

# High performance Parallel Linear Sorter Core Design

David Castells-Rufas  
Cephis - UAB  
Edifici ETSE, UAB  
Bellaterra, Spain  
+34 93 581 3563  
**David.Castells@uab.es**

Màrius Montón  
Cephis-UAB  
Edifici ETSE, UAB  
Bellaterra, Spain  
+34 93 581 3563  
**Marius.Monton@uab.es**

Lluís Ribas  
Cephis-UAB  
Edifici ETSE, UAB  
Bellaterra, Spain  
+34 93 581 1078  
**Lluís.Ribas@uab.es**

Jordi Carrabina  
Cephis - UAB  
Edifici ETSE, UAB  
Bellaterra, Spain  
+34 93 581 3082  
**Jordi.Carrabina@uab.es**

## Abstract

Sorting hardware sorters exploit inherent concurrency to improve the performance of sequential, software-based sorting algorithms. They are often based on Batcher's odd-even or bitonic merging networks to attenuate the area-greedy hardware solutions.

In this paper, a new hardware sorter architecture is presented. It is composed of smaller sorter circuits inspired on insertion sorting algorithms that contain as many data-slice cells as data to sort. Such sorters are easily scalable and require minimal control schemes.

We shall outline the properties of the sorter modules in order to unveil some architectures that overcome their limitations. Such architectures are based on parallel combinations of such sorters to speed up the yield of the result systems. Different area and speed requirements lead to a variety of architectural considerations that produce optimal designs for each case.

An application of such sorters designs is presented and implemented on FPGAs. The synthesis results are compared with other parallel sorting architectures.

## 1. Introduction

Sorting is a basic operation in many computing systems. The sorting of a data set can be seen as a sequence of comparisons and move decisions to produce the final sorted set. Sequential

software sorting algorithms based on conventional general-purpose processors are limited to execution of a single operation (comparison, movement, flow control) at a time. On the other hand, hardware sorters can exploit the ability to compute many operations in parallel in order to compute the result much faster than software ones.

Bitonic sorting networks (BS) are one of the most well-known hardware parallel sorters [1],[2],[3]. Such networks are based on simple, so-called *2-sorter* units grouped in complex network of several stages. BS networks can be viewed as a binary tree of 2-sorters with  $N$  leaves at primary inputs. Therefore, they have a complexity of  $O(N \cdot \log_2 N)$  and a  $O(\log_2 N)$  delay time because of their logic depth in case of sorting  $N$  elements.

Pipelines can be used to greatly reduce the delay time between sorts, thus a pipelined bitonic sorter (PBS) [5] increases the throughput with the penalty of inserting a latency of just  $O(\log_2 N)$ .

However, area complexity of PBS is  $O(N \cdot \log_2 N)$ , which might become excessive for nowadays capacity of integration from relatively low values of  $N$  such as 64 or 128 and data widths large enough to handle references to data space in today's massive data management systems [4], [9].

To reduce area complexity, a much more in-depth analysis should be made: one of the basic ideas is to reduce the number of stages of the sorting network by re-circulating [7],[8] partially sorted set

of data through the network. There is an area overhead due to the interconnection network that must be used to enable data to re-circulate through a few stages of what would be the whole network. We shall call these sorting networks as *folded networks*.

With a convenient folding method as shown in [6], area complexity can be reduced to  $O(N)$ . However, these *bitonic folded sorters* (FBS) have a slightly greater delay times and lower throughput than PBS because of the more complex controller and of the added interconnection network.

These sorting networks deal with the problem of sorting  $N$  available data at a time with a fairly amount of circuitry and a very good time complexity. Unfortunately, they cannot handle as efficiently with progressively incoming data, as happens in most systems. What is more, their area complexity might hinder their implementation in FPGA or be much to demanding for ASIC in some cases.

In this work we address such problem by focusing our interest in area complexity, even with some penalty in time. As a result, minimal area, modular sorters can be produced. They are especially well suited for “progressive” sorting  $N$  data, and to take the  $N$  top data out of much bigger sets of data.

The paper is structured as follows. In section 2 we present the basic sorter module, the so-called *shifter sorter*. It has  $O(N^2)$  complexity with linear time and area complexities, but exhibits a modular, simple scalable architecture that enable rapid embedding into a variety of systems and parallel compositions.

Section 3 is devoted to a discussion of the problem of selecting the  $N$  most significant data from a larger set by using FBS, while highlighting the advantages and limitations of FBS in a SoC framework are highlighted. In section 4 we shall describe the usage of shifter sorters to solve the selection problem in common, generic SoC frameworks.

Finally, in section 5, results from different FPGA implementations of shifter sorter based systems are shown and compared with previous FBS

results. In the concluding section, we shall outline the main contribution of this work.

## 2. Shifter Sorter

The principle of the shifter sorting unit is to build up a list of sorted elements by inserting in order each element from the input set. Its name comes from using a shift register to store the sorted set. An element can be inserted every clock cycle, so as we have  $N$  input elements the sorted set will be built in  $N$  clock cycles.

Figure 1 presents the basic shifter sorter node that stores a pair (key, data). Sorters are commonly used to sort a vector of  $N$ -tuples by a given key rather than a vector of single keys, so it is interesting to consider it when designing a sorter.

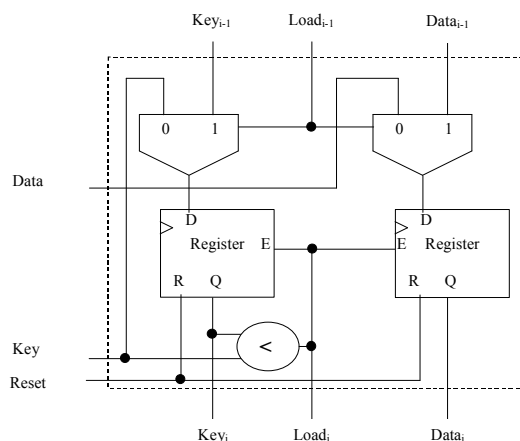


Figure 1 Shifter Node

The sorter node has a very simple operation: the input key is compared with the stored key and if it is greater a shift down operation is performed. Depending of the activity of above nodes the stored pair is the incoming pair or the pair shifted down by the above node.

$N$  sorter nodes are grouped to form the sorting unit like presented in Figure 2. The structure is very regular and although some signals (like new key and data) are broadcasted to every node of the sorter, the rest of the signals can be trivially

connected by adjacent placement, thus enabling a very compact ASIC (or FPGA) design.

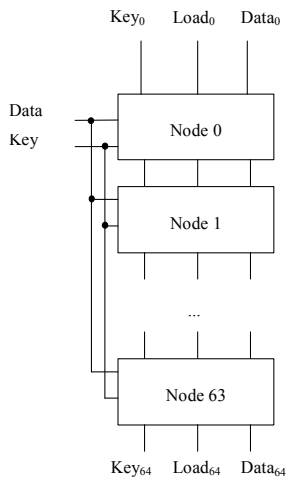


Figure 2 Shifter Sorter Design

As pointed before the shifter sorter has a  $O(N)$  complexity. The resource count for the whole sorter is depicted Table 1.

Table 1 Resources used by a N pair shifter sorter

Element	Key part count	Data part count
Registers	N	-
Comparators	N	N
Multiplexors	N	N

A drawback of the SS design is that the data is inserted in serial to the sorting unit. This forces the use of a serialization stage when the input is in parallel.

The sorted result set can be extracted from the SS either in parallel by taking data lower node outputs or in serial by adding a simple controller that feeds the unit with the maximum key value and gets the output from the last node.

To overcome the modest performance features of the SS when comparing with PBS or FBS a parallel technique is proposed.

The parallel system consists on replicating b SS units. The input set is splitted by b into disjoint sets that are feed in serial into replicated sorting units.

As each sorter process a disjoint set from the original input set the maximum values can be

dispersed in the different sorting units. A final merge of the values stored in the sorting units is necessary to obtain the sorted set.

The merge can be easily done by shifting down values in a domino fashion and adding a feedback loop to the first sorting unit that will store the final sorted set.

The complexity of the parallel sorter is  $O(bN)$  and the input set is sorted in  $O(N/b + N \cdot b)$

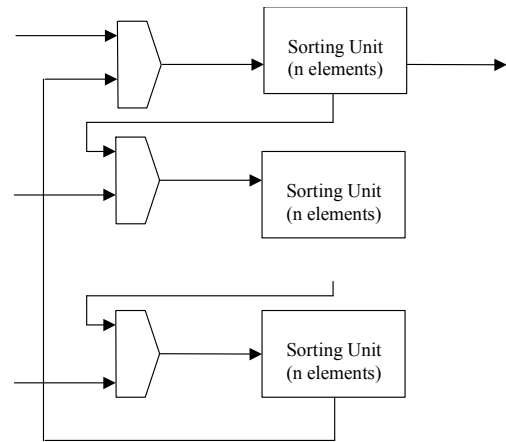


Figure 3 Parallel Shifter Sorters

### 3. Selecting first P values of a set

Selecting the first M most significant results of a large data set of N elements is a problem that most people face everyday. Querying an Internet Search Engine, reviewing the most recent bank account operations, and many other data intensive applications put emphasis in first N values of a sorted result set, considering other values to be irrelevant and so ignoring them.

As  $M \gg N$ , is not feasible to have a M parallel sorter due to area restrictions of current chips and the limitation of necessary I/O. Considering a N value low enough to able a on-chip implementation the generic overall max selector can be designed as shown in Figure 4 by using a sorter like FBS of  $2 \cdot N$  elements and a feedback loop.

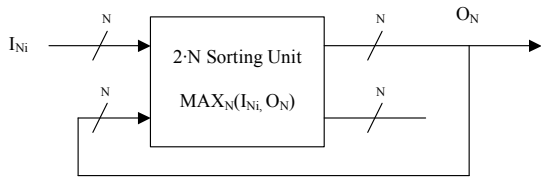


Figure 4 Overall  $MAX_N$  selector design using FBS

We will denote  $I_M$  to be the large input set containing  $M$  elements, and  $O_N$  the result set containing the  $N$  maximum elements from  $I_M$ . The idea is to split  $I_M$  into much smaller chunks of  $N$  elements (denoted as  $I_{Ni}$ ) that are feed to the sorting unit. The feedback loop allows computing the overall maximum set between the input chunk and the already sorted set  $O_N$ .

Notice that the function  $MAX_N(I_{Ni}, O_N)$  takes  $2 \cdot N$  input elements.

If a shifter sorter is used there is no need of feedback since the fact that it always maintains the list of the sorted overall maximum. On the other hand it introduces a serialization phase of  $I_{Pi}$  since Shifter Sorter only process a single element at a time or  $b$  elements, in case of using a parallel SS design. This allows a more compact design having  $N$  elements with the penalty of reducing throughput.

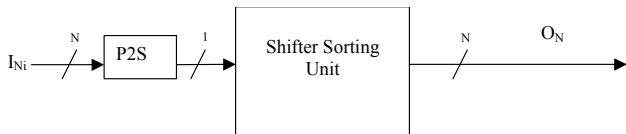


Figure 5 Overall  $MAX_N$  selector design using SS

#### 4. Integration of a FBS into a SoC system

Figure 6 shows the integration of a sorter unit into a generic SoC environment. The system interconnection network (or BUS) is a crucial element of the system because it becomes the bottleneck of the system.

Consider having a BUS width of  $W$  bits and that the elements to sort consist of pairs (key, data) of width  $l_k$  and  $l_d$  bits respectively.

Let  $q$  be the number of simultaneous elements traversing the bus.

$$q = W / (l_k + l_d)$$

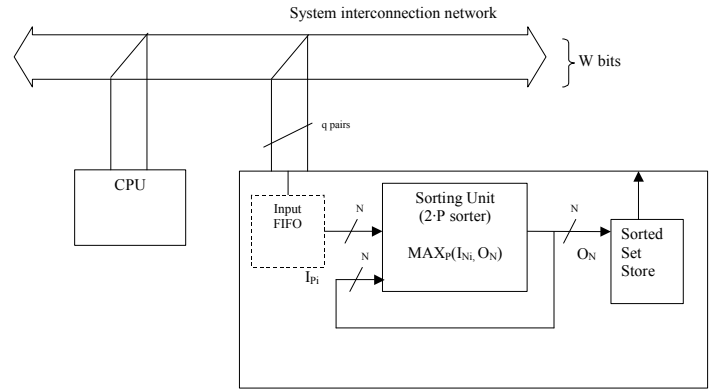


Figure 6 Integrating FBS based selector in SoC

There is a high probability that  $q$  being much lower than  $P$ .

For instance let's say we want to build a sorter core for a system with  $W = 64$  to sort  $P=64$  pairs of  $l_k=8$  and  $l_d=8$ . For that system  $q$  would be 4.

Having  $q \ll N$  means that sorter unit can't start processing the input set until  $N$  elements are received in the sorting unit. Moreover, depending on the bus width and  $N$ , the incoming elements must be stored into a FIFO as shown in Figure 6 which adds additional logic and contributes to an increment of the complexity of the FBS to  $O(3N)$ .

The time need by the FIFO+FBS system must be also revised:

The FIFO is filled in  $O(N/q)$  time whereas the FBS consumes its data and it is ready for a next chunk in  $O(\log_2 N)$  time.

If the FIFO is filled faster ( $N/q < \log_2 2 \cdot N$ ) as the result of of having a big  $q$ , the throughput is ruled by the FBS and the potential BUS bandwidth is not used.

On the contrary if  $N/q > \log_2 2 \cdot N$  the FIFO is slower than FBS and it becomes the limiting factor wasting FBS computation power.

The optional operation point of the system is when  $N/q = \log_2 2 \cdot N$

So, to sum up,

- when  $q=N$  we need no FIFO and time is  $O(\log_2 N)$
- when  $q < N$  time is  $O(N/q)$

As we stated above in common SoC systems is much more probable to have a  $q \ll N$ . So we consider having a time  $O(N/q)$ .

### 5. Integrating a shifter sorter on a SoC

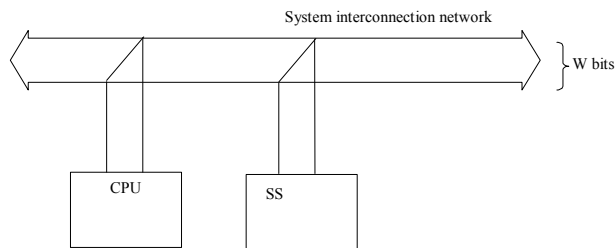


Figure 7 Integrating SS based selector in SoC

To consider the integration of SS into a SoC system we must take into account the  $q$  number of elements traversing the bus and the ability to scale the SS for parallel operation.

It must be considered that the final merge of a parallel SS is only executed after  $M$  elements have been processed, being  $M \gg N$ . The final merge phase then can be treated as irrelevant for the processing time calculation.

When  $q > b$  the SS can not consume the provided data, so either data is serialized by a serializer (with a hardware cost) or data is produced in  $b$  elements wasting part of the  $q$  elements of the bus, so the effective  $q$  becomes  $b$ .

If  $q < b$  additional resources are not useful and only  $q$  sorters are used.

So whatever the values of  $q$  and  $b$  are given the effective parallelization factor is  $\min(q, b)$ .

So time is  $O(N/\min(b,q))$   $O(bN)$ .

Comparing with the FBS based solution we have a very similar time order.

- FBS:  $O(N/q)$
- SS:  $O(N/\min(b,q))$

So, for  $q < N$  SS and FBS both compute in  $O(N/q)$ .

### 5. Hardware Implementation

The simplicity of SS allows a very fast implementation of the design. In contrast to FBS that need a rather complex controller and additional FIFOs for controlling the flow to the sorter input. Various SS have been implemented with  $b=1$  and various  $N$  sizes on FPGAs. Synthesis results are shown in Table 2.

Table 2 Implementation of the shifter sorter in a Xilinx Virtex-E XCV2000E FPGA width different input widths (16 to 128 keys)

N (keys)	Max. Period (ns)	Max. Speed (MHz)	Area (Slices)	FPGA usage
16	7.463	133.994	415	2%
32	7.463	133.994	830	4%
64	7.463	133.994	1660	8%
128	7.463	133.994	3320	17%

Comparing obtained results with FBS results by [6] the SS's clock period is smaller than FBS's thanks to the shorter logic depth given by the simple design. Another interesting point is that area occupancy factor between FBS and SS. If we take  $N$  as 64 a FBS based selector uses a 128 element sorter whether the equivalent SS based selector uses a 64 element sorter. Table 3 highlights the factors between both implementations.

Table 3 Comparison of implementation results

	FBS 128	SS64	SS64 / FBS 128
Area (Slices)	5390	1660	~1/3
Speed (Mhz)	73Mhz	133Mhz	1,8

As the equivalent SS implementation takes a fraction of the area used by FBS one, there is still place to replicate more SS instances within the same space.

Thus,  $b$  can be increased to 3 or 4 in order to maximize the throughput of the system. Table 4

gives an idea of how throughput can be boosted by increasing the b value beating FBS alternative design and still saving more area.

**Table 4** Scaling SS to increase throughput

Design	Area (Slices)	Speed (MHz)	Throughput (pairs/s)
FBS128	5390	73 Mhz	292 Mp/s
SS64x2	3320	133 Mhz	266 Mp/s
SS64x3	4980	133 Mhz	399 Mp/s
SS64x4	6640	133 Mhz	536 Mp/s

## 7. Conclusion

There are many data-management applications in which rapid sorting are required. Particularly, those in which searching operations are done are very demanding in terms of amount of data to be processed but the result usually consists of a relatively small set of them. We have analyzed the solutions based on Batcher's bitonic sorting networks and shown that area complexity matters when facing implementation. Furthermore, we have proposed a different, new solution based on simple insertion sorting.

The proposed shifter sorter is based on a simpler element than the 2-sorter and does not require controlling circuitry. Its data-slice structure would drastically minimize area and wire length in ASIC implementations and helps in efficient FPGA usage.

Moreover, the shifter sorter has a smaller logic depth than sorting network counterparts, thus being able to operate at higher clock frequencies.

On the other hand, we have shown that shifter sorters can be easily combined in parallel architectures with a minimum controlling scheme that consists of additional muxs and a counter. Such parallel structure with linear link exhibits the drawback of latency equivalent to the overall data-slice module count.

In common SoC frameworks, bus widths strongly limit the amount of data to be taken by the sorter at a cycle. Therefore, sorting networks might

require the use of a FIFO to get the data to sort, while the shifter sorter acts as a priority queue that does not require further processing. As a result, shifter sorters have better performance. What is more, they are ready to use in a design flow because of their scalability and minimum controlling circuitry.

## 8. References

- [1] K.E. Batcher, "Sorting Methods and their Applications" AFIPS Spring Joint Computing Conference, vol. 32, pp. 307-314, 1968.
- [2] Bitton, DeWitt, Hsiao, and Menon, "A taxonomy of parallel sorting," *Computing Surveys*, vol. 16, no. 3, pp. 287-318, 1984.
- [3] G. Brassard, and P. Bratley, *Fundamentals of Algorithmics*. Prentice-Hall, 1997, ch. 11.
- [4] K. Claessen, M. Sheeran, and S. Singh, "The Design and Verification of a Sorter Core," *11<sup>th</sup> Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*. Springer-Verlag Lecture Notes in Computer Science. Vol 2144, September 2001, pp. 355-369.
- [5] R. Kannan, "A pipelined single-bit controlled sorting network with  $O(N \log^2 N)$  bit complexity," *Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM '97*. Proceedings IEEE, vol.1, pp. 253 - 260, April 1997.
- [6] C. Layer and H-J Pfeleiderer "A reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data"
- [7] J.-D. Lee, and K. E. Batcher, "Minimizing communication of a recirculating bitonic sorting network," *Proc. of International Conf. on Parallel Processing*, pp. I-251-I-254, 1996.
- [8] J.-D. Lee, and K. E. Batcher, "Minimizing communication in the bitonic sorter," *IEEE Trans. on Parallel and Distributed Systems*, vol. 11, no. 5, May 2000.
- [9] —, *A Sorter Example in Lava*, [www.xilinx.com](http://www.xilinx.com), 2004.