11—3

# On the Synthesis of a Controller for Handling Borders in Systolic Architectures for 1-D Discrete Wavelet Transform

M. Ferretti, and D. Rizzo
DIS - University of Pavia, Italy
{ ferretti,rizzo} @elzira.unipv.it

## Abstract

In this paper we describe the control subsystem of a systolic architecture that implements the 1-D discrete wavelet transform (DWT) on the basis of the Recursive Pyramid Algorithm (RPA) and that correctly manages the border problem adopting a periodic extension strategy to obtain a perfect reconstruction of the signal. Usually, the proposed architectures for DWT (either 1-D or 2-D) assume a zero-padding extension.

Periodic extension of the input signal allows to set up a new algorithm (PE-RPA) that correctly recovers the input signal without additional computations. A detailed analytical description of this algorithm shows that a certain amount of control is mandatory, to have the underlining systolic structure perform the 1-D DWT computations correctly in time.

In this paper, we show the design of the controller required for such a scheduling. The methodology used for such a design is a VHDL RTL description, with the subsequent optimised synthesis. The preliminary results show that the overall complexity of the controller compares to that of the systolic architecture in the case of a 10 stage filter (L=10) with a 20 bit data representation extended to 36 bits for multiplication. The dependency of the controller on the input signal size N is logarithmic, since the local memory for handling borders is O(L logN). The overhead hidden within the handling of borders is clearly non negligible, and must be taken into account.

## The problem of borders in wavelets

In this paper, we address the VLSI implementation of the 1-D Discrete Wavelet Transform [1,2], and we further delimit the scope of the analysis to orthogonal wavelets. The purpose of this work is to highlight the VLSI cost of correctly handling the border problem, that arises in all discrete transforms when applied to finite length input signal. Among the best VLSI implementations of the DWT are those based on systolic implementations [3-5] of the Recursive Pyramid Algorithm [6]. This algorithm is based on the assumption that the computations of the various levels of the transform are scheduled as soon as possible, that is when enough values from preceding levels are available to initiate a new convolution.

The systolic arrays derived on the basis of this assumption have 100% efficiency and minimal output latency. Unfortunately, all this works well if one ignores the effect of finite length of the input signal and assumes that it is zero-extended when the convolution kernel embedded into the systolic array requires data that are not available. The architectures proposed so far for DWT ignore this problem.

In a preceding paper [7] we have shown with analytic detail the effect of ignoring the border problem and two possible solutions, the former based on an extension of the RPA algorithm that computes more coefficients and restores efficiency within the systolic array by compressing the last stages of computations at each level; the second solution extends *periodically* the input and guarantees the perfect reconstruction while computing exactly N coefficients, the theoretical minimum [8]. We refer to this solution as to PE-RPA. Both solutions prevent using the bare RPA schedule, that must be abandoned. In the following, we describe briefly the PE-RPA algorithm, and then go into the detail of the architecture required to support it. While the major component is still the systolic datapath for wavelet convolution, the irregular timing necessary to handle the periodic extension requires further hardware resources: more memory and a modified controller. We have obtained a synthesized version of both starting from a VHDL description. The comparison with the systolic datapath shows that the mandatory handling of borders requires roughly as many hardware resources as the systolic array, at least for convolution kernels of reasonable size.

### *The Periodic Extended RPA*

The equations to compute a 1-D DWT, using orthogonal wavelet filters of length $L$, are the following:

$$a_j(n) = \sum_{m=0}^{L-1} a_{j-1}(2n-m)h(m) \qquad (1)$$

$$d_j(n) = \sum_{m=0}^{L-1} a_{j-1}(2n-m)g(m) \qquad (2)$$

$$1 \le j \le \log N,\ 0 \le i \le N_j\text{-}1$$

where $a_j(i)$, $d_j(i)$ represents the $i^{th}$ approximation and detail coefficient at level $j$; $h()$,$g()$ are the low and high pass $L$-tap filters obtained from the chosen wavelet; $a_0()$ is the N-sample input signal; $N_j = N\,2^{-j}$ is the number of wavelet coefficients in level $j$. Since the computations of either set of coefficients have same structure, the analysis can be done considering only the approximation coefficients $a_j(i)$. We consider the case with N power of 2 and with the orthogonal filter length L even.

Of the three possible ways to extend the signal (at the beginning, at the end, or symmetrically at both ends), only the extension at the end of the input sequence is viable for an on-line implementation, since the other two introduce immediately a long latency. Due to the structure of the wavelet convolution, that embeds a downsampling by two, the number of values required for the extension is exactly L-2, for a filter with L taps.

Since we consider a border extension on one side only, the first wavelet coefficient $a_{j+1}(0)$ is obtained by a convolution with the complete wavelet filter placed over the first $L$ values of level $j$; so, the last coefficient $a_{j+1}(N_{j+1}\text{-}1)$ is computed using the last two values from level $j$, and $L$-2 more values, namely $a_j(0)$, $a_j(1)$ .. $a_j(L\text{-}3)$ (see Fig. 1).

The equations for the decomposition and periodic extension of PE-RPA are the following:

$$a_j(i) = \sum_{m=0}^{L-1} a_{j-1}(2i+L-1-m)h(m) \qquad (3)$$

$$d_j(i) = \sum_{m=0}^{L-1} a_{j-1}(2i + L - 1 - m)g(m) \quad (4)$$

$$a_{j-1}(k) = a_{j-1}(k - N_{j-1}) \quad \text{Per. Extension} \quad (5)$$

$$1 \le j \le \log N,\ 0 \le i \le N_j\text{-}1,\ N_{j-1} \le k \le N_{j-1} + L\text{-}3$$

The reconstruction within PE-RPA requires that the signal be interleaved with zeros, as usual in wavelets, and that it be also periodic extended at the *left* side with *L-2* values, including zeros.

In contrast to the bare RPA, the PE-RPA algorithm has a fairly complex activation sequence of the operations to be carried out in the underlining systolic array. In the following, we briefly hint to the result of the timing analysis, in order to show that a controller is indeed necessary.

We assume that the systolic data path carries out a multiply-and-add operation in a clock cycle; the systolic arrays multiplexes the computations of the convolutions of the detail and approximation signals. Furthermore, thanks to the downsampling by 2 typical of sub-band processing, the array interleaves the computations necessary to produce the first level of the transform to those required for the upper levels. This is the basic strategy in RPA as well. The PE-RPA carries out the overall transform in two major phases: i) handling of the finite length input sequence; ii) residual processing after the whole input has been read.

During the first phases, the computations that generate coefficient $i$ of level $j$ are scheduled within the systolic array at time $T_{i,j}$ as follows:

$$\Delta_j = \Delta_{j-1} + 2^j(L-1) + 2^{j-1} = (2L-1)2^j - (2L-1) \quad (6)$$

$$T_{i,j} = \Delta_j + 2^{j+1} i \quad (7)$$

where $\Delta_j$ is an initial latency at each level, due to the convolution mode depicted in Fig. 1.

The first phase terminates at time T* :

$$T^* = 2(N + L - 2) - 1 \quad (8)$$

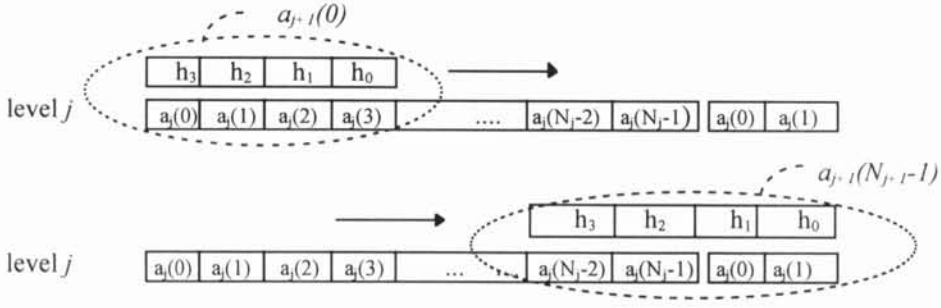


Fig. 1:Starting and stopping convolution, for the first and the last coefficient computation over a level, with right side periodic extension, and $L$=4.

Once all inputs have been read in and processed, not all the transform has been computed. The coefficients yet to be generated are split into two sets: i) coefficients that belong to levels for which the computations have been started, but are not complete; ii) coefficients of levels for which no computation has been scheduled yet. We denote with $j^*$ the discriminating level, that is the first one for which no coefficient has been produced; also, the number of coefficients not yet produced in level $j$ is denoted with $K(j)$. Details on the analytic derivations of $j^*$ and $K(j)$ are available in [7]. This second phase will be denoted as PA phase in the following.

The schedule of equation (7) cannot be applied to this residual sets of coefficients, because it would issue operations with much too low a frequency within the systolic array. It is the task of the controller to compact the remaining computations; the simplest way to do so is to complete the transform on a level by level basis. Fig. 2 shows the overall schedule of PE-RPA for an input signal of length $N$=16 and a filter kernel $L$=4.

The activation times for the computations of the residual coefficients of levels $j<j^*$ are.

$$T_{i,j} = 2(N + L - 2) - 1 + 2 + \sum_{m=2}^{j-1} 4K(m) + 4\left[ i - \left( \frac{N}{2^j} - K(j) \right) - 1 \right] \quad (9)$$

where $N_j - K(j) \le i < N_j$ and j<j*.

Analogously, the activation times for levels $j \ge j^*$ are:

$$T_{i,j} = 2(N + L - 2) - 1 + 2 + \sum_{m=2}^{j^*-1} 4K(m) + \sum_{m=j^*}^{j-1} 4\frac{N}{2^m} + 4i \quad (10)$$

where $0 \le i < N_j$ and $j^* \le j \le \log N$

## Description of the architecture

In this section, we highlight the overall structure of the modified architecture that computes the DWT of a 1-D finite signal, managing borders with PE-RPA. As a reference, we use the systolic architecture described in [4], with proper modifications to handle timings and data distribution. Fig. 3 depicts the major components of the architecture. In the following, we describe each component and analyze in a more detailed way the complexity of the new controller and subsequently a possible implementation given in VHDL description.

The main components are: i) a systolic array of $L$ Processing Elements (PEs); ii) a controller; iii) a set of $L$ local memories for storing intermediate approximation coefficients; iv) a border memory; v) a PA memory, used to store partial approximations results during the second scheduling phase.

The *systolic array* supports the convolutions and the downsampling required to produce both the detail and the approximation signals. Each PE is connected to its right and left neighbors and to the local memory. The left-to-right path carries the primary inputs forwarded to the array from the controller. The path from local memory carries elements from already computed higher level coefficients. The right-to-left paths carry: i) partially accumulated results, which eventually get to the controller and are dispatched either to local memory (approximation signals) or to the output (detail signals); ii) control tokens, generated by the controller and described later, which are the *state* information for establishing the processing mode of the PEs. Each PE performs a synchronous multiply-accumulate operation

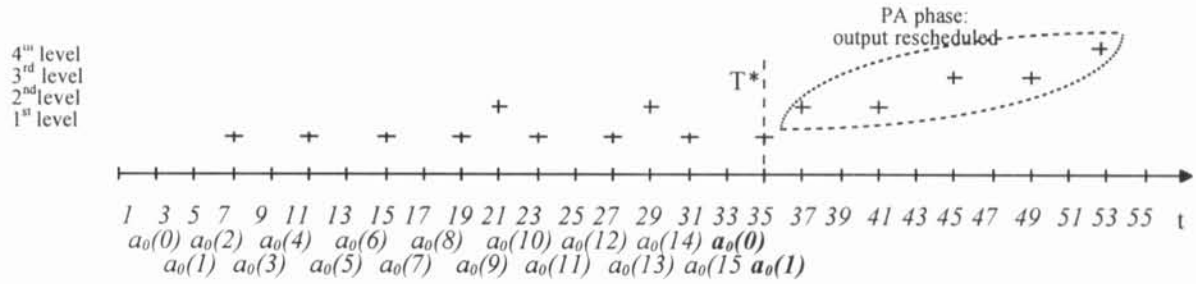


Fig. 2: Example of PE-RPA scheduling with input length $N$=16, $L$=4 and right side periodic extension; $\bar{j}$=3.

on inputs depending on the *state* it receives from its right neighbour. In every case it transmits the input from left to right, even when it computes higher level coefficients. The timing of computation is such that the array achieves 100% efficiency: the primary input is fed from the controller with 50% throughput; the accumulated output is produced at the leftmost PE with 50% throughput. Indeed, the systolic convolution on a single data stream (approximation signal) coupled with the downsampling by two yields 25% efficiency, thus making each PE perform useful work once in four clock cycles. The remaining three clock cycles are used profitably by scheduling the PE for the detail convolution on the primary input and for the approximation and detail convolutions on a higher level data. This restores the efficiency to 100%.

The *controller* determines the scheduling of wavelet coefficients computation within the array by forwarding control tokens to rightmost PE ($PE_{L-1}$). Each token consists of a <*state*,*level*> pair. Each PE goes through a sequence of four states:

**$LP_1$**: read from input data line and perform 1st level low pass computation (filter *h*, approximation signal);

**$HP_1$**: re-use input data previously read to perform 1st level high pass computation (filter g, detail signal);

**$LP_j$**: read from local memory data line and perform low pass computation for level j>1 (filter h, approximation signal);

**$HP_j$**: re-use local memory data previously read to perform high pass computation for level j>1 (filter g, detail signal).

At the begin of operations, when the array reads in the first primary input, it must be in the following

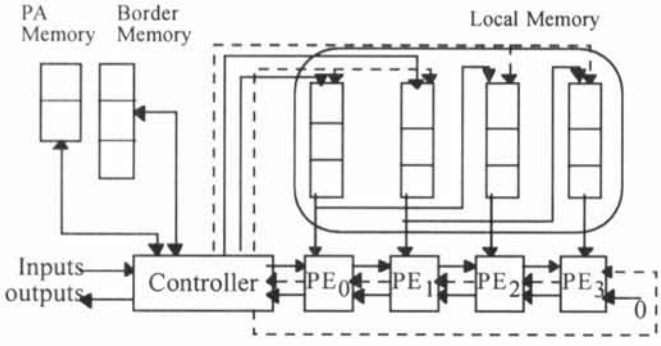


Fig. 3: Systolic architecture to perform DWT, PE-RPA ($L$=4).

situation:

| PE : | **0** | **1** | **2** | **3** | 4 | 5 | .... |
|---|---|---|---|---|---|---|---|
| State: | **$LP_j$** | **$HP_j$** | **$LP_1$** | **$HP_1$** | **$LP_j$** | **$HP_j$** | .... |

When in state $LP_j$, the PE uses the token value *level* to get the correct address for reading from the local memory. The sequence of levels is generated by the controller during either scheduling phases.

The description of how the controller discriminates between the first phase and the PA phase is deferred to a subsequent section. We only anticipate that the controller also manages the extra memories, denoted with *PA memory* and *Border memory*, to be described shortly.

The *local memory* is arranged into $L$ modules, each *(LogN-1)* words in size. New data (approximation signals for the various levels) are written alternatively into the first two memory modules from the controller. The paths from module $i$ to module $i$+2 are required to support data shifts for convolutions in each level: a value in level $j$ at position $i$ (in $PE_i$) will be used for a convolution in the same level at position $i$+2 ($PE_{i+2}$) because of the subsampling. It is convenient to set-up a path outside the array of PEs, to preserve array regularity. During the first phase of operation, $t \le T^*$, with reference to memory contention, the value read at PE in state $LP_j$ will be written in a PE in state $LP_{\bar{j}}$; the receiving memory module is therefore available, since it is not used by the local PE. Actually, in a limited set of cases arising when processing levels $j > j^*$ and at times following $T^*$, there is indeed contention due to further shifts of data that are required and that can be scheduled during states $LP_{\bar{j}}$. This contention can be eliminated because the item being written into memory is also the item to be read by the PE.

The *PA memory* module is used to store coefficients of levels $j > 1$ that cannot be stored in the first two local memory modules during the PA phase. To keep the management of this new memory module as simple as possible, the number of memory locations can be set to 2($L$-1). Indeed, $L$-1 is an upper bound to the number of residual coefficients to be scheduled for each level in the PA phase; using twice as much locations allows for a simple double buffering scheme. A detailed analysis shows that this figure could be decreased at the expense of a more complex control policy within the controller.

The *border memory* is used to manage border extensions for each level; it is written when the array reads in the first $L$-2 primary inputs or when it produces the first $L$-2 values for a generic level $j$ ; it is read when the array computes the last ($L$-2)/2 values of level $j$+1. It can be therefore set up as a one port memory of ($L$-2) log$N$ words.

The changes to the basic architecture (RPA) consist in a larger memory and in a more complex controller. As to memory, the overall amount of words necessary is $L$(log$N$-1)+2($L$-1)+($L$-2)log$N$; in most practical cases, $L \approx \log N$, and the dimension of the memory can be approximated to 2$L$ log$N$, that is twice as much as in the basic RPA. The actual cost of the extended

architecture is hidden within the controller, which has more counters and which drives more control points.

## The breakdown of the controller

Figure 4 shows the partitioning of the architecture of the controller. The external control signals are *reset* and *clock* (not shown, because shared by all of blocks) and *start, end_input*.

The overall operation of the system consists of an initialisation and of a processing phase. The initialisation begins when the external input *reset* is asserted; the controller sets up the array by propagating the <state,level> tokens and signals that the operations can begin by asserting the *start* output when the propagated tokens have reached $PE_0$. The external unit starts sending the input data at every other clock cycle and closes the processing phase by asserting the input signal *end_input*.

The PE-RPA controller is partitioned into the following modules: state sequencer, scheduler, PA scheduler, data path interface, border manager. Each of these modules can be further decomposed into sub-modules, respecting the hierarchy shown in Figure 5.

### *The state sequencer*

This module is activated by the *reset* signal and produces the sequence of states used by the PEs within the array. The period of the sequence is 4 and each item is prepared and output at every clock cycle. It is the responsibility of the Scheduler to make sure that enough clock cycles are counted (and consequently, enough state tokens generated and forwarded to the array) before the first input is used. As already described, the state sequence allows to intermingle the computations of the approximation and detail signals within the array, by having each PE reuse each input item twice in consecutive states for the same level.

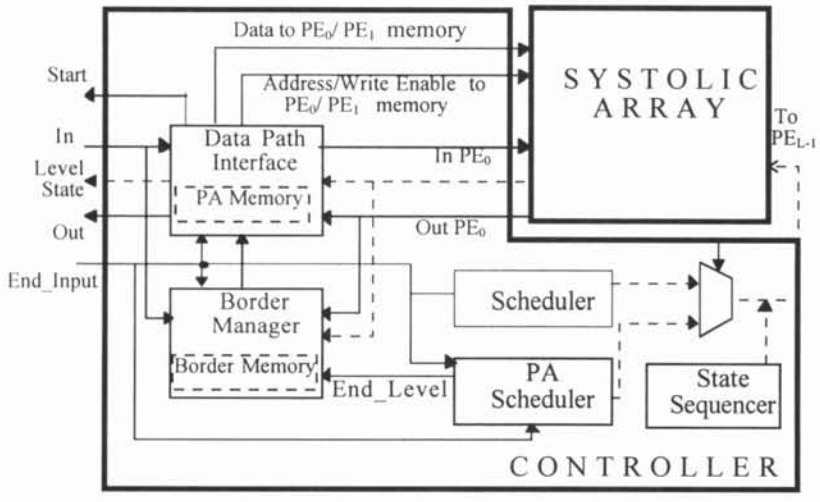


Figure 4: PE-RPA controller's partitioning.

### *The scheduler*

This is the controller component that generates the sequence of control information to be used by each PE. That is, it generates for the first scheduling phase ($t \leq T^*$), the correct sequence of levels in the <state, level> pairs to perform PE-RPA computations. Since it has to take into account the initial latency for each level, it is composed by a *module-L* counter. This component starts counting the first L input data samples and acts as a level enabler: when it has reached the L-th state, it enables the current level (initially the first), allowing the *level sequencer* submodule to start scheduling the first level computations. It then starts counting for the next level, and so on.

The level sequencer is a set of module-2 counters, that schedules a level computation for each level $j < j^*$ that has been enabled (that is, for which the first L elements of the previous level have been already computed). The module-2 counter takes into account when a new couple of previous level elements can be used to perform a new computation. A point worth noting regards the need for shifting data, in local memories. To better understand when such situation occurs, we can consider the following example: with L=10, when the first 14 input data samples have been read, the first level computations have already been enabled by the module-10 counter and the level sequencer has scheduled the first level coefficients $a_1(0)$ $a_1(1)$. These values have been written, by the datapath interface (which will be described later), respectively to $PE_1$ and $PE_0$ local memories. Since a couple of values is not sufficient to allow a second level computation, the second level is not enabled, but there is the need to make these couple of values shift along the local memories, in order to get into the correct PE position, when the second level will be enabled (this couple of values should reach $PE_{L-1}$ and $PE_{L-2}$, respectively). In fact, if it doesn't happen, the computations of the next couple of first level coefficients $a_1(2)$ $a_1(3)$, would rewrite the first couple, causing a loss of data. So the level sequencer should signal that a second level shift is needed, although no second level value should be computed. Therefore the controller has to send to PEs another control information, *Level_enabled*, that in this case has to be set to false, while it is true for the enabled levels.

Glue logic and a clock divider complete the Scheduler in order to realise a hierarchy priority mechanism for level scheduling so that when two enabled levels can compute a new value, the lower level takes precedence.

### *The PA scheduler*

The other scheduling module, *PA Scheduler*, is activated after T* (actually, a little before, because we have to consider the latency to propagate from the controller to $PE_0$ in the systolic array). It generates the PA scheduling sequence of levels as previously described.

The functionality is the same as that of the Scheduler, but there exists a subtle difference in managing control. Indeed during PA scheduling there is the need to manage borders extension; so when a level computation is reaching the last (L-2)/2 coefficients, the PA scheduler should signal the need for border management and border data. Since the state of datapath is delayed with respect to the actual scheduling, the controller makes use of a L-clock delay line to transmit the request for borders data to the Border Manager. There is also the problem of managing data shift for the subsequent level. We use states $LP_1$, to perform this task when needed

### *The data path interface*

The functions carried out by this module can be subdivided into two major operations: forwarding the input samples to the array and splitting the output into the detail signal, which is actually sent to the outside world, and into the approximation signal, which is kept

within the system for higher level computations. The handling of this stream of data can be further broken down into two major phases: in the first one, the approximation is stored directly and alternatively into the local memories of the first two PEs; after T*, the output from PE0 has to be stored into the PA memory for proper reuse when the computations of the level using them will be scheduled. The 2(L-1) memory locations that make up the PA memory allow for a straightforward embedding of this memory within the data path interface module, which is the only module that interfaces with it; control signals to operate on the PA memory are therefore best kept local to data path interface. During the second phase of operation, the data path interface receives data from the *border manager*, precisely when the last coefficients of each level are produced. Data from the border manager are required also in the first phase, just before T*, that is when the first level computations are terminated.

The *end_input* signal is used to discriminate between the first and the second phase of operation; it triggers a sequence of internal control states that are subsequently conditioned also by signals from the PA scheduler. The data path interface contains 2 internal registers each log N-1 bit in length; one keep tracks of the destination memory for each level, and toggles the level bit identified with the level token emerging from the array, thus alternating between $PE_0$ and $PE_1$; the second register stores an activity bit that is set when a data item is stored into the corresponding level for later use by the array, and is reset when the array output carries a token showing that the data item has been used.

### Border Manager

The main purpose of this module is to store the first $L$-2 coefficients produced at each level, for later reuse when extending the convolution at the end of the same level. To do so, a border memory is required, consisting of ($L$-2) log$N$ words. This module interfaces directly with the input, because the first $L$-2 data have to be stored directly into the border memory, for the convolution on the input (level 0); the remaining data are received from $PE_0$, with its whole state information. Data stored into the border memory are extracted by this module upon request from the external (*end_input* is asserted) or from the PA scheduler (*end_level* asserted, meaning that in a level the last coefficient that does not required border data, has been scheduled); a couple of data is forwarded to the data path interface for storing into the local memories with two consecutive states on the negative clock transitions, in time for the data path interface to sample them on the following positive clock transition.

## The VHDL implementation

The PE-RPA controller has been described in VHDL, adopting a mixed behavioral and RTL style of description. The leading criteria for partitioning the controller is based on functionality and feasibility of synthesis. Actually, the resulting structure described in the following is the outcome of a synthesis process that has forced an increased depth in the hierarchy, whenever the complexity of the synthesised design unit was too high.

### *The design hierarchy*

The basic operations just outlined previously correspond in most cases to a design entity instantiated in the controller top level entity. Each design entity has been furthermore decomposed into sub-components when necessary. Figure 5 reports the hierarchy partitioning and the association between functions and design entities. As an instance, the state sequencer is described as a process in the top-level entity; analogously the scheduler is not described as a separate entity but its components are directly instantiated inside the top-level. One of the most involved module is the border manager. It has been described with 3 sub-entities: one to get and store the first L-2 values for each level (data input comprised), one to manage the sending of input data stored, and one to manage the sending of higher levels coefficients to the data path interface (further decomposed for levels $j<j^*$ and $j \geq j^*$).

### *Processes and design entities*

Many entities are composed of a certain number of cooperating processes, which trigger each other and correspond to operations that are mutually exclusive. For example, the data path interface has been described by four processes, each of them corresponding to a particular situation: one process reads input data and forwards it to the array (reading also border data to conclude the first level computations); one reads output values produced by $PE_0$, that need to be stored in the local memory of $PE_0$ or $PE_1$, during the first scheduling phase; one concludes the second level computations, sending also border values to the local memories and storing the computed coefficients into the PA memory; the last process manages the output produced, the sending of data to local memories and the receiving of border values, during the second scheduling phase till the end, making a distinction between levels over and below $j^*$.

### *Memory inference*

We decided to implement the memory required by the controller using registers rather than RAM components. This choice is justified by the little amount of memory needed, and the requirements of access pattern. This choice is valid also for the PE local memory, where we may need read and write accesses to different (but in some cases, to the same) addresses in the same local bank memory. The depth of the word line is 20 bits and the total amount of additional memory, with respect to the reference RPA architecture, is $(L-2)logN + 2(L-1)$ words.

### *The reference description of the PE-local memory pair*

The description of the processing elements has been made considering a unique entity which combines the combinational path (multiplier and adder) with the local memory. There are three separate processes: one to manage the local memory, one to manage the operation based on input state information and one to samples input data line, according to clock events.

The local memory has dimension log$N$-1 word and is realised with latches. The memory is asynchronous: every time the signal *enable_write* is raised, the corresponding data line is sampled and stored in the address specified. Read operations instead are initiated synchronously by the corresponding PE; furthermore, every time there is a read from a memory location at $PE_i$, the data read is shifted and stored into to the same location of the memory of $PE_{i+2}$.

The asynchronous memory is useful to avoid partial stalls, during the PA schedule phase, for levels $j>j^*$ with $N_j<L-2$, since the propagation of data needed from

the previous level over the array of local memories must be "as fast as possible", in order to get the data into the correct position for convolution without delays.

The PE has a couple wavelet filter coefficients (low pass and high pass corresponding to filter position i) used alternatively, on the base of the state ($LP_x$ or $HP_x$) received. Currently, they are instantiated as constants.

## The synthesis results

The design hierarchy just described has been synthesised using SYNOPSYS Design Compiler™, with a target technology library from SGS-Thomson (HCMOS 0.35 micron).

Table I shows the major figures obtained by the synthesis of the design units described previously.

The overall constraint used in the synthesis process is a symmetrical clock with a 20ns period that has been met everywhere. The choice of this constraint has been done by considering the speed of the fastest multiplier available to perform the 36 bits internal product in the PE (about 10 ns): the other 10 ns have been allocated considering the broadcast of data read from the local memories, that could potentially travel from the first $PE_0$ to the last $PE_{L-1}$ when computing last levels. This requirement for a potentially dangerous broadcast operation is however non critical in practical situations, since the wavelet transform is hardly carried out up to

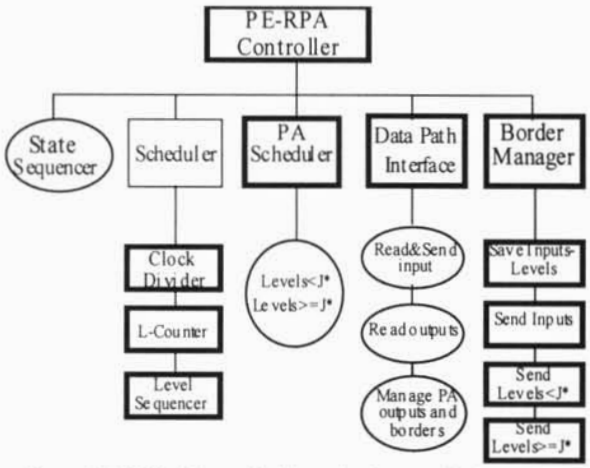


Figure 5: PE-RPA Controller hierarchy: boxes with heavy stroke are entities, bubbles are processes.

the highest level.

The synthesis strategy has been changed across the design units; the default structuring and resource sharing has been used in most cases, except in the data path interface, where the sharing has been reduced. Boundary optimisation has been applied to the design units that embed a hierarchy, such as the border manager.

Currently, the overall hierarchy has not been collapsed into a single design unit; we expect a certain amount of further optimisation by this last phase of the synthesis.

The most important conclusion that can be drawn by the figures shown in table I is the relative complexity of the PE-local memory systolic array with respect to the overall controller. In a 10 stage array (L=10), that supports a fairly large convolution kernel, capable of handling standard image dimensions (N=512), the area taken by a PE-local memory pair is roughly 1/8 of that of the controller. Stated otherwise, the requirement of handling borders costs as much a the bare processing for the DWT convolution, a result that cannot be disregarded.

## Conclusions

In this work we have designed the hardware structure required to handle the border effect for perfect reconstruction of a DWT transformed finite signal. We have shown that the synthesised hardware is comparable to that mandatory for the execution of the convolution with a fairly standard kernel. We expect that the complexity of the controller could be reduced considerably by assuming that the DWT transform is not carried out up to the last level; indeed, in some applications such as image compression, the number of levels computed hardly goes beyond 3 or 4 in images of up to 1024x1024 pixels in size.


### ACKNOWLEDGEMENT

This work has been partially supported by grant ASI 1996 RS 192. SGS-Thomson has supported the work through the technology library for synthesis.


## References


[1] O.Rioul, M.Vetterli, "Wavelets and Signal Processing", IEEE SP Magazine, October 91.

[2] G.Strang, T.Nguyen "Wavelets and Filter Banks", Wellesley-Cambridge Press, 1996.

[3] M.Vishwanath, R.M.Owens, M.J.Irwin "VLSI Architectures for the Discrete Wavelet Transform", *IEEE Trans. On Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 42, No.5, pp. 305-316, May 1995.

[4] R.Lang, E.Plesner, H.Schroder, A.Spray, "An efficient systolic architecture for the one-dimensional wavelet transform", SPIE, Vol 2242, Wavelet Applications (1994), pp. 925-935.

[5] J.Fridman, E.S.Manolakos "Discrete Wavelet Transform: Data Dependence Analysis and Synthesis of Distributed Memory and Control Array Architectures", *IEEE Transaction on Signal Processing*, Vol. 45, No.5, May 1997.

[6] M.Vishwanath, "The Recursive Pyramid Algorithm for the Discrete Wavelet Transform", *IEEE Trans. on Signal Processing*, Vol. 42, No. 3, pp. 673-677, March 1994.

[7] M.Ferretti, D.Rizzo, "Handling Borders in Systolic Architectures for 1-D Discrete Wavelet Transform", submitted to IEEE Trans. on Signal Processing, available as RIDIS-121-98, Dip. Informatica e Sistemistica, University of Pavia, 1998.

[8] C.Taswell, K.C.McGill, "Length-Preserving Wavelet Transform Algorithms for Zero-Padded and Linearly-Extended Signals", Available at http://www.toolsmiths.com/papers.shtml


| Design | Comb. Area | Seq. Area | Total Area |
|---|---|---|---|
| State Sequencer | 414 | 756 | 1170 |
| Scheduler | 6246 | 12726 | 18972 |
| Scheduler PA | 23256 | 8064 | 31320 |
| Data Path interface | 343602 | 172494 | 516096 |
| Border Manager | 609426 | 366480 | 975906 |
| | | | |
| PE-RPA Controller | 982944 | 560520 | 1543464 |
| Array 10 PEs + local memories | 1416780 | 548640 | 1965420 |

Table I: synthesis results (area expressed in $\mu^2$).