A Versatile Linear Insertion Sorter Based on a FIFO Scheme

Roberto Perez-Andrade, Rene Cumplido, Fernando Martin Del Campo, Claudia Feregrino-Uribe
Department of Computer Science
National Institute for Astrophysics, Optics and Electronics INAOE, Puebla, Mexico
{j_roberto_pa,rcumplido,fmartin,cferegrino}@inaoep.mx

Abstract

A linear sorter based on First In First Out (FIFO) scheme is presented. It is capable of discarding the oldest value, inserting the incoming data while keeping the values sorted in a single clock cycle. This type of sorter can be used as coprocessor or as module in specialized architectures for order statistics filtering. The architecture is composed of identical processing elements thus can be easily adapted to any length according to specific application needs. The use of compact identical processing elements results in a high performance yet small architecture. Results of implementing the architecture on a Field Programmable Gate Array (FPGA) are presented and compared against other reported hardware based sorters.

Keywords: **Hardware sorters, Linear Sorters, FIFO**

1. Introduction

Sorting is one of the most important operations used in computers. Given their practical importance, algorithms for sorting values have been the focus of extensive research resulting on several algorithms proposed to address specific problems. First, serial sorting algorithms were investigated, then parallel sorting algorithms became a very active area of research, finally models for parallel computations were developed [1].

For certain applications like median filters, ATM (Asynchronous Transfer Mode) switching, order statistics and, in general, continuous data processing, sometimes software only implementations of sorting algorithm do not achieve the required processing speed [2]. In order to speed up the sorting operation, some custom hardware architectures have been proposed in recent years. The relatively simple logic required for sorting and the inherent concurrency of the algorithms have allowed to explore a number of custom architectures in order to improve the performance obtained by software implementations. Hardware sorters are evaluated according to area requirements (number of Flip-Flops, comparators, control logic, gates, LUTs, etc), processing time (including latency and maximum operating frequency) and power consumption. Hardware sorters can be grouped in two kinds of architectures: sorting networks (including some systolic architectures) and linear arrays. The main idea behind sorting networks is to sort a block of data passing through a network of processing elements (PE) connected in such way that individual values take their corresponding place. Linear sorters are based on the idea that data to be sorted come in a continuous stream one value at a time, each value is inserted into its corresponding place in a register group (register file) at the same time that one of the stored values is deleted. This work is based on the latter idea: sorting the values as they are introduced into the register file, discarding the oldest value in the register file while maintaining the values sorted in a single clock cycle. This FIFO scheme can be used in applications that are continuously processing data in serial fashion such as order statistic filtering. This type of applications commonly require accessing a value from a specific position within a sorted array, more than one value simultaneously, or even the whole set of values in the array to perform parallel operations, thus making traditional FIFO memories with a single output port unsuitable. The proposed architecture for the insertion sort algorithm has a FIFO-like behavior, i.e. discards the oldest value when a new one arrives, while allowing flexible access to its contents.

2. Related Work

Several hardware architectures for performing sorting algorithms have been proposed. These architectures can be grouped in two families according to the algorithm they use: sorting networks and linear sorters. The sorting networks are based on a network constituted by several PEs located in the nodes of the network. The goal of each PE is to order two input values in ascending (or descending) order by placing the larger (or smaller) value in a specific output. This technique supposes that a block of data is available for being

978-0-7695-3170-0/08 $25.00 © 2008 IEEE
DOI 10.1109/ISVLSI.2008.14

sorted in parallel fashion. Sorter networks can be pipelined in order to reduce their critical path and latency, thus resulting in a better throughput. The disadvantage of this approach is that the network can potentially require a large number of PEs and, depending on the algorithm, several clock cycles for sorting the whole block of data. Besides, if one input value changes the whole block of data must be resorted. The efficiency of these sorters can be measured by its total size (numbers of PEs) and by its depth (maximum number of PE from input to output).

Linear sorters are useful when sorting data streams, where sorting must be carried out after each input value is received. Linear sorters are composed of a group of cells, each of them capable of deciding if an internal register should hold its current value or update it, either using the input value to the sorter or a value stored in adjacent cells. The advantages of this approach are that it uses fewer area resources and data are always sorted.

2.1. Sorting Networks

In [3], Batcher was the first to introduce the concept of sorting networks. In this work he presented the odd-even merging and bitonic networks. The odd-even merging network consists of two networks that sort all the values contained in odd and even positions separately, applying an interactive rule. The bitonic network works similarly to the odd-even, it merges two monotonic sequences, one in ascending order and the other in descending order. These two monotonic sequences are built by sorting the input values in ascending and descending lists, and being merged later. The nodes of both networks are built using PEs. The odd-even network can only sort a *N* fixed number of data. If *N* changes, the network must be rearranged. For this reason Kuo and Huang in [4] proposed a modification of the odd-even sorting network. They proposed a network that can sort any *M* input data that is smaller than *N*, which is the maximum number of data that the netwotk can sort. In their work [5] Tabrizi and Bagherzadeh use a different sorting scheme. Basically, they use a tree as a network implemented in an ASIC, where the leaves of the tree are the inputs and the main node is the output. This scheme works in a parallel input and produces a serial output, thus requiring several clock cycles to flush the tree after the beginning of the process.

In [6] Hirschil and Yaroslavsky propose three different sorting architectures. One of these architectures does not work as a sorting network neither it sorts the elements, instead it ranks the input data. This Parallel Rank Computer (PRC) receives, in a parallel fashion, a vector of *N* numbers and produces their ranks in two clock cycles. The rank of each number is calculated by comparing every pair of numbers and summing the comparison values.

2.2. Linear Sorters

The other two sorting architectures proposed in [6] are based on shift register architectures operating in a FIFO scheme. One of these architectures, called Serial Rank Computer (SRC), sorts a number according to the incoming value and its calculated rank. The other architecture, a Serial FIFO Sorter (SFS), stores an input vector of data in the order that it is received. This scheme is different from regular FIFO schemes as it keeps the data ordered by magnitude, still data leave the sorter in a FIFO fashion.

A VLSI sorter implementation is presented in [2] by Colavita et al. They propose a shift register architecture based on a Basic Sorting Unit (BSU) which contains two registers to store the data and an associated tag, a comparator, and a small logic circuit. This implementation is able to continuously process an input data stream while producing a sorted output data stream. The data stream is sorted according to the tags preserving the order of words with identical tags.

Chin-Sheng and Bin-Da Liu in [7] propose a sorter that uses a column of *N* PE to progressively sort *N* data. These PE are composed of two registers, and a Compare-Swap Cell (CS), which is built by a comparator and a swap unit. These PE are layout in cascade so their outputs are attached to the inputs of their successors. The idea of the PE is to allow the previous data being held by the PE or shifted to the successor PE at each clock cycle. In [8] Lluís Ribas et al. propose a register file (linear shifter) built on data-slice cells. This scheme requires minimal control logic and it is easily expandable. The idea of this sorter is based on the insertion sorting algorithm, which for every unsorted value, looks the right position in the sorted list in order to perform the insertion of the unsorted value into its corresponding place. This architecture only shifts values to one direction, discarding the smallest value. The data-slice cell is composed of a multiplexor, a register and a comparator, resulting in a compact and simple architecture. A similar sorting scheme is proposed in [9]. In this work, the data contained in the register file can be left o right shifted depending if the value is going to be inserted or deleted. Both, the value to be inserted and to be deleted, are specificated by the input signal. To perform the insert or delete process, the cell must perform four basic operations: shift right, shift left, load and initialize. In the next section our proposed solution is presented, it takes some ideas from previous works, especially from [8], but it has been modified to work in a FIFO fashion.

3. Proposed Insert Sort Algorithm

The proposed linear sorter is based on the insertion sort algorithm. This algorithm performs, for every unsorted value, a procedure that looks for the appropriate position in the sorted list to insert the input data [8]. The algorithm is presented in the next pseudo-code:

```
function InsertSort
for each unsorted D {
    i = 0;
    while(i < n) and (D > R[i]) ) {
        R[i] = R[i+1];
        i = i+1;
    } R[i-1] = D;
}end function;
```

The algorithm inserts incoming data in the vector R of infinite length. However, in practice, this characteristic can not be met, thus a deleting condition must be used. In [8], the condition used for deleting is to erase the smallest value stored, meanwhile in [9], the data to be erased is indicated by an external input signal.

In the proposed architecture, a FIFO scheme is used i.e. the oldest value is discarded, allowing for the incoming value to be inserted in its corresponding position. In order to achieve a FIFO-like operation, it is necessary to keep a life period value for each sorted data. If the value is shifted, then the corresponding life period value is shifted too. The life period value is increased by one every time that a new data is inserted. When the life period value has expired, i.e. it reaches a value equal to the number of elements in the array, the corresponding value is discarded, making an empty space in the vector and thus allowing the insertion of a new value. For this scheme, three different operations are performed in order to keep the array sorted: shift the value and life period value to the left, to the right, or to hold the values. To know the direction the values should be shifted to, every element in the array must know on which side, on relation to itself, the value that is going to be discarded is located. Also, it must know on which side the incoming value must be stored.

This functionality can be achieved by creating and array of PEs, called Sorting Basic Cell (SBC). The behavior of a *i-th* SBC can be described by the four different functions shown next, where *CNT[i]* represents the life period value of the *i-th* SBC, *R[i]* the data stored on the *i-th* SBC, *cnti* is a flag that indicates that life period value from a SBC to the right has expired, *D_right* and *D_left* are the output values to the right and left sides of the SBC respectively:

```
function SBC_SendData
D = Incoming Data;
if (R[i] < D){
    D_right = R[i];
    D_left = D;
}else{
    D_right = D;
    D_left = R[i];
}end function;
```

The *SBC_SendData* function is in charge of sending to its left and right neighbors the value it currently stores and the incoming data, *D_left* and *D_right* SBC respectively. If the first condition is met, it indicates that this SBC must send to its right its current value (*R[i]*) and to the left the incoming value else it must send to its left its current value and to the right the incoming value.

```
function SBC_ResetPeriodLife
D = Incoming Data;
if(CNT[i] = 0) or ((R[i] < D) xor (cnti = 1)){
    if (Ri < D) and not (R[i+1] < D){
        CNT[i] = 0;
    }
    if not (Ri < D) and (R[i-1] < D){
        CNT[i] = 0;
    }
}end function;
```

The first condition of *SBC_ResetPeriodLife* checks if the CNT[i] value must be updated while the inner conditions check for those cases where the SBC's counter must be set to zero. This action takes place when the incoming value will be stored on the *i-th* SBC therefore setting the life period value to zero is needed.

```
function SBC_UpdateValues
D = Incoming Data;
if(CNT[i] = 0) or ((R[i] < D) xor (cnti = 1)){
    if (Ri < D){
        R[i] = R[i+1];
        CNT[i] = CNT[i+1];
    }else{
        R[i] = R[i-1];
        CNT[i] = CNT[i-1];
    }
} end function;
```

In the *SBC_UpdateValues* function the first condition checks if the R[i] value must be updated by with the value coming from its left o right neighbor as indicated by the second condition. Even though the first condition is the same as the one in the shown in *SBC_ResetPeriodLife* function, they are separated because there is priority order, if both conditions are met then only the *SBC_ResetPeriodLife* function should be performed.

```
function SBC_PropagateFlag
if(CNT[i] = 0){
    cnti = 1;
} else{
    cnti = 0;
}end function;
```

This final function, *SBC_PropagateFlag*, checks if the life period value of the SBC has expired. This flag, *cnti*, is used by the functions as one of the conditions checked to update the SBC.

In order to fulfill the FIFO sorting functionality, the SBCs must be interconnected (figure 1) in a simple linear structure, called register file. This linear structure can be easily expandable as long as needed depending on the application.

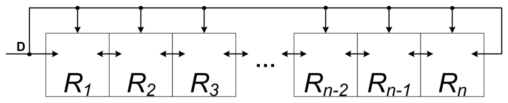


Figure 1. Register File.

For each incoming data D, one of these SBC's must discard its value. At the same time, all the SBC's hold their previous value, or store the value coming from the cell to the left, or store the value coming from the cell to the right. Only one clock cycle is needed to perform these actions (discarding the oldest data, holding, right or left shifting). Under this FIFO sorting functionality, there are three insertion cases that are considered and solved by the SBC (shown in figure 2, where the grey cell indicates the data to be discarded) are:

1. The incoming value is inserted to the left of the cell that discards its stored value. In this case data from R_i to R_{n-2} must be shifted to the right side and the incoming data is inserted in R_i.
2. The incoming value is inserted to the right of the cell that discards data its stored value. In this case data from R_i to R_3 must be shifted to the left side and the incoming data is inserted in R_i.
3. The incoming value is inserted at same position of the discarded value i.e. R_i. The rest of the cells hold their values.

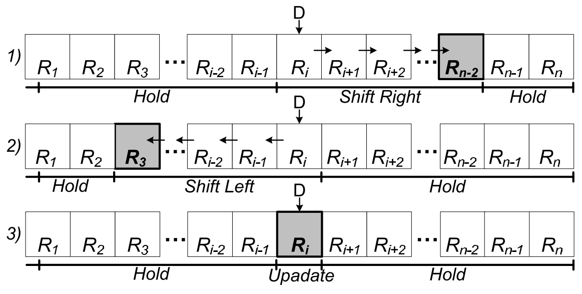

D
1)
R_1 R_2 R_3 R_{i-2} R_{i-1} R_i R_{i+1} R_{i+2} R_{n-2} R_{n-1} R_n
Hold
Shift Right
Hold
2)
Hold
Shift Left
Hold
3)
Hold
Upadate
Hold

Figure 2. Insertion Cases.

It is important to emphasize that the proposed sorter differs from other sorters as it implements a FIFO-like scheme where the oldest value in the register file is discarded to make room to each incoming data.

4. Sorting Base Cell

The proposed SBC has a register with synchronous load to store the data, a counter with synchronous reset and load to store the period life of the data, a comparator, four 2-1 multiplexers and control logic, figure 3.

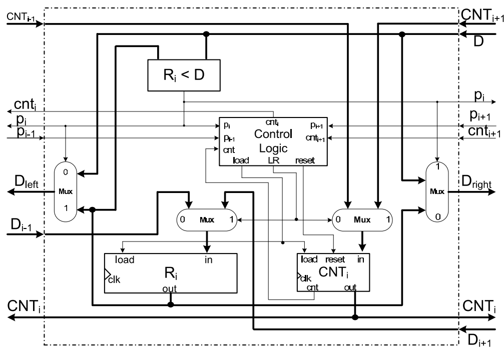


Figure 3. Architecture of the SBC.

This SBC control logic consist of four boolean equations, which control the register, the counter and the multiplexers. These equations can be viewed as representations of the conditions functions previously explained. The equation 1 is used as a condition in *SBC_ResetPeriodLife* and *SBC_UpdateValues* functions. This equation controls when the register and the counter must take the neighbor value, while the origin of this data (left or right side) is selected by the equation 2, which is used in function *SBC_UpdateValues*. The function *SBC_ResetPeriodLife* is represented by equation 3, it indicates if the counter must be set to zero. Finally, the equation 4 detects and propagates if the life period value of one of the SBCs to the right has expired as indicated by the *SBC_PropagateFlag* function.

$$load = (p_i \oplus cnt_{i+1}) + cnt_i \tag{1}$$

$$LR = (p_i \cdot load) \tag{2}$$

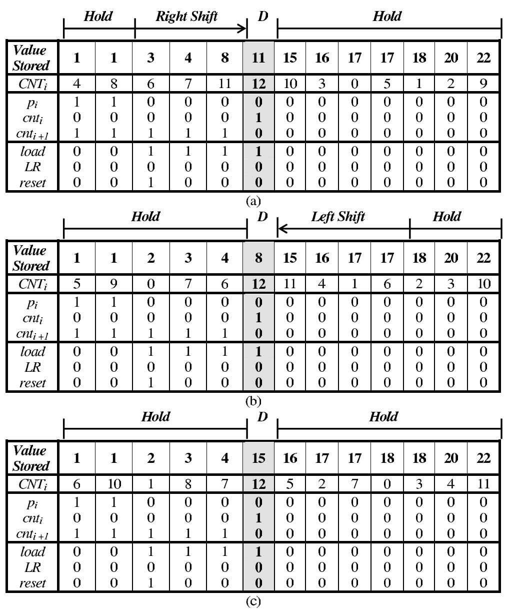
$$reset = load \cdot [(p_{i-1} \cdot \overline{p_i}) + (p_i \cdot \overline{p_{i+1}})] \tag{3}$$

$$cnt_i = cnt_{i+1} + cnt \tag{4}$$

where p_i is the comparator output as described by the *SBC_SendData* function. The signals p_{i+1} and p_{i-1} correspond to the right and left SBC neighbors respectively and cnt_{i+1} is the flag coming from the SBC immediately to the right. This signal helps to detect if the life period value of one SBC to the right has expired. In order to perform correctly the insert sort algorithm, the left most p_{i+1} signal's value is always 1 and the right most p_{i-1} signal's value is 0.

This can be viewed as the left most value having the largest value while the right most has the smallest one. To ensure proper behavior, all registers R$_i$ must be initialized to zero, while life counter values CNT must be initialized according to CNT[i] = i.

Figure 4 exemplifies how the SBC's control signals work allowing the register file to perform the sorting algorithm. This figure illustrates the three previously mentioned insert cases in a 3 steps sequence. The first row contains the sorted values currently stored in the register file, the second row contains the corresponding life period values and following rows contain the values of the control signals. The grey column indicates the data to be discarded, whose period life value is 12. In this example there is a given sorted sequence (figure 4.a). The first incoming value is *D = 2* is inserted in its corresponding position figure 4.b, then the second value *D = 18* is also inserted in its corresponding position figure 4.c and the final incoming value is *D = 11* which is placed in the SBC that just discarded its value.


Hold
Right Shift
D
Hold
Left Shift

Value Stored	1	1	3	4	8	11	15	16	17	17	18	20	22
CNT_i	4	8	6	7	11	12	10	3	0	5	1	2	9
p_i	1	1	0	0	0	0	0	0	0	0	0	0	0
cnt_i	0	0	0	0	0	1	0	0	0	0	0	0	0
cnt_{i+1}	1	1	1	1	1	0	0	0	0	0	0	0	0
load	0	0	1	1	1	1	0	0	0	0	0	0	0
LR	0	0	0	0	0	0	0	0	0	0	0	0	0
reset	0	0	1	0	0	0	0	0	0	0	0	0	0

(a)

Value Stored	1	1	2	3	4	8	15	16	17	17	18	20	22
CNT_i	5	9	0	7	6	12	11	4	1	6	2	3	10
p_i	1	1	0	0	0	0	0	0	0	0	0	0	0
cnt_i	0	0	0	0	0	1	0	0	0	0	0	0	0
cnt_{i+1}	1	1	1	1	1	0	0	0	0	0	0	0	0
load	0	0	1	1	1	1	0	0	0	0	0	0	0
LR	0	0	0	0	0	0	0	0	0	0	0	0	0
reset	0	0	1	0	0	0	0	0	0	0	0	0	0

(b)

Value Stored	1	1	2	3	4	15	16	17	17	18	18	20	22
CNT_i	6	10	1	8	7	12	5	2	7	0	3	4	11
p_i	1	1	0	0	0	0	0	0	0	0	0	0	0
cnt_i	0	0	0	0	0	1	0	0	0	0	0	0	0
cnt_{i+1}	1	1	1	1	1	0	0	0	0	0	0	0	0
load	0	0	1	1	1	1	0	0	0	0	0	0	0
LR	0	0	0	0	0	0	0	0	0	0	0	0	0
reset	0	0	1	0	0	0	0	0	0	0	0	0	0

(c)

Figure 4. Register File Example Functionality.

5. Results

For the purpose of validation and comparison against other works, the proposed architecture was modeled using the VHDL Hardware Description Language and synthesized with Xilinx ISE 8.2 targeted for a Virtex-II XC2V3000 device. Table 1 summarizes the FPGA hardware resource utilization and timing performance for the proposed architecture and related sorters. Note that a direct comparison between our proposed sorter and other sorters is not possible because there is not another linear sorter which performs the same functionality. Data for the Bitonic, Odd-Even, Column, and Shifter sorters were taken from [8], while data for the SFS, PRC, and SRC sorters were taken from [6]. Even though the proposed sorter is not the fastest, it is able to discard a value and to insert the new value in a single clock cycle while maintaining the data sorted. Block sorter on the other hand would require a large number of clock cycles to sort the data even if a single value is replaced.

Table 2 shows a comparison of the proposed architecture against other works in terms of the number of hardware elements they require. In the table, *n* refers to the number of data being sorted. The information for the other sorters is taken from [8]. The linear shifter requires the least hardware elements, followed by the column shifter. Although the proposed architecture requires more hardware elements that the column shifter, it is capable of sorting *n* values in as many clock cycles, similar to the linear shifter. The bitonic and the odd-even sorters require less time to sort the *n* values, however they require that all values to be sorted are available at the same time, which is not always possible specially in applications that produce data in stream fashion, also they require a larger number of hardware elements.

6. Conclusion

Sorting is one of the most important operations used in computers. When implementing statistical signal processing algorithms, it is commonly required to access values from a sorted array in a number of different ways. Some algorithms may require accessing the largest or smallest value in the array, the value stored in a specific position, or even values within a range. Additionally, as incoming data are processed in a stream fashion, a FIFO like behavior is required where the oldest value in the array has to be removed before making room to any new value. In this work a compact and efficient hardware implementation of a linear sorter based on FIFO scheme was presented. The architecture, composed of an array of identical processing elements, implements the insert sort algorithm in a compact an efficient way by performing a number of tasks in a single clock cycle. The architecture can be easily adapted to any length according to specific application needs and used as coprocessor or as module to implement a register file in specialized architectures. The architecture nature exploits the parallel nature of the insert sort algorithm and achieves excellent performance due to the use of identical processing elements that perform a number of tasks in parallel without need for a complex control unit.

Table 1. Performance Results with others Sorting Architectures.

Sorter	FPGA Used	Speed (MHz)	Latency Clock Cycles	Gate Count	Flip Flops	LUTs Count	Data Sorted	Word Size (Bits)
Bitonic	Virtex 2	127	14	153k	-	-	32	16
Odd-Even	Virtex 2	147	14	137k	-	-	32	16
Column	Virtex 2	66	32	23k	-	-	32	16
Shifter	Virtex 2	216	32	12k	-	-	32	16
SFS	VirtexE	115	32	18k	700	1,875	5x5	8
PRC	VirtexE	159	2	32k	440	3,270	5x5	8
SRC	VirtexE	96	32	13k	408	1,302	5x5	8
FIFO Scheme	**Virtex 2**	**126**	**32**	**25k**	**672**	**2,726**	**32**	**16**

Table 2. Comparison with others Sorting Architectures.

	Sorter				
Number of	Bitonic	Odd-Even	Column	Linear Shifter	FIFO Scheme
Multiplexers	$n\ (\log^2 n + \log n)/2$	$n\ (\log^2 n - \log n + 4)/2 -2$	$2n$	n	$\boldsymbol{4n}$
Comparators	$n\ (\log^2 n + \log n)/4$	$n\ (\log^2 n - \log n + 4)/4 -1$	n	n	$\boldsymbol{n}$
Register	$n\ (\log^2 n + \log n)/2$	$n\ (\log^2 n - \log n + 4)/2 -2$	$2n$	n	$\boldsymbol{2n}$
Counters	0	0	0	0	$\boldsymbol{n}$
Clock Cycles	$(\log^2 n + \log n)/2$	$(\log^2 n + \log n\ )/2$	$4n$	n	$\boldsymbol{n}$

7. Acknowlegments

First author thanks the National Council for Science and Technology from Mexico (CONACyT) for financial support through the scholarship number 204500.

References

[1] Donald E. Knuth: *Art of Computer Programming, Volume 3: Sorting and Searching*, Addison Wesley Professional, Second Edition, 1998.

[2] Colavita, A.A.; Cicuttin, A.; Fratnik, F.; Capello, G.: *SORTCHIP: A VLSI Implementation of a Hardware Algorithm for Continuous Data Sorting*, IEEE Journal of Solid-State Circuits, 2003, Vol 38 No. 6, pp. 1076-1079.

[3] Batcher, K. E.: *Sorting Networks and their Applications*,Proceedings of the AFIPS Spring Joint Computer Conference, 1968, Vol 32, pp. 307-314.

[4] Chung J. Kou; Zhi W. Huang: *Modified Odd-Even Merge-Sort Network for Arbitrary Number of Inputs*,IEEE International Conference on Multimedia and Expo, 2001. ICME 2001, pp. 929-932.

[5] Tabrizi, N. Bagherzadeh, N.: *An ASIC Design of a Novel Pipelined and Parallel Sorting Accelerator for a Multiprocessor-on-a-Chip*, ASICON 2005. 6th International Conference On ASIC, 2005. Vol 1, pp. 46-49.

[6] Hirschil B., Yaroslavsky L.P.: *FPGA Implementations of Sorters for Non-Linear Filters*, Eusipco 2004 : Proceedings of the XII European Signal Processing Conference Vol 1. pp 541-544. Vienna, Austria.

[7] Chi-Sheng Lin; Bin-Da Liu;: *Design of a Pipelined and Expnadable Sorting Architecture with Simple Control Scheme*, IEEE International Symposium on Circuits and Systems. ISCAS 2002, Vol 4, pp. 26-29.

[8] L. Ribas, D.Castells, J. Carrabina: *A Linear Sorter Core Based on a Programmable Register File*, XIX Conference on Design of Circuits and Integrated Systems, DCIS 2004. pp. 635-640. Bordeaux, France.

[9] Chen-Yi Lee; Jer-Min Tsai: *A Shift Register Architecture for High-Speed Data Sorting*, The Journal of VLSI Signal Processing, 1995, Vol 11 No. 3 pp. 273-280.