International Journal of Engineering Research and Technology. ISSN 0974-3154, Volume 12, Number 6 (2019), pp. 814-819
© International Research Publication House. http://www.irphouse.com

The VLSI Design of a High-Speed Sorting Engine By Using Demultiplexer Array

Myungchul Yoon

Dept. of Electtronics and Electrical Engineering, Dankook University: Yongin, Gyeonggi-do, Republic of Korea.
ORCID: 0000-0001-7952-4349

Abstract

A novel sorting engine called DAS (Demultiplexer Array Sorting engine) is presented in this paper. The algorithm used in DAS is based on radix-2 sort algorithm combined with counting sort algorithm. DAS implements the algorithm using an array of 1-to-2 deMUX. DAS uses $N \times N$ deMUX array to sort N of k-bit data, and completes sorting operation in k cycles regardless of N. Although sorting is finished in a fixed number of cycles, the clock cycle time is proportional to N, so that time complexity of DAS is $O(Nk)$. Although DAS has $O(N^2)$ hardware complexity due to the array of deMUX, the simple deMUX circuit makes DAS hardware-efficient implementation. DAS uses minimal memory because it does not move any data during the sorting operation, but regenerates input data to memory in sorted order. Therefore, only the memory to store the final result is required. The speed of DAS is evaluated by simulations. The worst case cycle time is 20.8 ns for sorting 100 data, and it linearly increases with N. For sorting 1000 of 32-bit data, about 200ns cycle time is required so that the sorting is completed in 6.4 μsec.

Keywords: Sorting, Sorting Engine, Sorting Network, Non-comparison Sorting, Sorting Architecture.

I. INTRODUCTION

Sorting is a widely used essential operation in computer-related applications. Therefore, sorting has been extensively studied in both sequential [3] and parallel [4][5] models of operation, and many software-based and hardware-based solutions have been developed. Most of solutions are a comparison-based solution which compares two data and swap their positions. It is proved that comparison-based sequential sorting algorithm can achieve up to $O(N \log N)$ time-complexity to sort N of k-bit numbers [6]. Some sorting algorithms such as counting sort, radix sort, bucket sort, etc. do not compare data. The non-comparison sorting can achieve up to $O(N)$ time-complexity [6]. Although there are many sorting algorithms reported in the literature, their performance is discussed based on software implementations. Many sorting algorithms are inefficient for hardware implementation so that the number of hardware implementations of sorting algorithms is limited.

Because sorting is a computationally intensive and expensive operation, finding an efficient hardware sorting engine has been a significant challenge to overcome the computational bottleneck of the sorting problems. Hardware implementations also can be divided into comparison-based implementations and non-comparison implementations. Most of comparison-based hardware implementations use hardware parallelism, which requires many k-bit comparators and secondary storages to swap data. Although this type of implementations can achieve $O(N\log N)$ time complexity through parallelism, it is hard to use them for a large number of data due to the large hardware overheads. Non-comparison hardware implementations, however, do not compare data one another, but use a special algorithm to sort. Most of them use bit-serial approach for bit-unit operations so that their hardware burden is much lighter than that of comparison-based implementations which perform word-unit operations.

A new non-comparison sorting engine is presented in this paper. The new engine sorts N data in k cycle regardless of N using $N \times N$ demultiplexer (deMUX) array. Although the number of clock cycle is fixed, the clock cycle time is proportional to N, so that its time complexity is $O(Nk)$. The new engine neither compares nor swaps input data. Therefore, only $N \times k$ bit memory is required to store sorted data, and no other memory for swapping is necessary. Instead of moving input data, the sorted data are regenerated and inserted directly to the memory during the sorting process.

The related sorting engines are reviewed in Section 2. The algorithm and implementation of the new sorting engine is described in Section 3. Performance evaluation of new engine is presented in Section 4, and Section 5 concludes this paper.

II. PREVIOUS WORKS .

Most of hardware sorting implementations use parallelism for high speed operation. Sorting networks are a well-studied class of parallel sorting devices. Sorting network is a regularly connected network of processing elements which compare the magnitude of data and exchange their order. Batcher's bitonic sorting network and odd-even merge sorting network [7][8] have $O(\log^2 N)$ time complexity, while AKS sorting network [9] has $O(N\log N)$ time complexity. Leighton [10] developed hardware-efficient sorting network with $O(\log N)$ time

complexity. Besides, a number of sorting networks have been developed.

Comparison based hardware solution such as sorting networks requires many comparators and extra storages for swapping, which is limited in scalability for sorting a large number of data. Some hardware sorting solutions do not compare any data to one another. Most of them use bit-level parallelism while the majority of comparison-based implementations use word-level parallelism. Tugba et al. [1] proposed a sorting engine implemented by rank order filter and multi-input majority voters. Their engine completes sorting in N+k-l cycles. Srikanth et al. [2] presented a way to sort binary numbers by making a special N×N matrix called rank matrix. The final rank matrix contains information on the positions of N input in sorted sequence. Their sorting engine complete the rank in N+2k cycles. Even though they have $O(N)$ time complexity, the constant hidden in $O(.)$ notation is very small because of the bit-level operation. Because of bit-unit operations, their processing elements are very small and fast which is suitable for building a scalable architecture to large number of data.

III. THE DESIGN OF SORTING ENGINE USING DEMULTIPLEXER ARRAY

A non-comparison type sorting engine is presented in this paper. The new sorting engine is implemented by using 1-to-2 demultiplexer (deMUX) array so that it is named as DAS (DeMUX Array Sorting engine). The algorithm and VLSI implementation of DAS are as follows.

III.I THE SORT ALGORITHM FOR DAS

The sorting algorithm used in DAS is basically radix-2 sort algorithm combined with counting sort algorithm. That is, input data are sorted digit (bit) by digit from MSB (most significant bit) to LSB (least significant bit), and each digit is sorted according to counting sort algorithm. One of the characteristics of DAS is that DAS regenerates input data in sorted order rather than exchanges or moves them to other memory locations. Therefore, DAS need only N×k bit memory in sorting N of k-bit binary data.

The algorithm of DAS for sorting N data in descending order is as follows. For the sorting in ascending order, just switch 0 and 1 each other

Sorting Algorithm

- To sort N of k-bit binary data, prepare N×k bit memory, M, to store the sorted results. The N rows of M will be filled with sorted N data after k cycles. N inputs and N rows of M are considered as a group. At the beginning of sorting, only one group exists.
- In the first cycle following operations are performed
  - count the number of 1s in the MSB out of N input data (assume that the result is p).
  - insert 1 to MSBs of the first p rows out of N rows, and insert 0 to MSBs of the rest N-p rows.
  - divide N inputs and N rows into two independent groups, where one group includes the first p-rows out of N rows and p inputs of which MSB is 1, and the other group contains the rest N-p rows and N-p inputs.
- At the end of the first cycle, two subgroups possibly exist. Likewise, maximum 2^i subgroups can exist at the end of i-th cycle. From the second cycle, the same operations as in the first cycle are applied to each subgroup independently and simultaneously.
- Generally, the following procedure is performed in each cycle to every subgroup.

  For a subgroup with q inputs and q rows
  - In the i-th cycle ($1 \le i \le k$), count the number of 1s in i-th MSB out of q input data (assuming that the result is r).
  - insert 1 to i-th MSBs of the first r rows out of q rows and insert 0 to i-th MSBs of the rest q-r rows.
  - divide q inputs and q rows into two independent groups, one group includes the first r-rows out of q rows and r inputs of which i-th MSB is 1, and the other group contains the rest q-r rows and q-r inputs.
- After k cycles, the memory M is filled with sorted data in descending order.

Note that the sorted data is gradually generated in M cycle by cycle so that there is no extra memory to swap the data.

III.II IMPLEMENTATION OF THE DEMUX ARRAY SORTING ENGINE

The block diagram of the newly designed sorting engine with demultiplexer array (DAS) is shown in Fig. 1. DAS consists of three major parts: sorting-cell array, zone control logic, and

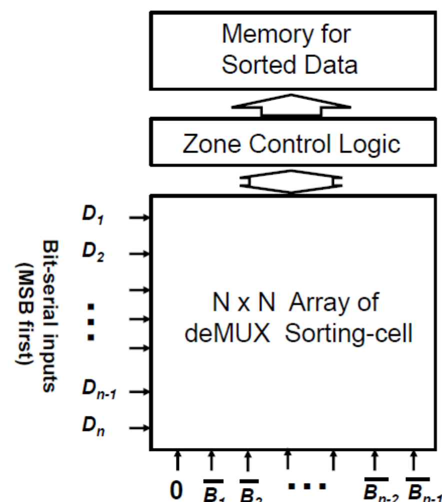


Fig. 1. Block diagram of the demultiplexer array sorting engine

International Journal of Engineering Research and Technology. ISSN 0974-3154, Volume 12, Number 6 (2019), pp. 814-819
© International Research Publication House. http://www.irphouse.com

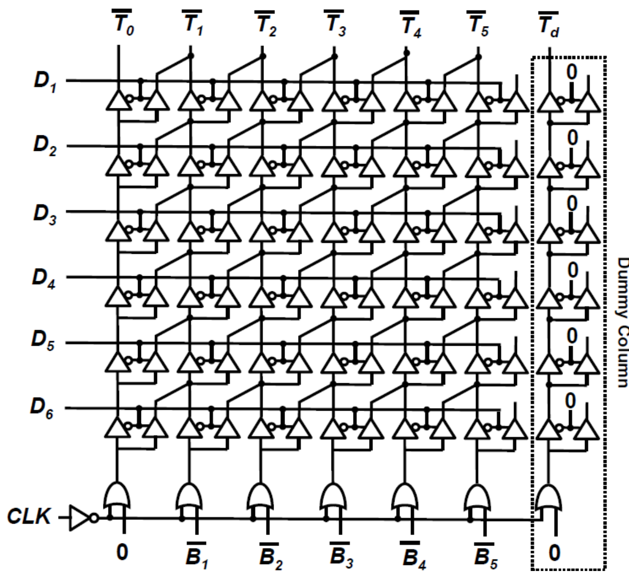


Fig. 2. A simplified circuit of 6×6 deMUX array

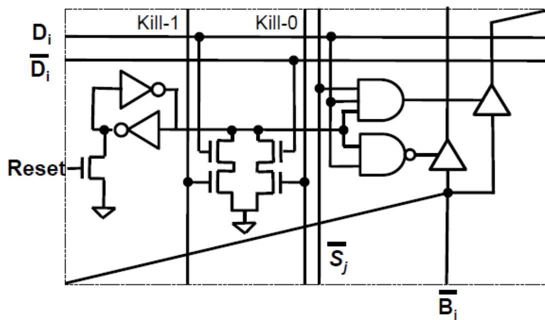


Fig. 3. The logic circuit of the sorting-cell (controllable deMUX)

memory. The sorting-cell array is used to count the number of 1s in a given digit of N inputs. With the results of the sorting-cell array, the zone control logic (ZCL) divides the array into 1-zone and 0-zone, and deactivates some selected deMUX cells in the array. The memory is made by N of k-bit shift registers. In each cycle, 1 is fed to the shift registers in the 1-zone, and 0 is fed to those in the 0-zone.

For sorting N binary inputs, the sorting-cell array is made by N×N controllable 1-to-2 deMUXs. A simplified deMUX array is drawn in Fig. 2 to explain its operation. In each cycle, one bit per input (from MSB to LSB) is fed to the array as in Fig. 2. The purpose of the deMUX array is to count the number of 1s in these inputs. The input bit D_i is applied to all deMUXs in the i-th row. The $\overline{B_i}$ represents a starting value. At the beginning of sorting, all $\overline{B_i}$ s are set to 1 except the leftmost one ($\overline{B_0}$). The leftmost one ($\overline{B_0}$) has 0 value permanently. In each cycle, the **0**s in the bottom starts its travel to the top. A deMUX cell passes its input to vertical-upward if D_i=0 while it does right-upward when D_i=1. Therefore, the **0** at the starting

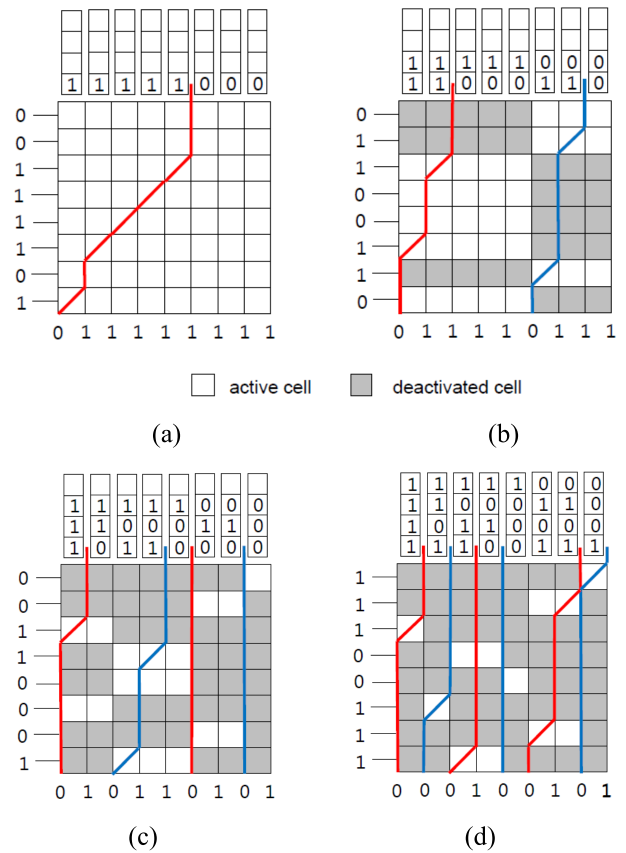


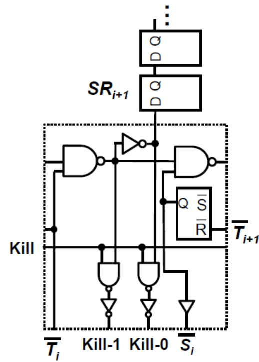
Fig. 4. Example of the DAS operation in sorting 8 values (1, 5, 15, 10, 8, 13, 5, 11) by descending order (a) 1st cycle, (b) 2nd cycle, (c) 3rd cycle, (d) 4-th cycle

positions is moved one column to right whenever it meets a row with D_i=0.

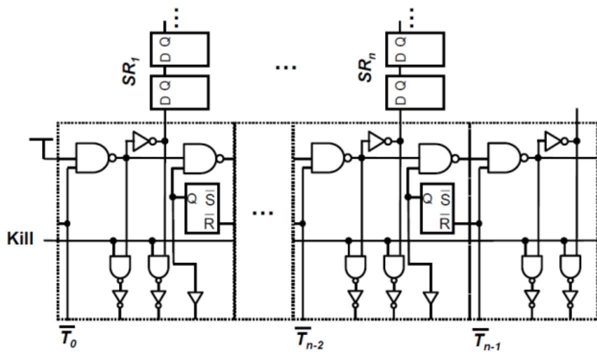
By this ways, the **0** at $\overline{B_0}$ reaches to $\overline{T_m}$ when the number of 1s in the D_is is m, regardless of locations of 1s. A dummy column is added to the N×N cell array as in Fig. 2. The column is used as a timing simulator such that the dummy $\overline{T_d}$ signal becomes 0 at the same time with $\overline{T_m}$.

Fig. 3 shows the circuit of actual sorting-cell. The sorting-cell is devised by modifying basic deMUX circuit such that the deMUX function can be deactivated by some control signals. If a cell is deactivated, $\overline{B_i}$ is passed to vertical-upward regardless of the value D_i. All cells are activated by reset signal at the beginning a sorting operation. The cells can be deactivated by either Kill-1 or Kill-0 signal. The Kill-1 (Kill-0) signal is used to deactivate the cells with D_i =1 (D_i =0). Once a cell is deactivated, it cannot be reactivated until the sorting is finished. The signal $\overline{S}$ is used to divide an array into two subarrays. If $\overline{S_j}$ becomes 0, the j-th column is separated from (j+1)-th column so that the input cannot propagates to the right of j-th column.

Let us explain the operations of DAS by using a simple sorting example. Assume that there are eight 4-bit binary numbers, (D_1,



(a)



(b)

Fig. 5. The zone control logic (ZCL) and memory (a) The circuit of a ZCL cell with a *k*-bit shift register (b) the structure of ZCL and memory

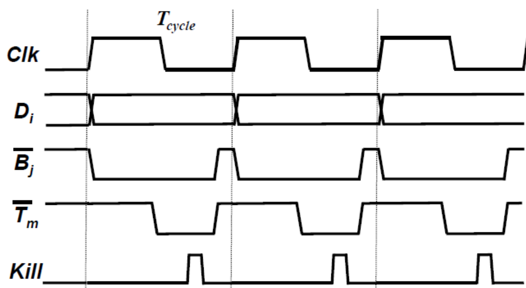


Fig. 6. The timing diagram of major signals

D_2, ... D_8) = (1, 5, 15, 10, 8, 13, 5, 11), which need to be sorted in descending order. Fig. 4 shows how the eight numbers are sorted by using the sorting-cell array. Since the data are 4-bit values, DAS sorts them in 4 cycles. The processes of DAS in each cycle are depicted through Fig. 4-(a) ~ 4-(d).

Before the first cycle, all cells are activated by “Reset” signal, and all $\overline{B_i}$ s except $\overline{B_0}$ are set to 1. At the first cycle, the MSBs of eight inputs are fed to the array as in Fig. 4-(a) and the only **0** at the left-bottom cell propagates through the array. Since there are five 1s out of 8 inputs, the **0** terminates at $\overline{T_5}$ as in Fig. 4-(a). The $\overline{T_5}$ signal makes zone control logic (ZCL) to divide the array into two zones (groups), Z_{1L} and Z_{1R}. Z_{1L} is called left-zone (or 1-zone) in cycle 1 and Z_{1R} is right-zone (or 0-zone) in cycle 1. The separation is performed by making $\overline{S_5}$ into 0. All columns of the array and the shift-registers in the left of $\overline{T_5}$ belong to Z_{1L}, while those in the right of $\overline{T_5}$ belong to Z_{1R}. The inputs for all shift-registers in Z_{1L} are set as 1 by ZCL while those in Z_{1R} are set as 0 as in Fig. 4-(a). In addition, ZCL activates Kill-0 signal over 1-zone while it triggers Kill-1 signal over 0-zone to selectively deactivate useless cells in the rest of cycles. The cells deactivated in cycle 1 are shown in Fig. 4-(b).

In the second cycle, $\overline{B_5}$ becomes 0 by $\overline{S_5}$, hence, the leftmost inputs of two subarrays are become **0**. As in Fig. 4-(b), the **0**s at $\overline{B_0}$ and $\overline{B_5}$ terminates at $\overline{T_2}$ and $\overline{T_7}$ respectively. Similarly in the cycle 1, Z_{1L} and Z_{1R} are bisected by $\overline{T_2}$ and $\overline{T_7}$ respectively, and the left pieces are grouped as Z_{2L} and the right pieces as Z_{2R}. The shift-registers in Z_{2L} are shifted with input 1, and those in Z_{2R} are done with input 0. Then, Kill-0 is generated over Z_{2L}, and Kill1-1 over Z_{2R} which result in increasing deactivated cells as in Fig. 4-(c). The same processes are repeated in the cycle 3 and the cycle 4. At the end of the cycle 4, the registers are filled with sorted inputs as in Fig. 4-(d).

The structure of ZCL is a cascade of N ZCL-cells such as in Fig. 5. The role of ZCL is to divide the deMUX array and the memory (shift-registers) into subgroups, and to manipulate each subgroup independently. The boundary of a subgroup is stored in the $\overline{S}\overline{R}$ -latch of ZCL-cell. All latches of ZCL are set to 1 only at the beginning of sorting operation. In each cycle, the latch in a ZCL-cell is reset to 0 when its $\overline{T_i}$ becomes 0. A subgroup is defined as the columns of cell-array and memory from a 0-latches to the next 0-latch. In each cycle, a newly developed $\overline{T_i} = 0$ signal, distinguishes a subgroup into left-zone (1-zone) and right-zone (0-zone). If any $\overline{T_i}$ in a subgroup does not become 0 until $\overline{T_d}$ of the dummy column become 0, ZCL regards all D_i's in the subgroup are 1 by default, so that the subgroup is included in 1-zone. After 1-zone and 0-zone is identified, the Kill pulse is generated to trigger Kill-0 over 1-zone and Kill-1 over 0-zone. The Kill signals deactivates sorting-cells selectively to exclude unnecessary inputs in the rest of cycles. Fig. 6. shows the timing diagram of major signals.

DAS has $O(N^2)$ hardware complexity by sorting-cell array. Despite its $O(N^2)$ complexity, DAS is very hardware-efficient because the circuit of the cell is very simple as in Fig. 3. Furthermore, the memory size required for DAS is just the size needed to store the sorted data.

Regardless of the number of data, DAS completes sorting operation in k cycles. If T_{clk} is the cycle time of clock, DAS requires kT_{clk} in sorting. However, the clock cycle time T_{clk} is proportional to N, because the propagation time of **0** from the bottom to the top of the array is proportional to N. Therefore, the time complexity of DAS becomes $O(Nk)$. The worst case delay of DAS happens in the first cycle, and when MSBs of all inputs are 0. In the worst case, $\overline{B_0}$ should pass N deMUX cells and then, the signal triggered by $\overline{T_0}$ should pass N ZCL-cells. After the array is divided into subarrays, the propagation

Table 1. The worst case clock-cycle time for sroting N data

N	100	200	400	600	800	1000
T_{cycle} (ns)	20.8	40.3	79.4	118.4	157.5	196.4

lengths through ZCL-cells are shortened so that the cycle time could be decreased.

IV. EXPERIMENTS

The speed of DAS is evaluated through simulation. The simulation is performed by HSPICE with IBM's "1.2V-0.13μm 8RF-LM" model parameter.

Since sorting is completed in k-cycles, regardless of the number of data, the clock cycle time determines the speed of DAS. The worst case clock cycle time T_{clk} is obtained by critical path simulation for a various number of data. The results are shown in Table 1. The worst case cycle time for sorting 100 data is about 20.8 ns, and it increases linearly with N. For sorting 1000 of 32-bit data, T_{clk} is about 200ns (5 MHz) and therefore, the sorting is completed in 6.4 μsec. Because DAS needs to change clock cycles according to the number of data for fast operation, generally, a separate internal clock dedicated to DAS is required.

The speed of DAS is compared to other sorting engines. Among many previous sorting engines, two sorting engines in [1] and [2] are selected for comparison, since these engines use a non-comparison algorithm and bit-serial operation like DAS. The comparison is summarized in Table 2. The major difference between DAS and the other implementations is that DAS has long clock cycle time (T_{clk}) but completes sorting in the smallest number of cycles, while the others have short T_{clk} but require many clock cycles. While T_{clk} is proportional to N in DAS, it is slightly increases in [1] and [2]. Contrary, the number of cycle to complete sorting is fixed in DAS but it is proportional to N in the other engines.

Because the three implementations use different data size in speed simulation, the sorting time per unit data (T_D/N) is used to compare their speeds. As illustrated in Table 2, DAS has the shortest T_D/N. Note that T_D in [2] is not the time to get sorted data but the time to complete rank matrix. Although the rank matrix holds enough information to obtain sorted data, it is not clear in the literature how much time is required to decode the matrix and to get sorted data.

The simplicity of the sorting cell makes DAS a hardware-efficient implementation. The simulation results shows that it is faster than other existing sorting engines, and it can complete sorting 1000 of 32-bit data in 6.4 μsec.

V. CONCLUSION

The architecture and implementation of a new sorting engine called DAS are presented in this paper. DAS uses well-known radix-2 sort algorithm combined with counting sort algorithm. The algorithm is implemented using $N \times N$ deMUX array.

Table 2. Comparison of DAS to other sorting engines

	DAS	engine in [1]	engine in [2]
Time complexity	$O(Nk)$	$O(N+k)$	$O(N+2k)$
Technology	130 nm	350 nm	65 nm
Data size (N)	100	63	16
Data width (k)	16	16	16
No. of cycles	16 (k)	78 ($N+k-1$)	48 ($N+2k$)
T_{cycle}	20.8 ns	5 ns	1.6 ns
Total delay (T_D)	332.8 ns	390 ns	76.8 ns
T_D / N	3.33 ns	6.19 ns	4.8 ns
Final result	sorted data	sorted data	rank matrix

DAS sorts N data in k cycles regardless of N, but the clock cycle time is proportional to N. The cycle time is 20.8ns for sorting 100 data, and increases to about 200ns for sorting 1000 data. Therefore, 1000 of 32-bit data can be sorted in 6.4 μsec.

DAS has $O(N^2)$ hardware complexity. Although it requires N^2 sorting-cells, the simplicity of sorting-cell reduces hardware burdens. The memory size required for DAS is just the size needed to store the sorted data. With the simple sorting-cell structure, its optimal memory requirement makes DAS a scalable architecture.

REFERENCES

[1] T. Demirci, I. Hatirnaz, and Y. Leblebici, 2003, "Full-custom CMOS realization of a high-performance binary sorting engine with linear area-time complexity," in *International Symposium on Circuits and Signals*, pp. 453-456.

[2] S. Alaparthi, K. Gulati, and S. P. Khatri, 2009, "Sorting binary numbers in hardware – A novel algorithm and its implementation," in *International Symposium on Circuits and Systems*, pp. 2225-2228.

[3] D. E. Knuth, 1973, *The Art of Computer Programming,* vol. 3: Sorting and Searching, Addison-Wesley.

[4] R. S. Francis, and I. D. Mathieson, 1988, "Benchmark parallel sort for shared memory multiprocessors," *IEEE Tans. on Computers,* vol. C-37, pp. 1619-1626.

[5] S. G. Akl, 1985, Parallel Sorting Algorithms. New York: Academic Press.

[6] Wikipedia, https://en.wikipedia.org/wiki/Sorting

[7] K. E. Batcher, 1968, "Sorting Networks and Their Applications," *Proc. AFIPS Conf.*, pp. 307-314.

[8] K. E. Batcher, 1990, "On Bitonic Sorting Networks," *Proc. Int'l Conf. Parallel Processing*, pp. 376-378.

© International Research Publication House. http://www.irphouse.com

[9] M. Ajtai, J. Komlos, and E. Szemeredi, 1983, "Sorting in c log n Parallel Steps," *Combinatorica*, vol. 3, pp. 1-19.

[10] F. T. Leighton, 1985, "Tight Bounds on the Complexity of Parallel Sorting," IEEE Trans. Computers, vol. 34, pp. 344-354.