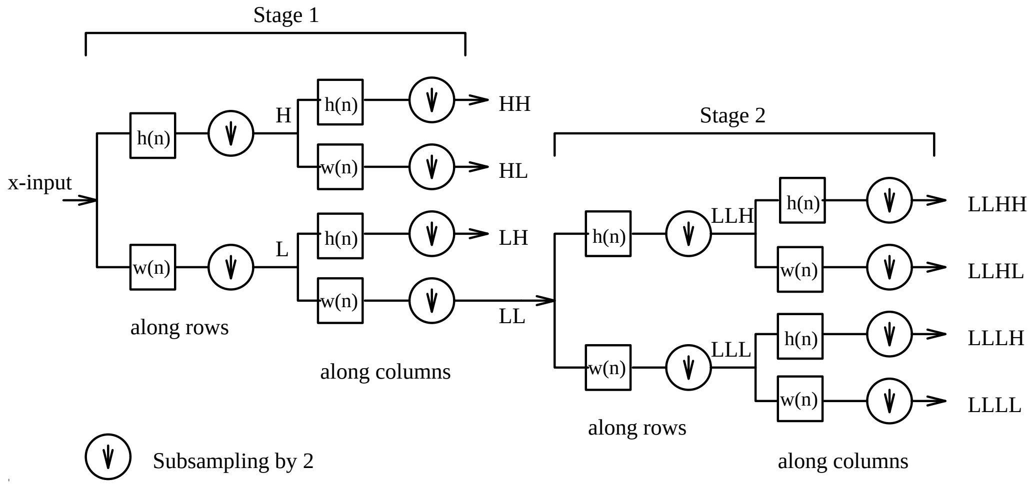

Figure 13: **The 2-D DWT filter bank structure for $J = 2$ stages.**

but a semisystolic version of the serial filter (or possibly more correctly the serial filter is the fully systolic version of the parallel filter [1]). A modified version of the routing networks that have been developed for the semisystolic serial architectures can be used here. For example, in the mux based routing network of serial architectures, in any given time step, the storage cells in different rows are selected by the muxes. In the mux based routing network here, in any given time step, the storage cells in the same row are selected by the muxes. This clearly results in lower control complexity compared to the network of the serial architecture.

3 Architectures for the 2-D Discrete Wavelet Transform

3.1 Introduction

The separable 2-D Discrete Wavelet Transform can be implemented by applying the two channel decomposition of 1-D DWT (or the pyramid algorithm) in the horizontal and vertical dimensions. The resulting four channel subband decomposition of this transform is described in Figure 13. The filters $h(n)$ and $w(n)$ are the same as those used for 1-D DWT.

In the 2-D DWT filter bank structure, in each stage, the row computations precede the column computations. In Stage 1, 1-D DWT is computed along the rows of the input array to generate the H (high pass) and the L (low pass) outputs. If the image is of size $N \times N$, then the H and L output arrays are each of size $N \times N/2$. The HH and HL output arrays are obtained by computing 1-D DWT along the columns of the H array. Similarly, the LH and LL output arrays are obtained by computing 1-D DWT along the columns of the L array. The HH, HL, LH and LL arrays are each of size $N/2 \times N/2$. The LL outputs are sent to Stage 2. The four outputs of Stage 2, namely,

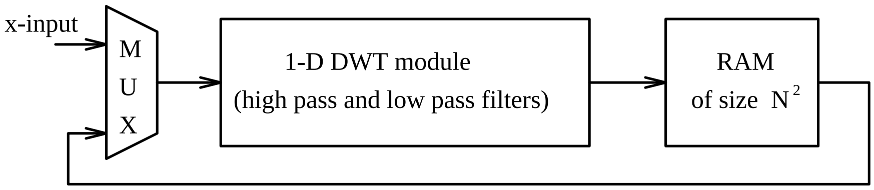

Figure 14: **Folded architecture: direct implementation**

$LLHH$, $LLHL$, $LLLH$ and $LLLL$, are each of size $N/4 \times N/4$. Out of these, the $LLLL$ outputs are sent to Stage 3. Thus N^2 outputs are generated in Stage 1, $N^2/4$ in Stage 2, and so on.

The filter bank structure of Figure 13 can be implemented on general-purpose, single processor computers as well as in a parallel processing environment. In the parallel environment, if each filter module is mapped to a processor, then the computation load is not evenly distributed among the processors. In fact, the processors in Stage i are active only $\frac{1}{4^{i-1)}}$ of the time compared to the processors in Stage 1, $1 \leq i < J$.

3.2 Folded Architectures

In order to improve the hardware efficiency, a single high/low filter pair can be used to compute all the outputs [20]. Figure 14 describes such an architecture. Here the 1-D DWT module is used for both the row and the column computations. The architecture requires a storage of size N^2 words. The storage size is determined by the maximum number of outputs that need to be stored during any computation (in this case, the H and L arrays in Stage 1). The architecture also requires an address generator to enable appropriate access of data for the row and column computations. The two main problems with this architecture are its large storage requirement and its large latency. We next describe two architectures which have lower storage requirements and lower latency than the direct implementation (Figure 14). The two architectures achieve this by interlacing the higher stage outputs with the first stage outputs.

3.2.1 Serial-parallel architecture

The serial-parallel architecture of [20] derives its name from the fact that it consists of two serial filters to do the row computations and two parallel filters to do the column computations. Each of the serial and parallel filters consist of a high pass (K multipliers) and a low pass filter (K multipliers). Figure 15 gives the block diagram of this architecture. Two rows of inputs are fed in at the same time. The serial filters $S1$ and $S2$ read data along rows and compute the outputs in row major order. While $S1$ computes only the high pass and the low pass outputs of the first stage (H

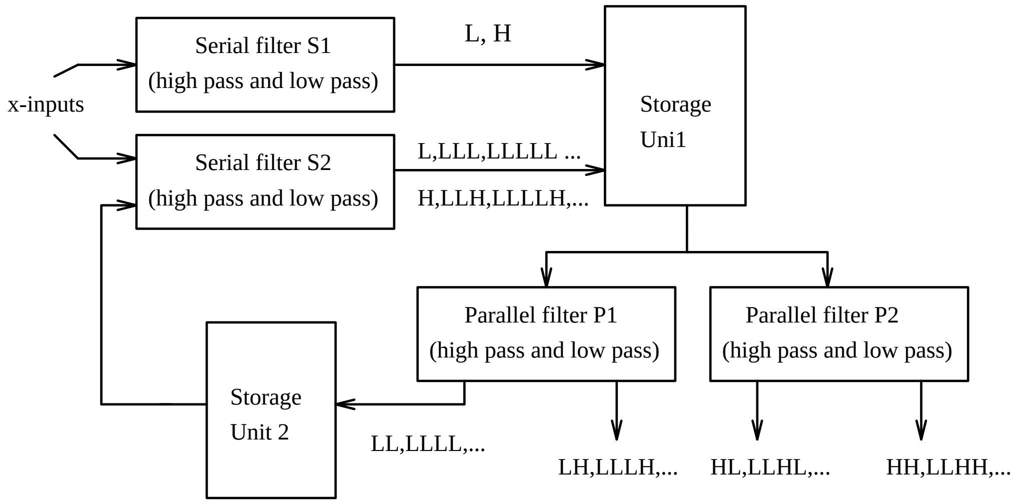

Figure 15: **The serial-parallel architecture for 2-D DWT**

and L), $S2$ computes the high pass and low pass outputs of all the stages (H, L, LLH, LLL, etc.). The outputs of $S1$ and $S2$ are stored in row major order in Storage Unit 1. The parallel filters $P1$ and $P2$ also compute the outputs in row major order, but these filters read data along columns from Storage Unit 1. $P1$ computes the outputs LL, LH, $LLLL$, $LLLH$, etc, while $P2$ computes the outputs HL, HH, $LLHL$, $LLHH$, etc. Of these outputs, the LL, $LLLL$, etc. outputs are stored in row major order in Storage Unit 2. Storage Units 1 and 2 can be implemented with shift register routing networks or RAMs. In [20], Storage Unit 1 is implemented with a shift register routing network of size $\approx 2KN$.

The computations in the serial and parallel filters can be scheduled by a modification of the RPA scheduling scheme. This is because while the serial filters have a latency of 1, the parallel filters can have a latency that is greater than 1.

3.2.2 Parallel architecture

The parallel architecture described here is a modification of the serial-parallel architecture in that all the filters are parallel filters, and that only one row (instead of two) is input at a time. The second feature allows the parallel filters to be built with K programmable multipliers (instead of $2K$ multipiers) resulting in reduced hardware complexity. The architecture is described in Figure 16 and consists of parallel filter units $P1$ and $P2$ which compute along the rows, parallel filter units $P3$ and $P4$ which compute along the columns, and two storage units. $P1$ computes the low pass and the high pass outputs (L and H) along each row while $P2$ computes the outputs LLL, LLH, $LLLLL$, $LLLLH$, etc. along each row. The outputs of both $P1$ and $P2$ are stored in row major

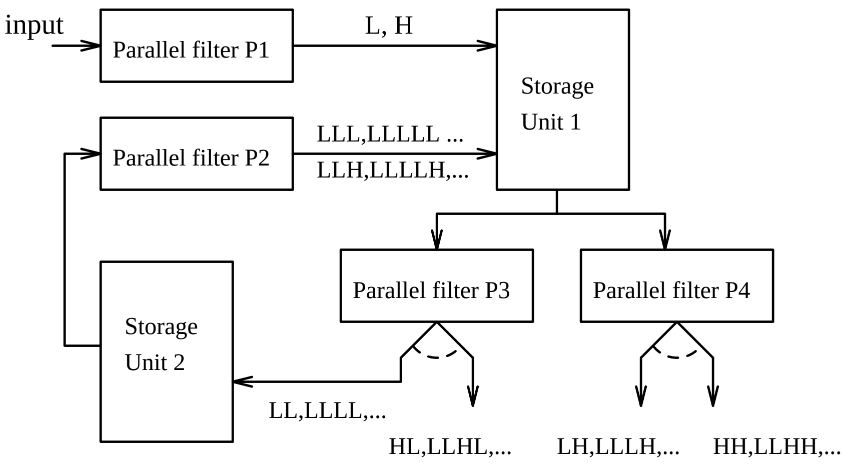

Figure 16: **The completely parallel architecture for 2-D DWT**

order in Storage Unit 1. $P3$ and $P4$ generate the outputs in row major order but read data along columns from Storage Unit 1. $P3$ computes the outputs LL, LH, $LLLL$, $LLLH$ etc., with the LL, $LLLL$ outputs being computed N cycles before the LH, $LLLH$ outputs. Similarly, $P4$ computes the outputs HL, HH, $LLHL$, $LLHH$, etc., with the HL, $LLHL$ outputs being computed N cycles before the HH, $LLHH$ outputs. The outputs LL, $LLLL$, etc. are stored in Storage Unit 2 in row major order. Storage Unit 1 is of size $\approx 2KN$ and Storage Unit 2 is of size $\approx N$. The storage units can be built with shift registers, or RAM (as in the case of 1-D DWT) and we will not describe it here.

In this architecture, $P2$ works in a lock step manner with $P3$ and $P4$. Thus if a no-conflict scheduling scheme is determined for $P3$ and $P4$, the scheduling scheme of $P2$ will automatically be conflict-free. It is to be noted that an extension of the MRPA scheduling algorithm that had been developed for 1-D DWT parallel architectures can be extended to design a scheduling algorithm for Filters 3 and 4.

Other existing parallel architectures include the one in [2] for non-separable 2-D DWT. The architecture consists of 2 parallel filters of size K^2 and a storage unit of size NK. Each parallel filter computes the outputs of 2 bands. An extension of the MRPA scheduling algorithm is used here to schedule the operations in the two filters.

3.3 Other Approaches

In an attempt to reduce the storage requirements, block filtering approaches have been proposed in folded [20] as well as in unfolded architectures [4]. The main drawback is using block filtering is that the data has to be available in a block form (instead of the usual row-major or column-major).

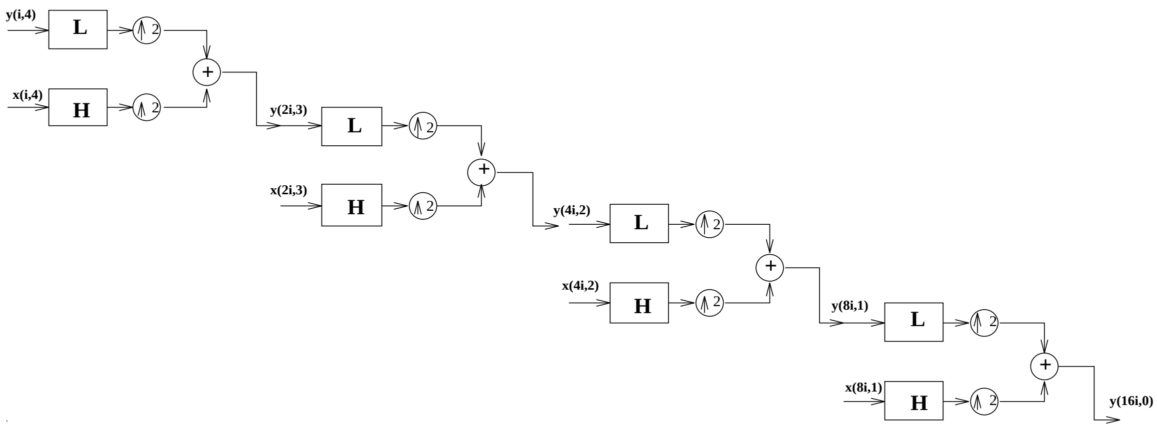

Figure 17: **The Inverse DWT filter bank**

This necessitates the use of data converters at every stage. The architecture of [4] uses register minimization techniques to reduce the number of registers in the data converters. This comes at the expense of complex routing and more importantly, scalability.

We end the section by describing the first (chronologically speaking) architecture that was proposed for 2-D DWT in [8]. The emphasis was on designing a multiplierless architecture for the 1-D filter modules. While this was possible with Daubechies 4-tap filters, it is not obvious how this method can be extended for general filters.

4 Architectures for the 1-D Inverse Discrete Wavelet Transform

The Inverse Discrete Wavelet Transform (IDWT) is computed as shown in Figure 17. It reconstructs (synthesis) a signal from the decomposition induced by the DWT. There are 2 kinds of scenarios wherein the IDWT is used: off-line (where all the DWT outputs are available) or in tandem with the DWT. In both scenarios, the latency of reconstruction (ie., the time to reconstruct the first output sample) is an important consideration. In the tandem case, minimizing the buffering between the DWT and the IDWT is also an important issue. This buffering is required because the IDWT module does not consume the DWT outputs immediately. This is because to begin the reconstruction of a J octave decomposition, the IDWT needs the first few J'th octave outputs of the DWT. In this section we describe running/on-line architectures for the IDWT. We start the section by describing the Inverse Recursive Pyramid Algorithm.

4.1 The Inverse Recursive Pyramid Algorithm (IRPA)

The design of the IRPA assumes that the hardware resources available for the IDWT are very similar to that required for the DWT. In particular, the number of MACs is the same. The IRPA is very similar to the RPA. An efficient implementation of the IRPA generates a j'th octave output every 2^j clock cycles, $0 \leq j < J$. Note that $j = 0$, corresponds to the reconstructed output samples. As we show in the next section, the reconstructed outputs are generated in parallel with the rest of the octaves.

The pseudo-code for the IRPA is shown below,

```
begin{Inverse Recursive Pyramid Algorithm}
          ridwt(0,J-1)
          for(i=1 to N/2 - 1)
               ridwt(i,1)
end{Inverse Recursive Pyramid Algorithm}

ridwt(i,j)
     begin{ridwt}
          if(i is 0)
               if(j is NOT 0)
                    Compute y(0,j)
                    ridwt(0,j-1)
               else
                    Compute y(0,0)
                    Compute y(1,0)
          else if(i is odd)
               k=(i+1)/2
               Compute y(k,j)
          else
               ridwt(i/2,j+1)
     end{ridwt}
```

The schedule generated by the IRPA for $J = 5$ is shown below. Here $y(i, j)$ corresponds to the i'th output of the j'th octave. $y(i, j) = yh(i, j) + yl(i, j)$, where $yh(i, j)$ is the output of the highpass filter and $yl(i,j)$ is the output of the lowpass filter. The odd or even filter taps are used to generate yl and yh when i is odd or even, respectively.

$y(0,4), y(0,3), y(0,2), y(0,1), X, y(1,1), y(1,2), y(2,1), y(1,3), y(3,1), y(2,2), y(4,1), y(1,4), y(5,1), y(3,2), y(6,1), y(2,3), y(7,1), y(4,2), y(8,1), I, y(9,1), y(5,2), y(10,1), y(3,3), y(11,1), y(6,2), y(12,1), I, y(13,1), y(7,2), y(14,1), I, y(15,1)$

The I marks indicate idle time slots, i.e., when no output is computed. The slot marked X is also an idle slot but it indicates the time when the first output sample (0'th octave outputs) is computed. An output sample is computed every clock cycle.

It is important to note that from the RPA and IRPA schedules we can deduce the maximum amount of buffering needed between the DWT and the IDWT (for the tandem case). A j'th octave output is generated by the RPA every 2^j cycles and it is consumed by the IRPA at the same rate with a delay of 2^J. Thus the amount of buffering needed between the DWT and the IDWT is atmost $2 * 2^J$.

4.2 Architectures

It is clear from the previous section that the IRPA is quite similar to the RPA except for the initial boundary condition and the manner in which 0'th octave outputs are produced. In this section we show how the inverse DWT (IDWT) can be computed in a manner symmetric to the DWT on some of the folded architectures, with a few changes. This is a useful property to have while building single chip solutions for the wavelet transform. The architecture is shown in Figure 18. The box labeled "O/E" alternately enables the odd and the even coefficients of the filter. The first set of highpass and lowpass filters compute all the octaves except the 0'th octave. Since a 0'th octave output has to be produced every cycle, it is computed on a seperate set of filters. Note that two 0'th octave outputs are generated for each 1'th octave output.

The storage units could be one of the many kinds of routing networks that have been described in the section on 1-D DWT architectures. Important of these are the MUX based storage units [18], storage units using minimum number of registers [13]. Similarly, the filters can be systolic [7], parallel [18], digit-serial [13] or minimized using lattice structures [3]. The basic operation of this architecture is similar to that of the 1-D DWT folded architectures and the trade-offs described between the various 1-D DWT architectures also hold for the 1-D IDWT.

In a running implementation, given the output of the DWT, the earliest that the first output of the IDWT can be produced is the J'th cycle. As mentioned in the beginning of this section, there

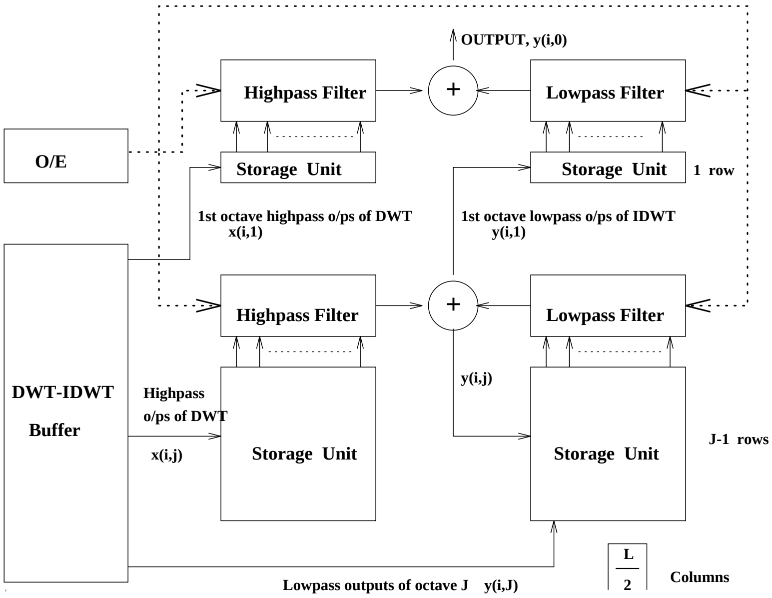

Figure 18: **Generic Architecture for folded IDWT**

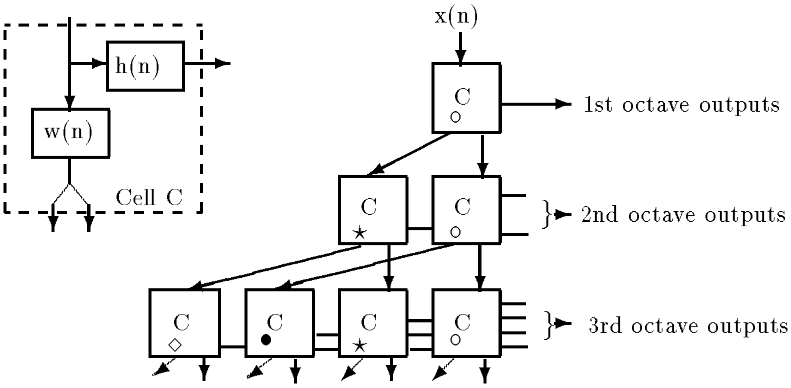

Figure 19: **Reorganized 'atrous' computational structure for 1-D CWT**

are situations where both the DWT and the IDWT are computed in tandem. In such a situation, if both the DWT and the IDWT are computed using running implementations, then the earliest that the first output of the IDWT can be produced is the $2J$'th cycle. Such minimal latency/buffering trade-offs are explored in [18]. One of the conclusions of this study was that the reconstruction latency, of the DWT-IDWT combination, can be minimized at the expense of an increase in the complexity of the control unit (scheduler) and a small increase in the buffering between the DWT and the IDWT.

5 Architectures for the 1-D Continuous Wavelet Transform

The Continuous Wavelet Transform (CWT) is defined as a wavelet transform with no decimation in time and scale. However, in this paper we consider a commonly used version of the CWT [14], where the scale resolution is logarithmic as in the case of DWT. At each scale k, the number of samples in the time dimension in N.

We begin this section by describing the reorganized "a trous" computational structure of Shensa [15]. Figure 19 describes this structure. Note that there are 2^{j-1} cells for the jth octave computation, with each cell working at the rate of $1/2^{j-1}$th that of the cell for the first octave computation. This is because the inputs to the jth octave cell are subsampled by 2^{j-1} compared to the original input. If the 2^{j-1} cells for the jth octave computation are combined to form a single computation unit, then this unit works at the same rate as the unit for the first octave computation. In the resulting implementation, each computation unit computes a total of N outputs with 1 output being computed every cycle, Since there are J computation units, J outputs (1 per computation unit) are computed simultaneously. In this section we will describe serial and parallel architectures that implement this J-computation unit architecture.

In applications where minimization of area is very important, the J-computation unit structure is folded into a single computation unit structure. In this section we describe folded architectures (serial and parallel) and a scheduling algorithm called the On-Line CWT Algorithm (OCA) for these architectures [2].

5.1 J outputs are computed every cycle

When J outputs are computed every cycle, both the serial and the parallel architectures consist of J computation units with each unit consisting of one low pass and one high pass filter. The inputs of the jth computation unit are fed from the low pass filter of the $(j-1)$th computation unit. Moreover the inputs of the jth computation unit have to be subsampled by 2^{j-1} compared to the x-input. The serial and the parallel architectures differ in the way the inputs are fed into the computation unit as well as in the way the subsampling is implemented.

Serial architecture:

The low pass and the high pass filters in the serial architecture are implemented by purely systolic arrays of size L. The inputs to the array flow from left to right, the outputs flow from right to left and the weights remain in the processors. In this implementation (as in all other systolic implementations) the inputs are slowed by a factor of 2. Figure 20 describes this implementation when the filter size is $L = 4$. In the jth unit, subsampling by a factor of 2^{j-1} is obtained by adding 2^{j-1} delay units in the input and the output path of each processor.

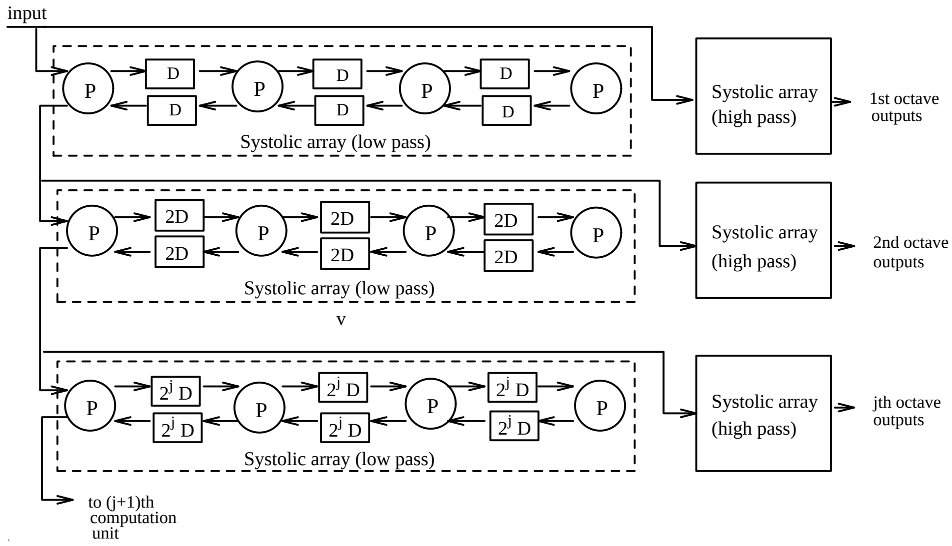

Figure 20: **Serial architecture for 1-D CWT when J outputs are computed every cycle. 'P' stands for a processor which is essentially a multiply-accumulator and 'D' stands for unit delay. The filter size $L = 4$.**

Semi-serial architecture:

This architecture is a modification of the serial architecture. Here the inputs are broadcast to all the processors, the weights stay in the processors, and the partial results get updated as they move from one processor to another in the array. In the jth unit, subsampling by a factor of 2^{j-1} is obtained by storing the partial results in registers of size 2^{j-1} in between consecutive processors. Figure 21 describes the systolic array architecture when $L = 4$.

Parallel architecture:

In the parallel architecture, all the L inputs are fed to the computation unit at the same time. In the the jth computation unit, subsampling by a factor of 2^{j-1} is achieved by storing the data in a delay line of size $2^{j-1}L$, and tapping the delay line every 2^{j-1} delay units. Figure 22 describes the architecture when the filter size is $L = 4$.

5.2 One output is computed every cycle:

When one output is computed every cycle, the architecture consists of only one computation unit and the outputs of J octaves (a total of JN outputs) are now computed by the same unit. A straight-forward way to implement this scheme would be to compute all the outputs of a particular octave before beginning the computations of the next octave. Such an implementation would require a large memory. In order to reduce the size of the memory, the computations of several octaves can be interlaced as in 1-D DWT. The on-line CWT algorithm (OCA), which we describe

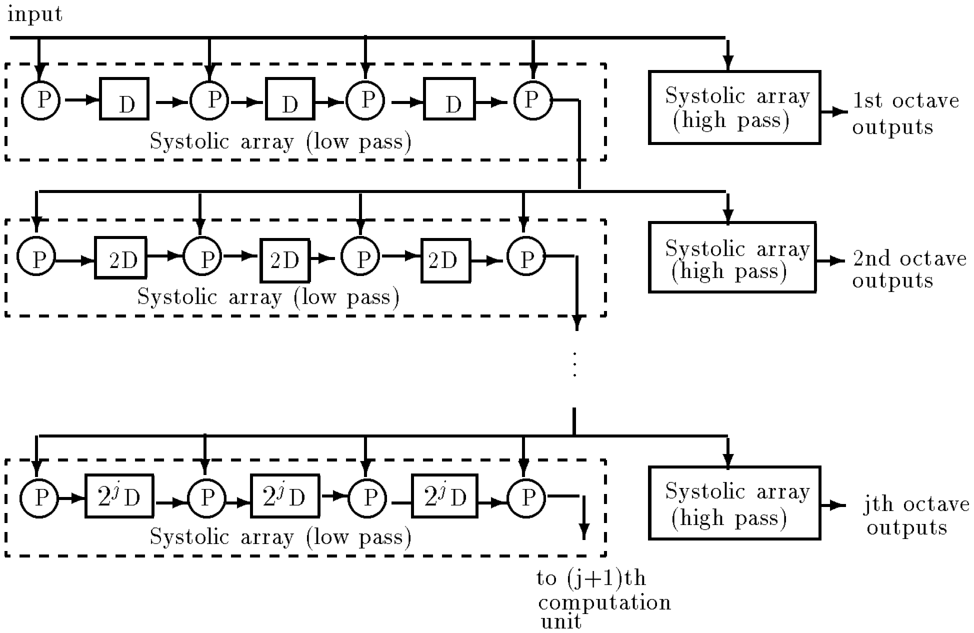

Figure 21: **Semi-systolic array architecture for 1-D CWT when J outputs are computed every cycle. 'P' stands for a processor which is essentially a multiplier-accumulator and 'D' stands for unit delay. The filter size $L = 4$.**

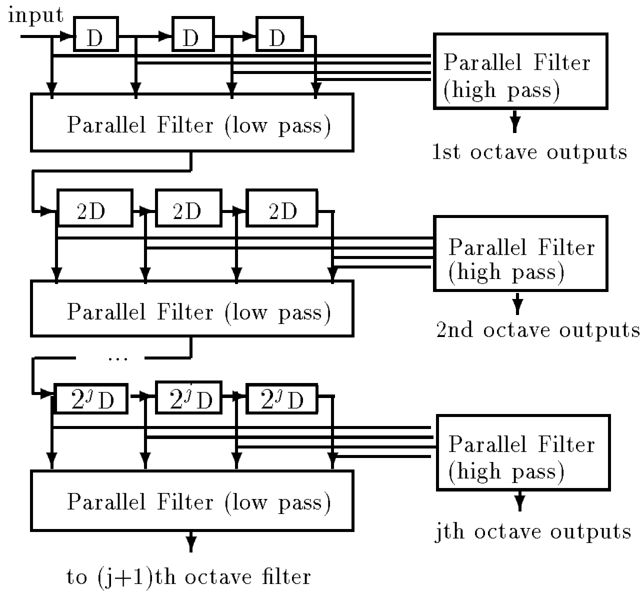

Figure 22: **Parallel filter architecture for 1-D CWT when J outputs are computed every cycle. 'D' stands for unit delay. The filter size $L = 4$.**

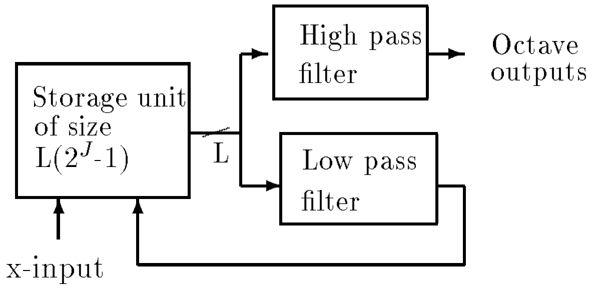

Figure 23: **Block diagram of the folded architecture for 1-D CWT**

next, was designed to enable such interlacing.

5.2.1 On-line CWT Algorithm

The On-line CWT Algorithm (OCA) is a running algorithm for computing the 1-D CWT which can be implemented on existing programmable DSP chips without any buffering as well as on folded architectures. Since the CWT algorithm produces NJ outputs (N outputs per octave), the OCA schedules the computations in such a way that the output of a particular octave is produced every Jth cycle. Consequently, the input has to be fed at the rate of one every Jth cycle. As in the case of RPA and MRPA (for 1-D DWT), the outputs are scheduled at the earliest possible instance. If the latency of the computation unit is 1, then there are no clashes, and the outputs of the various octaves are scheduled one after the other. For instance if $J = 4$, then the outputs are scheduled as follows: $y_1(1), y_2(1), y_3(1), y_4(1), y_1(2), y_2(2), y_3(2), y_4(2), y_1(3), \ldots$ and so on. If the latency of the computation unit is greater than 1, then the outputs have to be scheduled in a way such that in case of a clash, the lower output octaves have precedence over the higher octave outputs. Once the first output of any octave is scheduled such that there is no conflict with any of the lower octaves, the remaining outputs of that octave are scheduled every J cycles then on. In an example with $J = 4$ and latency $T = 4$, the outputs of the various octaves are scheduled as follows. $y_1(1), -, -, -, y_1(2), y_2(1), -, -, y_1(3), y_2(2), y_3(1), -, y_1(4), y_2(3), y_3(2), y_4(1), y_1(5), y_2(4),$ $y_3(3), y_4(2), y_1(6), \ldots$.

5.2.2 Folded Architectures

In this section we will describe two types of folded architectures for computing the 1-D CWT: the serial architecture and the parallel architecture. The block diagram of the folded architecture is described in Figure 23. Both the serial and the parallel architectures consist of a computation

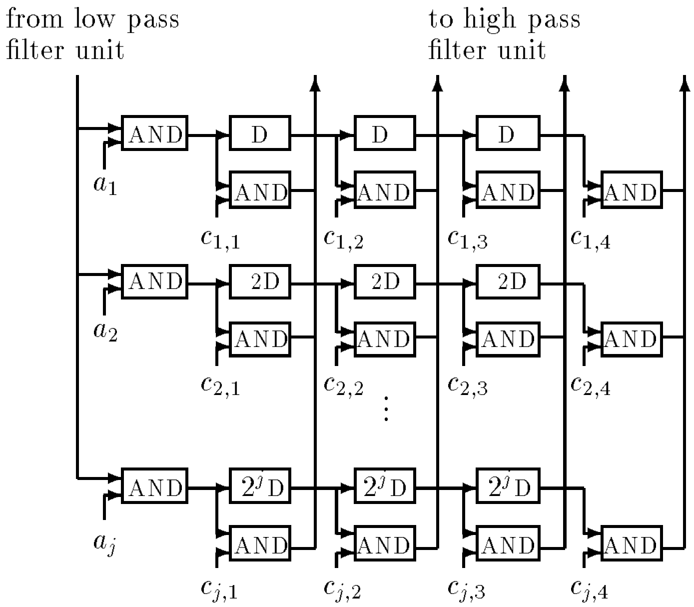

Figure 24: **Storage unit and the corresponding routing network of the folded architecture when** $L = 4$

unit to compute the low pass and the high pass filter outputs, and a storage unit store the inputs required for the computation of the octave outputs. The storage unit consists of J subunits of storage cells; the first subunit stores L x-inputs, the second subunit stores $2L$ y_1-outputs, the third subunit stores $4L$ y_2-outputs and so on. The number of cells in the jth subunit is $2^{j-1}L$, since the inputs for the computation of the jth octave are subsampled by a factor of 2^{j-1}. All the cells of the storage unit are clocked uniformly (in contrast to the storage unit for 1-D DWT). A data is read into the storage unit every cycle and into a subunit every J cycles. While L data are read out from the storage unit every cycle, the way the data are read out is different for the parallel and the serial architectures. Figure 24 describes the storage unit for the case when $L = 4$. Control signals $\{a_k\}$ and $\{c_{k,l}\}$ are generated at appropriate times by a finite state machine to read in(out) data to(from) the storage unit.

Serial architecture:

In this architecture, the weights stay in the processors, the inputs are fed in parallel, and the partial results get updated as they move from one processor to another in the array. This computation scheme demands that the inputs for the computation of a particular output are fed to the array in a skewed fashion. Also, L different outputs are computed at the same time in the array, and the L elements that are loaded in parallel from the storage unit come from J different subunits (if $L \geq J$) or L different subunits (if $L < J$).

Parallel architecture:

In this architecture, all L elements of the same subunit are fed to the low pass and the high pass filters at the same time. Thus for the computation of an jth octave output, control signals $c_{j,4}$, $c_{j,3}$, $c_{j,2}$ and $c_{j,1}$ are set to 1, and L elements of the jth subunit are fed in parallel to the filters. Note that these L elements are 2^{j-1} apart in the jth subunit.

6 Architectures for the 2-D Continuous Wavelet Transform

In this section we describe serial and parallel architectures for 2-D CWT for the case when both the input and the output are in the line scan mode. We study the case when J outputs are produced every cycle. The architectures that we describe here are extensions of the serial and parallel architectures that we developed for 1-D CWT. The architecture for computing J octaves consists of J computation units with each unit consisting of one low pass and one high pass filter. The inputs of the jth computation unit and fed from the low pass filter of the $(j-1)$th computation unit. Moreover the inputs of the jth computation unit have to be subsampled along the rows as well as along the columns by a factor of 2^{j-1}. The serial and the parallel architectures differ in the way the data is fed into the computation unit as well as in the way data subsmapling along the rows and along the columns is implemented.

Semi-serial architecture:

In the semi-serial architecture, each low pass (or high pass) filter consists of L^2 processors, grouped in L subunits with a semi-systolic array of L processors in each subunit. Note that the subunit architecture is the same as that of the 1-D CWT semi-serial architecture. The outputs of the subunits are added by a row of adders (see Figure 25). Subsampling along the rows is obtained by storing the partial results in 2^{j-1} storage cells in between consecutive processors in each subunit. Subsampling along the columns is obtained by storing the partial results in $2^{j-1}N$ storage cells in between two subunits. Figure 25 describes the semi-systolic array architecture when the filter size is (4×4).

Parallel Architecture: In the parallel architecture, each low pass (or high pass) filter consists of L^2 multipliers grouped in L subunits with L multipliers and $L-1$ adders per subunit. The outputs of each of the subunits is added by a tree of adders. As in the case of 1-D CWT, subsampling along the rows is achieved by storing the data in a delay line of size $2^{j-1}L$, and tapping the delay line every 2^{j-1} delay units. Since the inputs are fed in the line scan mode, subsampling along the

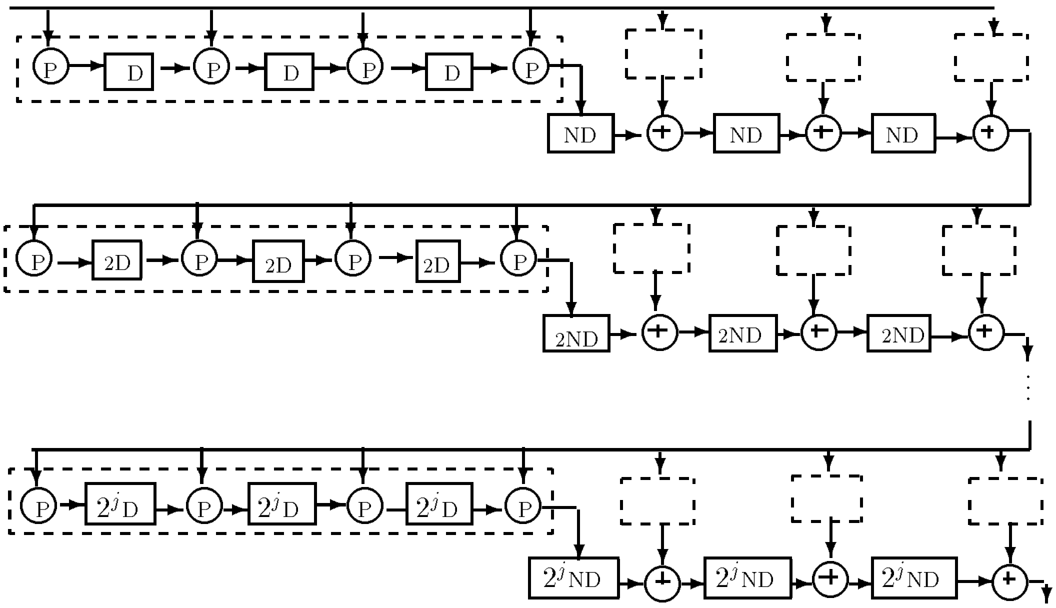

Figure 25: **Semi-systolic array architecture for 2-D CWT when the filter size is** (4×4)**. 'P' stands for a multiply-accumulate processor and 'D' stands for unit delay. The high pass filters have not been shown.**

columns is obtained by storing the data in a delay line of size $2^{j-1}LN$, and tapping the delay line every $2^{j-1}N$ delay units. Figure 26 describes the architecture when the filter size is (4×4).

7 Conclusions

Currently, there are no chips available in the market that are optimized towards computing arbitrary wavelet transforms. We believe that the reasons are two-fold, firstly, it is only recently that the compression standards and VLSI signal processing community have started taking an active interest in wavelet transforms, for eg. a fingerprint compression technique based on the DWT was recently adopted as the standard by the FBI. Secondly, there is a common misconception that subband transforms need more buffering than block transforms. The large number of folded VLSI architectures that we presented in this paper clearly dispel that myth. For eg., if the input data is presented in a raster scan order (as in practical systems), a 2-d DWT based on an 8-tap wavelet requires no more buffering than a 8x8 DCT.

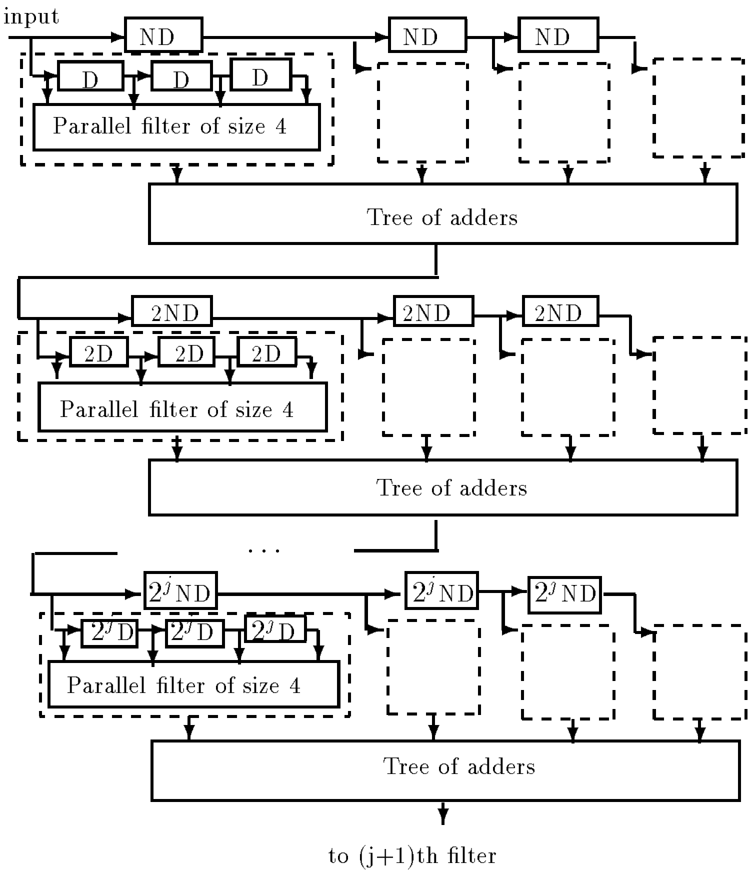

Figure 26: **Parallel filter architecture for 2-D CWT when the filter is of size** (4×4)**. The high pass filters have not been shown.**

References

[1] R.P Brent and H.T. Kung. The chip complexity of binary arithmetic. *Proc. 12th Annu. Ass. Comput. Mach. Symp. Theory Comput*, pages 190–200, May 1980.

[2] C. Chakrabarti and M. Vishwanath. Efficient realizations of the discrete and continuous wavelet transforms: from single chip implementations to SIMD parallel computers. *IEEE Trans. on Signal Processing*, 43:759–771, March 1995.

[3] T. Denk and K. Parhi. Architectures for lattice structure based orthonormal discrete wavelet transform. *Proceedings of the 1994 IEEE International Conference on Application Specific Array Processors*, pages 259–270, August 1994.

[4] T. Denk and K. Parhi. Calculation of minimum number of registers in 2-D discrete wavelet transforms using lapped block processing. *Int. Symp. on Circuits and Systems*, pages 77–81, 1994.

[5] F. Fridman and E. S. Manolakos. Distributed memory and control vlsi architectures for the 1-D discrete wavelet transform. *IEEE VLSI Signal Processing VII*, 1994.

[6] G. Knowles. VLSI architecture for the discrete wavelet transform. *Elec. Letters*, 26(15):1184–1185, Jul 1990.

[7] R. Lang, E. Plesner, H. Schroder, and A. Spray. An efficient systolic architecture for the one-dimensional wavelet transform. *Proceedings of SPIE conference on Wavelet Applications*, pages 925–935, April 1994.

[8] A.S. Lewis and G. Knowles. VLSI architecture for 2-D Daubechies wavelet transform without multipliers. *Elec. Letters*, 27(2):171–173, Jan 1991.

[9] S. Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Trans. Acoustics Speech and Sig. Proc*, 37(12):2091–2110, Dec 1989.

[10] S. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. Pattern Analysis and Mach. Intell*, 11(7):674–693, July 1989.

[11] Z. Mou and P. Duhamel. Short-length fir filters and their use in fast nonrecursive filtering. *IEEE Trans. on Signal Processing*, 39(6):1322–1332, June 1991.

[12] C. Nagendra, M.J. Irwin, and R. M. Owens. Digit pipelined discrete wavelet transform. *Proceedings of ICASSP*, April 1994.

[13] K. Parhi and T. Nishitani. VLSI Architectures for Discrete Wavelet Transforms. *IEEE Trans. on VLSI Systems*, 1(2):191–202, 1993.

[14] O. Rioul and P. Duhamel. Fast algorithms for wavelet transforms. *IEEE Trans. on Information Theory*, 38(2):569–586, Mar 1992.

[15] M.J. Shensa. The discrete wavelet transform: Wedding the a'trous and mallat algorithms. *IEEE Transactions on Signal Processing*, 40:2464–2482, October 1992.

[16] M. Vishwanath. *Time-Frequency Distributions: Complexity, Algorithms and Architectures.* PhD thesis, Pennsylvania State University, May 1993.

[17] M. Vishwanath. The recursive pyramid algorithm for the discrete wavelet transform. *IEEE Trans. on Signal Processing*, 42:673–676, March 1994.

[18] M. Vishwanath and C. Chakrabarti. A VLSI architecture for real-time hierarchical encoding/decoding of video using the wavelet transform. *Int. Conf. on Acoustics, Speech and Signal Processing*, pages 401–404, 1994.

[19] M. Vishwanath, R.M. Owens, and M.J. Irwin. Discrete wavelet transforms in VLSI. *Proc. Intl. Conf. on Application Specific Array Processors*, pages 218–229, Aug 1992.

[20] M. Vishwanath, R.M. Owens, and M.J. Irwin. VLSI architectures for the discrete wavelet transform. *IEEE Trans. Circuits and Systems*, 42:305–316, May 1995.