Manuscript
Click here to view linked References

A Versatile Linear Insertion Sorter Based on a FIFO Scheme

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

Abstract

A linear sorter based on a First-In First-Out (FIFO) scheme is presented. It is capable of discarding the oldest stored datum and inserting the incoming datum while keeping the rest of the stored data sorted in a single clock cycle. This type of sorter can be used as a coprocessor or as a module in specialized architectures that continuously require to process data for non-linear filters based on order statistics. This FIFO sorting process is described by four different parallel functions that exploit the natural hardware parallelism. The architecture is composed of identical processing elements, thus it can be easily adapted to any data lengths, according to the specific application needs. The use of compact identical processing elements results in a high performance yet small architecture. Some examples are presented in order to understand the functionality and initialization of the proposed sorter. Results of synthesizing the proposed architecture targeting a Field Programmable Gate Array (FPGA) are presented and compared against other reported hardware based sorters. Scalability results for several sorted elements with different bits widths are also presented.

Keywords: Hardware Sorters, Linear Sorters, FIFO

1 Introduction

Sorting is one of the most important operations performed by computers. Given their practical importance, algorithms for sorting data have been the focus of extensive research, resulting in several algorithms proposed to address specific problems. First, serial sorting algorithms were investigated. Then parallel sorting algorithms became a very active area of research, and several models of parallel computations have been considered and developed, deriving in sorting algorithms that later on were implemented in hardware. All developed serial algorithms implemented in software are evaluated by their time complexity and other properties such as the time-memory trade-offs (the amount of additional memory required to run the algorithm and the memory for storing the initial sequence), stability, and sensitivity to the initial distribution of the data (best case and worst case). In parallel processing, when processors share a common memory, the idea of contiguous memory locations is identical to that in serial processors. Therefore, this situation can be analyzed identically as the serial case.
When the processors do not share memory and they communicate with each other through an interconnection network, the time complexity is expressed in terms of parallel comparisons and exchanges between adjacent processors [1, 2].
For certain applications, like median filters, ATM (Asynchronous Transfer Mode) switching, order statistics filtering and, in general, continuous data processing, sometimes software-only implementations of sorting algorithms do not achieve the required processing speed [3]. 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 exploring a number of custom architectures. Hardware sorters are evaluated according

to area requirements (number of Flip-Flops, comparators, control logic, gates, and LUTs), processing time, including latency and maximum operating frequency, and power consumption. Hardware sorters can be grouped into 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 a datum takes its corresponding place. Linear sorters are based on the idea that data to be sorted come in a continuous stream, one datum at a time; each datum is inserted into its corresponding place in a register group (sorting array) at the same time that one of the stored data is deleted. Figure 1.a represents the sorting network idea, where data are firstly stored and then sorted by a sorting network in a parallel fashion. The gray blocks represent the first and last stored datum. The first stored datum is the first element leaving the file register, i.e. like in a FIFO scheme. Figure 1.b represents the linear sorter idea, where data are always sorted, thus the first and last datum are merged inside of the sorting array. On these sorters, a deleting mechanism must be used in order free space for incoming data. Some examples of these mechanisms are deleting the oldest datum, selecting one datum or deleting the greatest or the smallest one.

This work is based on the idea of sorting the data as they are introduced into the sorting array, discarding the oldest datum in the sorting array while maintaining the data sorted, all that in a single clock cycle. This FIFO scheme can be used in applications that are continuously processing data in serial fashion like non-linear filters such as the rank-order filters, weighted order statistics (WOS) filters and stack filters. These nonlinear

filters are based on order statistics, thus require to access an ordered list of the random variables $X_1, X_2, X_3, \ldots, X_n$. An ascending sequence can be represented as follows:

$$\boldsymbol{X_1 \leq X_2 \leq X_3 \leq, \ldots, \leq X_n} \quad (1)$$

Where indexes indicate the rank-order number. The idea on rank-order filters is to select a value X_i, where $i \in \{1, 2, 3, \ldots, n\}$ from the sequence in equation 1 and then to use this X_i value as a sample of the sorted data.

Several signal processing applications based on order statistics require a sorter with FIFO-like behavior. For example, in image processing, non-linear filters such as: rank order, max/min, mean, morphological, and adaptive trimmed mean, are commonly used as they offer benefits such as edge preservation, robustness, adaptation to noise statistics and preservation of image details [5, 6]. Other applications can be found in radar and sonar systems where the detection procedures involve the comparison of the received signal amplitude to a threshold. This threshold is obtained by using a Constant False Alarm Ratio (CFAR) algorithm, which requires keeping sorted the incoming echo samples [7]. More examples of applications in signal processing are: smoothing of time-series, maximum likelihood estimation, and one-dimensional non-linear filtering [8]. These applications 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. it discards the oldest datum when a new one arrives, while allowing flexible access to its contents.

2 Related Work

As mentioned earlier, 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, which consists on a comparator and are located in the nodes of the network. The goal of each PE is sorting two input data in ascending (or descending) order by placing the larger (or smaller) datum in a specific output. This technique supposes that a block of data is available for being 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, even if only one input datum 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 PEs from input to output). Both metrics are highly dependant on the number of data the architecture can sort. Figure 2 shows an eight elements input sorting network example, whose size is of 24 PEs and has a depth of 6 stages. Each PE is represented by two interconnected nodes.

Linear sorters are useful when sorting data streams and where sorting operation must be carried out after each input datum 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 datum or a datum stored in adjacent cells. The advantages of this approach are that it uses fewer area resources and data are always

sorted. Figure 3 shows a linear sorter example, which inserts the input datum in its corresponding place and discards the greatest datum stored.

2.1 Sorting Networks

In [9], Batcher introduced the concept of sorting networks. In his work he presented the odd-even merging and bitonic networks. The odd-even merging network consists of two networks that sort all the data contained in odd and even positions separately, applying an interactive rule. The bitonic network works, similarly to the odd-even, merging two monotonic sequences, one in ascending order and the other in descending order. These two monotonic sequences are built by sorting the input data in ascending and descending lists, and merging them. 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 [10] proposed a modification of the odd-even sorting network. They proposed a network that can sort any m input data smaller than n, which is the maximum number of data that the network can sort. In [11], 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 root node is the output. This scheme works in a PISO (Parallel Input-Serial Output) fashion, thus requiring several clock cycles to flush the tree after the beginning of the process. In [12] 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 [12] are based on shift register architectures operating in a FIFO scheme. One of these architectures, called Serial Rank Computer (SRC), includes two attributes: value and rank. The incoming data are arranged according to their arrival sequence accompanying each number with 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 [3] 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 in the same way. Data are sorted according to the tags preserving the order of words with identical tags.

Chin-Sheng and Bin-Da Liu in [13] propose a sorter that uses a column of n PEs to progressively sort n data. These PEs are composed of two registers and a Compare-Swap Cell (CS), which is built by a comparator and a swap unit. The cells are connected 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 [14], Lluís Ribas et al. propose a sorting array (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 datum, looks for the right position in the sorted list in order to perform the insertion of the unsorted datum into its corresponding place. This architecture only shifts data to one direction, discarding the smallest datum. The data-slice cell is composed of a multiplexer, a register and a comparator, resulting in a compact and simple architecture. A similar sorting scheme is proposed in [15], where data contained in the sorting array can be left o right shifted depending on the operation to perform, insertion or descarding. Both the datum to be inserted and the one to be deleted are specificated by the input signal. To perform the inserting or deleting process, the cell must perform four basic operations: shift right, shift left, load and initialize.

For applications that require continuously data processing, sorting networks are not the best option, as they may become a processing bottleneck. Although pipelining techniques can be applied, there is a latency time that must be considered as a trade-off. Linear sorters have a better performance for these data streaming applications. The linear sorters mentioned in literature have different features that make them suitable for different applications, in this case, a FIFO sorting behavior for performing the rank-order operation. The SFS presented by Hirschil and Yaroslavsky [12] works in a FIFO basis and it needs $n+1$ cells to sort n elements and two levels of memory elements. The sorters proposed in [13] and [14] only keep the greatest data discarding the smallest one from the linear sorter. Therefore the FIFO functionality is not achieved by these two linear sorters. The sorter proposed in [15] can achieve the FIFO functionality only if the external logic specifies that the oldest datum is the one to be discarded. Our proposed

solution achieves the FIFO functionality, as the SFS, but it requires less logic, working on one clock edge. It takes some ideas from previous works, especially from [12] and [13], but it has been modified to work in a FIFO fashion.

3 Proposed Insert Sort Functions

The proposed linear sorter is based on the insertion sort algorithm. This algorithm performs, for every unsorted datum, a procedure that looks for the appropriate position in the sorted list to insert the incoming datum [14]. 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;
     }
     if (i==0)
          discard D;
     else
          R[i-1] = D;
}end function;
```

On each iteration, this algorithm inserts an incoming datum into the vector R of length n (n>0) in ascending form and discards the datum with the smaller value, which is located in $R[0]$. To ensure a proper behavior it is assumed that the value stored in $R[n]$ exists and is larger than any value to be sorted, i.e. it represents infinity. In case that the incoming datum is smaller than $R[0]$, it will not be inserted into the array and will be discarded as indicated by the sentence *discard D*.

The *while* sentence can be easily converted to a *for* loop with bounded limits. This transformation makes the algorithm suitable to be parallelized. The next code shows the result of this transformation.

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

As vector *R* has a finite length, a discarding condition must be used in order to provide an empty space for the incoming datum. In [14], the condition used for deleting is to discard the smallest stored datum, meanwhile in [15], the datum to be discarded is indicated by an external input signal. In the proposed architecture, a FIFO scheme is used i.e. the oldest datum in the array is discarded, allowing for the incoming datum 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 datum is shifted, then its corresponding life period value is shifted as well. The life period value is increased by one every time a new datum is inserted. When the life period value has expired, that is, when it reaches a value equal to the number of elements in the array (n), the corresponding datum is discarded, making an empty space in the vector and thus allowing the insertion of a new datum. For this scheme, three different operations may be performed in order to keep the array sorted: shift the datum and life period value to the left, to the right, or hold the datum. To know the direction the data should be shifted to, every element in the array must know on which side, on relation to itself, the datum

that is going to be discarded is located. Also, it must know on which side the incoming datum must be stored.

This functionality can be achieved by creating an array of PEs, called Sorting Basic Cell (SBC). In order to fulfill the FIFO functionality, the SBCs must be interconnected (figure 4) in a simple linear structure, called sorting array. This linear structure can be easily expanded depending on the application. For each iteration, one of the SBCs must discard its value and, at the same time, all the SBCs must either hold their previous value, or store the value coming from one of the neighbor cells (left or right). Only one clock cycle is needed to perform these actions (discarding the oldest data, holding data, 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 5, where the gray cell indicates the location of the datum to be discarded):

1) The incoming datum is inserted to the left of the cell that discards its stored value. In this case, data from $R[i]$ to $R[n\text{-}2]$ must be shifted to the right side and the incoming data is inserted in $R[i]$.
2) The incoming datum is inserted to the right of the cell that discards its stored value. In this case, data from $R[i]$ to $R[3]$ must be shifted to the left side and the incoming datum is inserted in $R[i]$.
3) The incoming datum is inserted at the same position of the discarded value i.e. $R[i]$. The rest of the cells hold their values.

In order to support these three insertion cases, each SBC must perform four different functions. These functions describe the interaction that the *i-th* SBC has with its neighbors through different signals. Figure 6 shows the interconnections between two SBCs, where the *i*+1 and *i*-1 indexes represent incoming signals from the right and left SBC neighbor respectively. Please note that for the sake of clarity, in all figures that describe the cells and sorting array, the register or cell that holds the variable R[*i*] will be labeled as R_i. This also applies to other registers that hold variables that require an index *i*.

In order to explain the previously mentioned functions, we define the next variables: *CNT*[*i*] stores the life period value of the *i-th* SBC, the data stored on the *i-th* SBC is represented by *R*[*i*], *cnt*[*i*] 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 ports 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 sends 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 datum, otherwise it must send to its left its current value and to the right the incoming datum.

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

The SBC_ResetPeriodLife function sets to zero the life period value if certain conditions are met. The first condition of this function checks if the *CNT*[*i*] has reached its maximum value or if the incoming datum *D* must be inserted in the *i-th* SBC (exclusive or between the *cnt*[*i+1*] and *R*[*i*] > *D*). The inner conditions check those cases where the SBC's counter must be set to zero. This action takes place when the incoming datum will be stored on the *i-th* SBC therefore setting the life period value to zero is needed. These conditions allow to set to zero the life period value in only one SBC inside the sorting array at a same time.

```
function SBC_UpdateValues
D = Incoming Data;
if(CNT[i] == n) or (R[i] > D xor cnt[i+1] == 1){
     if (R[i] > D){
          R[i] = R[i+1];
          CNT[i] = CNT[i+1];
     }else{
          R[i] = R[i-1];
          CNT[i] = CNT[i-1];
     }
} end function;
```

The SBC_UpdateValues function is in charge of updating the store value and the life period value coming from one of the neighbors. In order to know which neighbor to take the new value form, two conditions must be met. The first condition checks if the *CNT*[*i*] has reached its maximum value or if the incoming datum *D* must be inserted in the *i-th* SBC (exclusive or between the *cnt*[*i+1*] and *R*[*i*]>*D*), similar to the SBC_ResetPeriodLife function. The second condition selects which neighbor (left or right) the *R*[*i*] and *CNT*[*i*] variables of the *i-th* SBC must take their value from. Even though the first condition is the same as the one shown in the 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] == n  or cnt[i+1] == 1){
     cnt[i] = 1;
} else{
     cnt[i] = 0;
}end function;
```

This final function, SBC_PropagateFlag, checks if the life period value of the SBC has expired or the right SBC life period value has expired; if so, then the *cnt*[*i*] flag is set to one. The flag *cnt*[*i*] must not be confused with *CNT*[*i*], because the first one is used by the functions as one of the conditions to update the *i-th* SBC, meanwhile the second one is the life period value of the *i-th* SBC. These four functions describe the interactions the *i-th* SBC has with its neighbors and how these interactions allow the sorting array to perform the three insertion cases previously described.

It is important to emphasize that the proposed sorter differs from other sorters as it implements a FIFO-like scheme where the oldest datum in the sorting array is discarded to make room for every incoming data. Although our sorter performs the same FIFO functionality of the SFS presented in [12], they differ in their internal functionality. The SFS stores an input vector of data in the order that it is received, discarding the oldest datum i.e. the FIFO scheme. At each clock cycle, one datum enters taking its corresponding place inside the sorter according to its value and other datum leaves the sorter. These two characteristics are met by our sorter too. Also, both sorters need to store the life period value, however, the SFS sorter requires two levels of memory elements: main and auxiliary; meanwhile our sorter only requires one memory level. Moreover, the SFS needs $n+1$ cells to sort n elements, requiring an overflow cell. On the other side our sorter needs only n cells to sort n elements. Although both the SFS and our sorter operate in one clock cycle, the SFS works during both clock edges: rising and falling edges. During the rising edge it is inserted the incoming datum by shifting the needed data to the overflow cell side, having $n+1$ sorted elements. In the next falling edge the oldest stored datum in the $n+1$ cells is discarded by shifting since the position of the overflow cell until the oldest datum. Our sorter is able to perform all described operations in a single clock edge, which is a desired featured in digital architectures.

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 ‘major than’ comparator, four 2-1 multiplexers and control logic (figure 7). In order to build the local control unit of the SBC, the conditions presented in the four functions previously

described were mapped into boolean equations. This control logic consists of four boolean equations, which control the register, the counter and the multiplexers. Equation 2 is used as a condition in SBC_ResetPeriodLife and SBC_UpdateValues functions. Also this equation controls when the register and the counter must take the neighbor value. The origin of the data (left or right side) is selected by equation 3, which is used in function SBC_UpdateValues. The function SBC_ResetPeriodLife is represented by equation 4 and it indicates if the counter must be set to zero. Finally, equation 5 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.

$$load = (p_i \oplus cnt_{i+1}) + expired \tag{2}$$

$$LR = p_i \bullet load \tag{3}$$

$$reset = load \bullet [(p_{i-1} \bullet \overline{p_i}) + (p_i \bullet \overline{p_{i+1}})] \tag{4}$$

$$cnt_i = cnt_{i+1} + expired \tag{5}$$

where the signal p_i is the comparator output as described by the SBC_SendData function. If all comparators inside the sorting array are inverted to a 'minor than' comparator, then the sorting array would work in a descending fashion. The signals p_{i+1} and p_{i-1} correspond to the right and left SBC neighbors respectively. The signal *expired* indicates when the life period value has reached its maximum value inside of one SBC 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 leftmost p_{i-1} signal's value is always 1 and the rightmost p_{i+1} signal's value is always 0. This can be viewed as the leftmost datum having the smallest value while the rightmost has the largest one.

To ensure proper behavior, all registers R_i must be initialized to zero, while life counter values *CNT* must be initialized according to $CNT[i] = i$. The *CNT* counter word size depends on the sorting array's length, being a function of

$$\lceil log_2 n \rceil \tag{6}$$

where n is the sorting array length.

Figure 8 and figure 9 exemplify how the SBC's control signals work in two different situations. In both figures, the first row contains the sorted data currently stored in the sorting array in ascending form, the second row contains the corresponding life period values and the following rows contain the control signals' values needed to perform the insertion operation. The gray column indicates the oldest data to be discarded, whose period life value is 12. Different clock cycles are represented by different tables in the same figure. Only one SBC can have the reset signal asserted at each clock cycle. When one SBC asserts the reset signal, it means that this SBC is where the incoming datum D will take place in next clock cycle. The *expired* signal is only asserted when the SBC's period life value has reached the same value of the sorting array length, meaning that this is the oldest datum stored (shown by the gray columns). Similarly to the *reset* signal, only one SBC can have the *expired* signal asserted at each clock cycle. Note how cnt_i signal is propagated through the sorting array to the left side once it is activated according to equation 5. Although *LR* signal is always calculated according to equation 3, it is only considered when in the same SBC the load signal is asserted.

Figure 8 exemplifies how the SBC's control signals work at each clock cycle allowing the sorting array to perform the sorting algorithm. Different clock cycles are represented

by different tables in the same figure. This figure illustrates the three previously mentioned insert cases in a 3 steps sequence. In this example the array already holds sorted sequence (figure 8.a). At the first clock cycle the incoming datum value is $D = 2$. The control signals take their corresponding values allowing the inserting, shifting and deleting operations. Note that at this moment, the incoming datum has not been inserted yet and the oldest datum is still in the sorting array. At the next clock cycle the sorting array is updated (figure 8.b) and the second incoming datum $D = 18$ is also inserted in its corresponding position performing similar actions as with the first incoming datum. In this case, there is another datum inside of the sorting array which has the same value. When this case occurs, the new datum is inserted to the left side of the datum with the same value, having the oldest datum always at the right most position. This behavior is because SBC has the comparator unit which performs the comparison in a strictly 'minor than' fashion between its stored datum and the incoming datum. Finally, in figure 8.c, the incoming datum value is $D = 11$ which is placed in the SBC that just discarded its datum. Note that this figure only shows the control signals values.

Figure 9 exemplifies how the SBCs must be initialized. After the reset signal is asserted, all the stored data in the SBCs take a zero value, while the period life values are set according to $CNT[i] = i$ (figure 9), i.e. the SBC position inside the sorting array. This special initialization for the $CNT[i]$ is because it is always needed to discard only one datum from the sorting array in order to make possible the insertion operation. At this moment, the incoming data value is $D = 6$, thus the control signals take their value in order to perform the insertion of this datum. Like in the previous example, in the next clock cycle, $D = 6$ is inserted and the datum that has the oldest period life value is deleted, following the sorter normal functionality. Figures 9.b-f show the insertion

process after the initialization, being $D = \{3, 5, 0, 1, 4\}$ the respective incoming datum value sequence for these figures. Note that in figure 9.d the incoming datum value $D = 0$ is inserted in the left most side of the sorting array. This is similar to the behavior shown in figure 8.b.

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 9.2 targeted for a Virtex-II XC2V3000 FPGA device and for a VirtexE XCV200E. The design was also synthesized for a Virtex 5 XC5VLX220 FPGA device in order to show results for scalability in a more up to date device. Table 1 summarizes the FPGA hardware resource utilization and timing performance for the proposed architecture and related sorters, using Virtex-II and VirtexE. 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 2, n refers to the number of values being sorted.

Data for the Bitonic, Odd-Even, Column, and Shifter sorters were taken from [14], while data for the SFS, PRC, and SRC sorters were taken from [12]. Note that a direct comparison between our proposed sorter and other sorters is not possible, except by the SFS sorter which performs the same FIFO functionality. Even though our proposed sorter is not the fastest among all the sorters shown in table 1, it uses less hardware resources (gates, FFs and LUTs) than the SFS sorter that performs similar functions. The SFS sorter is faster that the proposed linear sorter, but this SFS needs longest clock

period to ensure the signal stability as it works during both clock edges. Both sorters are able to discard a datum and to insert a new datum in a single clock cycle while maintaining the rest of the data sorted. Network sorters on the other hand would require a larger number of clock cycles to sort the data even if only a single datum is replaced. It is important to mention that the numbers of flip-flops required by the first 4 sorters in Table 1 are not explicitly reported in [14]. These values (marked with *), were estimated by analyzing the structure of the sorters and taking into account the number of values and word sizes shown in table 2.

Although the Bitonic and Odd-Even sorters have a greater maximum frequency operation and a smaller latency than our FIFO scheme, they need to re-sort the data once a datum has changed. This make them impracticable for a continuously data processing. Also both sorters require less time to sort the *n* data than the linear sorters, however they require that all data to be sorted is available at the same time, which is not always possible specially in applications that produce data in a stream fashion. Moreover they require a larger number of hardware elements.

According to table 2, the linear shifter requires the least quantity of hardware elements, followed by the column shifter. Although the proposed architecture requires more hardware elements than the column shifter, it is capable of sorting *n* data in as many clock cycles, similar to the linear shifter and the SFS sorter. Even though the proposed architecture and the SFS performs the sorting operation based on a FIFO way, differing in their internal functionality, the FIFO scheme presented requires less hardware elements than the SFS.

Table 3 shows the scalability results of the sorting architecture for the Virtex 5 device. For this comparison, different word sizes and sorting array lengths combinations were used. The scalability data results are grouped by number of sorted elements (amount of SBCs) and their word size in bits. All frequencies are greater than 150 MHz, thus area results are the main concern. Figure 10 shows the LUTs results in a graph for clarity purpose. By increasing the file register size, the number of LUTs used grows more than twice as the SBC amount is increased at the same proportion.

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 datum stored in a specific position, or even data within a range according to the application. Additionally, as incoming data are processed in a stream fashion, a FIFO like behavior is required where the oldest datum in the array has to be removed before making room for any new datum. In this work, a compact and efficient hardware implementation of a linear sorter based on a FIFO scheme was presented. The architecture, composed of an array of identical processing elements, implements the insert sort algorithm in a compact and efficient way by performing a number of tasks in a single clock cycle. The architecture is based on four functions whose characteristics are translated into four boolean equations, working as an internal control logic for each of these processing elements. The architecture can be easily adapted to any length and data width according to specific application needs and used as a coprocessor or as a module to implement a sorting array in specialized

architectures. The nature of this architecture exploits the parallel properties 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 the need of a complex control unit.

7 Acknowledgments

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

8 References

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

[2] Bitton D.; DeWitt J. D.; Hsiao D.K..; Menon J.: A Taxonomy of Parallel Sorting, ACM Computing Surveys, 1984, Vol 16 No. 3, pp. 287-318.

[3] 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.

[4] Thomson Leighton: Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes, Morgan Kaufman Publishers, 1992.

[5] Ioannis Pitas: Digital Image Processing Algorithms and Applications, Wiley-Interscience, Feb 2000.

[6] Rafael C. Gonzalez, Richard E. Woods: Digital Image Processing, Prentice Hall, Third Edition, 2007.

[7] Merrill Skolnik: Introduction to Radar Systems, McGraw-Hill, Third Edition, 2002.

[8] N. Balakrishnan, C.R. Rao: Handbook or statistics 17: Order Statistics: Applications, Elsevier Science Pub Co.1998.

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

Figure 1

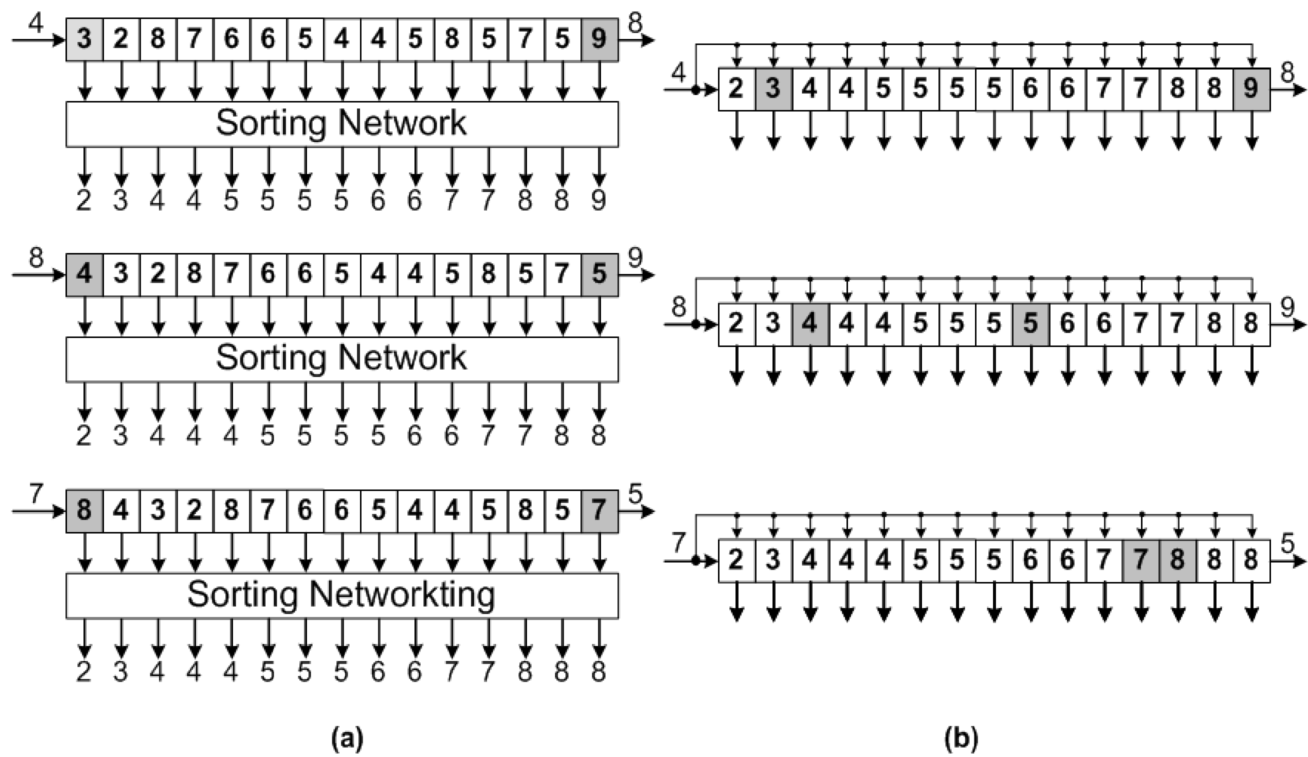

Figure 1. Sorting network and linear sorter

Figure 2

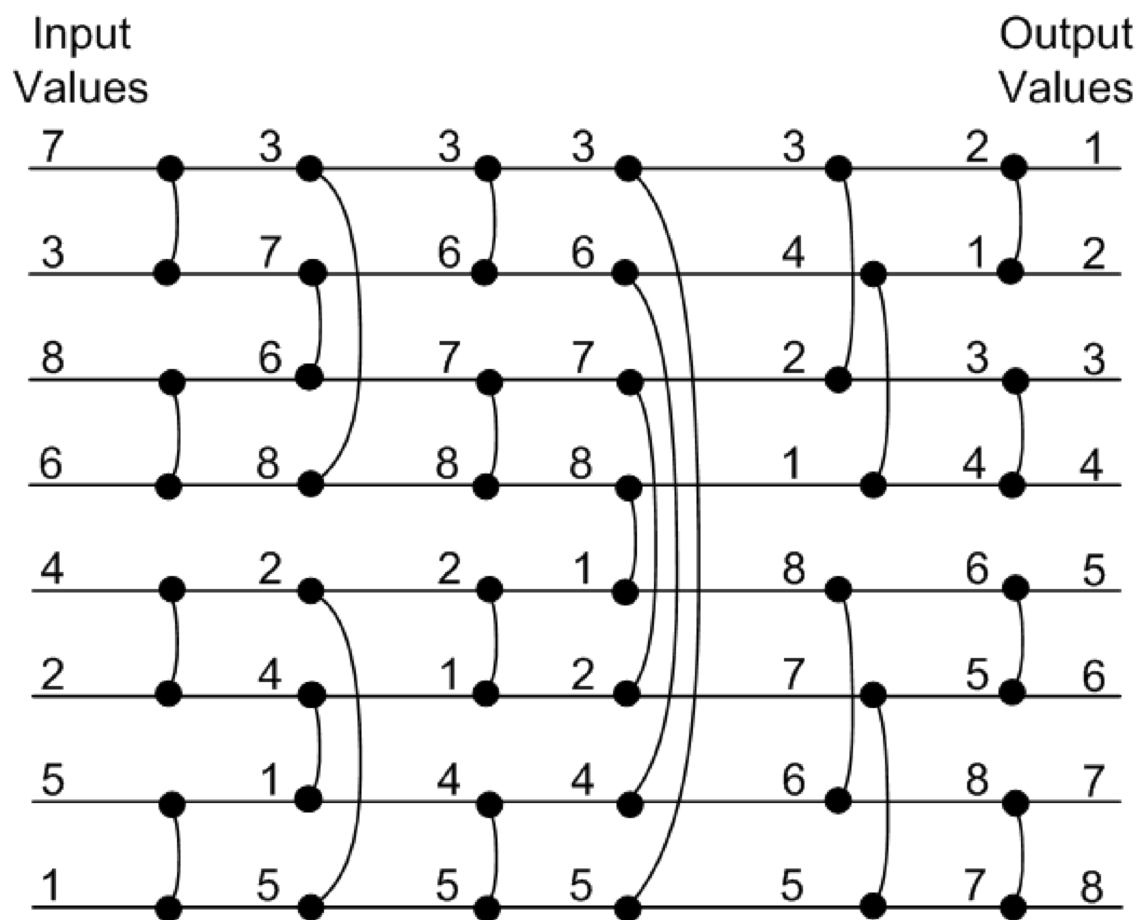

Figure 2. Sorting network example.

Figure 3

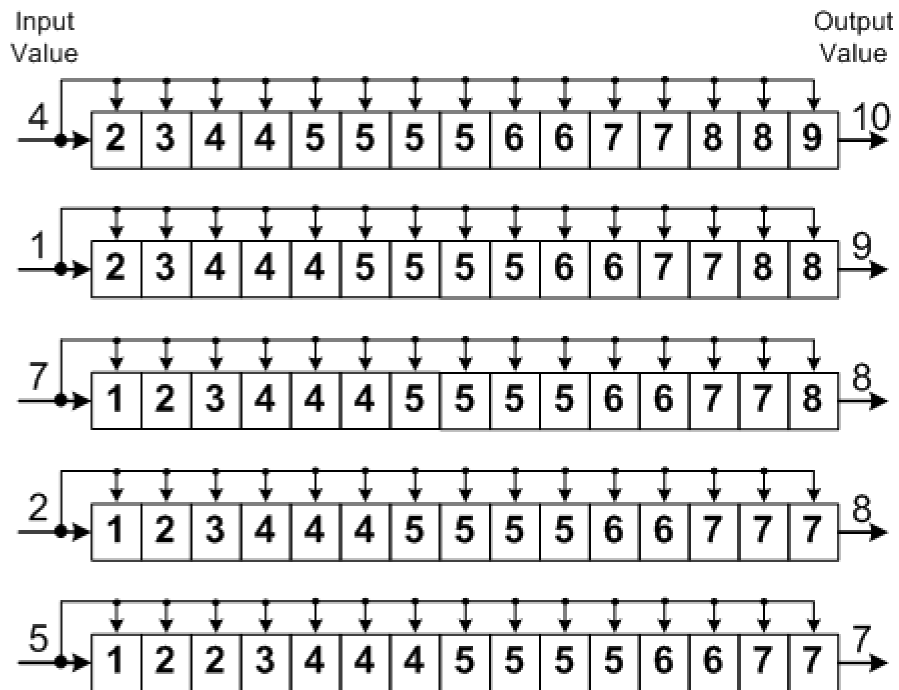

Figure 3. Linear sorter example.

Figure 4

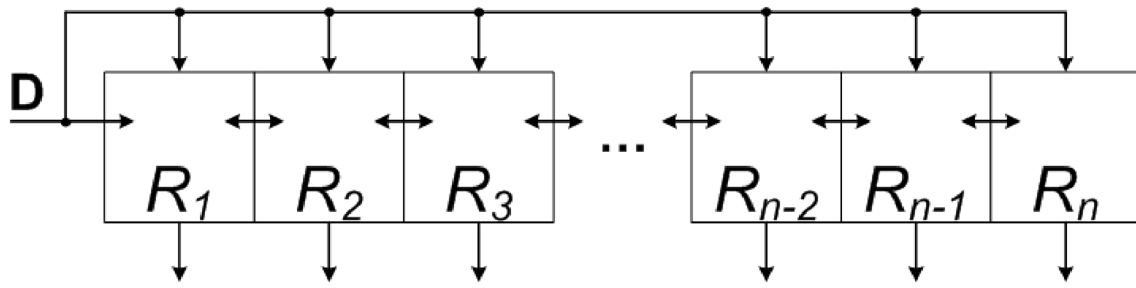

Figure 4. Sorting array.

Figure 5

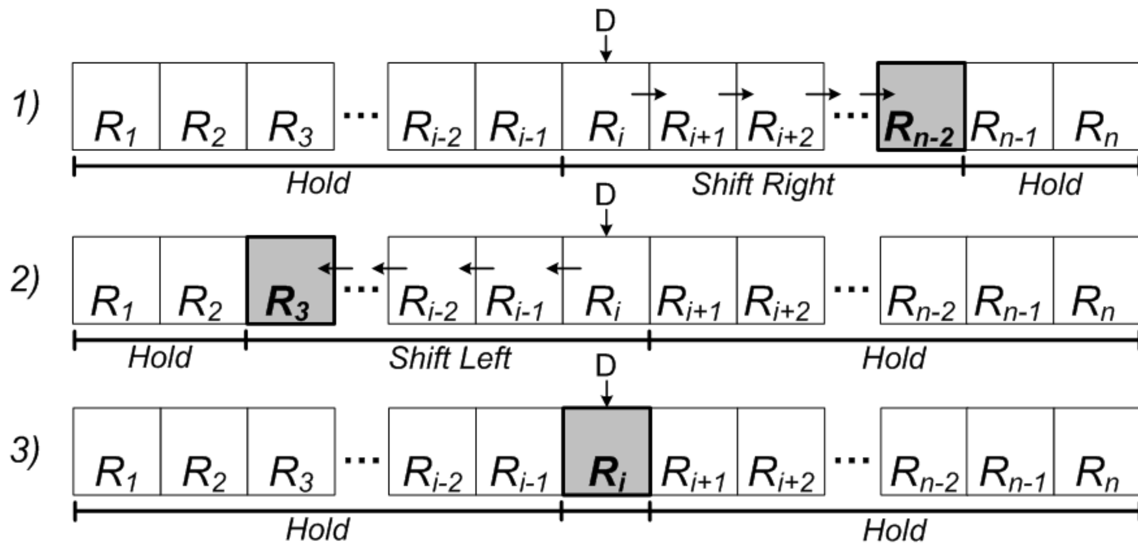

Figure 5. Insertion cases.

Figure 6

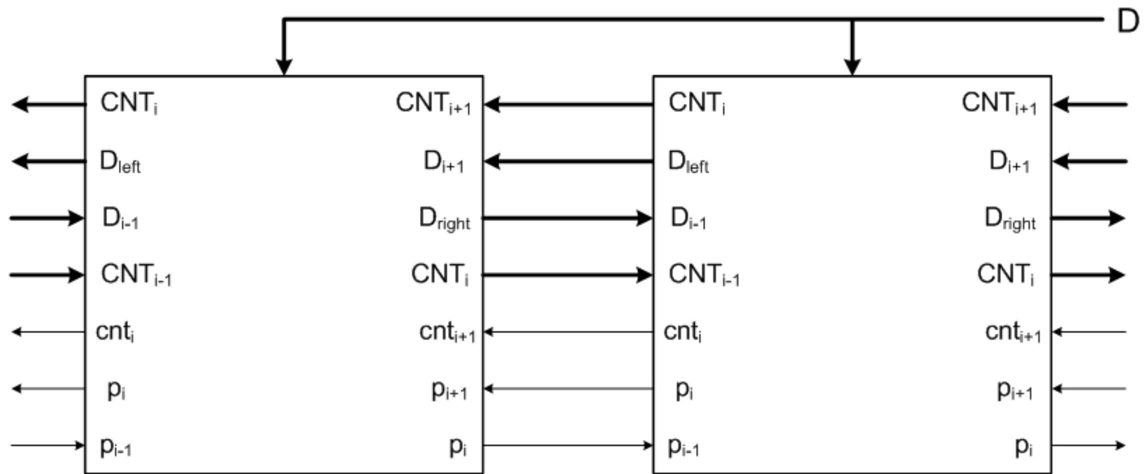

Figure 6. Connection of two SBCs.

Figure 7

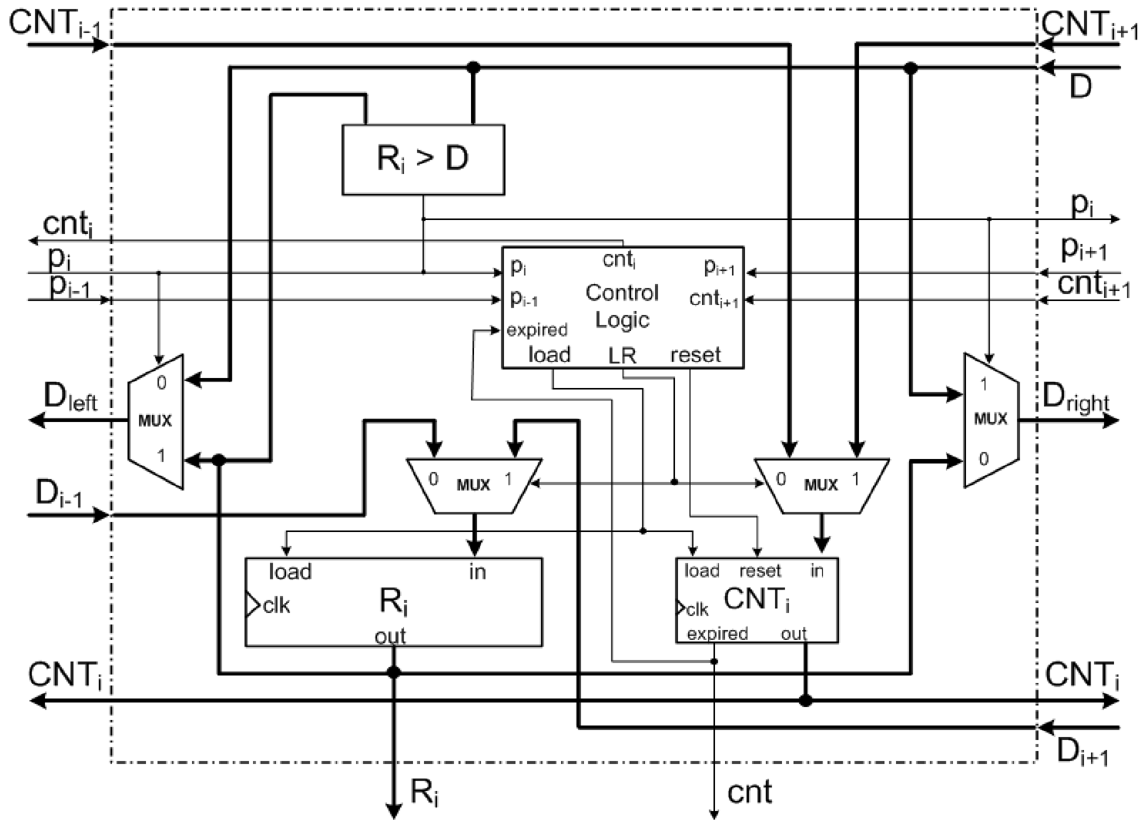

Figure 7. Architecture of the SBC.

Figure 8

Hold | *D* ↓ *Right Shift* → | *Hold*

Sorted Value	**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
expired	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
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)

Hold | ← *Left Shift* *D* ↓ | *Hold*

Sorted Value	**1**	**1**	**2**	**3**	**4**	**8**	**15**	**16**	**17**	**17**	**18**	**20**	**22**
CNT_i	5	9	0	7	8	**12**	11	4	1	6	2	3	10
p_i	1	1	1	1	1	**1**	1	1	1	1	0	0	0
expired	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
load	0	0	0	0	0	**1**	1	1	1	1	0	0	0
LR	0	0	0	0	0	**1**	1	1	1	1	0	0	0
reset	0	0	0	0	0	**0**	0	0	0	1	0	0	0

(b)

Hold | *D* ↓ | *Hold*

Sorted Value	**1**	**1**	**2**	**3**	**4**	**15**	**16**	**17**	**17**	**18**	**18**	**20**	**22**
CNT_i	6	10	1	8	9	**12**	5	2	7	0	3	4	11
p_i	1	1	1	1	1	**0**	0	0	0	0	0	0	0
expired	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
load	0	0	0	0	0	**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	0	0	0	**1**	0	0	0	0	0	0	0

(c)

Figure 8. Sorting array example functionality.

Figure 9

Hold — *D*

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
CNT_i	0	1	2	3	4	5	6	7	8	9	10	11	**12**
p_i	1	1	1	1	1	1	1	1	1	1	1	1	**1**
expired	0	0	0	0	0	0	0	0	0	0	0	0	**1**
cnt_i	1	1	1	1	1	1	1	1	1	1	1	1	**1**
load	0	0	0	0	0	0	0	0	0	0	0	0	**1**
LR	0	0	0	0	0	0	0	0	0	0	0	0	**1**
reset	0	0	0	0	0	0	0	0	0	0	0	0	**1**

(a)

Hold — *D* — *Hold*

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**6**
CNT_i	1	2	3	4	5	6	7	8	9	10	11	**12**	0
p_i	1	1	1	1	1	1	1	1	1	1	1	**1**	0
expired	0	0	0	0	0	0	0	0	0	0	0	**1**	0
cnt_i	1	1	1	1	1	1	1	1	1	1	1	**1**	0
load	0	0	0	0	0	0	0	0	0	0	0	**1**	0
LR	0	0	0	0	0	0	0	0	0	0	0	**1**	0
reset	0	0	0	0	0	0	0	0	0	0	0	**1**	0

(b)

Hold — *Left Shift* — *D* — *Hold*

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**6**
CNT_i	2	3	4	5	6	7	8	9	10	11	**12**	0	1
p_i	1	1	1	1	1	1	1	1	1	1	**1**	1	0
expired	0	0	0	0	0	0	0	0	0	0	**1**	0	0
cnt_i	1	1	1	1	1	1	1	1	1	1	**1**	0	0
load	0	0	0	0	0	0	0	0	0	0	**1**	1	0
LR	0	0	0	0	0	0	0	0	0	0	**1**	1	0
reset	0	0	0	0	0	0	0	0	0	0	**0**	1	0

(c)

D ↓ — Right Shift → — Hold

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**5**	**6**
CNT_i	3	4	5	6	7	8	9	10	11	**12**	1	0	2
p_i	0	0	0	0	0	0	0	0	0	**0**	0	0	0
expired	0	0	0	0	0	0	0	0	0	**1**	0	0	0
cnt_i	1	1	1	1	1	1	1	1	1	**1**	0	0	0
load	1	1	1	1	1	1	1	1	1	**1**	0	0	0
LR	0	0	0	0	0	0	0	0	0	**0**	0	0	0
reset	1	0	0	0	0	0	0	0	0	**0**	0	0	0

(d)

Hold — D ↓ — Hold

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**5**	**6**
CNT_i	0	4	5	6	7	8	9	10	11	**12**	2	1	3
p_i	1	1	1	1	1	1	1	1	1	**1**	0	0	0
expired	0	0	0	0	0	0	0	0	0	**1**	0	0	0
cnt_i	1	1	1	1	1	1	1	1	1	**1**	0	0	0
load	0	0	0	0	0	0	0	0	0	**1**	0	0	0
LR	0	0	0	0	0	0	0	0	0	**1**	0	0	0
reset	0	0	0	0	0	0	0	0	0	**1**	0	0	0

(e)

Hold — ← Left Shift — D ↓ — Hold

Sorted Value	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**3**	**5**	**6**
CNT_i	1	5	6	7	8	9	10	11	**12**	0	3	2	4
p_i	1	1	1	1	1	1	1	1	**1**	1	1	0	0
expired	0	0	0	0	0	0	0	0	**1**	0	0	0	0
cnt_i	1	1	1	1	1	1	1	1	**1**	0	0	0	0
load	0	0	0	0	0	0	0	0	**1**	1	1	0	0
LR	0	0	0	0	0	0	0	0	**1**	1	1	0	0
reset	0	0	0	0	0	0	0	0	**0**	0	1	0	0

(f)

Figure 9. Sorting array example initialization.

Figure 10

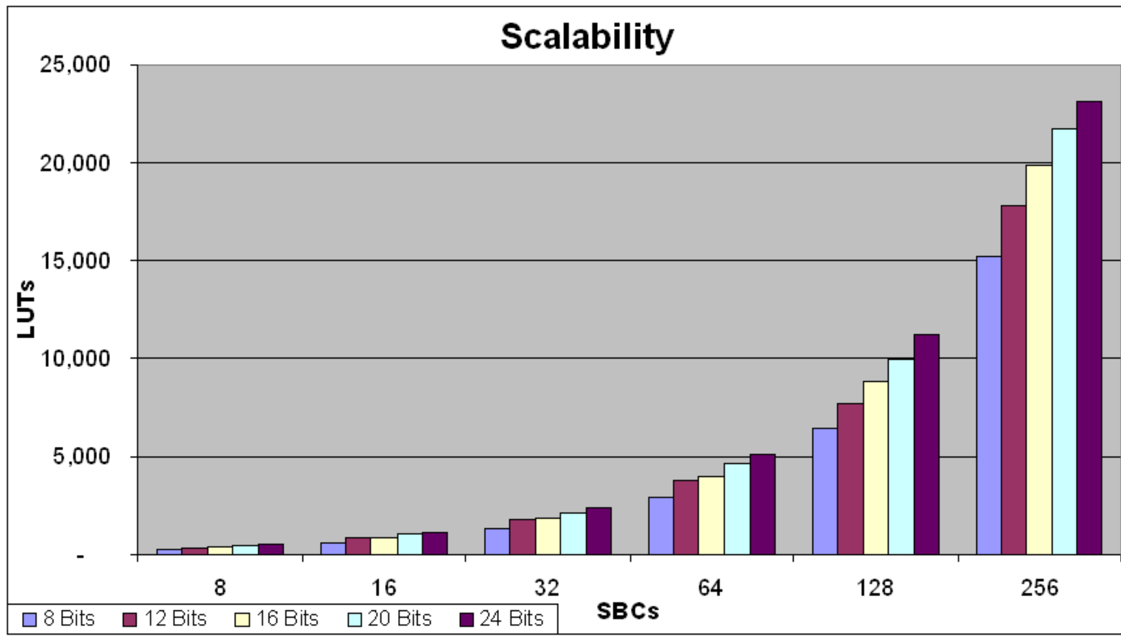

Figure 10. LUTs comparison results.

Table 1

Sorter	FPGA Used	Speed (MHz)	Latency Clock Cycles	Gate Count	Flip Flops	LUT's Count	Data Sorted	Word Size (Bits)
Bitonic	Virtex II	127	14	153k	7,680 *	NA	32	16
Odd-Even	Virtex II	147	14	137k	6,112 *	NA	32	16
Column	Virtex II	66	32	23k	1,024 *	NA	32	16
Shifter	Virtex II	216	32	12k	512 *	NA	32	16
SFS	Virtex E	115	49	35k	1,372	3,430	7x7	8
PRC	Virtex E	159	2	384k	1,634	11,809	7x7	8
SRC	Virtex E	96	49	19k	784	2,548	7x7	8
FIFO Scheme	**Virtex E**	**72**	**49**	**15k**	**325**	**1,895**	**49**	**8**
FIFO Scheme	**Virtex II**	**126**	**32**	**25k**	**672**	**2,726**	**32**	**16**
FIFO Scheme	**Virtex 5**	**234**	**32**	**24k**	**672**	**1,894**	**32**	**16**

Table 1. Performance results with other sorting architectures.

Table 2

Number of	Sorter					
	Bitonic	**Odd-Even**	**Column**	**Linear Shifter**	**SFS**	**FIFO Scheme**
Multiplexers	$n(\log^2 n + \log n)/2$	$n(\log^2 n - \log n + 4)/2-2$	$2n$	n	$5n+5$	$\mathbf{4n-2}$
Comparators	$n(\log^2 n + \log n)/4$	$n(\log^2 n - \log n + 4)/4-1$	n	n	$2n+2$	$\mathbf{2n}$
Registers	$n(\log^2 n + \log n)/2$	$n(\log^2 n - \log n + 4)/2-2$	$2n$	n	$4n+4$	$\mathbf{2n}$
Counters	0	0	0	0	$n+1$	$\mathbf{n}$
Clock Cycles	$(\log^2 n + \log n)/2$	$(\log^2 n + \log n)/2$	$4n$	n	n	$\mathbf{n}$

Table 2. Comparison with others sorting architectures.

Table 3

SBC Amount	8 BITS			12 BITS			16 BITS		
	FF	LUT	MHz	FF	LUT	MHz	FF	LUT	MHz
8	88	232	291	120	327	318	152	377	289
16	192	568	264	256	850	266	320	840	237
32	416	1,306	219	544	1,806	228	672	1,894	231
64	896	2,931	198	1,152	3,815	193	1,408	3,981	206
128	1,920	6,422	171	2,432	7,683	172	2,944	8,826	173
256	4,096	15,196	158	5,120	17,747	151	6,144	19,862	152

Table 3a. Scalability results of the sorting architecture.

SBC Amount	20 BITS			24 BITS		
	FF	LUT	MHz	FF	LUT	MHz
8	184	435	277	216	500	264
16	384	1,014	255	448	1,138	264
32	800	2,114	217	928	2,357	233
64	1,664	4,603	196	1,920	5,083	203
128	3,456	9,899	172	3,968	11,189	172
256	7,168	21,701	151	8,192	23,170	150

Table 3b. Scalability results of the sorting architecture.