Worst Case $O(N)$ Comparison-Free Hardware Sorting Engine

Sanchita Saha Ray, Dulal Adak, and Surajeet Ghosh, *Member, IEEE*

***Abstract*—This article proposes a novel comparison-free hardware sorting engine that sorts N unique n-bit elements (irrespective of signed and unsigned) consuming linear sorting latency of $O(N)$ clock cycles. It can even efficiently sort N data elements with a nonzero duplicity rate in less than $O(N)$ clock cycles. This sorting engine is designed using n-symmetric cascaded blocks utilizing few fundamental logic components. The entire design is synthesized for several data sets from pseudo-randomly generated data elements to unique elements, and also from random to completely sorted elements with various duplicity rates. The architecture appears impartial with respect to ordering of elements. Synthesis results indicate that the proposed approach consumes reasonably lower field programmable gate array resources than existing approaches. The architecture takes per-element sorting latency in sorting 512 unique signed elements as 22.56 ns (48 bit) and takes 26.80 ns (64 bit) to sort 256 unique signed elements. The engine achieves sorting throughput rates as ≈117-to-142 Million-Elements-per-second (MEps) (16 bit), 79-to-97 MEps (24 bit) for sorting 256-to-1K, whereas 66-to-73 MEps (32 bit) and 44-to-49 MEps (48 bit) for sorting 256-to-512 elements. However, it is 37 MEps (64 bit) in sorting 256 signed elements. This architecture consumes ≈1.52 μW for the unique signed numbers (SNs) as per-byte processing power and ≈1.55 μW for the SNs with nonzero duplicity rates.**

***Index Terms*—Comparison-free sorter, duplicate element detector, duplicate numbers sorter, hardware sorting engine, largest element detector (LED), $O(N)$ sorting engine, signed numbers (SNs) sorter, worst case $O(N)$ sorter.**

I. Introduction

SORTING is one of the prime functions and prerequisite in most of the data-centric applications those involve big data analysis, such as image processing [2], [3], video processing [4], database query operations [5], [6], processor scheduling [7], ATM switching [8], multiple-input–multiple-output (MIMO) K-best detector in wireless local area network [9], [10], etc. Due to the remarkable impact of sorting operations, computation intensive tasks are handled by highly parallel multicore processing systems for big data processing. However, scalability of multiple CPUs is restricted due to large communication delay, power consumption, and cost. Numerous sorting algorithms have been evolved in last few decades, those are mainly graphics processing unit (GPU) based [11] and multicore architecture based [12]. These GPU-based or multicore-based sorting designs consume large volume of hardware and software resources. Additionally, in hard real-time systems, worst case execution time (WCET: longest execution time to accomplish a specific task) involved in sorting operations is an important concern to enforce safety in critical real-time systems [13]. The WCET is critical in terms of both computational effects and on the accomplishment of operations meeting its deadline. Therefore, sorting engine with predictable WCET is an attractive feature for hard real-time systems.

Owing to many advantages of hardware sorters over software-based approaches, hardware sorting architectures have been an area of interest for many researchers and computer scientists. Though, the software oriented designs are not practical solutions for real-time sorting operations, but implementation on application-specific integrated circuit (ASIC) [8] or on field programmable gate array (FPGA) [4], [7], [14], [15] may improve the system performance. Most of these designs are mainly based on compare-and-swap (CAS) operations. To improve the performance of sorting architectures, parallel data-independent CAS blocks are devised to form a sorting network. In some applications, viz., MIMO in the wireless network uses an M-to-N max-set-selection partial sorting technique [16]–[19]. On the other hand, to improve the performance of sorting architectures, merge sort-based designs gain more attraction [14], [15], [20], [21]. However, all these sorting designs utilize CAS units for comparison operations, which incur large amount of processing time and hardware resources.

To the best of our knowledge, only a few comparison-free sorting approaches have been proposed so far, [1], [22], and [23]. The schemes presented in [22] and [23] use complex matrix-mapping operations based on the one-hot weight method using a number of matrices to carry out actual sorting operations. In contrast, an $O(N)$ sorting engine is found in [1] that can perform sorting operations consuming lower resources.

In summary, most of the prior works on hardware sorting designs require large hardware resources (in terms of logic gates, registers, memory elements, pipelined stages, etc.)

Manuscript received 27 July 2021; revised 2 October 2021; accepted 10 November 2021. Date of publication 30 November 2021; date of current version 20 September 2022. This work was supported by the Department of Science & Technology and Bio-technology, Government of West Bengal, India, under Grant 149(Sanc.)/ST/P/S&T/6G-16/2018. A part of this article is presented in IEEE Computer Society Annual Symposium on VLSI (ISVLSI) 2019 [1]. This article was recommended by Associate Editor Z. Shao. *(Corresponding author: Surajeet Ghosh.)*

Sanchita Saha Ray is with the Department of Information Technology, St. Thomas' College of Engineering and Technology, Kolkata 700023, India (e-mail: saharay.sanchita@gmail.com).

Dulal Adak and Surajeet Ghosh are with the Department of Computer Science and Technology, Indian Institute of Engineering Science and Technology, Shibpur, Howrah 711103, India (e-mail: adak.dulal.92@gmail.com; surajeetghosh@ieee.org).

Digital Object Identifier 10.1109/TCAD.2021.3131554

and require complex operations (with respect to CAS operations, matrix manipulation operations, etc.) and are based on some traditional sorting algorithms. Additionally, most of these approaches require a large amount of preprocessing time as a part of prerequisition for splitting the input data list into a number of partitions. Moreover, fundamental requirements of a real-time sorter are to handle both unsigned and signed numbers (SNs) as well as mixtures of any of them with a nonzero duplicity rate in an efficient manner (i.e., achieving maximum desired results by utilizing minimum hardware resources). Furthermore, the system is expected to be efficient in handling duplicate data entries and should also be able to meet the requirements of a high throughput sorter (in terms of Gbps with respect to the number of sorted data yield per unit time). Additionally, when implemented on hardware, resources should be utilized optimally. To address these challenges, a novel hardware-based comparison-free sorting engine has been proposed in this article.

The key contributions of this article are summarized as follows.

1) This is the kind of sorting architecture that does not use any of the existing or improved variants of traditional sorting algorithms.
2) Exhibits comparison-free sorting mechanism in void of any variants of comparators, complex circuitry, or any complex algorithm (e.g., matrix manipulation).
3) Efficiently handles unsigned and SNs as well as mixtures of any of them with duplicity rates.
4) The architecture is impartial to the ordering of data elements as no swap operation takes place.
5) Completely sorts N unique data elements (regardless of signed or unsigned) in a linear sorting delay time of $O(N)$ clock cycles (including worst case).
6) Completely sorts N data elements (with nonzero duplicity rate) in less than $O(N)$ clock cycles with respect to duplicity rates.

The remainder of this article is arranged in the following manner. Section II summarizes a study on relevant works in sorting hardware. The proposed comparison-free sorting engine has been described in Section III. This section describes two major contributions of the architecture in handling signed and duplicate elements. Section IV evaluates the performance of the proposed sorting architecture, and finally, Section V concludes this article with some concluding remarks.

II. Related Works

Generally, the software-based sorting approaches are slow and unable to satisfy the requirements of real-time systems. Therefore, hardware-based sorters are alternative solutions to boost the performance of the system. Existing hardware sorting schemes are designed for either FPGA-based applications or for application specific domains. A hardware algorithm for sorting N elements using either a p-sorter or a sorting network of fixed input–output size p is presented in [18], runs in $O((N\times\log_2 N)/(p\times\log_2 p))$ time. To reduce sorting latency, some recent hardware-based sorting approaches used merge sort found in [15] and [20]. Matsumoto *et al.* [15] proposed a sorter called K-Sorter, a FIFO-based parallel merge sorter optimized for FPGAs. K-Sorter sorts K keys in $(K+\log_2 K)-1$ latency, using $\log_2 K$ comparators in $(K\times\log_2 K)$ comparisons. Another hardware merge sorter [20] achieved superior performance compared to optimized algorithms on CPUs or GPUs, and the estimated latency is nearly $(K\times\log_{W'} K/W')$ units (where W': number of sequences and $T\propto W'$, T: throughput), i.e., $O((K/W')\log_{W'} K)$ cycles for sorting K records. However, both [15] and [20] require large numbers of comparators. Merge sort-based two schemes, namely, wide merge sorter and efficient hardware merge sorter, are found in [24] to reduce hardware resources. Unfortunately, it fails to meet the requirement of power consumption of the system. A hardware-based partial sorting requires $\lfloor N/2\rfloor$ clock cycles to find the largest element (LE), but it consumes reasonably larger amount of FPGA resources [19]. A real-time hardware sorter using the multidimensional sorting algorithm is presented in [25]. Though it reduces the required resources and increases memory efficiency, but it has negative impact on execution time and consumes huge power, while the number of input increases. Usui *et al.* [14] and Song *et al.* [21] proposed parallel merge sorter trees, but regardless of their exploited parallelism in FPGA-based merge sorters for achieving good best-case throughput, suffer a significant throughput drop especially when the data distribution among the input streams is skewed poorly. A stable throughput is achieved in [26] by involving a high bandwidth sort merger. However, it is difficult to scale up this design for a large volume of data streams.

Some sorters are designed for specialized applications, Pedroni *et al.* [27] used systolic sorter to exploit direct pipelining. In spite of their acceptable latency, the required number of registers $(N\times(N+1))$ and CAS units $(\lfloor N/2\rfloor\times(N-1))$ is very large. To reduce sorting latency and resource usage, in their another work in [28], adopted compact digital parallel-input sorters for low power 2-D applications in image processing and data switching. The circuit contains N-bit registers and $\lfloor N/2\rfloor$ CAS units. Bitonic sorting is one of the fastest CAS-based sorting techniques implemented in hardware [6], [29], [30]. Bitonic sorters are more computation intensive compared to merge sorters, but are better for parallel implementation as comparisons of elements are done in a predefined sequence of independent of data. The conventional Bitonic sorter is fully combinational and has a complexity of $O(N\times(\log_2 N)^2)$ [6]. In [29], a hardware-based bitonic sorter to find top-k LEs has been presented. A recent scheme for max–min-set-selection on FPGA has been presented to reduce overhead of programming complexity through bitonic sorting networks [30]. A low-latency sorting technique is found in [16] for designing N-to-M sorting and max-set-selection units based on bitonic and odd-even merge sorting algorithms. Unfortunately, this approach consumes huge amount of hardware resources. A hardware design of partial sorting network is presented in [17] where the comparing modules move indices of samples instead of input data. However, the required hardware resources increase dramatically. A low-cost sorter using unary processing is found in [2], which is a subclass of stochastic computing. Here, resources required for CAS blocks are reduced, but suffers from substantial delay and low throughput. To resolve this

TABLE I
COMPARATIVE ANALYSIS OF SOME RECENT SORTING ARCHITECTURES AND THE MAIN FEATURES OF THE PROPOSED SORTER

Scheme & Year	Method	CAS	Pre-Processing	Duplicate Numbers	Signed Numbers	Time Complexity	Space Complexity
[16] 2013	Partial Sort (Bitonic)	Yes	Required	No	No	$O(\log_2 N \times \log_2 M)$	$O(N \log_2^2 M)$
[21] 2016	Merge Sort	Yes	Required	Yes*	No	$O((N/W') \times \log_{W'} N)^{@}$	$O(W' \log_2^2 W')$
[20] 2017	Merge Sort	Yes	Required	–	–	$O((N/W') \times \log_{W'} N)$	$O(W'^2 \log_2 W')$
[22] 2017	Comparison-Free	No	Required	Yes$^{\$}$	No	$O(N)$	$O(N^2)$
[17] 2017	Partial Sort (Bitonic)	Yes	Required	–	–	$O(\log_2 N \times \log_2 M + \log_2 N)$	$O(N \log_2^2 N)$
[2] 2018	Unary Processing	Yes	Required	–	–	$O(2^W)$	$O(N \log_2^2 N)$
[24] 2019	Merge Sort	Yes	Required	–	–	$O(N \log_2 N)$	$O(N)$
[25] 2019	Matrix-based Sort	Yes	Required	No	No	$O(\sqrt{N} \times \log_2^2 \sqrt{N})$	$O(W \times N \times \log_2^2 \sqrt{N})$
[1] 2019	Comparison-Free	No	Not Required	Yes$^{\$}$	No	$O(N)$	$O(N)$
[29] 2020	Partial Sort (Bitonic)	Yes	Required	No	No	$O(M \times W' + N/M + \log_2^2 M + \log_2 M)$	$O(N \log_2^2 M)$
[30] 2021	Partial Sort (Bitonic)	Yes	Required	–	–	$O(N \log_2 M)$	$O(M \log_2^2 M)$
Proposed	Comparison-Free	No	Not Required	Yes$^{\#}$	Yes	$O(N)$	$O(N)$

CAS: Compare-And-Swap; N: Inputs; M: Outputs; W: Element Width; W': Number of parallel sequences; *: Requires FIFO re-balancing; $^{\$}$: Treats duplicate numbers as unique numbers; $^{@}$: Without considering stalls; $^{\#}$: Duplicate elements are handled using a special unit;

issue, they further developed a time-based unary scheme where the input data are encoded in time and represented with pulse signals.

Two comparison-free sorting approaches are found in [22] and [23], and use complex matrix-mapping operations to carry out the actual sorting operation, which are headed by one-hot weight preprocessing. This kind of preprocessing is not required in [1] and in the proposed sorting architecture. The proposed engine exhibits a comparison-free sorting mechanism that does not require any sort of comparators, any complex circuitry, or any complex algorithm. The proposed architecture also consumes at most $O(N)$ clock cycles to sort unsigned, signed, as well as mixtures of any of them with nonzero duplicity rates. A comprehensive summary is presented in Table I considering some of the existing hardware-based sorting architectures and the proposed design. Here, time complexity is represented in terms of the required number of iterations to sort N elements, whereas space complexity denotes the required amount of registers or memory space consumed.

III. PROPOSED COMPARISON-FREE HARDWARE SORTING ENGINE

An improved hardware-based comparison-free sorting technique is introduced in this article that completely sorts N data elements utilizing N or lesser iterations. The architecture finds the LE in the first iteration and thereafter, it finds the candidate LEs from the remaining data elements in subsequent iterations until all the elements are sorted. The proposed architecture works for both unsigned and signed data elements and could excel in terms of sorting performance while handling duplicate data elements. A simplified schematic of the proposed architecture is depicted in Fig. 1. This architecture receives input data set from an unsorted memory (UM) unit (upon receiving a read signal $\overline{\text{RD}}$ from the sort controller) through a switch that fundamentally behaves more like a bus arbitrator. Upon receiving data from UM, the sort controller initiates enable signal (EN) to perform actual sorting operation in the hardware sorting engine. In every clock cycle, it detects the local LE(s) from an updated data set followed by storing them in the

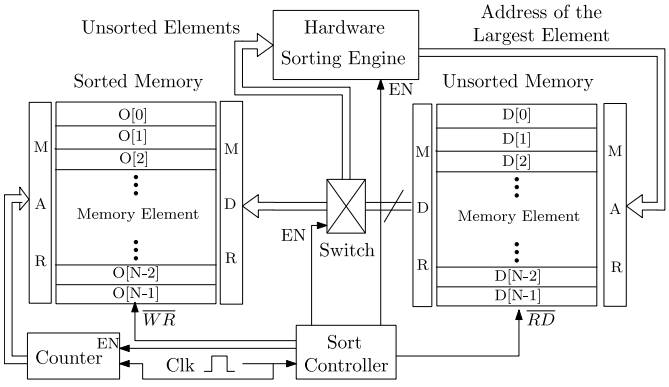

Fig. 1. Schematic of proposed comparison-free sorting engine.

sorted memory (SM) unit. Local LE is determined by placing its address in an UM register [namely, memory address register (MAR)]. After generating the address of LE, the controller sends a write signal ($\overline{\text{WR}}$) to store LE at the proper location in SM as indicated by the up-counter. The LE as being retrieved from UM is loaded into a register named memory data register (MDR) before it is eventually stored in SM.

A. Organization of Hardware Sorting Engine

Proposed sorting engine consists of a number of cascaded blocks (shown in Fig. 2), selected one after another. Given a set of N, n-bit wide data elements, the architecture employs n blocks. As a matter of fact, each block is filtering the smaller elements and forwarding the larger elements to the next block for further filtering and finally deciding the LE among the participating elements. Each of these blocks consists of N number of basic cells those operate in parallel fashion. The internal structure of a block is shown in Fig. 3(a). Each cell consists of a 2-input AND gate and a tiny switch (2:1 multiplexer), shown in Fig. 3(b). The data flow diagram of a block is shown in Fig. 3(c). Since, the cells present in a block run concurrently, therefore, delay incurred by each of these blocks is negligibly small, primarily composed of a 2-input AND gate delay (t_{AND^*}), an N-input OR gate realized using 4-input OR

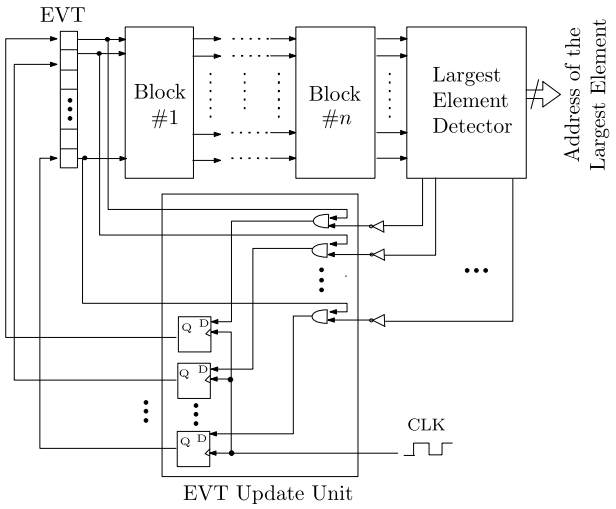

Fig. 2. Block level structure of the proposed sorting engine.

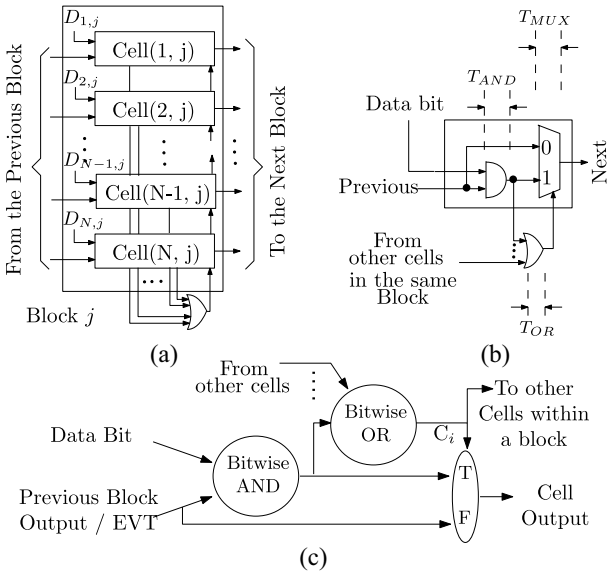

Fig. 3. Structure of (a) an intermediate block *j*; (b) a cell; and (c) data flow diagram of a block.

gates exhibits delay as ($\lceil \log_2 N/4 \rceil \times t_{\mathrm{OR}^\#}$), termed as selection delay (T_{sel}), and a multiplexer delay (T_{MUX}), shown in (1). Here, only T_{sel} ($\propto N$) is variable, while t_{AND^*} and T_{MUX} delay components are constant. For a large N, T_{sel} becomes a dominating factor. Per element sorting time, T_i is expressed in (2). T_{ENC} delay is expressed as per (3). Total sorting latency to sort N elements is denoted by $T_{\mathrm{sort}} (= N \times T_i)$

$$T_{\mathrm{block}} = t_{\mathrm{AND}^*} + T_{\mathrm{sel}} + T_{\mathrm{MUX}} \approx T_{\mathrm{sel}} \quad (1)$$

$$T_i = (n \times T_{\mathrm{block}}) + T_{\mathrm{ENC}} \quad (2)$$

$$T_{\mathrm{ENC}} = \left(\left\lceil \frac{\log_2 N}{4} \right\rceil \times t_{\mathrm{OR}^\#} \right) + T_{\mathrm{sel}} + t_{\mathrm{AND}^*} + t_{\mathrm{INV}}. \quad (3)$$

At the beginning of each iteration, the element vector table (EVT) reflects the list of input data elements those are yet to be sorted. The first block receives two inputs, one from UM and another from EVT. The outputs of this block (cells present in a block) are passed through an OR logic (hierarchy of parallel 4-input OR gates) as the selection input to all the multiplexers

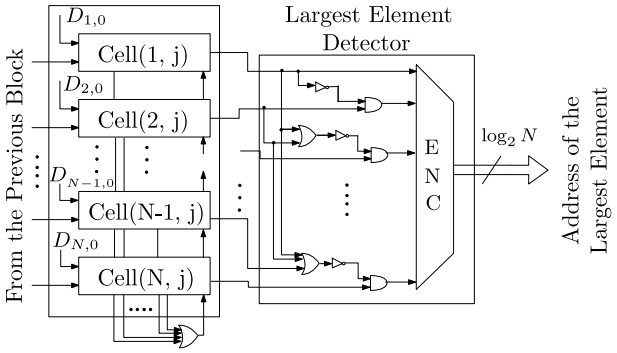

Fig. 4. Structure of terminal block with LE detector.

present in that block. These multiplexers also receive two inputs, one from the respective cell's AND gate and another from its previous block (corresponding cell). Continuing this way, in the final block, the output of only one of the switches is always found high (for unique data entries) to indicate the bit position of LE with the help of an LE detector (LED) unit (shown in Fig. 2) in a certain iteration. However, for the duplicate entries, the output of multiple switches might turn high, which is resolved by imposing a masking logic prior to an ordinary encoder circuit. This masking logic along with the encoder unit is termed as LED (shown in Fig. 4).

B. Operation of Hardware Sorting Engine

Let us take an example data set containing five data elements, namely, 5 (0101_2), 10 (1010_2), 7 (0111_2), 14 (1110_2), and 12 (1100_2). Here, EVT is a 5-bit register, where each bit represents a data element in UM and is initialized to 11111_2. In this particular example, there exist four blocks (for 4 bit data elements) in the architecture. The first block (from left) receives MSB (D_3) of these five data elements as one of the inputs and the second input from EVT. The output of the first block is 01011_2 and is fed to the second block (as the output of the OR plane is "1") along with the next higher order bit (i.e., D_2). Continuing this way, the output of the final block is 00010_2 at the end of the first iteration and reveals fourth data element, i.e., 14 is the LE. For unique data elements, under no such circumstances, multiple 1's would result in FO. In the next cycle, identification of the next LE and storing of the last identified LE in SM at an address indicated by the counter unit (shown in Fig. 4) are done in an overlapped fashion. At the beginning of the next cycle, EVT is updated by bitwise AND-ing it with the complemented output of FO, i.e., $\overline{\mathrm{FO}}$, i.e., 11101_2. This process continues until, all the bit position in EVT turns zero, or alternatively, all the data elements are sorted. After completion of the fifth iteration, data elements are completely sorted, shown in Fig. 5.

The sorting engine starts functioning upon receiving start operation signal (SO) at every iteration. The end of operation signal (*EO*) initiates the storing of LE at the incremental address in SM. The timing diagram of the sorting process is shown in Fig. 6. A signal named address latch enable (ALE) is used to latch the address of LE denoted by Addr(LE) signal. This information is maintained by MAR register of UM. Now, the LE is read out from UM by initiating $\overline{\mathrm{RD}}$ control

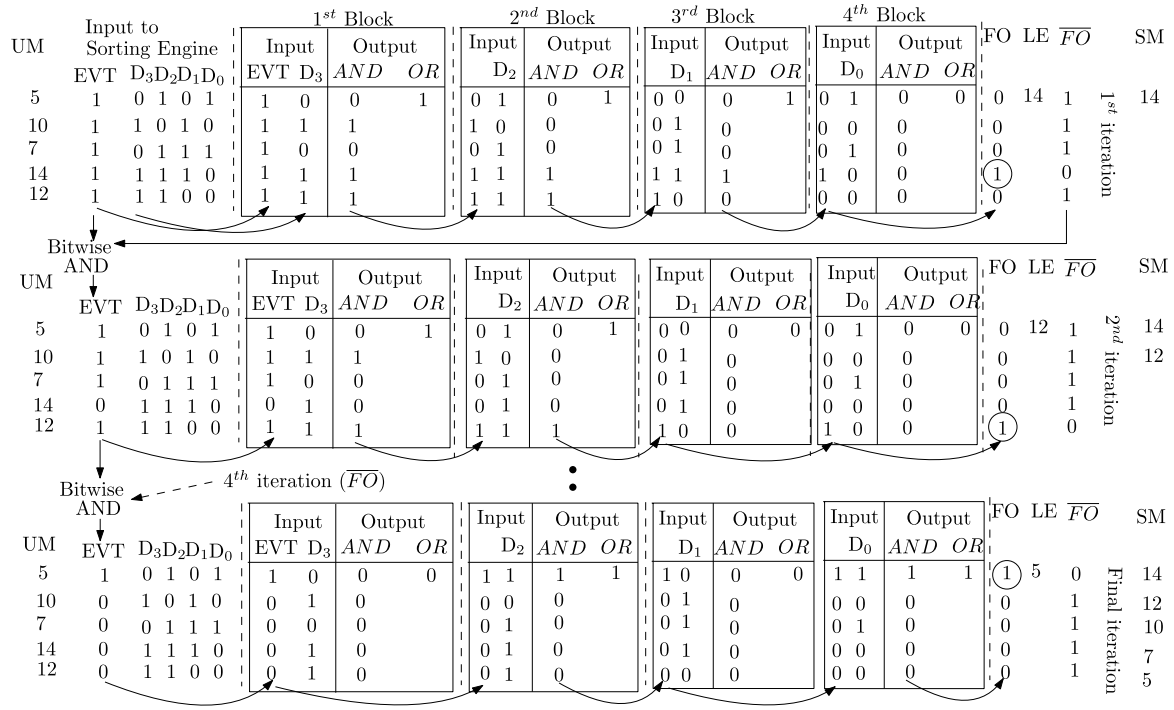

Fig. 5. Illustrative example showing internal operations performed for sorting unique unsigned data elements.

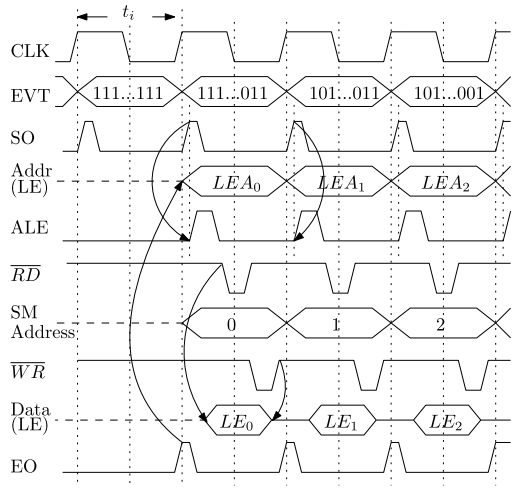

Fig. 6. Timing diagram of the proposed sorter handling unique data elements.

signal and thereafter, is loaded to MDR and eventually to SM (by initiating $\overline{\text{WR}}$ signal).

C. Handling of Signed Numbers

1) Functional Structure: Sorting of SNs has been discussed using a simplified schematic, shown in Fig. 7. The architecture behaves quite similar as its unsigned counterpart, however, there exists minor improvisation in the design to handle signed elements. For SNs, an additional block (compared to unsigned [1]) would be required to handle sign information of data elements. In this context, it is to be noted that architecture of [1] does not support handling of signed elements, whereas this engine has a dedicated block (initial block of the

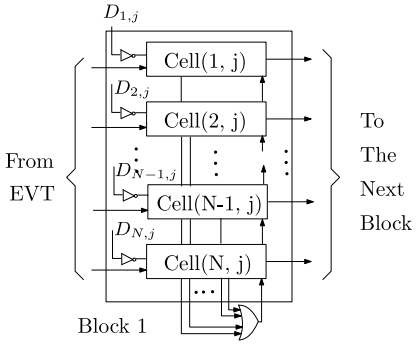

Fig. 7. Structure of the initial block for SNs.

design) to handle signed elements once the input set is given in two's complement form. Another minor modification has been adopted, where sign information is complemented before it is fed to the initial block. The architecture receives unsorted signed data elements and identifies relative LE in every clock cycle.

2) Delay Model: Handling of signed elements is almost similar to that of its unsigned counterpart. However, unlike unsigned data elements, an additional block delay is incurred due to sign information. Per-element sorting delay for unsigned numbers (UNs) as expressed in (2) could be modified as per (4) for signed data elements

$$T_i' = (n+1) \times T_{\text{block}} + T_{\text{ENC}}. \tag{4}$$

Now, per element sorting performance in terms of clock cycle counts for SNs can be measured as per

$$\frac{T_i}{T_i'} = \frac{n \times T_{\text{block}} + T_{\text{ENC}}}{(n+1) \times T_{\text{block}} + T_{\text{ENC}}} \approx 1. \tag{5}$$

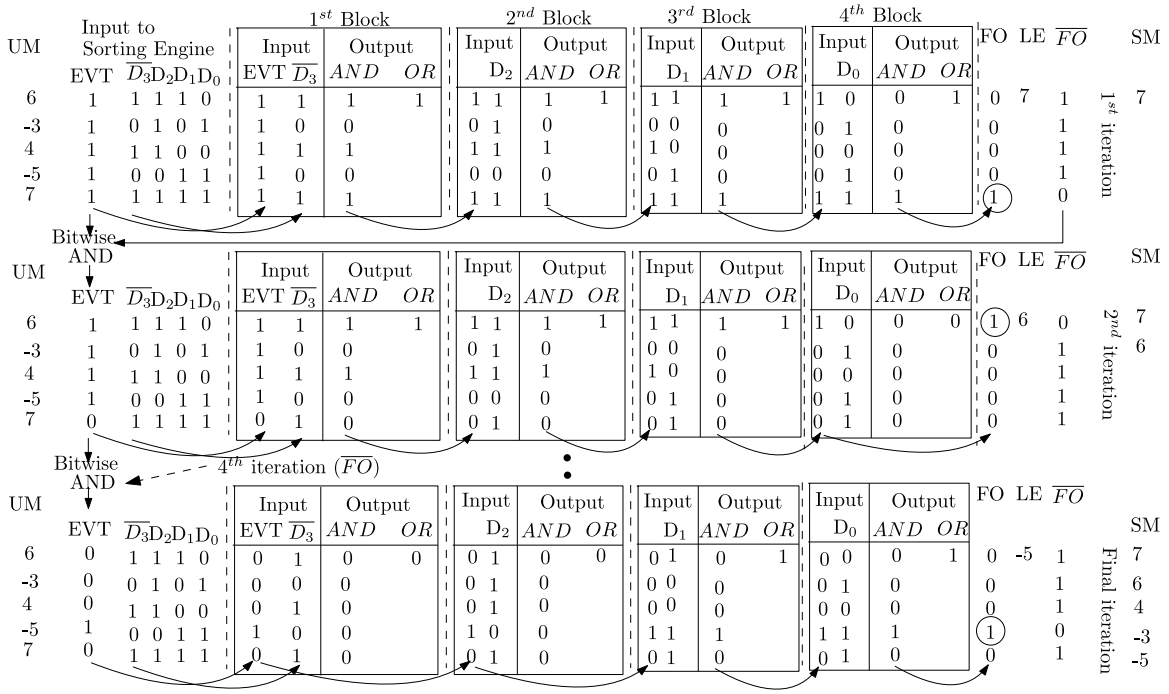

Fig. 8. Illustrative example of the proposed hardware sorting engine with SNs.

One may understand that per-element sorting time (iteration) irrespective of unsigned or SNs are roughly equal.

3) Working Principle With Illustrative Example: Let us consider an example input data set containing five signed data elements (in two's complement form) viz., 6 (0110_2), -3 (1101_2), 4 (0100_2), -5 (1011_2), and 7 (0111_2). In the signed architecture, the initial block receives one input from the complemented sign bits of the data elements and another input from the contents of EVT register. At the beginning, all bits in EVT are set, i.e., 11111_2, where each bit represents an unsorted data element in UM. At the end of each iteration, EVT register is updated according to the output been generated through $\overline{\text{FO}}$, shown in Fig. 8. After complementing sign bits of the data elements, those five data elements are expressed as 6 (1110_2), -3 (0101_2), 4 (1100_2), -5 (0011_2), and 7 (1111_2) for the sole purpose of the sorter design. In Fig. 8, completion of the first iteration reveals that the LE is "7" as indicated by FO. Thereafter, that bit position in EVT is updated to "0" to indicate that for the remaining iterations no longer that data element participates in the sorting operation. Similarly, the remaining elements are also sorted and stored in SM. The timing diagram of the sorting operation to handle signed data elements is similar that of its unsigned counter part (refer Fig. 6).

D. Handling of Duplicate Numbers

1) Functional Structure: Sorting of signed elements with a nonzero duplicity factor is discussed here. Like the previous architectures (described in Sections III-A and III-C), this architecture also receives unsorted data elements and detects relative LEs in every clock cycle. However, uniqueness comes from the ability to detect the duplicate LEs and sort each of them without consuming a complete per-element sorting latency (unlike the architecture described in [1]) and instead occupy only a meagre fraction of it. Reference [1] and the other prior hardware sorter designs [21], [22] are unable to discriminate between duplicate and unique elements (fails to avoid expensive compare and swap operations) in a given data set and hence, treat each entry as a unique element in terms of per element sorting latency. Hence, this feature of handling of duplicate elements results in improved system throughput. Controlling in terms of clock initiation and sending proper control signals to the respective units are governed by a sort controller.

For unique data elements, under no such circumstances, multiple 1's would result into FO at the end of any of the iterations. However, due to duplicate elements, the presence of multiple 1's in FO is a common scenario, which indicates duplicate LEs among the participating data elements. The largest and duplicate element detector (LDED) unit (contains a priority encoder circuit) is used to handle generation of multiple LEs per iteration, shown in Fig. 9. Say, for a certain iteration, if it is found that out of N elements, k' numbers of 1's are present in FO. Then, those k' elements are sorted without repeating the entire process for k' times. Sorting of these k' elements resets their corresponding bits in a temporary register (TR). Each bit resetting in TR is immediately followed by an OR-ing operation. Controlling over the internal clock is governed by the output of the OR-ing operation. Until all these k' elements (duplicate) are sorted (outcome of an iteration), the next iteration cannot be initiated based on the contents of FO. In other words, we may say that after sorting k' duplicate elements, another $(N - k')$ system clock cycles would be required (in worst case) if remaining data elements are unique. It is to be noted that presence of duplicate elements reduces the

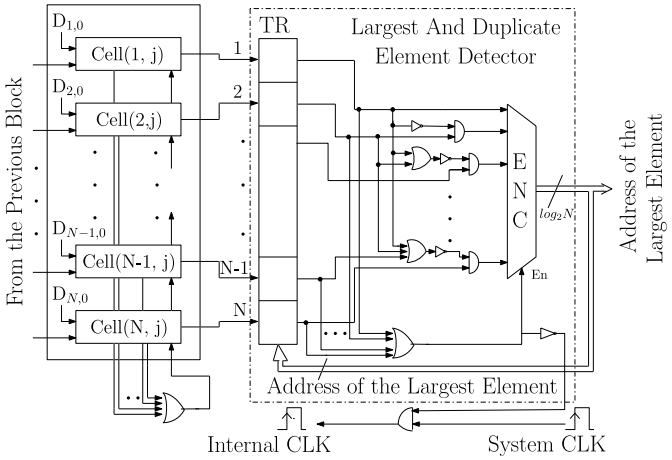

Fig. 9. Structure of the terminal block with LDED.

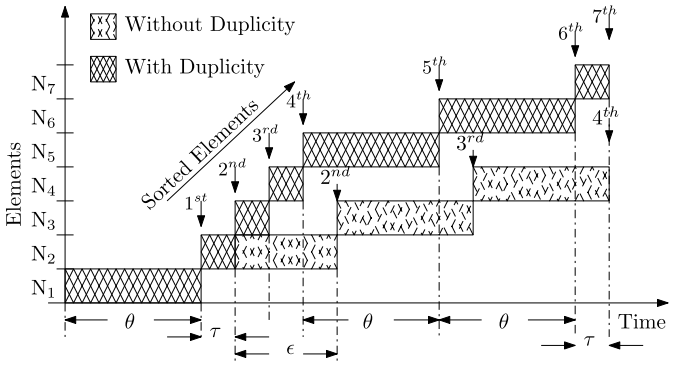

Fig. 10. Conceptualized timing chart of handling duplicate elements considering $\tau/\theta = 1/4$.

number of required sorting cycles, i.e., less than $O(N)$ clock cycles. In the best case (i.e., all elements in an input data set are same valued), the total number of required clock cycles is less than $O(N)$ clock cycles; however, in the worst case (i.e., all elements are unique except one element appears twice in input data set), the sorter consumes $O(N)$ clock cycles.

2) Delay Model: Unlike unique elements, duplicate elements do not consume a complete per-element sorting cycle and instead take only a meagre portion of it. For a given N elements, the system requires $T_{\text{sort}_{WD}}$ amount of time to completely sort the elements without having any duplicate entry in the input list, shown in

$$\begin{aligned} T_{\text{sort}_{WD}} &= N \times T_i \\ &\approx N(n \times T_{\text{block}} + T_{\text{ENC}} + T_{\text{DEC}}) \\ &\approx N(n \times T_{\text{sel}} + 2 \times T_{\text{ENC}}). \end{aligned} \tag{6}$$

However, to completely sort those many number of elements with k duplicate entries, the system would require T_{sort_D} time, which is certainly not same as $T_{\text{sort}_{WD}}$. The amount of time required for sorting each duplicate entity T_{DUP} is not as much as it is required by a unique element, expressed in (7). It is to be noted that in (7), prominent factors are $T_{\text{Add\_dec}}$, T_{ENC}, and T_{sel}. It is quite understood that $T_{\text{Add\_dec}}$ and T_{ENC} time delays are roughly equal and therefore, $T_{\text{Add\_dec}}$ could be approximated to T_{ENC}. T_{sort_D} time is shown in

$$\begin{aligned} T_{\text{DUP}} &= T_{\text{reg}} + T_{\text{Add\_dec}} + T_{\text{ENC}} + T_{\text{sel}} + t_{\text{AND}^*} + t_{\text{INV}} \\ &\approx T_{\text{sel}} + 2 \times T_{\text{ENC}} \end{aligned} \tag{7}$$

$$T_{\text{sort}_D} = (N-k)T_i + k \times T_{\text{DUP}}. \tag{8}$$

One can easily understand that T_{DUP} time is insignificant compared to per-element sorting time (T_i) (as per (2)) and instead, it is comparable with T_{block} (refer (1)). Another important observation is that T_{DUP} time is fixed irrespective of the width (in terms of bits) of data elements. Now, we can further express T_{sort_D} considering (7) and (8), shown in

$$\begin{aligned} T_{\text{sort}_D} &\approx (N-k)(n \times T_{\text{sel}} + T_{\text{ENC}}) + k(T_{\text{sel}} + 2 \times T_{\text{ENC}}) \\ &\approx (n \times N + k(1-n))T_{\text{sel}} + (N+k)T_{\text{ENC}}. \end{aligned} \tag{9}$$

Now, the system performance is measured in terms of reduced number of clock cycles. Speedup (S) is calculated considering time delays of $T_{\text{sort}_{WD}}$ and T_{sort_D}, shown in

$$\begin{aligned} S = \frac{T_{\text{sort}_{WD}}}{T_{\text{sort}_D}} &\approx \frac{N(n \times T_{\text{sel}} + 2 \times T_{\text{ENC}})}{[N \times n + (1-n)k]T_{\text{sel}} + (N+k)T_{\text{ENC}}} \\ &\approx \frac{N(n + 2 \times \gamma)}{N \times n + (1-n)k + (N+k)\gamma}; \left(\text{where, } \gamma = \frac{T_{\text{ENC}}}{T_{\text{sel}}}\right). \end{aligned} \tag{10}$$

Throughput of the proposed system considering duplicate entries is denoted by η_D, shown in

$$\begin{aligned} \eta_D = \frac{N}{T_{\text{sort}_D}} &\approx \frac{N}{(N \times n + (1-n)k)T_{\text{sel}} + (N+k)T_{\text{ENC}}} \\ &\approx \frac{N}{(N \times n + (1-n)k + (N+k)\gamma)T_{\text{sel}}} \\ &\approx \frac{N}{((n+\gamma)N + (1-n+\gamma)k)T_{\text{sel}}}. \end{aligned} \tag{11}$$

3) Working Principle: In the previous section, we have talked about major delay factors involved to completely sort N data elements with or without a duplicity factor. Now, we shall discuss system operation in handing N data elements with a duplicity rate of k/N. To understand the idea, let us consider a *major cycle*, θ, which represents a complete cycle required to sort a unique element and a *minor cycle*, τ, to sort a duplicate element, shown in Fig. 10. To draw resemblance with the previous discussions, we may use θ to denote T_i and τ to denote T_{DUP}. The ratio of *major cycle* and *minor cycle* is expressed in (12) as follows:

$$m = \frac{\theta}{\tau}. \tag{12}$$

In (12), m denotes upper bound in terms of the total number of duplicate elements those could be sorted within a *major cycle*. According to the example cited in Fig. 10, seven elements are present with four duplicate entries. Here, $m = 4$ (as $\theta = 4$ and $\tau = 1$) and therefore, in each *major cycles*, four duplicate elements could be sorted consuming four *minor cycles*. Hence, these seven elements are sorted consuming four *major cycles*. However, for unique elements, four elements could be sorted at the expiry of the fourth clock cycle.

4) Conceptual Working Principle: We have considered N elements with k duplicate entries. With respect to duplicity rates in the input set, we may have following cases.

1) *Case I ($k < m$):* For each duplicate entrant, the amount of per-element sorting time saved is denoted by ϵ (difference between *major cycle* and *minor cycle*). The total sorting time for $k < m$ ($T_{k<m}$) is expressed in

$$\begin{aligned} T_{k<m} &= (N-k)\theta + k\tau \\ &= N\theta - (\theta-\tau)k \\ &= N\theta - \epsilon k, \quad [\text{where, } \epsilon = \theta - \tau]. \end{aligned} \tag{13}$$

2) *Case II ($k = m$):* For each duplicate entrant, the amount of per-element sorting time taken is τ. Therefore, total sorting time ($T_{k=m}$) required can be expressed as per

$$\begin{aligned} T_{k=m} &= (N-k)\theta + k\tau \\ &= (N-m+1)\theta, \quad \left[\text{where, } m = \frac{\theta}{\tau}\right]. \end{aligned} \tag{14}$$

3) *Case III ($k > m$):* For each duplicate entrant, the amount of per-element sorting time saved is denoted by $[(m-1)/m]$. Therefore, total sorting time ($T_{k>m}$) required can be expressed as per

$$\begin{aligned} T_{k>m} &= (N-k)\theta + k\tau \\ &= (N-k)\theta + \frac{k\theta}{m} \quad \left[\text{where, } m = \frac{\theta}{\tau}\right] \\ &= \left[N - \frac{k(m-1)}{m}\right]\theta. \end{aligned} \tag{15}$$

Following (13)–(15), one may understand that sorting of each duplicate entry consumes only τ amount of time. However, the actual cycle consumption to sort the duplicate elements result into a quite different scenario with respect to duplicity rates, explained in the following section.

5) Experimental Working Principle: The explanation is further extended considering an input list of N elements with k duplicate entries from P unique elements. Let us also consider that each of these P unique elements has appeared in the list for d_i times, where $1 \le i \le P$. D is expressed as a set, where, $D = \{m, 2m, 3m, \ldots\}$. According to previous considerations, the number of major cycles (χ_θ) involved to sort only the duplicate entries may have two components, viz.: 1) complete cycle (χ_{θ_c}) and 2) partial cycle (χ_{θ_p}). Complete cycles include one or multiple *major cycles*, while partial cycles consume only a portion of a *major cycle*. The total number of complete cycles and partial cycles required to sort those k elements is expressed in (16) and (17), respectively. According to (16), *major cycle* are counted in terms of full cycles utilization, and therefore, fractional portion is not considered. However, in (17), only consumption of partial *major cycles* is counted, each of which is rounded off to a single complete *major cycle*

$$\chi_{\theta_c} = \sum_{i=1}^{P}\left(\left\lfloor \frac{d_i}{m} \right\rfloor\right) \tag{16}$$

$$\chi_{\theta_p} = \sum_{i=1}^{P}\left(\left\lceil \frac{(d_i \ \% \ m)}{m} \right\rceil\right). \tag{17}$$

In reality, sorting performance is measured in terms of the number of complete cycles. Therefore, χ_{θ_p} components are suitably scaled up depending upon the following three cases.

1) *Case I ($d \in D$):* Since each of those P elements is duplicated for multiple of m times, therefore, *major cycles* have been utilized maximally following case II of Section III-D4. Therefore, the system will not experience any *idle time* (T_{idle}). Here, the number of *major cycles* required to sort the duplicate LEs is shown in

$$\begin{aligned} \chi_\theta &= \sum_{i=1}^{P}\left(\left\lfloor \frac{d_i}{m} \right\rfloor + \left\lceil \frac{(d_i \ \% \ m)}{m} \right\rceil\right) \\ &= \left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor + \left\lceil \frac{d_1 \ \% \ m}{m} \right\rceil + \cdots + \left\lceil \frac{d_P \ \% \ m}{m} \right\rceil \\ &= \left(\left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor\right) + (0 + \cdots \text{upto } P \text{ terms}) \\ &= \left(\left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor\right). \end{aligned} \tag{18}$$

2) *Case II ($(d \notin D) \wedge (d < m)$):* Here, $d_i < m$, where $1 \le i \le P$, the sorter is idle for a finite duration of time (less than one *major cycle*) and is given in (19). However, to express total *idle time* of the system for sorting P unique elements those appeared in the list for d_i times each, could be expressed as in (20). The number of *major cycles* required to sort all the duplicate elements could be expressed as per

$$T_{\text{idle}} = [\theta - (d \ \% \ m)\tau] \tag{19}$$

$$T_{\text{idle}_{\text{TOTAL}}} = \sum_{i=1}^{P}[\theta - (d_i \ \% \ m)\tau] \tag{20}$$

$$\begin{aligned} \chi_\theta &= \sum_{i=1}^{P}\left(\left\lfloor \frac{d_i}{m} \right\rfloor + \left\lceil \frac{(d_i \ \% \ m)}{m} \right\rceil\right) \\ &= \left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor + \left\lceil \frac{d_1 \ \% \ m}{m} \right\rceil + \cdots + \left\lceil \frac{d_P \ \% \ m}{m} \right\rceil \\ &= (0 + \cdots \text{upto } P \text{ terms}) + (1 + \cdots \text{upto } P \text{ terms};) \\ &= P. \end{aligned} \tag{21}$$

3) *Case III ($(d \notin D) \wedge (d > m)$):* When $d_i > m$, where $1 \le i \le P$, then also, the *idle time* of the sorter is as per (19) and (20). However, count of *major cycles* to sort the duplicate elements could be expressed as per

$$\begin{aligned} \chi_\theta &= \sum_{i=1}^{P}\left(\left\lfloor \frac{d_i}{m} \right\rfloor + \left\lceil \frac{(d_i \ \% \ m)}{m} \right\rceil\right) \\ &= \left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor + \left\lceil \frac{(d_1 \ \% \ m)}{m} \right\rceil + \cdots + \left\lceil \frac{(d_P \ \% \ m)}{m} \right\rceil \\ &= \left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor + 1 \ \cdots \text{ upto } P \text{ terms}; \left(\frac{(d_1 \% m)}{m} < 1\right) \\ &= \left\lfloor \frac{d_1}{m} \right\rfloor + \cdots + \left\lfloor \frac{d_P}{m} \right\rfloor + P. \end{aligned} \tag{22}$$

6) Working Principle With Illustrative Example: In the proposed scheme, duplicate elements are treated differently compared to unique data elements. For a given data set containing five signed data elements with duplicate entries, namely, 5 (0101_2), 7 (0111_2), -6 (1010_2), 7 (0111_2), and -4 (1100_2), are shown in Fig. 11. In this particular example, on the verge of expiry of the first iteration, FO contains 01010_2. Multiple 1 in FO shows the presence of duplicate

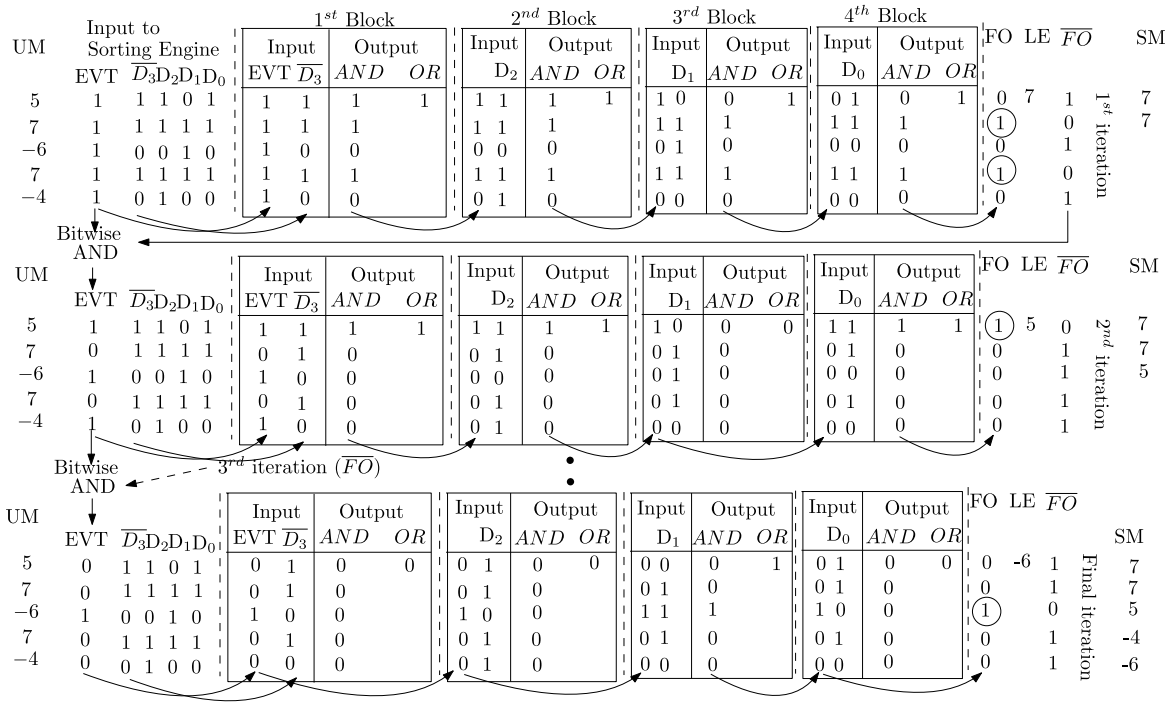

Fig. 11. Illustrative example of the proposed hardware sorting engine handling duplicate entries.

LEs. First, FO is fed to a priority encoder to identify the address of LE. Furthermore, this output of the encoder is used to reset the corresponding bit positions in FO, i.e., updated FO is 00010_2. Now, the value of OR-ing of FO will dictate whether that iteration will lengthen or terminate based on the presence of more LEs in the output result. Continuing with the same procedure, once again updated FO is fed to the encoder and accordingly resetting of corresponding bit turns 00000_2. The repeated resetting of respective FO bits continues until FO contains all 0's. Now, the original FO of this iteration (first iteration), i.e., 01010_2 is complemented and bitwise AND-ed with current EVT content to reveal how many elements are left to be sorted in the succeeding iterations. For a certain iteration, if FO contains a zero value, then it indicates end of that iteration and beginning of a new iteration if updated EVT still contains a nonzero value. On expiry of the fifth iteration, it is evident that all the elements are sorted.

A timing diagram illustrating working of the proposed sorter in handling duplicate elements is shown in Fig. 12. Here, external clock is used to find the LEs in consecutive clock cycles, while the internal clock cycle is used to handle the scenario of duplicate entries in a particular iteration. The content of EVT indicates the number of data elements is yet to be sorted (shown by 1 s). At the end of every iteration, FO contains information regarding the LE/(s) present in the unsorted data list. An ALE signal is used to latch the address of the largest element held by LE (indicated by FO followed by a priority encoder). In the first iteration, three elements are found largest and accordingly, FO is updated. Each FO update results into the generation of ALE signal for latching address of LE and SM. Now, LE is read out by initiating $\overline{\text{RD}}$ signal for the first data element and remains disabled for the remaining duplicate LEs. $\overline{\text{RD}}$ signal is initiated only once for every distinct

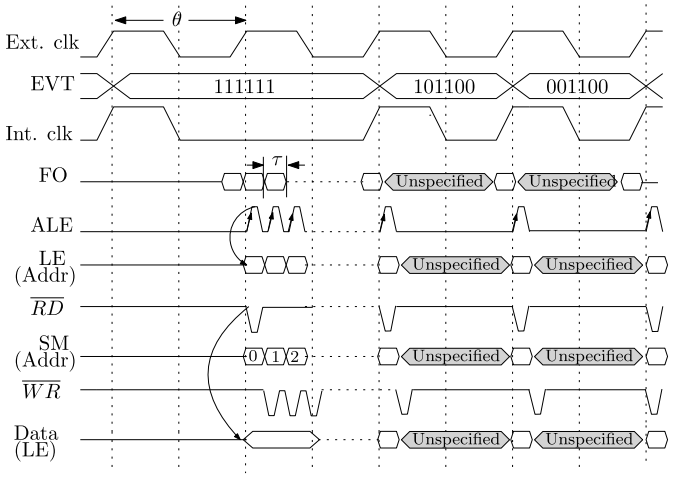

Fig. 12. Timing diagram for handling duplicate entries.

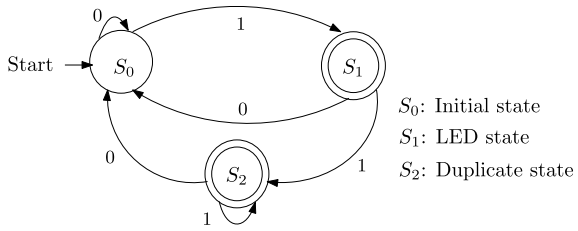

Fig. 13. Transition diagram showing a sorting cycle in the proposed sorter while handling duplicate elements.

elements present in the list. However, the $\overline{\text{WR}}$ signal is initiated to write these LEs in SM. Duplicate LEs in a certain iteration reduce per element sorting time for the succeeding elements. A transition diagram for duplicate elements is illustrated in Fig. 13. The system will undergo three possible states, namely, "initial state," "LED state," and "duplicate state" if there exist

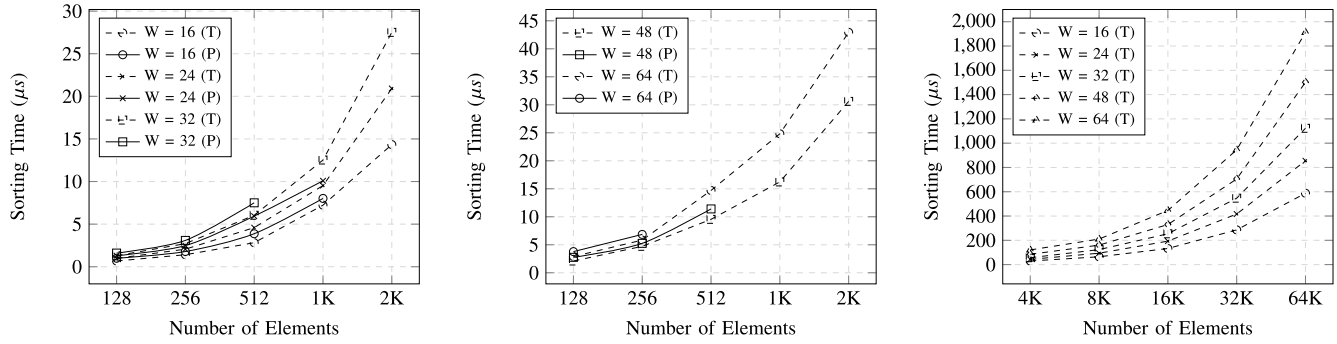

Fig. 14. Sorting latency for unsigned elements with different widths (*W*) in theoretical (*T*) and practical (*P*) setup.

TABLE II
PER-ELEMENT SORTING LATENCY AND THROUGHPUT (MEPS) FOR HANDLING UNIQUE DATA ELEMENTS

N (Range)	Width	Sorting Latency	Throughput (MEps)
256 to 1K	16	7 ns – 8.49 ns	142 – 117
	24	10.27 ns – 12.57 ns	97 – 79
256 to 512	32	13.60 ns – 15.10 ns	73 – 66
	48	20.20 ns – 22.56 ns	49 – 44
256	64	26.80 ns	37

TABLE III
FPGA RESOURCE UTILIZATION OF STATE-OF-THE-ART SORTING SCHEMES FOR 32-BIT DATA ELEMENTS (*N*), SYNTHESIZED IN VIRTEX-5 XC5VLX50T

Sorting Scheme	N	LUTs		Flip-Flops		Frequency
		Count	%	Count	%	(MHz)
[2] 2018#	16	1822	6.32	1033	3.58	100
	32	4250	14.76	2126	7.38	82
	64	8635	29.52	4264	14.76	70
	128	15787	54.82	8562	29.73	52
	256	Non-Synthesizable				
[21] 2016#	16	21252	73.79	7894	27.41	225
	32	Non-Synthesizable				
[20] 2017#	16	18824	65.36	27325	94.88	300
	32	Non-Synthesizable				
[25] 2019#	16	2429	8.43	2030	6.97	490
	32	7894	27.41	7893	27.41	482
	64	13358	46.38	17001	58.81	475
	128	25502	88.55	27302	94.79	450
	256	Non-Synthesizable				
[22] 2017$	16	2084	7.24	2342	8.13	96
	32	3382	11.74	3513	12.20	84
	64	6107	21.20	6391	22.19	71
	128	12502	43.41	13202	45.84	62
	256	26502	92.02	28102	97.58	50
	512	Non-Synthesizable				
[1] 2019$	16	241	0.84	612	2.12	134
	32	443	1.54	1156	4.01	110
	64	821	2.85	2358	8.19	93
	128	1556	5.40	4761	16.53	82
	256	2913	10.11	9473	32.89	72
	512	8089	28.09	18619	64.65	65
Proposed$	16	249	0.86	630	2.19	130
	32	459	1.59	1193	4.14	107
	64	853	2.96	2431	8.44	91
	128	1620	5.63	4908	17.04	80
	256	3041	10.56	9767	33.91	71
	512	8345	28.98	19207	66.69	64

#: Comparison-based; $: Comparison-free;

duplicate elements in the input list. Transition from S_0 to S_1 takes place on finding LE in each iteration. However, duplicate LEs enable transition from S_1 to S_2. The system will remain in S_2 until all duplicate LEs are resolved.

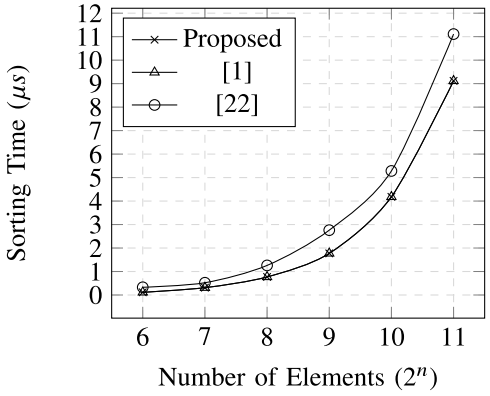

Fig. 15. Sorting time required by comparison-free hardware architectures for 2^n unsigned data elements, where n denotes data width, and $6 \leq n \leq 11$.

IV. PERFORMANCE ANALYSIS

The proposed sorting engine is simulated and synthesized in Xilinx ISE 14.7 and Virtex-5 XC5VLX50T development board. The experimental setup includes a variety of test data sets from pseudorandomly generated data elements to unique data elements, and also from random to completely sorted data elements as test stimuli. Furthermore, the architecture has been tested against different duplicity rates. It is worth mentioning that the architecture appears impartial to the ordering of data elements. Due to resource constraints of the XC5VLX50T platform, we could not test beyond 256 data elements of 64 bits, 512 data elements of 48 bits, and 1024 elements of 24 bits. However, theoretically we have analyzed the expected delay trend to sort up to 64K unsigned elements of different widths (shown in Fig. 14). The UM is implemented using registers, whereas the SM is maintained utilizing block RAM (BRAM) in the FPGA board.

We have considered delay components of 65-nm standard cell library at 1 V as per Virtex-5. Per-element sorting latency and throughput of the architecture have been depicted in Table II. The throughput has been demonstrated in terms of Million Elements per second (MEps). FPGA resource utilization of the proposed sorting engine and some recent sorting methods are reported in Tables III and IV considering 16–512 data elements for 32 and 64 bit, respectively. In both the tables, the operating frequency is reported based on the number of synthesizable elements handled by some of the existing comparison-based and comparison-free approaches. The comparison-free method in [22] requires a large amount

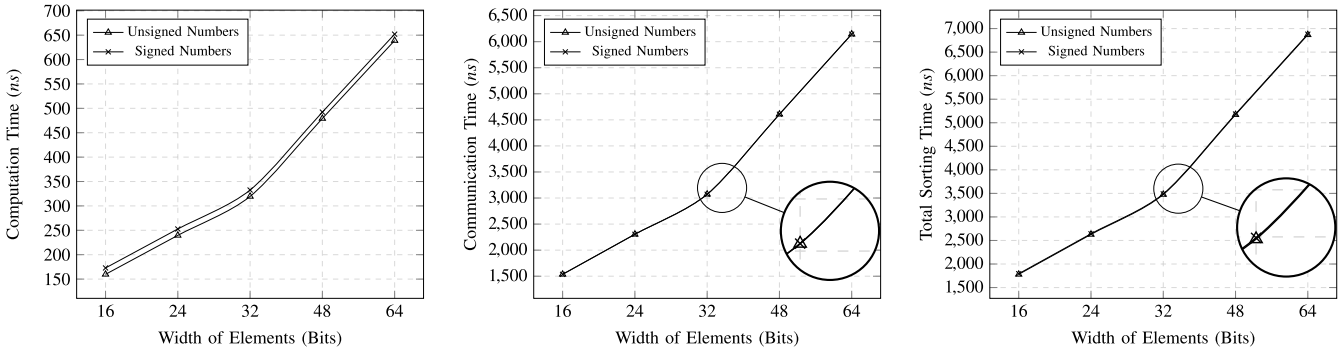

Fig. 16. Sorting time consumption profile of the proposed sorter to sort 256 elements of various data widths. (a) Computation latency. (b) Communication latency. (c) Total sorting latency.

TABLE IV
FPGA RESOURCE UTILIZATION OF STATE-OF-THE-ART SORTING SCHEMES FOR 64-BIT DATA ELEMENTS (N), SYNTHESIZED IN VIRTEX-5 XC5VLX50T

Sorting Scheme	N	LUTs		Flip-Flops		Frequency (MHz)
		Count	%	Count	%	
[2] 2018#	16	3643	12.65	2004	6.96	70
	32	7894	27.41	4251	14.76	68
	64	14572	50.89	8502	29.52	57
	128	27931	96.97	25503	88.55	50
	256	Non-Synthesizable				
[21] 2016#	16	Non-Synthesizable				
[20] 2017#	16	Non-Synthesizable				
[25] 2019#	16	4857	16.87	3643	12.65	333
	32	10322	35.84	13966	48.49	325
	64	22770	79.06	26109	90.66	300
	128	Non-Synthesizable				
[22] 2017$	16	3112	10.81	4125	14.32	52
	32	5957	20.68	7812	27.13	44
	64	11132	38.65	13834	48.03	35
	128	22502	78.13	25102	87.16	28
	256	Non-Synthesizable				
[1] 2019$	16	836	2.90	1710	5.94	67
	32	1542	5.35	3503	12.16	55
	64	3002	10.42	7080	24.58	46
	128	5716	19.85	12074	41.92	41
	256	11728	40.72	20001	69.45	36
Proposed$	16	844	2.93	1726	5.99	66
	32	1558	5.41	3535	12.27	53
	64	3034	10.53	7144	24.81	45
	128	5780	20.07	12202	42.37	40
	256	11856	41.17	20257	70.34	35
	512*	24298	84.37	27636	95.96	43

#: Comparison-based; $: Comparison-free; *: Synthesized upto 48 bits;

of FPGA resources to accommodate a good amount of registers, while [1] requires lower resources than the proposed one. However, [1] and [22] are suitable for sorting UNs and also have limited ability to exploit advantages out of the presence of duplicate elements in the input data set.

Sorting time latency for the two existing comparison-free methods [1], [22] and the proposed comparison-free scheme has been reported in Fig. 15. For a small input size ($\leq$ 256 data elements), apparently, [1], [22], and the proposed one exhibit more or less similar sorting latency (time difference < 1 μs). However, as the data size increases, the proposed sorting engine and [1] reasonably outperform due to large matrix manipulation time involvement in [22]. It is to be noted that the operating frequency is 225 MHz for the proposed one and it is 180 MHz in [22] for sorting 2^{11} elements of 11 bit.

The performance of the proposed sorting architecture has been evaluated considering following three cases.

1) *Case I—Unsigned Numbers:* Test stimuli considered as unique UNs.
2) *Case II—Signed Numbers:* Unique SNs are considered as the test stimuli.
3) *Case III—Signed Numbers With Duplicity (SND):* Sorting of duplicate elements being one of the main objectives of the proposed architecture. Duplicity rates have been chosen as 50%, 75%, and 87.5% in the input set to measure performance of the system. Duplicity rate as 50% indicates, each unique element has another entry in the input list (i.e., 1-for-1 duplicity). Similarly, for every unique element, if there exist another two entries in the list (i.e., 3-for-1 duplicity), denoted as 75% duplicity rate and 7-for-1 duplicity designated as 87.5% duplicity rate.

Fig. 16 shows a time consumption analysis for sorting 256 unsigned and signed data elements with variable widths. Fig. 16(a) and (b) depicts computation and communication time delays, respectively, and Fig. 16(c) depicts total sorting latency. Fig. 17(a)–(c), respectively, reports computation, communication, and total sorting delays for handling 256 data elements with 50%, 75%, and 87.5% duplicity percentages. Furthermore, Fig. 17(c) depicts total sorting delay incurred with respect to theoretical (T) and practical (P) setup. Time analysis portfolio for sorting 512 unsigned and signed data elements has been shown in Figs. 18 and 19. It is worth mentioning that the main driving portion toward total sorting time is the communication latency among the blocks. Since block count remains almost unchanged for unsigned and signed data elements, therefore, communication latency of both cases experiences similar behavior. A differential sorting latency for unsigned and signed elements has been depicted in Fig. 20 to demonstrate sorting delay gap for unsigned and signed elements of 256 and 512 elements.

It is to be noted that *major cycle* and *minor cycle* play key role in handling duplicate elements in the proposed architecture. Time analysis of major and minor cycles are given

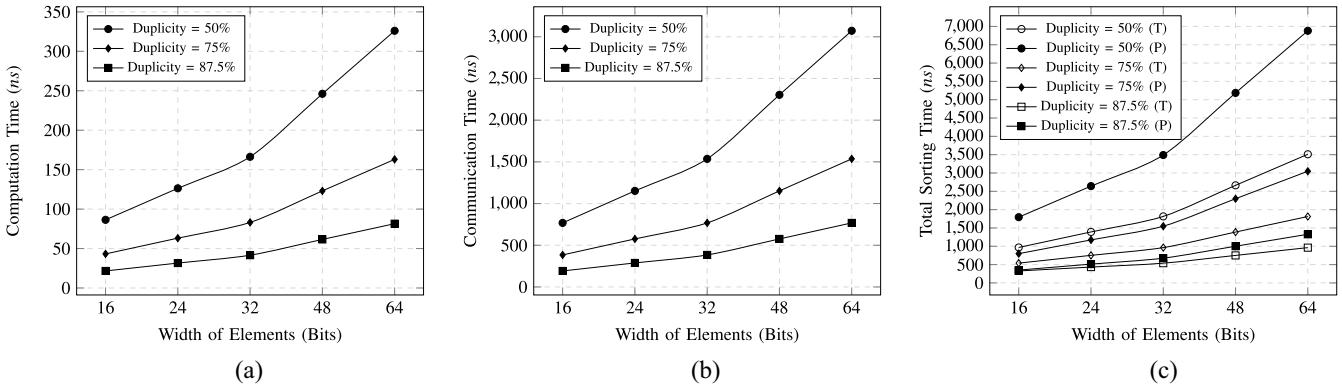

Fig. 17. Sorting time consumption profile to sort 256 SNs of various data widths under different duplicity rates. (a) Computation latency. (b) Communication latency. (c) Total sorting latency.

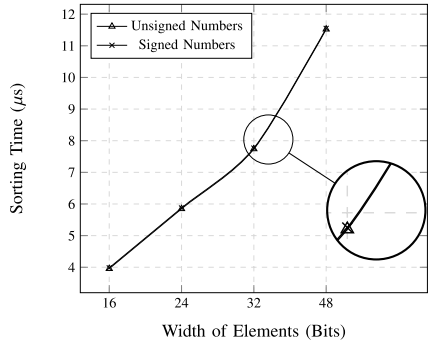

Fig. 18. Sorting latency considering 512 elements.

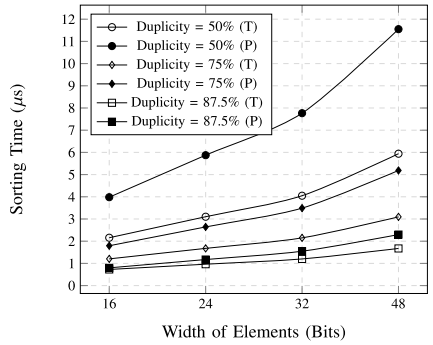

Fig. 19. Sorting latency with different duplicity rates for 512 elements.

TABLE V
MAJOR AND MINOR CYCLES CONSUMED BY THE PROPOSED SCHEME FOR HANDLING SIGNED DUPLICATE ELEMENTS

No. of Elements	Major Cycle Time Period (ns)					Minor Cycle Time Period (ns)
	Element Width					
	16	24	32	48	64	
256	7.00	10.30	13.60	20.20	26.80	0.621
512	7.77	11.40	15.10	22.56	–	0.663
1024	8.50	12.60	–	–	–	0.716

in Table V. We have analyzed power consumption using Xilinx XPower Analyzer for the proposed sorter against existing comparison-free architectures—[1], [22] and some existing state-of-the-art comparison-based sorting architectures—[2], [20], [21], and [25], reported in Fig. 21. Architecture [20] consumes reasonably higher power compared to other architectures. It is to be noted that both [20] and [21] are

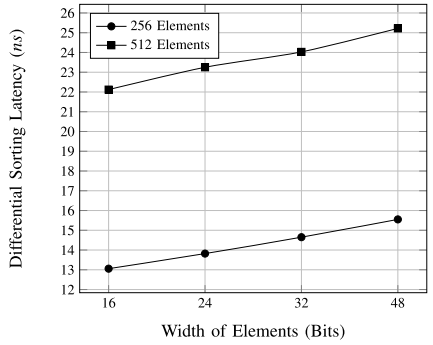

Fig. 20. Differential sorting latency in handling unsigned and signed elements.

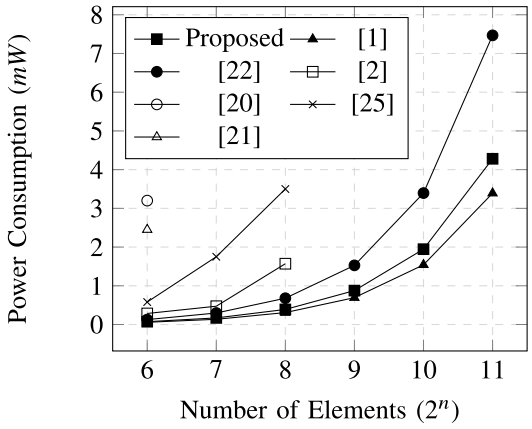

Fig. 21. Power consumption analysis of state-of-the-art architectures for 2^n data elements, where n denotes data width, and $6 \leq n \leq 11$.

quite resource hungry (as per synthesis results reported in Tables III and IV) and could not be synthesized beyond 2^n elements of n-bit (where, $n > 6$). Hence, power analysis of these two approaches could not be measured further. Similarly, approaches [2] and [25] are synthesizable for sorting up to 2^8 elements of 8 bit. Moreover, their power consumption is reasonably higher than the existing [1] and [22] as well as the proposed comparison-free sorting schemes. The differential power consumption between [22] and nonmatrix-based comparison-free approaches ([1] and proposed) increases exponentially with an increase in data

TABLE VI
CRITICAL-PATH DELAY PER BYTE AND POWER CONSUMPTION PER BYTE OF COMPARISON-FREE SORTING SCHEMES FOR 256 DATA ELEMENTS OF 32 BIT

Scheme & Year	Critical Path Delay / Byte (ns)	Power Consumption / Byte (μW)
[22]* 2017	5.402	2.625
[1]* 2019	3.779	1.204
Proposed*$	3.792	1.521
Proposed (SN)$	3.792	1.521
Proposed (SND-75%)	1.704	1.553
Proposed (SND-87.5%)#	0.656	1.983

*: Unsigned; SN: Signed; SND: SN with Duplicity; #: 48-bit;
$: Utilizing same Hardware Structure;

set. Though the comparison-free sorter [1] consumes least power, but it has limited functionality (in terms of efficient handling of duplicate elements and signed elements) than the proposed one. Table VI depicts delay and power consumption of comparison-free sorting schemes for per byte processing. Here, critical paths are shown considering 256 elements of 32 bit. The presence of duplicate elements in input data set results in reduced sorting time, since duplicate elements consume only *minor cycles* instead of *major cycles*. In other words, input data set with larger duplicity rates enables the system to run at higher frequencies. In summary, the proposed comparison-free architecture achieves significant improvement in terms of sorting efficiency and handling data diversity, i.e., unsigned, signed, and data set with duplicate entries.

V. CONCLUSION

A hardware-based comparison-free sorting technique with a worst case sorting time complexity of $O(N)$ has been proposed. The intended sorting engine sorts N unique n-bit data elements (irrespective of both signed and unsigned) consuming a linear sorting delay of $O(N)$ clock cycles. It can also be mentioned that the proposed sorting engine consumes $O(N)$ clock cycles in best case (i.e., already sorted input set) as well as in worst case (i.e., reverse order input set) scenarios. However, the architecture sorts duplicate entries consuming lesser clock cycles based on the duplicity rates. As the proposed sorter forward the data elements across n cascaded stages those involve only a few logic gates and tiny multiplexers (2:1) and therefore, requires reasonably small sorting latency. Due to overlapped operations of sorting and storing operations, the sorting engine eliminates additional memory cycles required for storing the sorted elements. Approximately, the architecture consumes ≈3.8 ns as per-byte processing latency for unsigned and SNs. However, it takes ≈1.7 ns and ≈0.76 ns as per-byte processing latency for SNs with duplicity percentage of 75% and 87.5%, respectively. This architecture consumes ≈1.52 μW power as per-byte processing for unique numbers and ≈1.55 μW for SNs with nonzero duplicity percentage. The experimental results show its minimal hardware resource utilization against some of the recent hardware-based sorting approaches with reasonably small sorting delay.

Our future work includes exploiting spatial parallelism in the proposed structure to further improve system throughput for larger volume of data set. Furthermore, to use this hardware sorting engine as a useful embedded component like an accelerator for data-aware applications. Moreover, to realize this system implementation in ASIC as a low-cost system-on-chip hardware sorter.

REFERENCES

[1] S. Ghosh, S. Dasgupta, and S. S. Ray, "A comparison-free hardware sorting engine," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2019, pp. 586–591.

[2] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Low-cost sorting network circuits using unary processing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 8, pp. 1471–1480, Aug. 2018.

[3] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. D. Riedel, "Computation on stochastic bit streams digital image processing case studies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 3, pp. 449–462, Mar. 2014.

[4] K. Ratnayake and A. Amer, "An FPGA architecture of stable-sorting on a large data volume: Application to video signals," in *Proc. 41st Annu. Conf. Inf. Sci. Syst.*, 2007, pp. 431–436.

[5] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surveys*, vol. 38, no. 3, p. 10, Sep. 2006.

[6] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics co-processor sorting for large database management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 325–336.

[7] Y. Tang and N. W. Bergmann, "A hardware scheduler based on task queues for FPGA-based embedded real-time systems," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1254–1267, May 2015.

[8] A. A. Colavita, A. Cicuttin, F. Fratnik, and G. Capello, "SORTCHIP: A VLSI implementation of a hardware algorithm for continuous data sorting," *IEEE J. Solid-State Circuits*, vol. 38, no. 6, pp. 1076–1079, Jun. 2003.

[9] R. C.-H. Chang *et al.*, "Implementation of a high-throughput modified merge sort in MIMO detection systems," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 9, pp. 2730–2737, Sep. 2014.

[10] M. Mahdavi and M. Shabany, "Novel MIMO detection algorithm for high-order constellations in the complex domain," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 5, pp. 834–847, May 2013.

[11] N. Faujdar and S. P. Ghrera, "A practical approach of GPU bubble sort with CUDA hardware," in *Proc. 7th Int. Conf. Cloud Comput. Data Sci. Eng. Confluence*, 2017, pp. 7–12.

[12] D. S. Banerjee, P. Sakurikar, and K. Kothapalli, "Fast, scalable parallel comparison sort on hybrid multicore architectures," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2013, pp. 1060–1069.

[13] L. Kohútka and V. Stopjaková, "Rocket queue: New data sorting architecture for real-time systems," in *Proc. IEEE 20th Int. Symp. Design Diagnost. Electron. Circuits Syst. (DDECS)*, 2017, pp. 207–212.

[14] T. Usui, T. V. Chu, and K. Kise, "A cost-effective and scalable merge sorter tree on FPGAs," in *Proc. 4th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2016, pp. 47–56.

[15] N. Matsumoto, K. Nakano, and Y. Ito, "Optimal parallel hardware K-sorter and top K-sorter, with FPGA implementations," in *Proc. 14th Int. Symp. Parallel Distrib. Comput.*, Jun. 2015, pp. 138–147.

[16] A. Farmahini-Farahani, H. J. Duwe, III, M. J. Schulte, and K. Compton, "Modular design of high-throughput, low-latency sorting units," *IEEE Trans. Comput.*, vol. 62, no. 7, pp. 1389–1402, Jul. 2013.

[17] S.-H. Lin, P.-Y. Chen, and Y.-N. Lin, "Hardware design of low-power high-throughput sorting unit," *IEEE Trans. Comput.*, vol. 66, no. 8, pp. 1383–1395, Aug. 2017.

[18] S. Olarlu, M. C. Pinotti, and S. Q. Zheng, "An optimal hardware-algorithm for sorting using a fixed-size parallel sorting device," *IEEE Trans. Comput.*, vol. 49, no. 12, pp. 1310–1324, Dec. 2000.

[19] A. Rjabov, "Hardware-based systems for partial sorting of streaming data," in *Proc. 15th Biennial Baltic Electron. Conf. (BEC)*, Oct. 2016, pp. 59–62.

[20] S. Mashimo, T. V. Chu, and K. Kise, "High-performance hardware merge sorter," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 1–8.

[21] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *Proc. IEEE 24th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, May 2016, pp. 95–102.

[22] S. Abdel-Hafeez and A. Gordon-Ross, "An efficient *O*(*N*) comparison-free sorting algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 6, pp. 1930–1942, Jun. 2017.
[23] S. Abdel-Hafeez, A. Gordon-Ross, and S. Abubaker, "A comparison-free sorting algorithm on CPUs and GPUs," *J. Supercomput.*, vol. 74, no. 11, pp. 6369–6400, 2018.
[24] E. A. Elsayed and K. Kise, "Towards an efficient hardware architecture for odd-even based merge sorter," in *Proc. IEEE 13th Int. Symp. Embedded Multicore/Many-Core Syst.-on-Chip (MCSoC)*, 2019, pp. 249–256.
[25] A. Norollah, D. Derafshi, H. Beitollahi, and M. Fazeli, "RTHS: A low-cost high-performance real-time hardware sorter, using a multidimensional sorting algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 7, pp. 1601–1613, Jul. 2019.
[26] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2014, pp. 151–160.
[27] V. A. Pedroni, R. P. Jasinski, and R. U. Pedroni, "Panning sorter: A minimal-size architecture for hardware implementation of 2D data sorting coprocessors," in *Proc. IEEE Asia Pac. Conf. Circuits Syst.*, Dec. 2010, pp. 923–926.
[28] V. A. Pedroni, R. P. Jasinski, and R. U. Pedroni, "Panning sorter: An approach to the design of minimal-hardware parallel-input data sorters," *Electron. Lett.*, vol. 46, no. 18, pp. 1262–1263, 2010.
[29] W. Chen, W. Li, and F. Yu, "A hybrid pipelined architecture for high performance top-K sorting on FPGA," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 8, pp. 1449–1453, Aug. 2020.
[30] D. Yan, W.-X. Wang, L. Zuo, and X.-W. Zhang, "A novel scheme for real-time max/min-set-selection sorters on FPGA," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 7, pp. 2665–2669, Jul. 2021.

Sanchita Saha Ray received the B.Tech. degree in electronics and communication engineering from the Sikkim Manipal Institute of Technology, Majitar, India, and the M.E. degree in computer science and engineering from West Bengal University of Technology, Kolkata, India.

She is currently working as an Assistant Professor with the St. Thomas' College of Engineering and Technology, Kolkata. Her research area includes network packet processing, HPC for biological sequences, and high performance computer architecture.

Ms. Saha Ray has received the Bronze Medal in M.E. (CSE), the IETE-Gowri Memorial Award 2008, and the Best Paper Award in IEEE ICCIC 2013. She is a member of the IETE (India).

Dulal Adak received the M.C.A. degree from Vidyasagar University, Midnapore, India, in 2016, and the M.Tech. degree in computer science and engineering from the Maulana Abul Kalam Azad University of Technology, Kolkata, India, in 2019.

He is currently working as a Junior Research Fellow with the Department of Science and Technology and Bio-Technology, Government of West Bengal Sponsored Project in the Department of Computer Science and Technology, Indian Institute of Engineering Science and Technology, Shibpur, Howrah, India. His research interests include cryptography, fuzzy logic, and FPGA-based system design.

Surajeet Ghosh (Member, IEEE) received the B.Tech. degree in computer science and technology from Kalyani University, Kalyani, India, the M.E. degree in computer science and engineering from the West Bengal University of Technology, Kolkata, India, and the Ph.D. degree in computer science and engineering from Jadavpur University, Kolkata.

He is currently serving as an Associate Professor with the Department of Computer Science and Technology, Indian Institute of Engineering Science and Technology, Shibpur, Howrah, India. His current research interests are in computational architecture for next-generation sequencing, hardware architecture for network routing schemes, Internet of Things, embedded systems, and FPGA-based system design.

Dr. Ghosh received IETE-Gowri Memorial Award 2008 and the Best Paper Awards at IEEE ICCIC 2013 and at International Conference on Advanced Computing 2009. He was associated with several international conferences as a Technical Program Committee Member, including IEEE flagship conferences. He served as a Session Chairman in various conferences, viz., IEEE ISVLSI, IEEE ANTS, and IEEE TENCON. He is a Fellow of the IETE (India).