An Efficient Architecture for Lifting-based Two-Dimensional Discrete Wavelet Transforms *

S. Barua J. E. Carletta
Dept. of Electrical and Computer Eng.
The University of Akron
Akron, OH 44325-3904
sb22@uakron.edu, carlett@uakron.edu

K. A. Kotteri A. E. Bell
Dept. of Electrical and Computer Eng.
Virginia Tech
Blacksburg, VA 24061-0111
kkotteri@vt.edu, abell@vt.edu

ABSTRACT

An architecture for the lifting-based two-dimensional discrete wavelet transform is presented. The architecture is easily scaled to accommodate different numbers of lifting steps. The architecture has regular data flow and low control complexity, and achieves 100% hardware utilization. Symmetric extension of the image to be transformed is handled in a way that does not require additional computations or clock cycles. The architecture is investigated in terms of hardware parameters such as memory size and number of ports, number of memory accesses, latency, and throughput. The proposed architecture achieves higher throughput and uses less embedded memory than architectures based on convolutional filter banks.

Categories and Subject Descriptors: B.5.1 [Design]: Data-path Design

General Terms: Design, Performance.

Keywords: Discrete Wavelet Transform, Lifting Structure.

1. INTRODUCTION

Since the emergence of the JPEG2000 image compression standard, considerable attention has been paid to the development of efficient system architectures for the discrete wavelet transform (DWT). The DWT has traditionally been implemented using convolutional filter banks [1][2]. However, a recently introduced framework called the *lifting scheme* (LS) [3] for constructing wavelet filters promises to reduce the number of operations involved in computing a DWT to almost one-half of those needed with a convolutional approach [4]. In addition, the lifting scheme is amenable to *in-place computation* [4], so that the DWT can implemented in low memory systems.

*This material is based upon work supported by the National Science Foundation under grants 0218672 and 0217894.

GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.

A two-dimensional (2-D) DWT parallel architecture based on the lifting scheme has been proposed in [5]. It is viable only for wavelet filters with no more than two lifting steps, and so could not be used for the 9/7 wavelet. It extends to two dimensions the one-dimensional recursive pyramid algorithm (RPA) proposed in [6]. [1] shows that the using the RPA for the 2-D DWT results in inefficient hardware utilization and complicated control circuitry. Other work in [7] is based on a lifting structure, but reduces embedded memory size at the expense of additional computations.

In this paper, we propose a new 2-D DWT architecture for a lifting scheme implementation of the 9/7 wavelet. We investigate implementation platform parameters such as memory size and number of ports, latency, and throughput. The advantages of the proposed architecture are 100% hardware utilization, regular data flow, and low control complexity. Additionally, because of the modular structure, the proposed architecture can easily be scaled to accommodate additional lifting steps and multiple levels. We show how the lifting scheme can be exploited to perform symmetric extension of images essentially for free, without additional computations or clock cycles.

The rest of the paper is organized as follows. Section 2 describes possible architectures for a two-dimensional discrete wavelet transform, and outlines the motivation for our choice. Section 3 describes our architecture in detail; it includes descriptions of the row and column processors and the implementation of symmetric extension. Section 4 describes the resulting properties of the architecture, and Section 5 draws conclusions.

2. BACKGROUND AND MOTIVATION

A survey of DWT architectures is reported in [8]. There are three basic architectures for the two-dimensional DWT : level-by-level, line-based, and block-based architectures. In implementing the 2-D DWT, it is always a challenge to perform efficient column processing after row processing. The difficulty is that row processing produces coefficients in a row-wise order, while traditional column processing requires those coefficients in a column-wise order. The three architectures address this difficulty in different ways.

A typical level-by-level architecture uses a single processing module that first processes the rows, and then the columns. Intermediate values between row and column processing are stored in a memory; since this memory must be large enough to hold wavelet coefficients for the entire image, external memory is usually used. Since column processing requires

that memory words be read in an order different from the one in which they were written by the row processor, high-bandwidth external memory access can not easily be used, and external memory access can become the performance bottleneck of the system.

A typical line-based architecture [9] includes separate processing modules for each level of transform to be done, and eliminates the need to store an entire image's worth of data in an external memory. The line-based architecture starts column processing as soon as a sufficient number of rows has been processed. For example, for a one-level decomposition using convolutional 9/7 wavelet filters, we can start column processing as soon as nine rows have been processed. These nine rows must be buffered until column processing begins, typically in an embedded memory. (In general, space for l_f rows of data is needed, where l_f is the length of the longest wavelet filter.) The column processing is done in row-wise manner on the buffered data; the column filter works on nine pixels of data from a single column at a time, and moves one column to the right at each clock. Although [9] reported that the amount of buffer space required can be reduced using the lifting scheme, no investigation was done.

The block-based architecture is like the level-by-level, except that it breaks the image into smaller blocks that are processed separately. This reduces the amount of intermediate data; the block size is typically chosen so that the data can be placed in embedded memory, rather than external memory. The main disadvantage is that if the boundaries between blocks are to be handled correctly, additional computations must be done, making this architecture computationally inefficient. A block-based architecture for the lifting scheme is reported in [10].

Our proposed architecture is a hybrid of level-by-level and line-based architectures. Our goal is a small yet high performance system. We keep the hardware size small by using a single computing module iteratively for the higher level computations, and going back to external memory between levels. However, unlike a typical level-by-level architecture, we do not use the external memory between row and column processing of a single level; this means that external memory data are always written and read in row-wise order, and high-bandwidth external memory access using burst mode can be used. We use line-based processing within a level, with embedded memory for the buffer space between row and column processing. In addition, we exploit the lifting scheme to further reduce the amount of buffer space needed.

3. OUR 2-D DWT ARCHITECTURE

The block diagram of the proposed 2-D DWT architecture is shown in Figure 1. The FPGA implements a row processor (RP), a column processor (CP), and a local memory module (MEM) used to buffer results between the two. The 2-D DWT is computed in row-column fashion (i.e., the DWT is carried out on the rows first and then on the columns). The image, which is stored in external memory, is read to the FPGA in row-by-row order. The row processor performs horizontal filtering to the rows and writes the approximation, a, and detail, d, coefficients to the local memory. Once a sufficient number of rows have been processed, the column processor starts vertical filtering; it fetches coefficients from the local memory and generates four subbands, aa, da, dd, and ad, as shown in Figure 2. The four subbands are written back to the external memory, again in row-wise order.

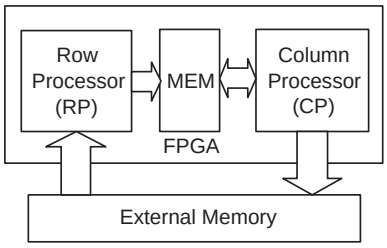


Figure 1: System architecture.

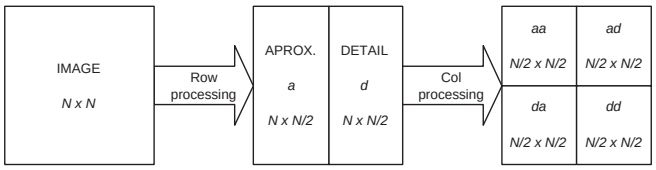


Figure 2: A one-level 2-D DWT.

Multiple levels are performed on this architecture in a non-interleaved fashion, with results between levels stored in the external memory. For each of the higher levels, an approximation subband is read from external memory and four higher level subbands are generated using the same computing modules. The operation continues until the desired level is finished.

3.1 Implementation of a lifting step

The lifting scheme implements a filter bank as a multiplication of upper and lower triangular matrices, where each matrix constitutes a lifting step. For the 9/7 wavelet, four lifting steps and one scaling can be used; its polyphase analysis filter banks $P(z)$ can be written as [4]:

$$P(z) = \prod_{i=1}^{2} \begin{pmatrix} 1 & s^i(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ t^i(z) & 1 \end{pmatrix} \begin{pmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{pmatrix} \quad (1)$$

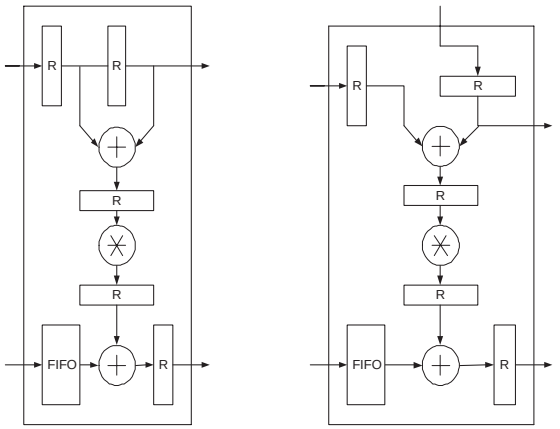
where $s^1(z) = \alpha(1 + z^{-1})$, $s^2(z) = \gamma(1 + z^{-1})$, $t^1(z) = \beta(1 + z)$, and $t^2(z) = \delta(1 + z)$. The parameters α, β, γ, and δ are two-tap symmetric filter coefficients and ζ and $1/\zeta$ are two scaling factors.

The lifting steps lead to the following equations to be implemented in hardware:

$$\begin{aligned} \text{Predict P1:}\ d_i^1 &= \alpha(x_{2i} + x_{2i+2}) + x_{2i+1} \\ \text{Update U1:}\ a_i^1 &= \beta(d_i^1 + d_{i-1}^1) + x_{2i} \\ \text{Predict P2:}\ d_i^2 &= \gamma(a_i^1 + a_{i+1}^1) + d_i^1 \\ \text{Update U2:}\ a_i^2 &= \delta(d_i^2 + d_{i-1}^2) + a_i^1 \\ \text{Scale G1:}\ a_i &= \zeta a_i^2 \\ \text{Scale G2:}\ d_i &= d_i^2/\zeta. \end{aligned} \quad (2)$$

The original data to be filtered is denoted by x_i, and the outputs are the approximation coefficients a_i and detail coefficients d_i. The superscripts on intermediate values show the lifting step number.

Each of the lifting steps has a similar computing pattern. They vary only in the values of the coefficients. A lifting step requires two additions and one multiplication. As our application requires high performance, we adapt a highly pipelined, multiplierless implementation of the lifting step.



(a) systolic architecture for row processing

(b) parallel architecture for column processing

Figure 3: Implementation of a lifting step.

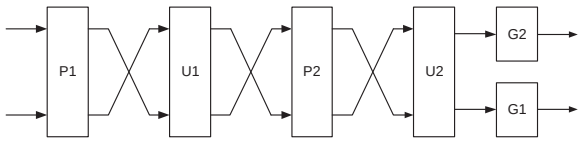


Figure 4: Architecture of the row processor.

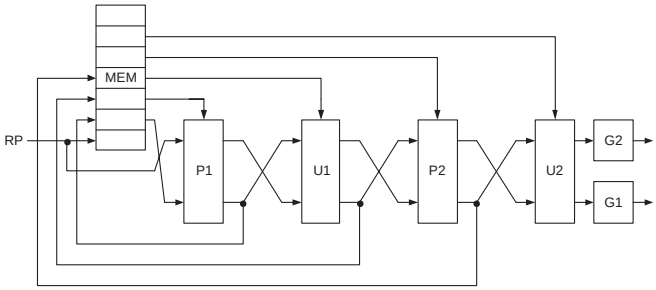


Figure 5: Architecture of the column processor.

Row and column processing require two slightly different implementations of the lifting step. During row processing, the pixels of an image are read from external memory in a row-wise fashion, and needed in a row-wise order, making a systolic architecture of Figure 3(a), which takes its inputs sequentially, ideally suitable. On the other hand, after row processing the pixels are stored in an embedded memory until enough column data is available to proceed with column processing. The inputs are best fed into the column processor in parallel, leading to the parallel architecture of Figure 3(b). The first-in-first-out (FIFO) buffers shown in Figure 3 are used to compensate for the latency of the pipelined multipliers within the lifting steps.

3.2 Row processor (RP)

Implementation of the row processor (RP) is straightforward; a functional block diagram is shown in Figure 4. It consists of six computing modules: P1, U1, P2, U2, G1 and G2. Each computing module implements the corresponding part of equation (2). Each lifting step (P1, U1, P2, and U2) is implemented using the systolic architecture of Figure 3, and the steps are cascaded to build the whole lifting structure. At each clock, P1 takes a pair of coefficients, even and odd, and produces a pair of outputs that is then fed to the next module U1. Similarly U1 feeds P2, which feeds U2. Finally, the outputs of U2 are scaled through G1 and G2, which are implemented as one-tap filters. The result is the one-dimensional DWT of the rows.

3.3 Column processor (CP)

Coefficients exit the row processor in row-wise order, and are stored in a embedded memory until enough rows are buffered so that column processing can begin. When using convolutional 9/7 wavelet filters, nine rows must be buffered. The column processing is then done in row-wise manner; the column filter works on nine pixels of data from a single column at a time. At each clock, the filter moves one column to the right. Our architecture reduces the amount of buffer space required by exploiting a lifting structure. The lifting structure staggers when the column data is needed; although we still need nine pixels of data from a column to complete one wavelet coefficient, we need only three pixels to start the first lifting step. The remaining pixels are used in later clocks by later lifting steps. We shall see that in all, buffer space for only seven rows is required.

A block diagram of the column processor is shown in Figure 5. The basic building blocks and their functions are the same as for the row processor except that the parallel lifting step architecture is used. The lifting step computations are scheduled on the modules of the column processor so as to minimize the amount of embedded memory needed to buffer data between the row and column processing while keeping both the row and column processor continually busy. We use ASAP scheduling: *each lifting step starts operation as soon as its inputs are available.* For the sake of simplicity of the figures, we assume that the latency of each computing module is one time unit. The schedule can be easily modified to account for additional latency.

The embedded memory is organized into seven lines of N words each. Each line has two parts: the first $N/2$ words are used to buffer approximation coefficients coming from the row processor, and the second $N/2$ words are used to buffer detail coefficients. At each clock, the row processor writes two new coefficients (one approximation and one detail). It moves from left to right along each line of the memory, and starts over at the beginning of the first line after it completes the seventh line.

As soon as the row processor (RP) completes the third row of coefficients, the first two lifting steps P1 and U1 start column processing. They travel left to right along the lines of memory, just behind the row processor. At each clock, P1 needs three inputs coming from a single column of the memory. One of the inputs is passed directly to it from the row processor. The other two are read from the embedded memory. Pl writes its result in the embedded memory, over the middle input (which will never be needed again). This result will be read by P2 at a later time. P1 also passes both the top input that it read and the result that it produced to U1. The other lifting steps follow along behind P1 in a similar way.

As an example, Figure 6 shows the contents of the memory as the row processor is in the process of computing coeffi-

cients $a_{6,5}$ and $d_{6,5}$ for an image of size 512 by 512 pixels, and the pattern of reads and writes in that time step. In the figure, subscripts (i, j) represent the ith row and jth column of the image. Coefficients without superscripts ($a_{i,j}$ and $d_{i,j}$) are approximation and detail coefficients directly out of the row processor. Coefficients with superscripts have been through one or more lifting steps, following the notation of Equation 2; for example, $da^1_{1,2}$ is the output of the column processor's first P1 step (d^1_2 in Equation 2) when working on the row processor's approximation coefficients from row 1.

Each lifting step requires three inputs and produces one output. In Figure 6, "R" indicates an input read from the embedded memory, while a lower case "r" indicates an input that has been forwarded from an earlier lifting step to the module in question. Data can be forwarded if the earlier module produced it one clock before, as is the case when P1 passes $da^1_{2,3}$ to U1, or read it from embedded memory one clock before, as is the case when P1 passes $a_{4,3}$ to U1. "W" indicates a write to the embedded memory, and "w" indicates data that is produced and passed on to the next module without actually writing that data to memory. The figure shows the lifting steps passing data one to the next, until finally the data is passed to the scaling modules G1 and G2. The scaling modules write 2-D DWT coefficients to the external memory.

In the time it takes for the column processor to complete one row of approximation coefficients (thereby travelling to the halfway point of the line of memory), the row processor, which has been writing one approximation and one detail coefficient at each clock, has reached the end of the line. While the column processor continues across the row, working on the detail coefficients from the row processor, the row processor continues on, completing its computations on the next row of the image. By the time the column processor completes one entire row, the row processor has completed two rows. The column processor then moves down two rows, as indicated in Equation 2. In this way, the row and column processors stay completely synchronized; no stalling is ever necessary.

Seven lines of memory are enough to successfully buffer data between row and column processing. All seven are used simultaneously when the column processor is working on the righthand side of the line, on the details coefficients coming out of the row processor. Then, the row processor working one row ahead of the column processor, so it is writing one line of the memory. The column processor itself is reading and writing from six additional lines of the memory, in a pattern like that of Figure 6.

3.4 Implementation of symmetric extension

For image compression, best results are achieved by symmetrically extending the image at the boundaries before filtering. For the convolutional 9/7 wavelet filters, the image is extended by four pixels on all four sides. While the extension is being input to the filter, the output of the filter is ignored (alternatively, the filter's computational hardware can be disabled); thus, four clock cycles at the beginning and four clock cycles at the end are wasted in processing each row and column of the image. When lifting structure filters are used, there is a simple way of implementing symmetric extension that does not waste any clock cycles. The outputs that each lifting step produces near the boundary are themselves symmetric. We can therefore apply the lifting structure filters to the original, unextended image, and exploit symmetry to determine what the coefficients that we didn't calculate should be. To demonstrate the approach, let us consider how the extension along the left edge of a row of data $x_0, x_1, x_2, ...$ is used to compute the first approximations coefficient. Computations are shown in the order in which they are done. The "steps" shown are not an exact notion of clock cycles because they do not account for latency of operations; we show each new lifting step output as soon as all the quantities needed to compute it are available. Straightforward implementation proceeds as follows:

step :	0	1	2	3	4	5	6	7	8	9	10	...
data :	x_4	x_3	x_2	x_1	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
P1 :			d^1_1		d^1_0		d^1_0		d^1_1		d^1_2	...
U1 :					a^1_1		a^1_0		a^1_1		a^1_2	...
P2 :							d^2_0		d^2_0		d^2_1	...
U2 :									a^2_0		a^2_1	...
G1 :									a_0		a_1	...

where the results are calculated using the equation (2). Note that the outputs of the first three lifting steps exhibit symmetry at the boundary. The output of predict steps P1 and P2 exhibits even symmetry (the edge coefficients d^1_0 and d^2_0 are reflected across the boundary), and the output of update step U2 exhibits odd symmetry (the edge coefficient a^1_0 is not reflected). We take advantage of this symmetry to compute a_0 without doing any redundant computations (and without wasting clock cycles). Instead of inputting the row data with a symmetric extension on the left edge, we input it starting directly with x_0, and we compute instead:

step :	0	1	2	3	4	5	6	...
data :	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
P1 :			d^1_0		d^1_1		d^1_2	...
U1 :			a^1_0		a^1_1		a^1_2	...
P2 :					d^2_0		d^2_1	...
U2 :					a^2_0		a^2_1	...
G1 :					a_0		a_1	...

by modifying the equations for a^1_0 and a^2_0 to:

$$\begin{aligned} a^1_0 &= 2\beta d^1_0 + x_0 \\ a^2_0 &= 2\delta d^2_0 + a^1_0. \end{aligned} \tag{3}$$

Modifications at the right edge of the row are similar. Without exploiting symmetry, all stages of the hardware are busy through step 514. We have:

step :	...	507	508	509	510	511	512	513	514
data :	...	x_{507}	x_{508}	x_{509}	x_{510}	x_{511}	x_{510}	x_{509}	x_{508}
P1 :	...		d^1_{253}		d^1_{254}		d^1_{255}		d^1_{254}
U1 :	...		a^1_{253}		a^1_{254}		a^1_{255}		a^1_{254}
P2 :	...		d^2_{252}		d^2_{253}		d^2_{254}		d^2_{255}
U2 :	...		a^2_{252}		a^2_{253}		a^2_{254}		a^2_{255}
G1 :	...		a_{252}		a_{253}		a_{254}		a_{255}

However, if we exploit symmetry a number of intermediate

line	approximations								details							
0	$aa^2_{0,0}$	$aa^2_{0,1}$	$aa^2_{0,2}$	$aa^2_{0,3}$	$aa^2_{0,4}$	$aa^2_{0,5}$	$\cdots$	$aa^2_{0,255}$	$ad^2_{0,0}$	$ad^2_{0,1}$	$ad^2_{0,2}$	$ad^2_{0,3}$	$ad^2_{0,4}$	$ad^2_{0,5}$	$\cdots$	$ad^2_{0,255}$
1		U2:R														
	$da^2_{0,0}$	$da^2_{0,1}$	$da^2_{0,2}$	$da^2_{0,3}$	$da^2_{0,4}$	$da^2_{0,5}$	$\cdots$	$da^2_{0,255}$	$dd^2_{1,0}$	$dd^2_{1,1}$	$dd^2_{1,2}$	$dd^2_{1,3}$	$dd^2_{1,4}$	$dd^2_{1,5}$	$\cdots$	$dd^2_{1,255}$
2		U2:r	P2:R													
		$aa^1_{1,1}$	$aa^1_{1,2}$	$aa^1_{1,3}$	$aa^1_{1,4}$	$aa^1_{1,5}$	$\cdots$	$aa^1_{1,255}$	$ad^1_{2,0}$	$ad^1_{2,1}$	$ad^1_{2,2}$	$ad^1_{2,3}$	$ad^1_{2,4}$	$ad^1_{2,5}$	$\cdots$	$ad^1_{2,255}$
	G1:r	U2:w														
	$aa^2_{1,0}$	$aa^2_{1,1}$														
3			P2:r	U1:R												
			$da^1_{1,2}$	$da^1_{1,3}$	$da^1_{1,4}$	$da^1_{1,5}$	$\cdots$	$da^1_{1,255}$	$dd^1_{3,0}$	$dd^1_{3,1}$	$dd^1_{3,2}$	$dd^1_{3,3}$	$dd^1_{3,4}$	$dd^1_{3,5}$	$\cdots$	$dd^1_{3,255}$
		U2,G2:r	P2:W													
	$da^2_{1,0}$	$da^2_{1,1}$	$da^2_{1,2}$													
4				U1:r	P1:R											
				$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$\cdots$	$a_{4,255}$	$d_{4,0}$	$d_{4,1}$	$d_{4,2}$	$d_{4,3}$	$d_{4,4}$	$d_{4,5}$	$\cdots$	$d_{4,255}$
			P2:r	U1:W												
	$aa^1_{2,0}$	$aa^1_{2,1}$	$aa^1_{2,2}$	$aa^1_{2,3}$												
5					P1:R											
					$a_{5,4}$	$a_{5,5}$	$\cdots$	$a_{5,255}$	$d_{5,0}$	$d_{5,1}$	$d_{5,2}$	$d_{5,3}$	$d_{5,4}$	$d_{5,5}$	$\cdots$	$d_{5,255}$
				U1:r	P1:W											
	$da^1_{2,0}$	$da^1_{2,1}$	$da^1_{2,2}$	$da^1_{2,3}$	$da^1_{2,4}$											
6					P1:r	RP:W								RP:W		
	$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$		$-$	$d_{6,0}$	$d_{6,1}$	$d_{6,2}$	$d_{6,3}$	$d_{6,4}$	$d_{6,5}$		$-$

Figure 6: Scheduling of reads and writes for the embedded memory when the column processor is working on approximations. "R" and "W" designate reads and writes to the memory, while "r" and "w" designate data that is passed from one lifting step to another without going through the memory.

computations need not be done:

step :	...	507	508	509	510	511	512	513	514
data :	...	x_{507}	x_{508}	x_{509}	x_{510}	x_{511}	(start next row)		
P1 :	...		d^1_{253}		d^1_{254}		d^1_{255}		
U1 :	...		a^1_{253}		a^1_{254}		a^1_{255}		
P2 :	...		d^2_{252}		d^2_{253}		d^2_{254}		d^2_{255}
U2 :	...		a^2_{252}		a^2_{253}		a^2_{254}		a^2_{255}
G1 :	...		a_{252}		a_{253}		a_{254}		a_{255}

using modified equations:

$$\begin{aligned} d^1_{255} &= 2\alpha x_{510} + x_{511} \\ d^2_{255} &= 2\gamma a^1_{255} + d^1_{255}. \end{aligned} \tag{4}$$

There is no need to symmetrically extend the right end of the row input data, and the next row of data starts entering the pipeline even before this row has completed (in step 512). While there is a lag of four steps in waiting for the approximations, no cycles are wasted; the pipeline is always fully utilized, with no stalling.

The lifting step hardware of Figure 3 can be easily modified to implement this approach to symmetric extension.

4. RESULTS

We now discuss the hardware properties of our system. In what follows, the image is N by N pixels, and we want to perform an L-level two-dimensional DWT. The first level works on all N^2 pixels of the original image; each successive level works on the approximations aa produced by the level before, so that each level uses one-fourth of the pixels of the level before it. Levels are indexed from 0 to $l-1$. The jth level computes $N^2/4^j$ coefficients; altogether, all L levels compute a total of

$$C = \sum_{j=0}^{L-1} \frac{N^2}{4^j} = \frac{4}{3}N^2(1 - \frac{1}{4^L}) \tag{5}$$

coefficients.

4.1 External memory

In the proposed architecture, the image is stored in external memory. Results are written back to external memory after each level. The level computations are done in-place in the external memory, with the new coefficients directly overwriting the old ones; as a result, a memory no larger than the original image size is sufficient.

Each level does one read and one write of the external memory for each coefficient that it produces. The total number of external memory accesses needed to compute all L levels is:

$$N_{ext-mem-acc} = 2C = \frac{8}{3}N^2(1 - \frac{1}{4^L}). \tag{6}$$

One important distinction between our architecture and others is that ours reads each pixel only once even when implementing symmetric extension. For architectures using convolutional filter banks, the image must be extended on all four sides; one way to accomplish this is using additional reads of the external memory along the image boundaries.

4.2 Embedded memory

For the 9/7 wavelet, seven lines of embedded memory, each with N words, is required. The architecture can easily be extended to accommodate more lifting steps. Lifting steps come in predict and update pairs; each additional pair of lifting steps requires an additional pair of lines of

embedded memory. In general, the proposed architecture requires $l_s + 3$ lines of memory, where l_s is the number of lifting steps. This should be compared to convolutional architectures, which require $l_f + 1$ lines of embedded memory, where l_f is the longest filter length. For the 9/7 wavelet, the embedded memory of our lifting-based architecture is 30% smaller than that required for a similar architecture using convolutional filter banks.

If each line is implemented in a separate bank of embedded memory, as shown here, each memory bank needs two write ports and one read port. It is also possible to use memory banks with one write port and one read port by implementing two banks for each line, splitting the line at the halfway point into approximations and details.

Of the coefficients produced by the jth level, half are approximations and half are details. For each coefficient, the row processor must do one write to the embedded memory. For approximation coefficients, the schedule of Figure 6 is used, and (looking at the "R"s and "W"s in the diagram), the four lifting steps do a total of eight embedded memory accesses. (In general, l_s lifting steps do $2l_s$ accesses.) For detail coefficients, the four lifting steps do nine ($= 2l_s + 1$) accesses; one more read is required for the first step because the column processor is not following directly behind the row processor. The total number of embedded memory accesses is:

$$\begin{aligned} N_{emb-mem-acc} &= C + (2l_s)\frac{C}{2} + (2l_s+1)\frac{C}{2} \\ &= \frac{2}{3}N^2(4l_s+3)(1-\frac{1}{4^L}). \end{aligned} \quad (7)$$

A similar architecture using convolutional 9/7 filters would require about five percent more embedded memory accesses; for each coefficient, ten accesses are required (the row processor does one write, and the column processor does nine ($= l_f$) reads, compared to 9.5 accesses for our lifting-based architecture.

4.3 System performance

In the proposed architecture, the levels are computed iteratively, and computations on a given level can begin as soon as the row processor is done with the level before. Computations are completely pipelined, with no stalling between row and column processing or between levels, and with the row and column processors continually busy once the pipeline is full, so that 100% hardware utilization is achieved. As detailed in section 3.4, no extra cycles are required to implement symmetric extension. Therefore, the number of clock cycles needed to perform the L-level 2-D DWT is simply

$$N_{\text{clock cycles}} = L_s + C = L_s + \frac{4}{3}N^2(1-\frac{1}{4^L}), \quad (8)$$

where L_s is the latency of the system. The latency of the system depends on the latencies of the row and column processors, which in turn depend on how heavily they are pipelined, and on the lag between row and column processing required to gather enough column data to start the first column lifting step. The column processor starts producing its first output after the row processor generates l_s lines of pixels, so that lag is $l_s N$ cycles long.

The number of clock cycles required for an architecture that uses convolutional filter banks is higher, because it must do explicit additional computations in order to implement symmetric extension. The additional computations include l_f pixels involved in the extension of each row and column of the approximations, at each level. The total number of additional computations is

$$2(l_f - 1)\sum_{j=0}^{L-1}\frac{N}{2^j} = 2N(1-\frac{1}{2^L}). \quad (9)$$

For a five-level 2-D DWT of a 512 by 512 image, 4.5% of the computations done by a convolutional filter banks are solely to calculate the symmetric extension. Because the proposed lifting-based architecture need not do these calculations, it will require fewer clock cycles to complete, and have proportionately higher throughput.

5. CONCLUSIONS

An efficient architecture implementing a lifting-based two-dimensional discrete wavelet transform is presented. Use of the lifting structure allows implementation of symmetric extension without additional computations, giving lifting-based architectures a significant advantage over convolutional filter bank-based architectures in terms of throughput. Using the proposed scheduling for column processing operations, the lifting-based architecture also requires significantly less embedded memory than a similar convolutional filter bank-based architecture. The architecture proposed is easily scalable to accommodate additional lifting steps.

6. REFERENCES

[1] P.-C. Wu and L.-G. Chen, "An Efficient Architecture for Two-Dimensional Discrete Wavelet Transform," *IEEE Trans. Circuits and Syst. Video Tech.*, Vol. 11, No. 4, pp. 536-545, April 2001.

[2] M. Vishwanath, R.M. Owens, and M.J. Irwin, "VLSI Architecture for Discrete Wavelet Transform," *IEEE Trans. Circuits and Systems*, Vol. 42, pp. 305-316, May 1995.

[3] W. Sweldens, "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Construction," *Proc. SPIE*, Vol. 2569, pp. 68-79, 1995.

[4] I. Daubechies and W. Sweldens, "Factoring Wavelet Transforms into Lifting Steps," *J. Fourier Analysis and Applications*, Vol. 4, No. 3, pp. 243-269, 2001.

[5] M. Ferretti and D. Rizzo, "A Parallel Architecture for the 2-D Discrete Wavelet Transform with Integer Lifting Scheme," *J. VLSI Signal Processing*, Vol. 28, pp. 165-185, 1994.

[6] M. Vishwanath, "The Recursive Pyramid Algorithm for the Discrete Wavelet Transform," *IEEE Trans. Signal Processing*, Vol. 42, No. 3, pp. 673-676, March 1994.

[7] W.-H. Chang, Y.-S. Lee, W.-S. Peng, and C.-Y. Lee, "A Line-based, Memory Efficient and Programmable Architecture for 2D DWT using Lifting Scheme," *Intl. Symp. Circuits and Systems*, vol. 4, pp. 330-333, 2001.

[8] N.D. Zervas, G.P. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, and C.E. Goutis, "Evaluation of Design Alternatives for the 2-D Discrete Wavelet Transform," *IEEE Trans. Circuits and Syst. Video Tech.*, Vol. 11, No. 12, pp. 1246-1262, December 2001.

[9] C. Chrysafis and A. Ortega, "Line-Based, Reduced Memory, Wavelet Image Compression," *IEEE Trans. Circuits and Syst. Video Tech.*, Vol. 9, No. 3, pp. 378-389, March 2000.

[10] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform," *IEEE Trans. Signal Processing*, Vol. 50, No. 4, pp. 966-977, April 2002.