A Scalable And Programmable Architecture For 2-D DWT Decoding

Massimo Ravasi, Livio Tenze and Marco Mattavelli

Abstract—**The compression of still images by means of the Discrete Wavelet Transform (DWT), adopted in the JPEG-2000 and MPEG-4 standards, is becoming more and more widespread because it yields better performances than other compression methods, such as DCT. The demand of efficient architectures for 2-D DWT coding and decoding for a variety of different applications and embedded systems is rapidly increasing. This paper presents the implementation of a 2-D DWT decoder for Mallat tree decomposition, suitable for low power applications, such as portable devices. The decoder design has been synthesized and validated in 0.35 μm CMOS technology. The architecture is scalable according to the desired maximum image size, the maximum DWT kernel length and arithmetic accuracy, and it is programmable at run-time to process different image sizes and to use different DWT kernels.**

Index Terms—**Image coding, Wavelet transforms, Discrete transforms, Digital Signal Processors**

I. Introduction

TEXTURE coding based on wavelet transform is playing a leading role for its higher performances in terms of signal analysis, multi-resolution features and improved compression compared to existing methods such as the DCT based compression schemes adopted in the old JPEG standard. This success is testified by the fact that the wavelet transform has now been adopted by MPEG-4 for still texture coding and by JPEG-2000. Indeed, superior performance at low bit-rates and transmission of data according to client display resolution are particularly interesting for mobile applications. The wavelet transform shows better results because, thanks to its time-scale representation, it is intrinsically well suited to non-stationary signal analysis, such as images [10]. Although it is a rather simple transform, DWT implementations may lead to critical requirements in terms of memory size and bandwidth possibly yielding costly implementations. Extended state of the art researches showed that DWT coding and decoding algorithms can be redesigned by changing the scheduling of operations, yielding more efficient implementations with reduced memory requirements [3], [4], [6]. Further work proposed a variety of strategies, dealing with the trade-off among implementation complexity, cache memory requirements and external memory requirements [1], [2]. Thus, efficient implementations must be investigated to fit different system scenarios. In other words the goal is to find different architectures each of them specifically optimized for any specific system requirement in terms of complexity and memory bandwidth.

Because the iterative sub-band decomposition is the core process of wavelet transforms, the coding/decoding stage has to be performed on several layers as shown in Fig. 1 in the case of 1-D Mallat tree decomposition where only approximation signals (*Lx*) are recursively split into two sub-signals. Along the tree decomposition it can be noticed that in intermediate layers some data are just temporary. For instance, in layer 1 represented in Fig. 1, *L*1 signal data is produced while coding input signal and is successively split into *L*2 and *H*2. Similarly, such temporary data can be found also in the reconstruction process.

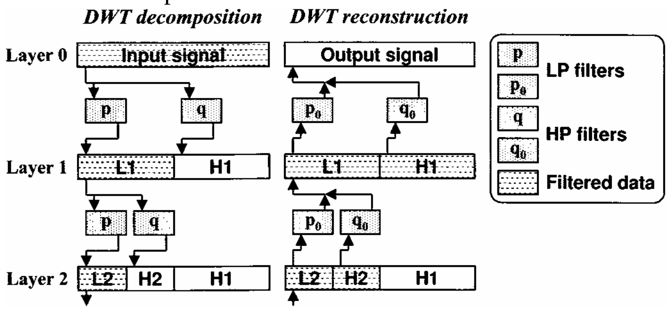

Fig. 1. 1D DWT with Mallat tree decomposition. The number of samples of the encoded signal is equal to the one of the input signal.

2D DWT coding is usually based on separable basic scaling functions and wavelet bases so that it can be performed iterating two orthogonal 1-D DWT. This fact implies the presence of additional temporary samples between horizontal and vertical processing. As shown in Fig. 2 for the 2-D DWT reconstruction, not only the temporary signal *LL*1 is needed, produced by decoding layer 2, but also temporary signals *L*1 and *H*1, produced as result of horizontal processing of layer 1 and required as input to vertical processing of the same layer are necessary.

M. Ravasi and M. Mattavelli are with the Integrated Systems Laboratory (LSI) of the Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland (telephone: +41-21-6936978, e-mail: massimo.ravasi@epfl.ch).

L. Tenze is with the D.E.E.I. of the University of Trieste, I-34100 Trieste, Italy (telephone: +39-40-6767147, e-mail: tenze@ipl.univ.trieste.it).

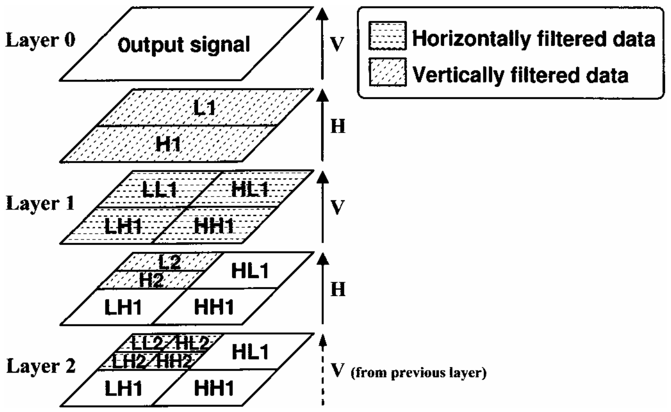

Fig. 2. 2D DWT with Mallat tree decomposition. The size of intermediate layers decreases twice faster than in 1D case and the amount of data to be filtered tends asymptotically to 4/3 of the size of the input signal. Since data must be filtered both horizontally and vertically, the total amount of filtered samples tends to 8/3 of the size of input signal.

Practical system limitation and requirements encountered by the designer include memory size and bandwidth for the storage of the temporary data, and the efficient use of both on-chip and off-chip storage [1]-[7]. Therefore redesigning the data processing scheduling and the memory storage scheme allows a joint optimization of the algorithmic and architectural features according to specific system requirements. The optimum choice of these factors can be achieved by analyzing different strategies. Each of these strategies corresponds to an implementation characterized in parametric form in terms of generic architectural features such as on-chip memory size, on-chip data-path bandwidths, overall filter complexity, external memory size and external data-path bandwidths [1]-[3]. This paper presents the implementation of an hardware DWT decoder based on “Sliding Windows Layer-by-Layer” architecture (SW-LbL) [1], [2]. Among the different approaches studied in literature, the SW-LbL approach [1], [2] offers a good trade-off between system performance and complexity. The main result is a sensible reduction of the bandwidth and of the size of external memory, at the price of a small increase of implementation complexity.

This paper is organized as follows: Section II presents the “Sliding-Windows Layer-by-Layer” architecture [1], [2] chosen to implement the DWT decoder, comparing it with the classical architecture. Section III describes the VHDL implementation of the decoder. Section IV reports the results of the synthesis in terms of performance and system requirements. Section V concludes the paper summarizing the main results achieved.

II. “Sliding-Windows Layer-by-Layer” DWT decoder

A. *Classical approach*

The classical approach to 2D DWT decoding (see Fig. 2) is to process each layer in the tree decomposition separately and to process the vertical and horizontal layers successively one after the other. The performance of this approach is strongly limited by the management of temporary data required between two successive layers and between horizontal and vertical filtering.

Let’s consider an input image with resolution of $W \cdot H$ samples encoded on *L* layers with Mallat tree decomposition. As shown in Fig. 2, while processing layer 1 the amount of temporary data between horizontal and vertical filtering is equal to the size of the image to be decoded. Even with relatively small image resolutions, the memory required to store these temporary data might be in general too large to be implemented on-chip, therefore an external memory (*EM*) must be used and its size, measured in samples, is

$$EM_{s,Classical} = WH \,. \tag{1}$$

The amount of data to be filtered on each layer decreases by a factor of 4 from one layer to the next one and the total amount of processed data along the whole tree reconstruction process is given by:

$$D = \sum_{l=1}^{L} \frac{WH}{4^{l-1}} = \frac{4^{L}-1}{3 \cdot 4^{L-1}} WH \approx \frac{4}{3} WH \,. \tag{2}$$

The last term of the expression 2 approximates in excess the exact result with an error smaller than 0.4% already for $L \geq 4$, hence, the approximated value for *D* will be used in the following expressions. Because of horizontal and vertical filtering, these data must be read and written twice on each layer, thus:

$$D_{r,Classical} = D_{w,Classical} = 2D = \frac{8}{3} WH \,. \tag{3}$$

Assuming that the compressed image bitstream is read by the decoder from the outside and that the decoded image is sent outside as the decoder output signal, the external memory is used only to store temporary data and the total amount of samples exchanged with the external memory is:

$$\begin{aligned} EM_{r,Classical} &= D_{r,Classical} - WH \\ EM_{w,Classical} &= D_{w,Classical} - WH, \end{aligned} \tag{4}$$

that is equivalent to:

$$EM_{r,Classical} = EM_{w,Classical} = \frac{5}{3} WH \,. \tag{5}$$

The implementation of the decoder consists mainly of two 1-D linear filters with a small support interval (e.g., DWT kernels suggested in JPEG 2000 are not longer than 18 samples). These filters can be used both for vertical and horizontal processing because these two operations never occur at the same time. Thus, the implementation of a decoder according to the Classical approach is rather simple and its main disadvantage is the need of a large external memory and, above all, of a large data exchange with it. Indeed external memory is less efficient than on-chip cache memory both in terms of available bandwidth, power consumption and overall system complexity. The management of temporary data is thus the main bottleneck of the Classical approach.

B. *Sliding-Windows Layer-by-Layer approach*

A redesigning of the DWT algorithm, in order to change the operation scheduling, allows to reduce the temporary data lifetime and to optimize the overall system performance [1]-[3]. The SW-LbL approach [1], [2] allows to significantly reduce both external memory size and bandwidth at the price of a slightly more complex implementation and the need of a small on-chip memory. The main idea behind this approach is to exploit data dependencies between horizontal and vertical processing in order to try to use temporary samples as soon as they are available.

The scheme of Fig. 3 shows how to manage temporary samples between horizontal and vertical filtering in order to reduce their lifetime. Let's suppose that the horizontal filters are applied first. Horizontal filters produce samples along rows, while vertical filters need input samples along columns. To generate these columns of samples with horizontal filters, a set of horizontal filters could be used, with a couple of *p*-*q* filters for each line. In this way, scheduling the horizontal filters line by line, columns of temporary data capable of feeding vertical filters are generated. Actually it is not needed to implement a couple of filters for each line because they never work in parallel, only one line is active at a time. It is just needed to store 2 columns of samples necessary as input by all the virtual horizontal filters and read them line by line to load them in parallel in a true real couple of horizontal *p*-*q* filters. These two columns behave like two windows sliding over the input image, because they cache successive columns of input samples. This is the origin of the name for this approach. Supposing to put the sliding-windows memory on chip, to reconstruct layer *n* it is necessary to exchange with external memory only input layer data (layer *n*-1 coded signals) and output layer data (reconstructed layer *n*), while all the temporary samples between horizontal and vertical filtering are managed by on-chip memory.

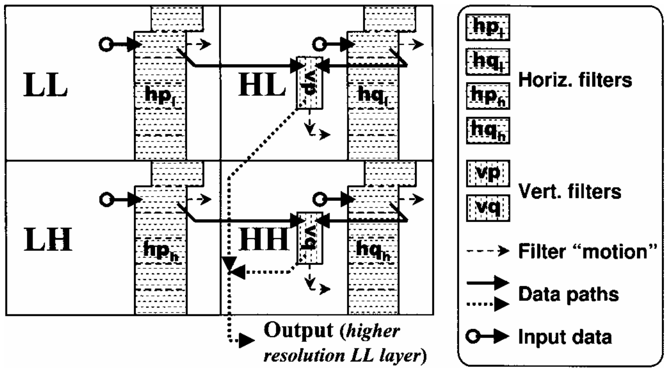

Fig. 3. "Sliding-Windows Layer-by-Layer" DWT decoder. The lifetime of temporary data between horizontal and vertical filtering is minimized by using them as soon as possible after their creation. The two columns of coupled horizontal filters are virtually implemented by means of a cache memory and of one couple only of real filters working line by line.

In general the memory required to store temporary data between two successive layers is still too large to be implemented on-chip. Looking at Fig. 2 and keeping in mind that temporary data between horizontal and vertical filtering no longer need to be stored on external memory, it can be noticed that the maximum amount of temporary data to be managed is given by *LL*1 signal storage that can be expressed as:

$$EM_{s,SW-LbL} = WH/4 . \tag{6}$$

Concerning data exchange with external memory, it can be observed that with respect to the Classical approach it is no longer needed to read and write temporary data between horizontal and vertical filtering. That is, in previous expression 5, instead of considering $2D$ samples (because of expression 3) only D samples must be considered:

$$\begin{aligned} EM_{r,SW-LbL} &= EM_{w,SW-LbL} \\ &= EM_{r,Classical} - D = \frac{5}{3}WH - D, \end{aligned} \tag{7}$$

that , according to equation 2, results in:

$$EM_{r,SW-LbL} = EM_{w,SW-LbL} = \frac{1}{3}WH . \tag{8}$$

Considering Fig. 3 it can be observed that the size of the memory required to store the two columns of samples *hp* and *hq* depends on the size of the reconstruction filters. Such size depends on the chosen DWT kernel, and on the height of the highest layer to reconstruct, that is on the height *H* of the image. Let K_s be the kernel size:

$$K_s = |p| + |q| , \tag{9}$$

and let $K_{s,MAX}$ be the maximum supported kernel size, it results that:

$$K_{s,MAX} = |p|_{MAX} + |q|_{MAX} . \tag{10}$$

According to the DWT reconstruction process, input data must be up-sampled before being processed by reconstruction filters *p* and *q*. Therefore *p* and *q* filters may be replaced by two couples of filters p_e-q_e and p_o-q_o, whose length is half of that of the original filters, working alternately on the same input data to produce respectively output samples in even and odd positions. Therefore, the total size of the inter-layer temporary memory is given by:

$$IM_{s,SW-LbL} = \frac{K_{s,MAX}}{2} H \,, \tag{11}$$

and considering that typical kernel lengths range up to 20 samples [8], this memory, referred here as *Internal Memory* (*IM*), can be conveniently and efficiently implemented on-chip.

Every line of the Sliding-Windows memory holds the input data of the corresponding horizontal virtual filter, thus emulating the functionalities of a FIFO. For each *L-H* couple of input samples, two output samples are produced, and this implies that for each output sample a new input sample has to be stored in virtual filter's FIFO, according to expression:

$$IM_{w,SW-LbL} = \sum_{l=1}^{L} \frac{WH}{4^{l-1}} \approx \frac{4}{3} WH \,. \tag{12}$$

Having to work along columns, whole lines of data in Sliding-Windows memory have to be reloaded, line by line, into the horizontal filter to produce each column of output data. Consequently, the amount of data read from Sliding-Windows memory is larger than the amount of written data:

$$IM_{r,SW-LbL} = \sum_{l=1}^{L} \frac{WH}{4^{l-1}} \cdot \frac{K_s}{2} \approx \frac{2}{3} WH \cdot K_s \,. \tag{13}$$

This relatively high bandwidth requirement can be easily satisfied by the high-speed performances provided by available on-chip memory cores.

By means of a small on-chip cache, the SW-LbL approach achieves a reduction of the external memory size to a quarter of that of the Classical approach and the memory bandwidth to a fifth. In the following section the most significant issues about the hardware implementation of the SW-LbL decoder are discussed. Moreover, the main results achieved and performances related to system requirements are presented.

III. SW-LbL Decoder implementation

The architecture of a SW-LbL decoder is composed by four main blocks: the horizontal filters block (HOR_FIL), the vertical filters block (VER_FIL), the state machine that synchronizes all the operations at layer level (FIR_CONTROL) and the state machine ruling the whole system functionalities at high level, from parameters setup to decoding along all layers (SCHEDULE). It will be shown that the most complex part of this implementation is the FIR_CONTROL, because the synchronization of all the operations (mainly the synchronization of vertical and horizontal filtering, in steady state as well as at the borders) is not a simple task.

At the synthesis stage, the SW-LbL decoder can be scaled in terms of maximum image sizes, maximum kernel length, samples word-size and kernel coefficients word-size. At run-time, the decoder can be programmed to process any image size within the maximum image size, with any kernel of any length within the maximum kernel length.

While it has been shown that DWT kernel filters can be efficiently implemented using the Lifting Scheme [8], [9], it is chosen here to implement the filters with the classical filters scheme. Despite the fact that the classical filter scheme needs a larger Sliding-Windows memory than the Lifting Scheme (the total number of taps of Lifting-Scheme's filters is about half of that of the corresponding classical scheme), the classical scheme presents several advantages that make it preferable for several reasons:

- Both in the classical scheme and in the Lifting-Scheme, the filters are typically non-causal. Having to implement the whole kernel system as a causal system, the lifting scheme presents some obvious problems because not only each filter must be "delayed" to make it become causal but also an extra delay must be introduced on the line to which the filter's output is added in order to preserve the correct synchronization. This extra delay can either be implemented reusing the FIFO line of the previous step's filter, when available and with enough taps, or introducing another ad-hoc FIFO line. In the classical scheme both filters just need to be delayed of the same number of samples, which needs no extra delays and does not give any further synchronization problem. The same does not apply for the Lifting-Scheme.
- The programmability of the kernel is drastically more complex in Lifting-Scheme. In the classical scheme it is just needed to store the kernel's coefficients, store the divisor at the end of each filter and synchronize the output in order to take into account the delay introduced to make filters causal. In Lifting-Scheme such operations must be performed at each step, in addition it is required to synchronize any step in order to respect the delay introduced by the previous step. As explained above, depending on the kernel either the FIFO line of previous step's filter can be reused or another FIFO line can be used, implying the need of more multiplexers and making the overall set-up by far more complex than the classical filter scheme.
- According to JPEG-2000 specifications, samples at signal borders must be mirrored. Depending on the filter's impulse response, samples may be mirrored with respect either to first (last) sample or to half sampling interval before (after) first (last) sample. While with the classical scheme we can easily reuse the taps of the filters to implement this feature, with the Lifting-Scheme this implementation is not straightforward.
- As shown with equation 12, for each processed sample, it is necessary to write one sample only in the Internal Memory. With the Lifting-Scheme, one sample has to be written for any step with impulse response of more than one sample as well as for all the extra delay lines. Besides increasing the bandwidth to write on Internal Memory, the complexity of the management of the data to be written would increase as well.

All the filtering operations are based on integer arithmetic. Experimental results with floating point kernels of JPEG-2000 showed that the PSNR loss of decoded image can be kept smaller than 2 dB by storing processed samples on 15 bits integers and scaling floating point coefficients in order to map them on 9 bits integers.

So as to manage the different timing requirements among the inputs of the filters and their outputs, and among the speed of the internal memory and the external one, it has been chosen to use two clocks: the first clock (CLK) controls the output of the filters and the load and store operations of the external memory, whereas the second clock (CLK'), 20 times faster than CLK, controls the input of the horizontal filter and the load and store of the internal cache.

The main blocks of the decoder are the horizontal and the vertical filtering blocks: every block implements the low-pass and a high-pass filter for DWT reconstruction. The filters have been implemented using a poly-phase approach as shown in Fig. 4.

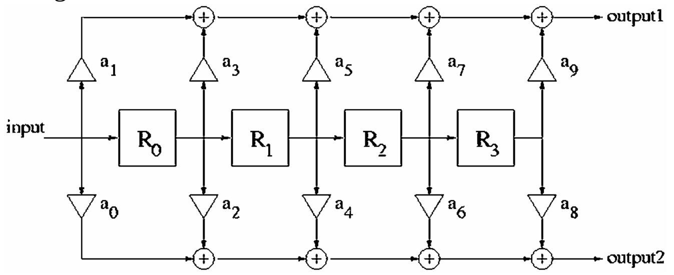

Fig. 4. Poly-phase implementation of kernel filters.

For every input sample two results are obtained. These results correspond to the two steps of a classical filtering process using the original data interlaced with zeros. The two outputs of both filters have to be appropriately added, as shown in Fig. 5, in order to produce samples in even and odd positions.

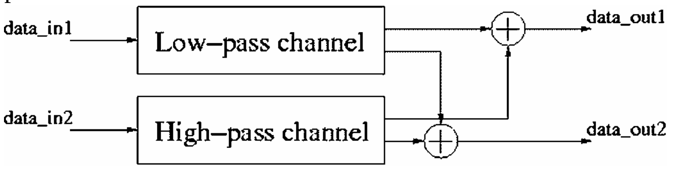

Fig. 5. Output data in even and odd positions (respectively data_out1 and data_out2) according to poly-phase implementation of kernel's low-pass and high-pass filters.

The input samples for the horizontal filters are stored in the internal cache while a whole line of samples has to be loaded at every CLK clock cycle to let both the horizontal filters produce an output value. Samples are loaded from internal cache at CLK' rate, in order to load all required samples within the available time according to CLK. The vertical filtering block is clocked just by CLK clock, to synchronize both the initialization process and the outputs.

Both horizontal and vertical borders of input signal must be mirrored in the same way but, because of the different implementations of horizontal and vertical filters, the mirroring operations are implemented in two completely different manners. For horizontal filtering, the border mirroring is implemented reading data twice from internal cache, first backward then forward for the beginning border, first forward than backward for the ending border. The number of memory accesses to the internal cache is not affected by mirroring, because the number of written and read samples is exactly the same like in the steady state. Within the vertical filtering block a set of multiplexers is in charge of carrying the input sample to the output of any register of the FIFO line (beginning border, multiplexers "B" in Fig. 6) or to reuse samples already in the FIFO line itself as input value (ending border, multiplexer "E" in Fig. 6). In both horizontal and vertical blocks, all the mirroring operations depend on parameters to be set during system initialization in order to correctly mirror input data according to the chosen kernel and image sizes.

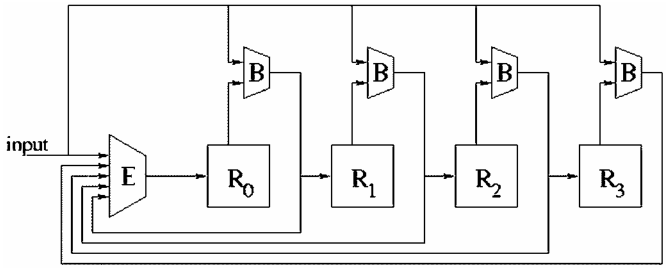

Fig. 6. Border mirroring in vertical filter block. The multiplexers "B" implement mirroring operations at the beginning of the column, by allowing carrying the input sample to any register's output, while the multiplexer "E" implements the mirroring operations at the end of the column, by allowing reusing as input sample the value stored in any register.

The most critical issue during the decoding process is the synchronization of operations between HOR_FIL and VER_FIL. To describe how these two blocks interact, it is necessary to distinguish between columns in even and odd position. In order to produce two decoded samples along a column, the filter blocks are fed with 4 samples, as shown in Fig. 7. HOR_FIL block produces two samples at the same time, the first in an even column and the second on the same row in the following odd column. Since the vertical block have to process one column at a time, the first sample produced by horizontal block can be passed immediately to vertical block while the other sample has to be stored in the internal cache in order to delay its use after current column has been completely processed. Seemingly VER_FIL needs two consecutive input samples on a column before being able to compute two output samples, therefore it has to wait for HOR_FIL to process a "square" of 4 samples on two rows before producing two output samples.

While producing an even column of output samples, HOR_FIL and VER_FIL work together. Each clock cycle HOR_FIL produces 2 temporary samples on even and odd columns and the two of them on the even column are immediately processed by VER_FIL. Producing the successive odd column, only VER_FIL works, using for input the samples previously produced by HOR_FIL and stored in the internal cache. In both cases, VER_FIL produces two samples at the same time each two clock cycles. The first sample is passed immediately to the output while the second one is delayed of one clock cycle, in order to produce, in steady state, one output sample at each clock cycle.

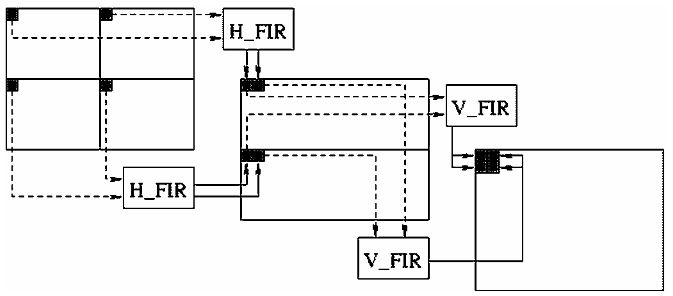

Fig. 7. Basic data flow in the decoding process. Horizontal filters (H_FIR) produce at the same time two columns of (temporary) output data while these columns have to be fed one after the other into vertical filters (V_FIR).

The FIR_CONTROL block is basically a state machine in charge of synchronizing all the steps to decode one layer. First it resets the filtering blocks, initializing them to process a new layer, then it controls the behavior of HOR_FIL and VER_FIL in order to synchronize the two blocks to decode a layer, take care of border mirroring, take into account the delay introduced by causal kernel filters and manage the accesses to internal cache.

The SCHEDULE block controls the whole decoding process interacting with FIR_CONTROL. Its main tasks are initializing FIR_CONTROL to start decoding a layer, controlling input and output operations and managing the transition from one layer to the next one when FIR_CONTROL notifies that a layer has been completely processed.

IV. Performance and Requirements

This section presents the results obtained with a VHDL implementation of the SW-LbL DWT decoder. The decoder has been synthesized and validated in 0.35 µm CMOS technology.

The parameters chosen to customize the implementation of the decoder and the tables of the corresponding results obtained after synthesis are reported below:

- Maximum image size: 768x512 pixels.
- Maximum kernel length: 10 taps both for high pass and low pass channel.
- Data samples' word-size: 16 bits.
- Kernel coefficients' word-size: 16 bits.

TABLE 1
HOR_FIL

Description	***Value***	***Unit***
Ports	101	#
Multipliers (16x16 bits)	20	#
Adders (32 bits)	4	#
Data registers (16 bits)	8	#
Kernel coefficient registers (16 bits)	20	#
Combinatorial area	2.54	mm^2
Noncombinatorial area	0.25	mm^2
Net Interconnect area	0.42	mm^2
Total area	3.20	mm^2

TABLE 2
VER_FIL

Description	***Value***	***Unit***
Number of ports	102	#
Multipliers (16x16 bits)	20	#
Adders (32 bits)	4	#
Data registers (16 bits)	8	#
Kernel coefficient registers (16 bits)	20	#
Combinatorial area	2.52	mm^2
Noncombinatorial area	0.26	mm^2
Net Interconnect area	0.38	mm^2
Total area	3.17	mm^2

TABLE 3
FIR_CONTROL

Description	***Value***	***Unit***
Ports	100	#
Combinatorial area	0.93	mm^2
Noncombinatorial area	0.14	mm^2
Net Interconnect area	0.18	mm^2
Total area	1.25	mm^2

TABLE 4
SCHEDULE

Description	***Value***	***Unit***
Ports	84	#
Combinatorial area	0.26	mm^2
Noncombinatorial area	0.16	mm^2
Net Interconnect area	0.08	mm^2
Total area	0.49	mm^2

We tested the synthesized chip using JPEG2000's 7x9 floating point kernel, with an image coded over 6 layers. The steady state throughput of the decoder is one sample per clock cycle and, according to equation 2, we find that the minimum time required to decode an image is

$$T_{\min} = D \qquad [\text{CLK cycles}]. \tag{14}$$

Because of the delays introduced to make the kernel's filters causal and because of mirror operations at layers' borders, the total simulated decoding time is slightly greater than the theoretical minimum $T_{\min}$:

$$T = 1.04 \cdot D \qquad [\text{CLK cycles}]. \tag{15}$$

The bandwidth required between the external memory and the chip and between the horizontal filters and internal cache meet the theoretical values shown respectively in equations 5 and 12, 13. The Sliding Windows on-chip cache memory is composed by two blocks of 2560 words of 16 bits.

V. Conclusions

This paper presents the implementation of a JPEG 2000 compliant hardware DWT decoder for Mallat tree decomposition. The decoder is capable of image decoding applications, both with integer and floating-point kernels. The customizable design allows setting the basic features of the decoder, such as maximum image size, maximum kernel length and computational precision. The synthesized decoder may be programmed to process images with different sizes, coded on a different numbers of layers and using different kernels, while always guaranteeing the correct mirroring operations at layers' borders. By means of an on-chip cache and of a specifically optimized scheduling of operations, the decoder requires a minimum number of accesses to external memory, making it suitable for low-power embedded applications.

References

[1] M. Ravasi, M. Mattavelli and D.J. Mlynek, “Scheduling strategies for 2D wavelet coding implementations,” Proceedings of X European Signal Processing Conference, vol. II, pp. 969-972, Tampere, Finland, September 2000.

[2] M. Ravasi, M. Mattavelli, D. J. Mlynek, A. Buttar and S. Soudagar, “Wavelet image compression for mobile/portable applications,” *IEEE Trans. on Consumer Electronics*, vol. 45, no. 3, pp. 794-803, August 1999.

[3] G. Lafruit, L. Nachtergaele, J. Bormans, M. Engels and I. Bolsens, “Optimal memory organization for scalable texture codecs in MPEG-4,” *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 9, no. 2, pp. 218 –243, March 1999.

[4] C. Chakrabarti, M. Vishwanath and R. Owens, “Architectures for wavelet transforms,” *VLSI Signal Processing VI, IEEE special publications*, NY, pp. 507-515, 1993.

[5] M. Vishwanath, “The recursive pyramid algorithm for the discrete wavelet transform,” *IEEE Transactions on Signal Processing*, vol. 42, no. 3, pp. 673-676, March 1994.

[6] T.C. Denk and K.K. Parhi, “Calculation of minimum number of registers in 2-D discrete wavelet transforms using lapped block processing,” *IEEE Int. Symposium on Circuit and Systems*, vol. 3, pp. 77-80, London, England, May 1994.

[7] G. Lafruit and J. Bormans, “Graceful degradation parameters for a scalable wavelet codec,” *ISO/IEC JTC1/SC29/WG11/MPEG97/M2655*, Fribourg, October 1997.

[8] TeraLogic, Inc. (C. Chui), “Integer Wavelet Transforms,” *ISO/IEC JTC1/SC29/WG1/N769*, Geneva, March 1998.

[9] W. Sweldens and P. Schröder, “*Building your own wavelets at home*,” in “*Wavelets in Computer Graphics*,” ACM SIGGRAPH Course Notes, pp. 15-87, 1996

[10] Y. Sheng, “*Wavelet Transform*,” Chapter 10 in “*The Transforms and Applications Handbook*,” A.D. Poularikas, CRC Press, 1996.