VLSI Architectures for Weighted Order Statistic (WOS) Filters [1]

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

Lori E. Lucke
Minnetronix, Inc.
Saint Paul, MN 55114
lelucke@minnetronix.com

Abstract

The class of median filters has been extended to include *weighted order statistics* (WOS) filters, to improve the flexibility of the filtering operation. The WOS filter weights each input within a sample window, and thus retains the original temporal order information. In this paper, we present efficient VLSI architectures for WOS filters which maintain a *weighted* rank for each sample in the sample window and update the weighted ranks for each window shift. We present novel (i) array architectures (ii) stack filter architectures and (iii) sorting network architectures for non-recursive and recursive WOS filters which implement the above procedure. Our analysis shows that the bit-serial stack filter implementation is the one with the smallest area while the bit-parallel stack filter is the one with the smallest input-output latency. The sorting network architecture (based on updating a sorted list) has the best area-time performance. Physical implementations verify our analysis.

[1] The authors' names appear in alphabetical order.

Index terms: Weighted Order Statistic Filters, Rank order filters, Recursive median filters, VLSI architectures, array architectures, sorting network architectures, stack filters, weighted rank.

Number of pages = 25, Number of figures = 14, Number of tables = 6

1 Introduction

Linear filters are by far the most popular filters in signal processing. However, linear filters cannot address all the problems in signal and image processing. Linear filters blur edges and fail to remove impulsive noise or noise with a heavy tailed distribution. The median filter is a popular non-linear filter that preserves edges, and efficiently attenuates impulsive noises. However, the median filter gives poor performance for non-constant signals corrupted by noise with a heavy tailed density function. This is because median filters use only the rank order information of the input data within a particular sample window, while discarding the original temporal order information.

Several classes of rank order based filters have recently been developed which attempt to incorporate both the rank and temporal information within the input data. These include stack filters [1, 2, 3, 4] and weighted order statistic (WOS) filters [5, 6, 7, 8]. In this paper we focus on the development of VLSI architectures for WOS filters. A WOS filter chooses an output based on the *weighted* ranks of the inputs within a window of samples. A WOS filter is described by

$$y_i = R^{th} \ smallest \ of \ \{W_{-N_1} \diamond x_{i-N_1}, \ldots, W_0 \diamond x_i, \ldots W_{N_2} \diamond x_{i+N_2}\} \qquad (1)$$

where R is the rank of the filter, $K = N_1 + N_2 + 1$ is the size of the sample window, $\vec{W} = \{W_{-N_1}, \ldots, W_{N_2}\}$ is an integer weight vector, and $\diamond$ denotes duplication of the sample by the corresponding weight value. Typically, $N_1 = N_2 = N$. The WOS filter differs from the median filter in that each sample now has a weight associated with it. These weights provide additional flexibility in the control of the filter behavior. WOS filters are used as robust smoothers or predictors [6] and have frequently been used in image processing applications.

The WOS filter has demonstrated its improved filtering capabilities compared to traditional median filters. Our intent in this paper is to study the implementation methods for WOS filters. This is essential for understanding the additional resources required to implement a WOS filter instead of a median filter. Such an analysis is also necessary to understand whether resource requirements are available to realize the perfomance gains of the WOS filter. In this paper we present several implementation strategies for WOS filters. We then analyze the time and area required for each implementation and discuss the tradeoffs. We also discuss the impact of pipelining on these implementations. These implementations were initially developed for application-specific architectures used in high speed real-time image processing applications. However, these methods can also be translated to efficient software algorithms, where the time and area tradeoffs described here are still applicable.

A large number of architectures have been proposed for order statistic (OS) or rank order filters. Rank order filters are a simplification of WOS filters with all the weights set to one. These architectures are classified as array architectures [9, 10], sorting network architectures [11, 10, 12, 13, 14, 15, 16], and stack filter-based architectures [17, 18, 3, 19, 20]. In array architectures, each element in the window has a rank associated with it, and with each window shift, the ranks of the elements are updated. In sorting network architectures, the elements in the window are completely sorted for each output, or a sorted list is updated for each output. Stack filters are a superset of the OS and WOS filters [5]. A stack filter translates the filtering operation to the binary domain through the use of threshold decomposition [21, 1]. Architectures for stack filters can be directly extended to OS filters.

The existing OS filter architectures can be extended to WOS filter architectures in a straightforward manner by repeating each sample according to its weight, and then operating on the repeated input set [22]. Such straightforward extensions are inefficient, since the window size increases to the sum of the weights, $S_W = \sum_{i=-N}^{i=N} W_i$, and leads to architectures that require $O(S_W)$ processors. In this paper we present new efficient architectures for WOS filters which operate on the original window of size K and thus require only $O(K)$ processors. These architectures maintain a weighted rank value for each sample within the sample window. Since each sample has a weight associated with it, as the window shifts for each new output, the weights of every sample within the window change. Consequently, the procedures for computing the weighted ranks of the samples are more complicated than computing ranks in OS filters. We describe three different methods by which the weighted ranks can be computed – by rank updating, by bit-level comparisons and by sorting. We map these procedures into new array architectures, stack filter architectures and sorting network architectures. We consider both non-recursive and recursive WOS filters.

This paper is organized as follows. Section 2 briefly describes three procedures for computing the WOS filters. Section 3 describes new array architectures, Section 4 describes new stack filter architectures, and Section 5 describes new sorting network architectures to implement these procedures for non-recursive and recursive WOS filters. Section 6 compares the area and time complexities of the three architectures.

2 WOS Filter Algorithms

Weighted Order Statistic (WOS) filters are an extension of Order Statistic (OS) filters – the main difference is that in WOS filters, each sample in the window has a weight associated with it and thus the output is a function of the *weighted* rank. For instance, if the filter window is of size 6,

$\vec{W} = \{3,3,3,4,2,2\}$, and if the samples in the ith window, $WIN_i = \{5,2,6,3,4,1\}$, then sample value '1' has ranks 1 through 2 associated with it, sample value '2' has ranks 3 through 5 associated with it, sample value '3' has ranks 6 through 10 associated with it, and so on. The output with rank 7 is thus sample value 3. In this section, we describe three different methods in which the weighted ranks can be computed – by rank updating, by bit-level comparisons and by sorting.

Method 1: In this method, each element in the sample window has a weighted rank, and as the window shifts, the weighted ranks are updated. This rank update method is an extension of the method that is used to update ranks in rank order or order statistics filters [9, 10]. In rank order filters, a separate rank is maintained for each sample location in the sample window. When the window is updated, each rank is updated based on comparisons with the new and the discarded samples in the sample window. In a WOS filter, the weighted rank of a particular sample is updated by an amount which is a function of both the comparison results and the weights of the new and the discarded samples, plus the changes in the weights of all samples that occur as each sample shifts position.

Consider the above example where $\vec{W} = \{3,3,3,4,2,2\}$ and $WIN_i = \{5,2,6,3,4,1\}$. Let $WIN_{i-1} = \{8,5,2,6,3,4\}$. Then the new sample in WIN_i is '1' and the discarded sample is '8'. In WIN_{i-1}, sample '6' has a weight of 4 and therefore has a range of ranks, 11 through 14, associated with it. We only need to store the smallest of this range (11), as the weighted rank of sample '6' (the remaining ranks are implied by the weight value). We update the weighted rank of sample '6' from WIN_{i-1} to WIN_i as follows. First notice that '6' is larger than the new sample, '1' ($m = 1$), and smaller than the old sample, '8' ($n = 0$). The weights of the new and the old samples are 2 and 3 so the rank update component due to the oldest and the new samples is $2m - 3n = 2$. Finally the rank update component, ΔF, due to the changes in the sample weights in successive window operations must be calculated. The difference in weights for each sample as a result of the window shift is 0 for sample 4; 2 for sample 3; 0 for sample 2; and 0 for sample 5. Note the weight difference only depends on the sample location, not on the sample value. Since samples 4, 3, 2, and 5 are all smaller than sample 6 their weight changes affect its rank, thus, $\Delta F = 0 + 2 + 0 + 0 = 2$. Therefore, the weighted rank of sample 6 in WIN_i becomes $11 + 2 + 2 = 15$.

Method 2: In this method a stack filter [1] is used. The input samples undergo threshold decomposition which changes its input into unary encoding, followed by filtering at each level of threshold decomposition with a bit-level logic function (BLL) that directly implements the weighted order statistic filter. The filtering is done at each level by summing the number of 0s in the weighted sample window. If this sum is greater than the rank of the filter, R, the output is a 0. For instance, in the example with $\vec{W} = \{3,3,3,4,2,2\}$ and $WIN_i = \{5,2,6,3,4,1\}$, there are 7 levels of threshold decomposition. The inputs to BLL unit 4 are $\{1,0,1,0,1,0\}$. The sum of the weights is 3+4+2=9 and if $R = 7$, a 0 is sent to the output logic unit. In fact, BLL units 1, 2 and 3 send 1s and BLL units 4, 5, 6, and 7 send 0s to the output logic unit. The output with rank 7 is thus 3.

Method 3: In this method, all the samples in a window are first sorted and then their weighted ranks updated. The weighted rank of a sample is a function of both its position in the sorted list and the weights of all samples that are smaller than itself. The weighted ranks are computed from scratch in each iteration. We explain this procedure with the help of the example where $\vec{W} = \{3,3,3,4,2,2\}$ and $WIN_i = \{5,2,6,3,4,1\}$. The sorted window (corresponding to WIN_i) is $\{1,2,3,4,5,6\}$. The corresponding weights are $\{2,3,4,2,3,3\}$. The weighted ranks are $\{1, 1+2 = 3, 3+3 = 6, 6+4 = 10, 10+2 = 12, 12+3 = 15\}$ or $\{1,3,6,10,12,15\}$. The element with rank 7 is thus the third element in the sorted list – which is 3.

3 Array Architectures

In this section we describe array architectures for non-recursive, and recursive WOS filters. All the architectures maintain a weighted rank for each element in the sample window, and update all the weighted ranks when the window shifts (see Method 1 in Section 2).

3.1 Non-recursive WOS filter

The array architecture for WOS filters with window size K consists of a semi-systolic linear array of K processors. The architecture is suited for both bit-parallel and bit-serial implementation. Furthermore, the architecture can be pipelined to any level, thereby supporting high throughput applications.

The processors store the samples in the order of arrival; processor $P(0)$ stores the new sample, and $P(K-1)$ stores the oldest sample. Each processor (except $P(0)$) updates its weighted rank based on the comparisons and weights of the new and discarded samples, plus the change in the number of samples that are smaller than itself. Processor $P(0)$ computes the rank of the new sample by counting the number of samples that are smaller than itself. Figure 1 contains a block diagram of this architecture. Each processor communicates with its right-adjacent neighbor, except for processor $P(0)$ which communicates with all the processors.

All processors (except $P(0)$) contain the following hardware elements: (i) data register a to hold the sample, (ii) weight register f to store the weight of that sample, (iii) rank register r to store the weighted rank of the sample, (iv) registers $f(P(0))$ and $f(P(K-1))$ to store the weights of samples in processors $P(0)$ and $P(K-1)$, (v) register array Δf to store the differences in weights of adjacent samples, (vi) register s to store the comparison results with all samples that arrived before it, (vii) a comparator to compare the current sample with the new sample, (viii) rank update unit, (ix) an output unit to output the element with the desired rank, R. Figure 2 describes the design of such a processor. Processor $P(0)$ consists of (i) data register a, (ii) rank register r, (iii) weights of all the window samples, (iv) register s, (v) a rank computation unit consisting of a tree of adders, and (vi) an output unit. Register s stores the results of the comparison to all samples that arrived before it and therefore eliminates the need to compare the current sample with the discarded sample.

The array architecture operates by updating all the weighted ranks concurrently and then using these to determine the element with the correct rank R. This operation is described as follows. Let x_{new} be the new sample, and let x_{old} be the discarded sample.

Step 1 Compare with new sample: Each processor (except $P(0)$) compares the contents of its register a with x_{new}, and stores the comparison result in 1-bit register u; $u = 1$ if $a \geq x_{new}$. The comparison result with x_{old} is stored in the rightmost cell of register s. This is loaded into 1-bit register v.

Step 2.1 Update rank: Each processor (except $P(0)$) updates its rank r by $r = r + uf(P(0)) - vf(P(K-1)) + \Delta F$, where $\Delta F = \sum_{i=0}^{K-2} s[i] * \Delta f[i]$.

Step 2.2 Compute rank of x_{new}: Processor $P(0)$ computes the rank of x_{new} by counting the number of samples that are smaller than its sample value. This is done concurrently with the other rank updates. $r = 1 + \sum_{j=1}^{K-1} \bar{u}(P(j)) * f[P(j)]$.

Step 2.3 Update comparisons with other samples: All processors (except $P(0)$) shift their register s right, and load u into the leftmost cell of s. Processor $P(0)$ loads the contents of $\bar{u}(P(j))$ into $s[j-1]$, $1 \leq j \leq K-1$.

Step 3 Match rank: Each processor compares its weighted rank r with the desired rank R, and if $r \leq R \leq r + f - 1$, then the sample value of that processor is output. Specifically, each processor computes $(R - r)$ and if $((R - r \geq 0) AND (R - r < f))$ is true, then the contents of register a of that processor are gated to the shared output bus. All processors transfer the contents of registers a, r, and s to their right-adjacent processors.

Figure 3 traces the operation of the array architecture for the window of samples given in the example in Section 2. The sample period of this architecture T_s is given by $T_s = T_{comp} + T_{rank} + T_{out}$, where T_{comp} is the time taken to compare two numbers, a and x_{new}, T_{rank} is the time taken to add K numbers using a tree of adders, and T_{out} is the time taken to compute $R - r$, compare $R - r$ with f and gate the appropriate register contents. It is clear that T_{rank} determines the sample period for this architecture. One possible way to reduce T_{rank} is to encode the rank update amount for various values of u, v, s, and use an adder to update the rank with the encoded amount. In such a realization, T_{rank} is approximated as an adder delay with the expense of an increase in area. The sample period of this design can be reduced by adding appropriate pipeline registers. For example assuming T_{rank} is the largest delay, adding pipeline registers before and after the rank update unit, reduces the clock cycle to the propagation delay of the rank update unit. For even higher throughput applications, latches can be added inside the rank update unit. The trade-off is a small increase in area (due to the latches) and an increase in latency (time from input to output).

3.2 Recursive WOS filter

In this section we describe an array architecture for recursive WOS filters. For the recursive WOS filter the sample window contains previously calculated outputs, as well as input samples. For each new output, both a y-sample and an x-sample enter and leave the sample window. The rank update procedure for recursive filters is very similar to that of non-recursive filters, except that the rank update component is now a function of two new samples and two discarded samples.

The proposed architecture consists of two semi-systolic linear arrays, Q and P, of sizes N and M, respectively, to store the y-samples and x-samples in the sample window. The recursive WOS filter can be used to emulate a pipelined recursive median filter [24]. For such cases, $M = N + A$, where A is the number of levels of pipelining [23]. The processors in both arrays store the samples in the order of arrival: $P(0)$ and $Q(0)$ store the new samples and $P(M-1)$ and $Q(N-1)$ store the oldest samples. While each processor communicates with only its right neighbor, processor $P(0)$ communicates with all the processors. Figure 4 describes the block diagram of this architecture.

Each processor here is similar to the processor in the architecture for the non-recursive WOS filter. In particular, each processor (except $P(0)$) contains: data register a; weight register f; weighted rank register r; registers $f(Q(0))$, $f(Q(N-1))$, $f(P(0))$ and $f(P(M-1))$ to store the weights of samples in $Q(0)$, $Q(N-1)$, $P(0)$ and $P(M-1)$; register array Δf_Q to store the differences in weights of adjacent y-samples; register array Δf_P to store the differences in weights of adjacent x-samples; register s_Q to store the comparison results of the sample with all the y-samples; register s_P to store the comparison results of the sample with all the x-samples; a comparator to compare the current sample with the new x-sample; a rank update unit to update the old rank; and an output unit to output the element with the desired rank. Figure 5 describes the design of such a processor. Processor $P(0)$ consists of: register a; register r; weights of all $(M + N)$ window samples; registers s_Q and s_P; rank computation unit consisting of a tree of adders; and an output unit. Registers s_Q and s_P store the results of previous comparisons and are updated in every iteration. This eliminates the need for a comparison to the two oldest samples. Also, comparison with the new y-sample is avoided by noting that if in the previous window the weighted rank r of a sample was greater than the desired rank R, then in the current window, the sample is larger than the new y-sample. This is because in the recursive filter, the new y-sample is the sample with rank R in the previous window.

The sequence of operations necessary to determine the element with rank R is similar to the non-recursive WOS filter and is described as follows. Let x_{new} and y_{new} be the two new samples, and x_{old} and y_{old} be the two discarded samples.

Step 1 Compare with new and old samples: Each processor, except $P(0)$, compares the contents of register a with x_{new}, and stores the comparison result in 1-bit register u_P; $u_P = 1$ if $a \geq x_{new}$. The comparison result with y_{new} is stored in register u_Q; $u_Q = 1$ if $a >= y_{new}$. (This comparison is performed indirectly in step 3.) The comparison results with x_{old} and y_{old} are already stored in the rightmost cells of s_P and s_Q. These are loaded into 1-bit registers v_P and v_Q: $v_P = s_P[M-1]$ and $v_Q = s_Q[N-1]$.

Step 2.1 Update rank: Each processor, except $P(0)$, updates its rank r by
$r = r + u_Q f(Q(0)) - v_Q f(Q(N-1)) + u_P f(P(0)) - v_P f(P(M-1)) + \Delta F$,
where $\Delta F = \sum_{i=0}^{N-2} s_Q[i] \Delta f_Q[i] + \sum_{i=0}^{M-2} s_P[i] \Delta f_P[i]$.

Step 2.2 Compute rank of new x-sample: Processor P_0 computes the rank of its sample, x_{new}, by counting the number of samples that are smaller that its sample value.
$r = 1 + \sum_{i=0}^{N-1} \bar{u}_P(Q(i)) f(Q(i)) + \sum_{j=0}^{M-1} \bar{u}_P(P(j)) f(P(j))$.

Step 2.3 Update comparisons with other samples: All processors, except $P(0)$, shift their registers s_Q and s_P right, and load u_Q into the leftmost cell of s_Q and u_P into the leftmost cell of s_P. $P(0)$ loads the contents of $\bar{u}_P(Q(i))$ into $s_Q[i]$, $0 \leq i \leq N-1$, and $\bar{u}_P(P(j))$ into $s_P[j]$, $0 \leq j \leq M-1$. This rank calculation is done concurrently with the other rank updates.

Step 3 Match rank: Each processor compares its rank r with the desired rank R. If $r >= R$, then it sets $u_Q = 1$, otherwise it sets $u_Q = 0$. If $r \leq R \leq r + f - 1$, then the contents of register a of that processor is output, and the contents of registers a, r, u_Q, s_Q and s_P are loaded to processor $Q(0)$. All processors transfer the contents of registers a, r, u_Q, s_Q, and s_P to their right-adjacent processors.

Figure 6 traces the computation for an example with $N = 2$, $M = 4$ and $R = 5$. The sample period of this architecture T_s is given by $T_s = T_{comp} + T_{rank} + T_{out}$, where T_{comp} is the time taken to compare two numbers a and x_{new}, T_{rank} is the time taken to add $(N + M + 3)$ numbers using a tree of adders, and T_{out} is the time taken to compute $R-r$, compare $R-r$ with f and gate the appropriate register contents. The size of T_{rank} is reduced by encoding the rank update amount for various values of u_Q, v_Q, u_P, v_P, s_Q, s_P, and using an adder to update the rank with the encoded amount.

The feedback in this filter prohibits straightforward pipelining as is used in the non-recursive filter. A technique such as approximate lookahead [24] would be necessary to support pipelining.

4 Stack Filter-Based Architectures

In this section, we describe stack-filter based architectures for non-recursive and recursive WOS filters. The procedure consists of threshold decomposing the input samples, and then filtering each level of threshold decomposition with a Boolean function [1] (see Method 2 in Section 2). The function can be realized using AND and OR operators as in a Positive Boolean Function (PBF) unit [1], or it can be built with addition and comparison operators [18]. The actual size of the PBF unit may be reduced using the techniques in [19].

4.1 Non-recursive WOS filter

The stack filters are mapped onto hardware in the following way. The bit-parallel realization consists of one threshold decomposition unit for the new input sample, 2^b-1 bit-level logic units, where b is the sample wordlength, and an output unit as shown in Figure 7. There is a shift register s at the input to the BLL unit to hold the 1-bit signals of the K samples. In every sample period, s shifts in a 1-bit signal corresponding to the new sample and shifts out a 1-bit signal corresponding to the discarded sample. The BLL unit sums the weights with equivalent level values of 0 within the sample window to determine whether the current output level should be a 0. The sum of the weights, denoted by $NUM0$, is compared with R, the rank of the filter. Then if $NUM0 \geq R$, a '0' is sent to the output logic unit, otherwise a '1' is sent. The comparison of $NUM0$ with R is avoided by initializing $NUM0$ with $-R$. In that case, if $NUM0 \geq 0$, then a '0' is sent to the output logic unit.

The BLL unit updates the $NUM0$ for each window using one of two methods. The first method directly adds the weights. The second method maintains a sum and updates this sum depending on comparisons with the oldest and the newest bits in a fashion similar to the rank update unit used in the array architectures.

Method 1: Each BLL unit contains K registers to store the weights of K window elements. Let $f[i]$ be the value of the weight of the ith window element. Then $NUM0 = -R + \sum_{i=0}^{K-1} \bar{s}[i] * f[i]$. This is computed using a tree of adders.

Method 2: In this method each BLL unit updates the number of 0s by a method very similar to that described in Section 2.1 for array architectures. Each BLL unit has an additional Δf register to store the difference in weights of adjacent samples. Let x_{new} be the new sample, and let x_{old} be the discarded sample. Then in register s, the 1-bit signal corresponding to x_{new} is in location 0 and the 1-bit signal corresponding to x_{old} is in location K. $NUM0$ is now stored in a register and is updated as follows: $NUM0 = NUM0 + \bar{s}[0] * f[0] - \bar{s}[K] * f[K-1] + \sum_{i=0}^{K-2} \bar{s}[i+1] * \Delta f[i]$. Recall that $NUM0$ is initialized to $-R$, and if $NUM0 \geq 0$ a '0' is sent to the output logic unit. Figure 8 illustrates this procedure by tracing the operations in a particular BLL unit in an example with $K = 5$ and $R = 3$.

Bit-serial implementation: The bit-parallel implementation based on adding the appropriate weights in every sample period (Method 1) is mapped onto a bit-serial implementation using a technique similar to that described in [17]. This is a very area-efficient implementation. The bit-serial architecture consists of a single BLL unit which adds the weights, and an input logic unit. Figure 9 describes this architecture. The computations are done MSB (Most Significant Bit) first, with the output of the pth MSBit used to modify the inputs for the computation of the $(p+1)$th MSBit. Each input channel has a flag c which controls the relation between the input data $\{a_{i,p}\}$ and the input to the PBF unit $\{d_{i,p}\}$. When $c_i = 1$, $d_{i,p} = a_{i,p}$, and when $c_i = 0$, $d_{i,p} = a_{i,m}$, where $a_{i,m}$ is the first input bit that differed from the output bit h_m.

Algorithm:

begin

  for $i = 1, \ldots, K$ do in parallel set $c_i = 1$

  for $p = 1, \ldots, K$ do in sequence

    $NUM0 = -R$

    for $i = 1, \ldots, K$ do in parallel

      if $(c_i = 1)$ then $d_{i,p} = a_{i,p}$, else $d_{i,p} = \tilde{d}_i$

      $NUM0 = \sum_{i=1}^{K} \bar{d}_{i,p} * f[i]$

      if $NUMO \geq 0$, then $h_p = 0$ else $h_p = 1$

      if $((c_i = 1) AND (h_p \oplus a_{i,p} = 1))$ then

        set $c_i = 0$, $\tilde{d}_i = a_{i,p}$.

end.

While the bit-parallel implementation can be pipelined in a straightforward manner, the bit-serial implementation is inherently pipelined and cannot be optimized further.

4.2 Recursive WOS filter

The b-bit stack filter realization of a recursive WOS filter is also an extension of the architecture for the non-recursive WOS filter. Figure 10 describes the block diagram of this architecture. There are two shift registers at the input of the BLL units: register s_x to hold the 1-bit signals of the M x-samples, and register s_y to hold the 1-bit signals of the N y-samples. In every sample period, s_x shifts in a new 1-bit signal, and s_y shifts in the output of the BLL unit. The recursive stack filter realization may be used to support clustered lookahead implementations of the recursive median filter [24].

The BLL unit is built using one of two methods discussed in the previous

section. As in the case of the non-recursive WOS filter, the first method adds the appropriate weights for every output, and the second method updates a stored sum for every output. Let f_x and f_y be the weights of the x-samples and the y-samples, respectively. In the first method, each BLL unit computes $NUM0$, $NUM0 = -R + \sum_{i=0}^{M-1} \bar{s}_x[i] f_x(i) + \sum_{j=0}^{N-1} \bar{s}_y[j] f_y(i)$, where R is the rank of the filter.

In the second method, each BLL unit updates $NUM0$ by a method that is very similar to that described in Section 3.2. for array architectures. The only difference is that the inputs to the BLL unit are only 0 or 1 (instead of b-bit samples). In this method, each BLL unit has an additional Δf_y register to store the differences in weights of adjacent y-samples, and an additional Δf_x register to store the differences in weights of adjacent x-samples. Let x_{new} and y_{new} be the two new samples, and x_{old} and y_{old} be the two discarded samples. In register s_x, the bit corresponding to x_{new} is in location 0, and the bit corresponding to x_{old} is in location M. In register s_y, the bit corresponding to y_{new} is in location 0, and the bit corresponding to y_{old} is in location N. $NUM0$ is initialized with $-R$. It is updated as follows: $NUM0 = NUM0 + \bar{s}_x[0]f_x[0] - \bar{s}_x[M]f_x[M-1] + \bar{s}_y[0]f_y[0] - \bar{s}_y[N]f_y[N-1] + \Delta F$, where $\Delta F = \sum_{i=0}^{M-2} \bar{s}_x[i+1]\Delta f_x[i] + \sum_{j=0}^{N-2} \bar{s}_y[j+1]\Delta f_y[j]$. If $NUM0 \geq 0$, then a '0' is sent to the output unit, otherwise a '1' is sent. $M = 4$ and $R = 4$.

The two implementations described above have comparable area and time complexity. For the case when the weights of the samples are large, and the difference between the weights are small, the second implementation is less complex compared to the first one. The feedback in these implementations prevents additional pipelining.

5 Sorting Network Architectures

In this section, we describe sorting network-based architectures for non-recursive and recursive WOS filters. These architectures implement the WOS filter by first sorting the samples of each window and then computing the weighted rank of each sample (see Method 3 in Section 2).

5.1 Non-recursive WOS filter

A trivial sorting network architecture for WOS filters consists of repeating the input according to its weight, and then sorting the entire set of inputs. This would result in a very large sorting network. We proposed a more efficient architecture in [22] where only the original window is sorted, and the ranks are determined by summing the weights of samples in sorted order. In this paper we extend our previous design [22] and develop an architecture with fewer comparators and a reduced sample period.

The proposed sorting network architecture for the non-recursive WOS filter with window size K consists of (i) a sorting network to update a sorted list of K samples, (ii) a rank computation unit to compute the ranks of the K samples, and (iii) an output unit which determines the element with rank R. Such a network is described in Figure 11.

Sorting network: The sorting networks based on Batcher's odd-even merge sort can be used to sort the input samples from scratch [11, 10]. The compare-swap unit in the odd-even merge sorting network is modified to pass the weights along with the inputs. As in [22], the comparator in the compare-swap unit compares the two inputs, and the swap unit swaps the weights along with the inputs. However, if Batcher's odd-even merge sorting network is used, then all K samples of a window are sorted for every output computation. Since the samples of consecutive windows overlap by $K-1$ samples, it is not necessary to completely sort these samples for each output. Instead, the samples could be stored in order as a sorted list, and in every sample period, the sorted list could be updated by removing the oldest element and inserting the newest element.

We next propose a modified version of the method in [16] for updating a sorted list. In [16], each element of the sorted list either shifts up, or shifts down, or does not shift, or is replaced by the new element, depending on the comparison results with the oldest and the new elements. In the proposed method, each element is only compared with the new element. Comparison with the oldest element is avoided by having the oldest element send a 1(or 0) to all elements after (or before) it is in the sorted list. Each element also has an K-bit arrival register, which denotes the relative time of arrival of that element. For instance, the oldest element has a 1 in the MSB of the arrival register, while the newest element has a 1 in the LSB (Least Significant Bit) of the register. The ith logic unit outputs the contents of the ith element, or the $(i-1)$th element or the $(i+1)$th element, or the new element, depending on the comparison results with the oldest and the new element.

Rank computation unit: The weighted ranks of the sorted list are computed by an adder chain as in [22]. The delay in such an implementation is Kt_{add}, where t_{add} is the delay of an adder. If the ranks are computed by a tree of adders, then the delay is reduced to $(\log K)t_{add}$. Let $r(x)$

be the weighted rank associated with sample x. Figure 12 shows the tree of adders required to compute the weighted ranks of the samples in an example with $K = 8$.

Output unit: The output unit compares the weighted ranks of the sorted list with R, the desired rank, and gates the appropriate sample to the output. In [22] we proposed an output unit which sequentially compares the ranks of all the samples and gates the first input whose rank is less than or equal to R. Here we propose a method which concurrently compares the ranks of all samples and gates the appropriate output. The reduces the latency of the implementation from $O(K)$ to $O(1)$ where K is the size of the sample window. The output unit consists of K comparators, to compare each rank r of a sample with R. The ith comparator compares $r(u_i)$ with R, and sets $e_i = 1$ if $r(u_i) > R$. e_0 is set to 1. u_i is gated to the output bus if $e_i \oplus e_{i+1} = 1$, $0 \leq i \leq K$. Figure **??** describes the output unit.

The latency of the proposed rank computation unit and output unit is $\approx (\log K)t_{add} + t_{comp}$, where t_{comp} is the delay in a comparator, and t_{add} is the delay in an adder. In comparison, the worst case latency of these two units in our previous design [22] is $t_{comp} + (K-1)\max\{t_{add}, t_{comp}\}$.

5.2 Recursive WOS filter

The proposed sorting network architecture for the recursive WOS filter is an extension of the network that we developed for recursive median filters [11]. However, since each sample has a weight associated with it, updating the sorted list is accompanied by updating the weighted ranks associated with each sample. The proposed network consists of two units as shown in Figure 14.

1. Unit 1: A sorting network to sort M x-samples followed by computation of the ranks of each sample. Unit 1 consists of an array that updates the sorted list of M x-samples, and a tree of adders to compute the ranks of the samples. These two units are the same as those for the non-recursive WOS filters described in Section 5.1.

2. Unit 2: N insertion stages to insert the N y-samples one at a time into the sorted list. For each insertion, the ranks of the samples in the sorted list have to be updated. Unit 2 consists of N insertion stages to insert the N y-samples. In each insertion stage, an input is inserted into a sorted list. For the computation of y_i, stage 1 inserts y_{i-N}, stage 2 inserts $y_{i-(N-1)}$, and finally, stage N inserts y_{i-1}. The ranks of all samples that are larger than the y-sample are updated by the weight of the y-sample.

Each insertion stage consists of comparators, logic units and adders to update the ranks. The sample period is determined by the sum of the times that are taken to compare two numbers and to update the rank based on the comparison. For a MSB-first implementation, the numbers may be represented in the redundant number system in order to reduce the delay in each insertion stage.

The sorting network implementation of the recursive WOS may be used to emulate a scattered lookahead form of the recursive median filter [24].

6 Implementation Comparisons

To complete the discussion of the WOS filter architectures it is necessary to compare the complexity of each implementation. In this section we discuss the area and time complexity of the architectures introduced in this paper. The area and time analysis can be used to evaluate both hardware and software implementations of these architectures.

In Tables 1 and 2 we compare the number of computational units used for the array architecture, the stack filter architecture (Method II), and the sorting network architecture for both the non-recursive and recursive WOS filters. In these tables the following filter parameters are used: b is the filter sample wordlength, c is the wordlength of the weights, K is the non-recursive window length, and M,N represent the recursive window length. The architectures are subdivided into comparators, adders, threshold decomposition logic, output logic, and adder trees where adder_tree(m, n) is an adder tree that adds m numbers each n bits wide. In the tables, each filter is assumed to be a complete bit-parallel implementation. Obviously smaller implementations are possible if folded [25] or bit-serial architectures are used but use of such structures decreases the throughput. It is clear from the tables, that a direct comparison of the size of each architecture is dependent on the choice of the filter parameters. The stack filter is clearly larger than the other two architectures for large b since it depends exponentially on b. For large K and large b, the sorting network architecture is the smallest of the three bit-parallel architectures.

We compare the input-output latency of the three architectures for both the nonrecursive and the recursive cases in Tables 3 and 4. We model the time complexity by input-output latency – the time from the introduction of a new input to the generation of its corresponding output. The parameter $T_{comp}(n)$ is the time to compare two n-bit numbers, $T_{add}(n)$ is the time to add two n-bit numbers, and $T_{addtr}(m, n)$ is the time to add m elements each of width n using a tree of adders. It is clear that the stack filter architecture has the smallest latency while the array architecture has the largest latency. The sorting network architecture has a latency that is slightly smaller than the array architecture. A quantitative comparison of the three candidate architectures for non-recursive WOS filters for the case when $K = 5$, $b = 8$ and $c = 8$ has been included in Table 5. The area is in terms of number of 2-input NAND gates (obtained by synthesizing each component in Mentor Graphics) and the input-output latency is in terms of number of full adder delays. Note that the values in the table match very nicely with the above observation.

We also compare the pipelineability of the different architectures. The parallel non-recursive filter architectures can be pipelined to any level by adding latches after each computation unit. The sorting networks are the easiest to pipeline since their structure is so regular. The datapaths of the array and stack filter architectures are not so regular and must be analyzed for delay to determine the best location of the pipelining latches. Due to their inherent feedback, the recursive filters cannot be pipelined by the simple addition of latches. Instead, approximate lookahead techniques (which are an extension of the traditional look ahead techniques) must be used. Both the order of the filter and the weights of the filter may change as a result of this. Moreover, the hardware complexity increases with each level of additional pipelining.

Finally, to verify the complexity analysis of the WOS filters we physically implemented two of the architectures using VHDL and logic synthesis. The filters were synthesized to the 1.2 um CMOSN standard cell library using Mentor Graphics CAD tools and the area and time complexity were measured. We chose to implement an array architecture and a stack filter architecture (Method II) for the recursive WOS filter. The VHDL code was designed using the filter parameters discussed above. For the array architecture the following parameters were used: $c = 8$, $M = 4$, $N = 2$, $b = 8$. For the stack filter architecture the parameters were identical except for b which was changed to $b = 3$. This was simply to reduce the synthesis time since the filter size is greatly affected by b. The results of the area and time measurements are shown in Table 6. Here a 2 input NAND gate delay is 2 nsec. The stack filter was faster and smaller than the array filter. However if b=8 was used for the stack filter it would require 1.3 million transistors which is clearly much larger than the array architecture. These empirical results match nicely with the complexity analysis in Tables 1,2,3, and 4.

7 Conclusion

In this paper we have presented new array architectures, new stack filter architectures, and new sorting network architectures for WOS filters. The array architecture is more suitable for large window sizes, since the size grows linearly with the window size. The stack filter architectures have the smallest latency, since the filtering is done in the binary domain. The sorting network architectures are easy to pipeline, and are most efficient for small and medium-sized windows. All the implementations for non-recursive WOS filters are suitable for high throughput applications since they can be pipelined to any level by adding latches in the feed forward path. The recursive implementations, on the other hand, cannot be pipelined as easily. Pipelining has to be supported through the use of *approximate* clustered and scattered lookahead realizations [24, 23]. Finally, all the architectures presented here can be extended for adaptive WOS filtering [6, 7, 8], by supporting computations with real (and not just integer) weights.

References

[1] P.D. Wendt, E.J. Coyle, and N.C. Gallagher Jr. Stack filters. *IEEE Trans. Acoust., Speech, and Signal Proc.*, ASSP-34(4), Aug. 1986.

[2] E.J. Coyle and J-H. Lin. Stack filters and the mean absolute error criterion. *IEEE Trans. Acoust., Speech, and Signal Proc.*, 36(8):1244–1254, Aug. 1988.

[3] J.P. Fitch. Software and VLSI algorithms for generalized ranked order filtering. *IEEE Trans. on Circuits and Sys.*, CAS-34(5):553–559, May 1987.

[4] E.J. Coyle, J-H Lin, and M. Gabbouj. Optimal stack filtering and the estimation and structural approaches to image processing. *IEEE Trans. Acoust., Speech, and Signal Proc.*, 37(12), Dec. 1989.

[5] O. Yli-Harja, J. Astola, and Y. Neuvo. Analysis of the properties of median and weighted median filters using threshold logic and stack filter representation. *IEEE Trans on Signal Processing*, pages 395–410, Feb. 1991.

[6] Y. Lin, J. Astola, and Y. Neuvo. Optimal weighted order statistic filters under the mean absolute error criterion. In *Proc. of the Int'l Conf. on Acoust., Speech, and Signal Proc.*, pages 2529–2532, 1991.

[7] Y. Lin, J. Astola, and Y. Neuvo. Adaptive weighted median filtering under the mean absolute error criterion. In *Proc. of the IEEE Workshop on Visual Signal Processing and Communications*, pages 184–187, June 1991.

[8] Y. Lin, J. Astola, and Y. Neuvo. Adaptive weighted order statistics filters. In *Int'l. Workshop on Algorithms and Parallel VLSI Architectures*, pages 327–332, June 1990.

[9] C. Chakrabarti. High sample rate array architectures for median filters. *IEEE Trans. on Signal Processing*, pages 707-712, March 1994.

[10] L.E. Lucke and K.K. Parhi. A new VLSI architecture for rank order and stack filters. In *Proc. Int. Symp. on Circuits and Systems*, pages 101–104, May 1992.

[11] C. Chakrabarti. Sorting network based architectures for median filters. *IEEE Trans. on Circuits and Systems*, pages 723-727, August 1993.

[12] M. Karaman, L. Onural, and A. Atalar. Design and implementation of a general purpose median filter unit in CMOS VLSI. *IEEE Journal of Solid State Circuits*, 25(2), April 1990.

[13] K. Oflazer. Design and implementation of a single-chip 1-D median filter. *IEEE Trans. on Acoustics, Speech, and Signal Proc.*, ASSP-31(4), October 1983.

[14] K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[15] D. Richards. VLSI median filters. *IEEE Trans. Acoust., Speech, and Signal Proc.*, 38(7), Jan. 1990.

[16] G. R. Arce and P. J. Warter. A median filter architecture suitable for VLSI implementation. In *Proc of 23rd Annual Allerton Conf. on Comm., Control, and Computing*, pages 172–181, 1984.

[17] K. Chen. Bit-serial realizations of a class of nonlinear filters based on positive boolean functions. *IEEE Trans on Circuits and Systems*, pages pp. 785–794, June 1989.

[18] C. Chakrabarti. Efficient stack filter implementations of rank-order filters. In *Proc. of the Int. Symp. on Circuits and Systems*, May 1993.

[19] D. Gervorkian and et. al. Parallel algorithms and vlsi architectures for stack filtering. *IEEE Trans. on Signal Processing*, 43(1):286–295, January 1995.

[20] K. Egiazarian, O. Vainio, and J. Astola. Implementation of cascaded and recursive stack filters. *Circuits, Systems, and Signal Processing*, 15(1):93–110, January 1996.

[21] J.P. Fitch, E.J. Coyle, and N.C. Gallagher Jr. Median filtering by threshold decomposition. *IEEE Trans. Acoust., Speech, and Signal Proc.*, ASSP-29(6), Oct. 1982.

[22] L.E. Lucke and K.K. Parhi. VLSI architectures for weighted order statistics filters. In *Proc. of the IEEE Winter Workshop on Nonlinear Digital Signal Processing, Finland*, Jan. 1993.

[23] C. Chakrabarti and L. E. Lucke. *in VLSI Signal Processing, VI*, chapter Highly Concurrent Architectures for Recursive Median Filters. IEEE Press, 1993.

[24] L.E. Lucke and R. Kroenke. Training Recursive Structures for Weighted Order Statistic Filtering. *Proc. of the IEEE Int'l. Symp. on Circuits and Systems, London*, June 1994.

[25] K.K. Parhi, C.Y. Wang, and A.P. Brown. Synthesis of control circuits in folded pipelined DSP architectures. *IEEE Journal of Solid-State Circuits*, 27(1):29–43, January 1992.

Architecture	Hardware Requirements			
	Comparators	Adders	Registers	Others
Array	wid b: $K-1$ wid c: K	add_tree(K,c): K width c: K	width b: K width c: K^2+3K-2 width K: K	Output logic (MUX etc.): K
Stack (parallel)	None	add_tree$(K+1,c)$: 2^b-1 add_tree$(2^b-1,1)$: 1	width K: 2^b-1 width c: $(2^b-1)(K+2)$	Threshold decomp. unit
Sorting network	wid b: K wid c: K	width c: $K/2 \log K$	width b: K width $\lceil logK \rceil$: K	exor, MUX: K

Table 1: Hardware requirements of the three candidate architectures for nonrecursive WOS filters

Arch	Hardware Requirements			
	Comparators	Adders	Registers	Others
Array	wid b: $M+N-1$ wid c: $M+N$	add_tree(M+N+3,c): $M+N$ width c: $M+N$	wid b: $M+N$ wid c: $(M+N)^2$ $+4(M+N)-3$ wid $M+N$: $M+N$	Output logic (MUX etc.): $M+N$
Stack (par)	None	add_tree(M+N+3,c): 2^b-1 add_tree$(2^b-1,1)$: 1	wid $M+N$: 2^b-1 wid c: $(2^b-1)(M+N+3)$	Threshold decomp. unit
Sorting network	wid b: $MN+$ $N(N-1)/2$ wid c: $2M+N$	width c: $M(\frac{1}{2}\log M+N)$ $+N(N_1)/2$	wid b: $M(N+1)$ $+N(N-1)/2$ wid $\lceil logM \rceil$: M	MUX: M logic: MN $+N(N-1)/2$

Table 2: Hardware requirements of the three candidate architectures for recursive WOS filters

Architecture	Input-Output Latency
Array	$T_{comp}(b)+T_{addtr}(K,c)\ +T_{add}(c)+T_{comp}(c)$
Stack (parallel)	$T_{thresh}(b)+T_{addtr}(K+1,c)\ +T_{addtr}(2^b-1,1)$
Sorting network	$T_{comp}(b)+(\log K)T_{add}(c)\ +T_{comp}(c)$

Table 3: Input/Output latency of the three candidate architectures for nonrecursive WOS filters

Architecture	Input-Output Latency
Array	$T_{comp}(b)+T_{addtr}(M+N+3,c)\ +T_{add}(c)+T_{comp}(c)$
Stack (parallel)	$T_{thresh}(b)+T_{addtr}(M+N+3,c)\ +T_{addtr}(2^b-1,1)$
Sorting network	$T_{comp}(b)+(\log M)T_{add}(c)\ +NT_{comp}(b)+T_{comp}(c)$

Table 4: Input/Output latency of the three candidate architectures for recursive WOS filters

Architecture	Area (in terms of 2-input NAND gates)	Input-Output Latency (in terms of full adder delays)
Array	3716	48
Stack (parallel)	7794	36
Sorting network	1034	40

Table 5: Comparison of the three candidate architectures for non-recursive WOS filtes for the case when $K=5$, $b=8$ and $c=8$

Architecture	Transistor Count	Input-Output Latency
Array ($b = 8$)	43,540	120 nsec
Stack ($b = 3$)	36,800	52 nsec

Table 6: Actual measured area and latency for the recursive WOS filters using the 1.2 um CMOSN standard cell library

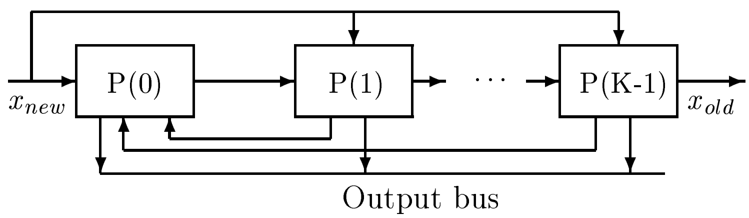

Figure 1: Block diagram of the linear array architecture for non-recursive WOS filter

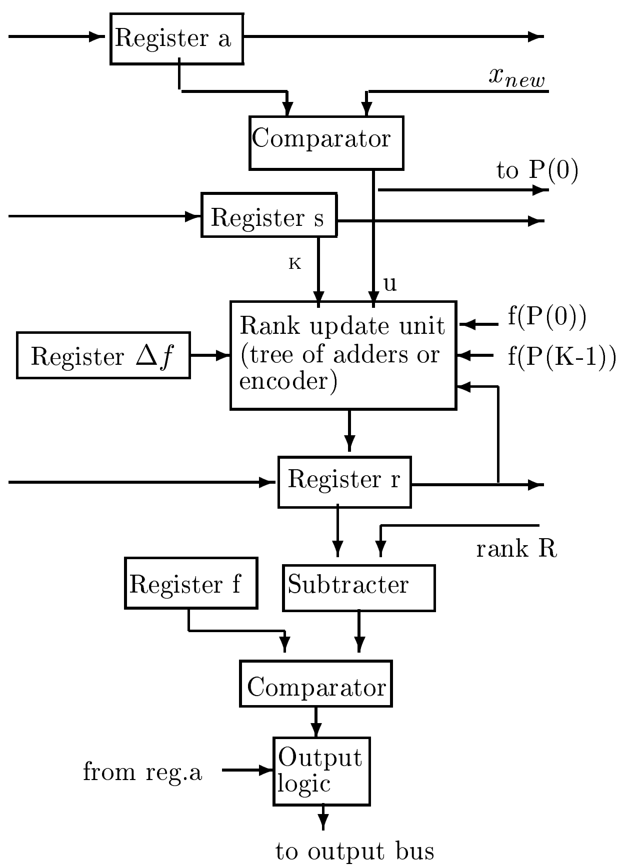

Figure 2: Design of a processor in the architecture for non-recursive WOS filter

WIN_{i-1}={8,5,2,6,3,4}

WIN_i={5,2,6,3,4,1} x_{new} of WIN_i is 1 x_{old} of WIN_i is 8

	P(0)	P(1)	P(2)	P(3)	P(4)	P(5)
Register f	2	2	4	3	3	3
Δf={0,2,-1,0,0}						
Step 1:						
Register a	1	4	3	6	2	5
Register s:	-	010100	000100	110110	000000	110100
Comp. with x_{old}:	-	0	0	0	0	0
Comp. with x_{new}:	-	1	1	1	1	1
Step 2						
Register r:	-	6	4	11	1	8
Compute ΔF	-	2	0	2	0	2
Update r:	1	10	6	15	3	12
Update s:	000000	101010	100010	111011	100000	111010
Step 3:						
Comp. with R=7			Match			
Output			3			

Figure 3: Sequence of operations that take place during the computation of the element with rank R=7 in an example where the window size is 6.

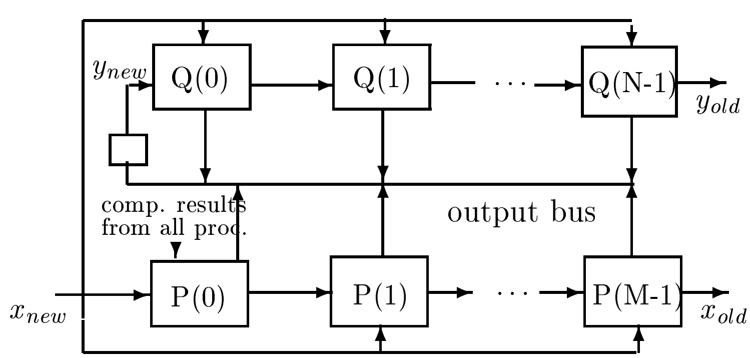

Figure 4: Block diagram of the array architecture for recursive WOS filter. The window size is $M+N$, where $M=N+A$. All the processors have been pipelined to A levels.

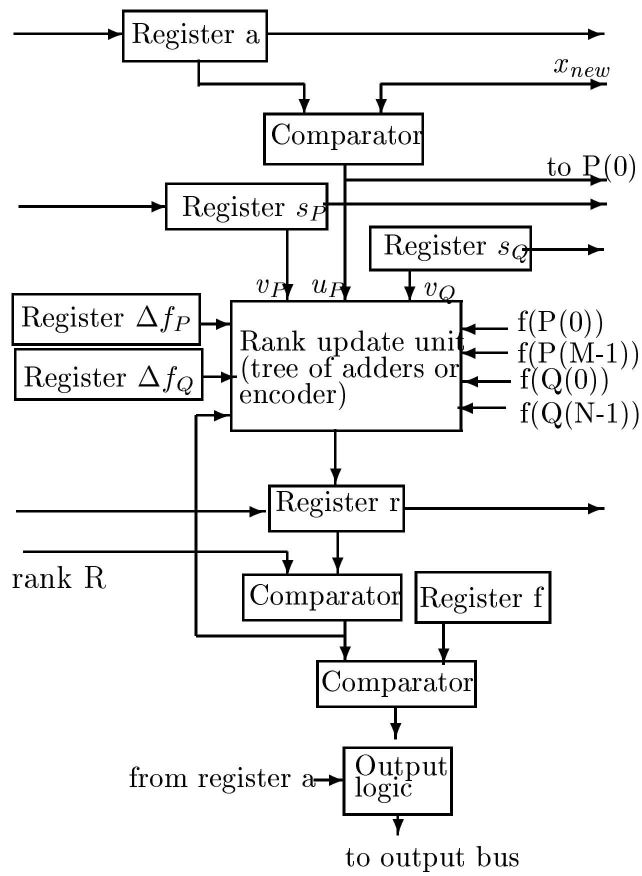

Figure 5: Design of a processor in the architecture for recursive WOS filter

WIN_{i-1}={8,9,6,7,5,1}
WIN_i={9,6,7,5,1,4} x_{new}=4, y_{new}=6, x_{old}=6, y_{old}=8

	Q(0)	Q(1)	P(0)	P(1)	P(2)	P(3)
Register f	2	1	1	2	2	1
Δf_Q={-1}						
Δf_P={1,0,-1}						
Step 1:						
Register a	6	9	4	1	5	7
Register s_Q	00	01	-	00	00	00
Register s_P	1100	1111	-	0000	1000	1101
Comp. with y_{old}, v_Q	0	1	-	0	0	0
Comp. with x_{old}, v_P	0	1	-	0	0	1
Comp. with y_{new}, u_Q	0	1	-	0	0	1
Comp. with x_{new}, u_P	1	1	-	0	1	1
Step 2:						
Register r	4	8	-	1	2	5
Compute ΔF	1	0	-	0	1	1
Update r	6	9	3	1	4	8
Update s_Q	00	10	00	00	00	10
Update s_P	1110	1111	0100	0000	1100	1110
Step 3						
Compare with R=5	-	-	-	-	Match	-
Output					5	
Compute u_Q	1	1	0	0	0	1

Figure 6: Sequence of operations that take place during the determination of the element with rank R in an example where $N = 2$, $M = 4$ and $R = 5$.

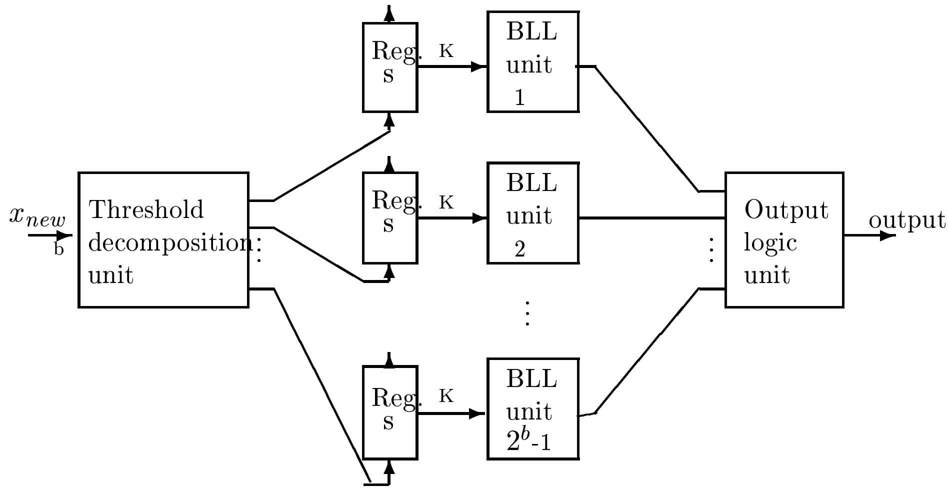

Figure 7: Block diagram of a bit-parallel stack filter-based architecture for non-recursive WOS filter when the window size is K.

K=5, R=3
f={1,2,1,1,}
Δf={2-1,1-2,1-1,2-1}={1,-1,0,1}

Iteration $i-1$:
  Register s: {1,0,1,1,0—1}
  $NUM0_{i-1}$=1

Iteration i:
  Register s: {0,1,0,1,1 — 0}
  $NUM0_i = NUM0_{i-1} + \bar{s}[0]f[0] - \bar{s}[5]f[4] + \sum_{i=0}^{3} \bar{s}[i+1]\Delta f[i]$
  $= 1 + 1 - 1 + (-1) = -1$
  output bit is 1

Figure 8: Sequence of operations that take place in a BLL unit in a WOS filter implementation when $K = 5$ and $R = 4$.

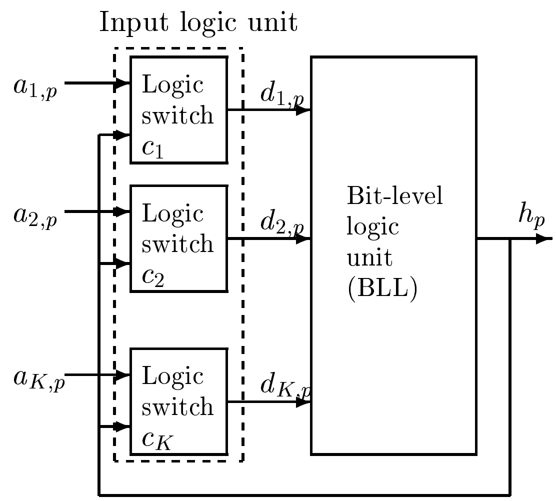

Figure 9: Block diagram of a bit-serial stack filter-based architecture for non-recursive WOS filter when the window size is K

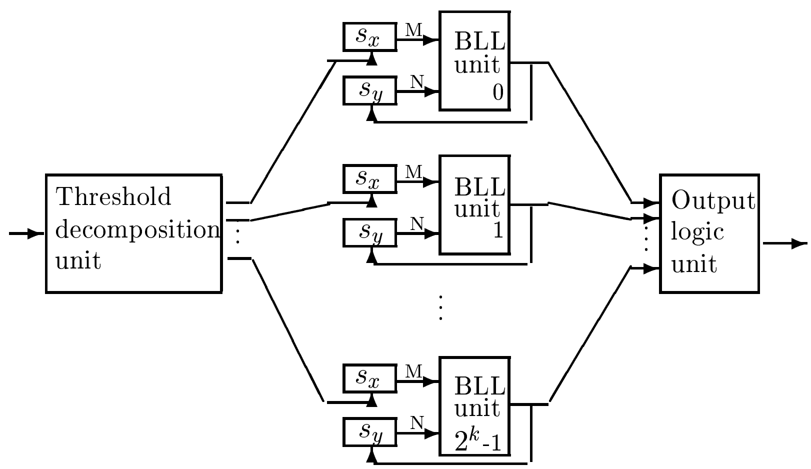

Figure 10: Block diagram of a bit-parallel stack filter-based architecture for recursive WOS filter. The window consists of M x-samples and N y-samples.

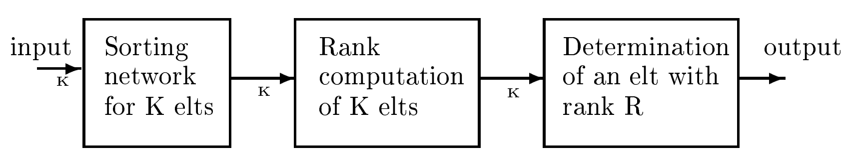

Figure 11: Block diagram of the sorting network architecture for non-recursive WOS filters when the window size is K

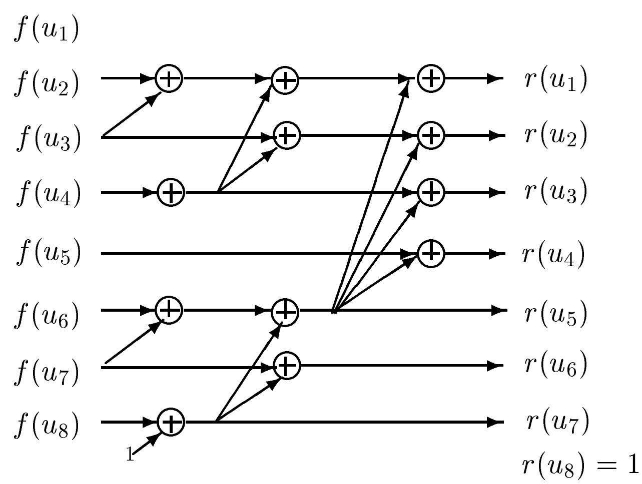

Figure 12: Tree of adder implementation of the rank computation unit in an example with $K = 8$. Here $u_1 > u_2 > \ldots > u_8$, $f(x)$ is the weight of sample x, and $r(x)$ is the rank of sample x. $r(u_i) = 1 + \sum_{j=i+1}^{8} f(u_j)$.