An Efficient O(N) Comparison-Free Sorting Algorithm

Saleh Abdel-Hafeez, *Member, IEEE*, and Ann Gordon-Ross, *Member, IEEE*

***Abstract*— In this paper, we propose a novel sorting algorithm that sorts input data integer elements on-the-fly without any comparison operations between the data—comparison-free sorting. We present a complete hardware structure, associated timing diagrams, and a formal mathematical proof, which show an overall sorting time, in terms of clock cycles, that is linearly proportional to the number of inputs, giving a speed complexity on the order of O(N). Our hardware-based sorting algorithm precludes the need for SRAM-based memory or complex circuitry, such as pipelining structures, but rather uses simple registers to hold the binary elements and the elements' associated number of occurrences in the input set, and uses matrix-mapping operations to perform the sorting process. Thus, the total transistor count complexity is on the order of O(N). We evaluate an application-specified integrated circuit design of our sorting algorithm for a sample sorting of N = 1024 elements of size K = 10-bit using 90-nm Taiwan Semiconductor Manufacturing Company (TSMC) technology with a 1 V power supply. Results verify that our sorting requires approximately 4–6 μs to sort the 1024 elements with a clock cycle time of 0.5 GHz, consumes 1.6 mW of power, and has a total transistor count of less than 750 000.**

***Index Terms*— 90-nm TSMC, comparison free, Gigahertz clock cycle, one-hot weight representation, sorting algorithms, SRAM, speed complexity *O(N)*.**

I. Introduction, Motivation, and Related Work

SORTING algorithms have been widely researched for decades [1]–[6] due to the ubiquitous need for sorting in many application domains [7]–[10]. Sorting algorithms have been specialized for particular sorting requirements/situations, such as large computations for processing data [11], high-speed sorting [12], improving memory performance [13], sorting using a single CPU [14], exploiting the parallelism of multiple CPUs [15], parallel processing for grid-computing in order to leverage the CPU's powerful computing resources for big data processing [16].

Manuscript received July 6, 2016; revised October 22, 2016 and January 16, 2017; accepted January 16, 2017. Date of publication February 22, 2017; date of current version May 22, 2017. This work was supported in part by the National Science Foundation under Grant CNS-0953447 and in part by Nvidia and Synopsys. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

S. Abdel-Hafeez is with Jordan University of Science and Technology, Irbid 22110, Jordan (e-mail: sabdel@just.edu.jo).

A. Gordon-Ross is with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA and also with the National Science Foundation Center for High-Performance Reconfigurable Computing, University of Florida, Gainesville, FL 32611 USA (e-mail: ann@ece.ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2661746

Due to the ever-increasing computational power of parrallel processing on many core CPU- and GPU-based processing systems, much research has focused on harnessing the computational power of these resources for efficient sorting [17]–[20]. However, since not all computing domains and sorting applications can leverage the high throughput of these systems, there is still a great need for novel and transformative sorting methods. Additionally, there is no clear dominate sorting algorithm due to many factors [21]–[24], including the algorithm's percentage utilization of the available CPU/GPU resources, the specific data type being sorted, amount of data being sorted.

To address these challenges, much research has focused on architecting customized hardware designs for sorting algorithms in order to fully utilize the hardware resources and provide custom, cost-effective hardware processing [2]–[27]. However, due to the inherent complexity of the sorting algorithms, efficient hardware implementation is challenging. To realize fast and power-efficient hardware sorting, a significant amount of hardware resources are required, including, but not limited to, comparators, memory elements, large global memories, and complex pipelining, in addition to complicated local and global control units.

Most prior work on hardware sorting designs are implemented based on some modification of traditional mathematical algorithms [28]–[31], or are based on some modified network of switching structures [32]–[34] with partially parallel computing processing and pipelining stages. In these sorting architectures, comparison units are essential components that are characterized by high-power consumption and feedback control logic delays. These sorting methods iteratively move data between comparison units and local memories, requiring wide, high-speed data buses, involving numerous shift, swap, comparison, and store/fetch operations, and have complicated control logic, all of which do not scale well and may need specialization for certain data-type particulars. Due to the inherent mixture of data processing and control logic within the sorting structure's processing elements, designing these structures can be cumbersome, imposing large design costs in terms of area, power, and processing time. Furthermore, these structures are not inherently scalable due to the complexity of integrating and combining the data path and control logic within the processing units, thus potentially requiring a full redesign for different data sizes, as well as complex connective wiring with high fan-out and fan-in in addition to coupling effects, thus circuit timing issues are challenging to address. Additionally, if multiple processors are used along with pipelining stages

and global memories, the data must be globally merged from these stages to output the complete final sorted data set [35], [36].

To address these challenges, in this paper, we propose a new sorting algorithm targeted for custom, IC-designed applications that sort small- to moderate-sized input sets, such as graphics accelerators, network routers, and video processing DSP chips [12], [33], [44], [46]. For example, graphics processing uses a painter unit that renders objects according to the object's depth value such that the object can be displayed in the correct order on the screen. In video processing, fast computation is required for small matrices in a frame in order to increase the resolution using digital filters that leverage sorting algorithms. Even though we present our design based on these scenarios, our design also supports processing large input sets by subsequently processing the data in multiple, smaller input sets (i.e., in sets of $N < 100\ 000$) using fast computations, and then merging these sets. However, since applications with larger input sets (on the order of millions) are usually embedded into systems with large computational resources, such as data mining and database visualization applications running on high-performance grid computing and GPU accelerators [17]–[20], these applications can harness those powerful resources for sorting.

Our sorting algorithm's main features and contributions include as follows.

1) Our design affords continuous sorting of input element sets, where each set can hold any type and distribution (ordering) of data elements. Sorting is triggered with a start-sort signal and sorting ends when a done-sorting signal is asserted by the design, which subsequently begins sorting the next input set, thus affording continuous, end-to-end sorting.
2) Our sorting design does not require any ALU-comparisons/shifting-swapping, complex circuitry, or SRAM-based memory, and processes data in a forward moving direction through the circuit. Our design's simplicity results in a highly linearized sorting method with a CMOS transistor count that grows on the order of O(N). Hence, the design provides low and efficient power components with the addition of regularity and scalability as key structure features, which provide easily and quick miagration to embedded micro-controllers and field-programmable gate arrays (FPGAs).
3) The sorting delay time is always linearly proportional to the number of input data elements N, with upper and lower bounds of $3N$ and $2N$ clock cycles, respectively, giving a linear sorting delay time of $O(N)$. This sorting time is independent of the input elements' ordering or repitition since the design always performs the same operations within these bounds as opposed to Quicksort and othersorting algorithms, which have large and nonlinear margin of bounds.

The remainder of this paper is organized as follows. Section II summarizes related works and the works' cost-performance bottleneck tendencies. Section III discusses our proposed comparison-free sorting algorithm with illustrative examples and Section IV provides a mathematical analysis. Section V details the hardware data path and control logic implementations along with timing diagrams. Section VI presents our simulation results, and Section VII discusses our conclusions, which elaborate on the overall results and our design's hardware advantages.

II. Related Work

In order to provide high scalability, it is critical to design a sorting method with timing and circuit complexity that scales linearly with the number of input elements N [i.e., the circuit timing delay and circuit complexity are on the complexity order of O(N)]. Although some recent works showed linear scalability, these works' O(N) notations hide a large scalar value [4], [27], [32], [34] and these methods have expensive circuit complexity with respect to multiprocessing, local and global memories, pipelining, and control units with special instruction sets, in addition to high-cost technology power factors.

Other recent works [2], [25], [37]–[42] divide the sorting algorithm design into smaller computation partitions, where each partition integrates control logic and the partition's comparison operations with feedback decisions from neighboring partitions. A global control unit coordinates this control to streamline the data flow between the partitions and the partitions' associated memories to store temporary data that is transferred between partitions. In addition to the complex circuitry required to maintain inter-partition connectivity and redundant intra-partition control circuitry, a complex global memory organization is required.

Alternative methods [43]–[45] attempt to eliminate comparators by introducing a rank (sorted) ordering approach. In [43], a bit-serial sorter architecture was implemented based on a rank-order filter (ROF), but comparators were still used to transform the programmable capacitive threshold logic (CTL) to a majority voting decision. That design used large array cells of ROF and CTL decisions with a pipelined architecture. The design in [44] counted the number of occurrences of every element in the unsorted input array, where the rank of each element was determined by counting the number of elements less than or equal to the element being considered. Thus, the comparison units were replaced by counting units with bit comparison. However, the design required a complicated hardware structure with pipelining and a histogram counting sequence. Alternatively, the design in [45] used a rank matrix that assigned relative ranks to the input elements, where the highest element had the maximum rank and the lowest element had the lowest rank of 1. The rank matrix was updated based on the value of a particular bit in each of the N input elements, starting with the most-significant bit. This bit-wise inspection required inspecting a complete column of the rank matrix in order for the lower ranks to update the higher ranks. However, that design could not be used when the number of elements was less than the elements' bit-width.

Some recent works [47]–[49] leverage previous works and integrate several different sorting architectures for different

requirements, such as speed, area, power. The work in [47] leveraged a bitonic sorting network to more efficiently map the methodology considering energy and memory overheads for FPGA devices. Further advances of that work [48] presented novel and improved cost-performance tradeoffs, as well as identification of some Pareto optimal solutions trading off energy and memory overheads. Additional work [49] developed a framework that composes basic sorting architectures to generate a cost-efficient hybrid sorting architecture, which enabled fast hardware generation customized for heterogeneous FPGA/CPU systems.

Even though all of these designs reported linear sorting delay times as the number of input elements increased, the authors did not include the initialization times for the required arrays/matrices, nor was the worst case sorting time evaluated. Furthermore, each design either required arrays to store the input elements, associated arrays for the rank operations and data routing, or had to globally merge the intermediate sorted array partitions. These array elements required a significant amount of local and global input–output data routing, SRAM-based memory, and control signals, where the local control logic communicated with each processing unit partition and the global control unit. This layout complicates adapting the design to different input data bit-widths. Additionally, since the control signals and data path wiring was intertwined, circuit design bugs were challenging to locate, in turn leading to high-cost design.

III. Comparison-Free Sorting Algorithm

The input to our sorting algorithm is a K-bit binary bus, which enables sorting $N = 2^K$ input data elements. The sorting algorithm operates on the element's one-hot weight representation, which is a unique count weight associated with each of the N elements. For example, "5" has a binary representation of "101," which has a one-hot weight representation of "1 00 000." For a complete set of $N = 2^K$ data elements, the one-hot weight representation's bit-width H is equal to the number of possible unique input elements. For example, a $K = 3$-bit input bus can sort/represent $N = 8$ elements, so each element's one-hot weight representation is of size $H = 8$-bit (i.e., $H = N$). The binary to one-hot weight representation conversion is a simple transformation using a conventional one-hot decoder. Using this one-hot weight representation method ensures that different elements are orthogonal with respect to each other when projected into an R^n linear space.

For brevity of discussion and ease of understanding our sorting method's mathematical functionality, we illustrate a small example in Fig. 1, which is based on linear algebra vector computations. This example shows our sorting algorithm's functionality using four 2-bit input data elements, with an initial (random and arbitrary) sequential ordering of [2; 0; 3; 1], which generates the outputted elements in the sorted matrix = [3; 2; 1; 0]. This sorting matrix is in descending order; however, the elements can also be represented in ascending order by having the mapping go from the bottom row to the upper row.

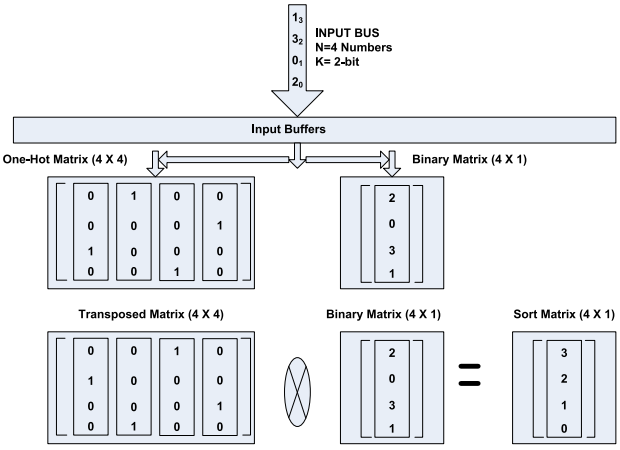

Fig. 1. Comparison-free sorting example using four 2-bit input data elements.

This example operates as follows. The inputted elements are inserted into a binary matrix of size $N \times 1$, where each element is of size k-bit (in this example $N = 4$ and $k = 2$ bit). Concurrently, the inputted elements are converted to a one-hot weight representation and stored into a one-hot matrix of size $N \times H$, where each stored element is of size H-bit and H=Ngiving a one-hot matrix of size N-bit $\times N$-bit. The one-hot matrix is transposed to a transpose matrix of size $N \times N$, which is multiplied by the binary matrix—rather than using comparison operations—to produce the sorted matrix. For repeated elements in the input set, the one-hot transpose matrix stores multiple "1s" (equal to the number of occurances of the repeated element in the input set) in the element's associated row, where each "1" in the row maps to identical elements in the binary matrix, an advantage that will be exploited in the hardware design (Section V). For example, if the input set matrix is [2; 0; 2; 1], then the transpose matrix is [0 0 0 0; 1 0 1 0; 0 0 0 1; 0 1 0 0]. Notice that the second row contains two "1s," such that when the transpose matrix is multiplied by the second row in the binary matrix, both "1" occurances in the transpose matrix are mapped to the "2" in the binary matrix. Therefore, the multiply operation can be simply replaced with a mapping function using a tri-state buffer (Section V). Additionally, the first row in the transpose matrix has no element in the first position (i.e., element 3 is not in the binary matrix since 3 is not in the input set). The absence of this element can be recorded using a counting register for each inputted element (Section V), and this register records the number of occurences of this element in the binary matrix, which in this case would be "0" for element 3.

For more insight on this algorithm, Fig. 2 shows C-code for a single-threaded implementation on a single CPU, where the transpose matrix is used as a vector matrix instead of a 2-D matrix such that the indices of the $TM_{N\times1}$ matrix record the counting elements of size $N\times1$. Hence, the initialization phase, which is structured in the first loop, requires less memory access time for the reads and writes in the loop body. The evaluation phase is conducted in the second loop, and in this phase, the elements are sorted and stored in the sorted vector $SS_{N\times1}$. The elements in the array vector $TM_{N\times1}$ are read sequentially, and concurrently the elements in the

```
1.  Input:  Integer Element BS[0 : n - 1 ]
2.  Output: Integer Sort SS[0 : n - 1 ]
3.  One-Hot Weight: char TM[0:n-1] initialize to zero
4.  for i = 0 to n  do
5.        TM[BS[i]] ←TM[BS[i] ] + 1
6.    endfor
7.  Z ←0
8.  for i = 0 to n  do
9.        if TM[i]>0  then
10.                SS[Z] ←i
11.                Z←Z+1
12.                TM[i] ←TM[i]-1
13.                i←i–1
14.         endif
15.  endif
```

Fig. 2. Comparison-free sorting C-code for a single-threaded single CPU.

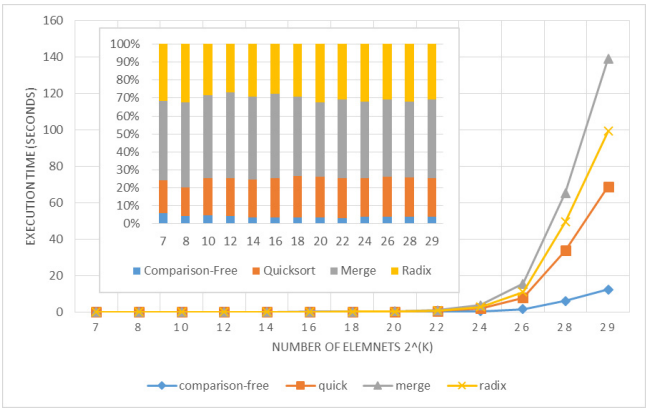

Fig. 3. Execution comparisons of for our comparison-free sorting design, Quicksort, merge sort, and radix sort.

sorted vector $SS_{N\times 1}$ are written sequentially, resulting in good spatial locality in the second loop of the C-code. Due to these structural designs, initial insight in our simulation results for a single-threaded single CPU, which is shown in Fig. 3, reveal the advantages of our proposed algorithm in execution time over other popular sorting algorithms such as Quicksort and other standard sorting algorithms reported in [50]

IV. Mathematical Analysis

In this section, we provide the mathematical proof for our sorting algorithm illustrating the case of N unique input elements as a proof of concept. We present this case as the base case proof for our sorting algorithm since other input element set cases (i.e., different numbers of duplicated elements) can be easily derived from this case.

Let

$$L = [a(1), \ldots, a(k)] \tag{1}$$

be a given list[1] of k positive integers and let

$$M = \max[a(1), \ldots, a(k)]. \tag{2}$$

Let $J = J_L$ be the (k x M) matrix whose entries $J_{r,s}$ are defined by

$$J_{r,s} = \begin{cases} 1, & \text{if } a(r) = s \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

[1]A list is a set in which repetition is allowed.

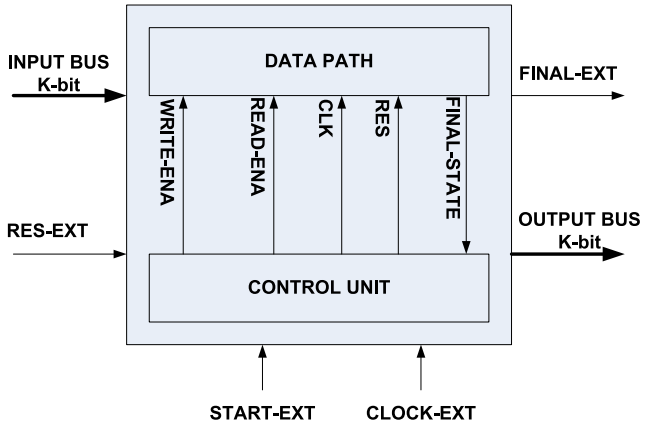

Fig. 4. Block diagram of the hardware structure for our sorting algorithm.

Thus, if s does not belong to L (i.e., there is no r such that $a(r) = s$), then the sth column of J will contain all "0s." If s belongs to L, then the sth column of J will have "1s" in exactly the locations r where $a(r) = s$.

Supposing that L had no repetitions, let

$$\begin{aligned} LJ &= [a(1), \ldots, a(k)] \\ J &= [b(1), \ldots, b(m)] \end{aligned} \tag{4}$$

which gives

$$b(\mathrm{s}) = \begin{cases} \mathrm{s}, & \text{if } s \in L \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

If $s \notin L$, then all of the values in the sth column of C_s of J are "0s," and $b(s) = L \cdot C_s^T = 0$. If $s \in L$, and if r is the unique value for which $a(r) = s$, then all of the values in the sth column of C_s of J are all "0" except for the value in the rth column, which is "1." Therefore, $b(s) = L \bullet C_s^T = a(r) = s$, which proves our claim.

For example, starting with $L = [6, 3, 4]$, then $J = J_L$would be the matrix

$$J = \begin{bmatrix} 0\,0\,0\,0\,0\,1 \\ 0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0 \end{bmatrix} \tag{6}$$

and

$$LJ = [0, 0, 3, 4, 0, 6]. \tag{7}$$

Let J^* be the matrix obtained by deleting the zero columns from J such that

$$LJ^* = [3, 4, 6]. \tag{8}$$

V. Hardware Functionality Details

The overall hardware structure for our sorting algorithm is divided into two parts: the data path and the control unit. Fig. 4 depicts the input–output signals of a complete block diagram for our sorting algorithm, which sorts of $N = 2^K$ input data elements. The basic design architecture operates in two sequential phases: the write-evaluate phase (Section V-A1) followed by the read-sort phase (Section V-A2). The control unit (Section V-B), is a simple state machine that controls the

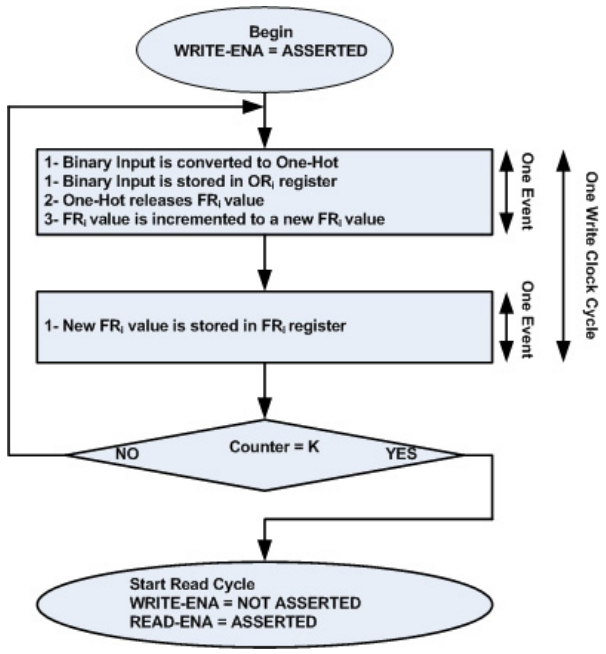

Fig. 5. Hardware flow for the write-evaluate phase.

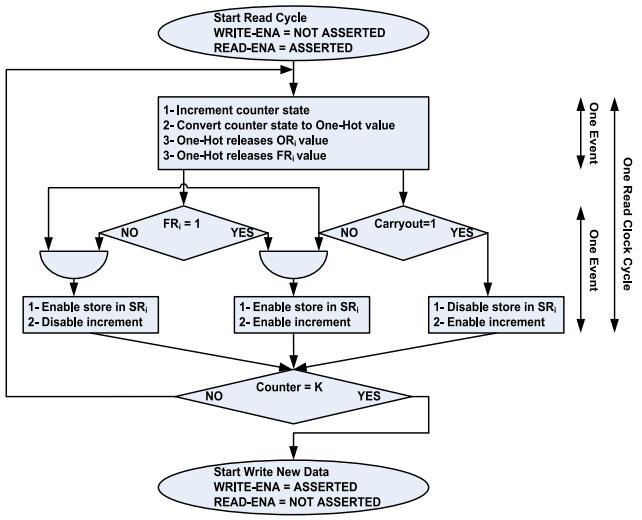

Fig. 6. Hardware chart for read-sort phase.

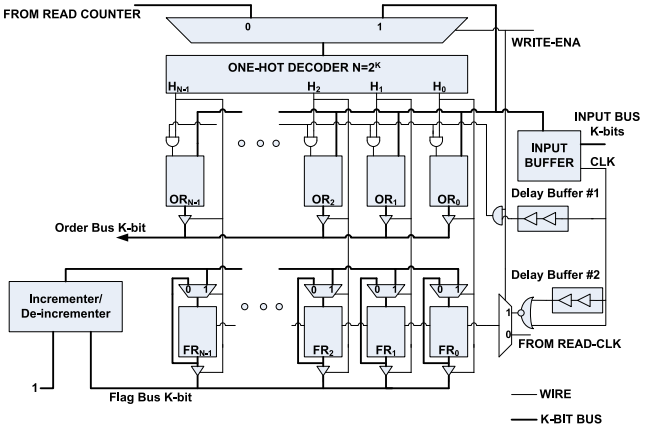

Fig. 7. Detailed block diagram of our sorting algorithm's write-evaluate phase.

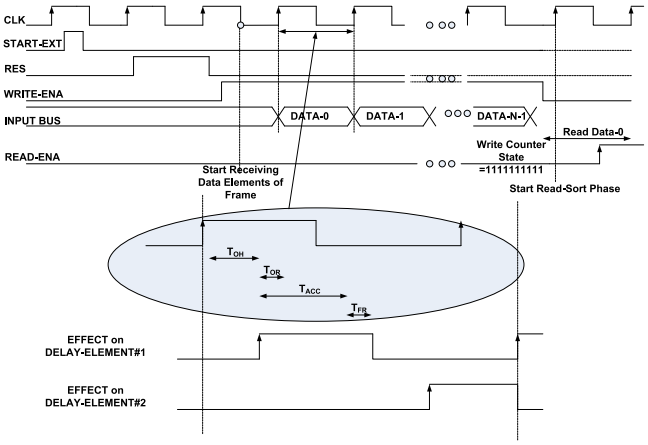

Fig. 8. Timing diagraph for our sorting algorithm's write-evaluate phase.

data path's phases using only a few D-type flip-flop (DFF) components. Sorting begins when the *START-EXT* signal is asserted and the design signals that sorting has completed by asserting the *FINAL-EXT* signal.

A. Data Path Operation

The data path contains several circuit components: a one-hot decoder, register arrays, a serial shifter, a parallel counter (*PC*), tri-state buffers and multiplexors, a one-detector, and an incrementor/decrementor circuit. In order to meet the setup-hold delay time bewteen the clock and data stabilization for the elements' storage registers, the delay element's components are a cascade of an even number of inverters. These circuit components are standard CMOS circuit components [51]–[53], which are commonly used components for advanced CMOS technologies beyond 90 nm, making our design scalable for further advanced low-cost CMOS technologies.

Before proceeding with a more detailed circuit structure of the write-evaluate and read-sort phases, we present generalized and overall illustrations for these phases in the flow charts in Figs. 5 and 6, respectively. The rectangles present the operations during each clock cycle event, in which two events occur per clock cycle, one on each cycle edge (i.e., asserted high and low). The steps within the rectangles show the sequences of the operations based on the data hardware flow shown in Figs. 7 and 9, where some operations have the same number indicating parallelism/independence between these operations within the clock cycle, meaning that it does not matter which operation occurs first. Additionally, these flow charts adhere to the timing constraints depicted in Figs. 8 and 10, respectively, where each event occurs at a clock edge. The diamonds are the condition expressions that change the data flow based on control flow events.

1) Write-Evaluate Phase: During the write-evaluate phase, each binary input element is converted to the element's one-hot weight representation by the one-hot decoder. The decoder's output enables an associated register in a register array to record the binary input element's occurrence. We refer to this register as an order register (OR_i) array, where the ith register stores the ith input element. Each register is a

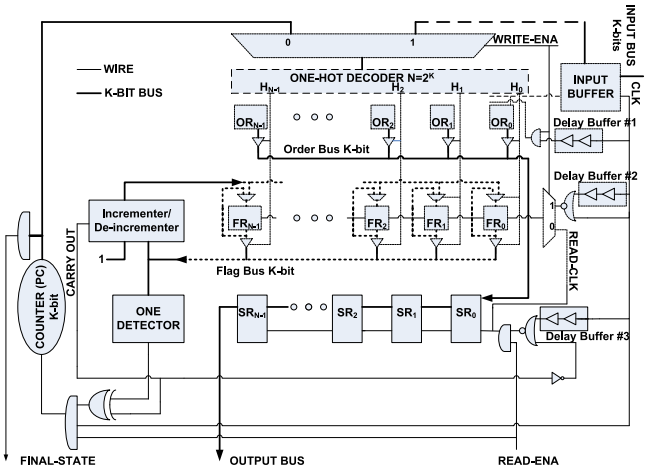

Fig. 9. Detailed block diagram of our sorting algorithm's read-sort phase.

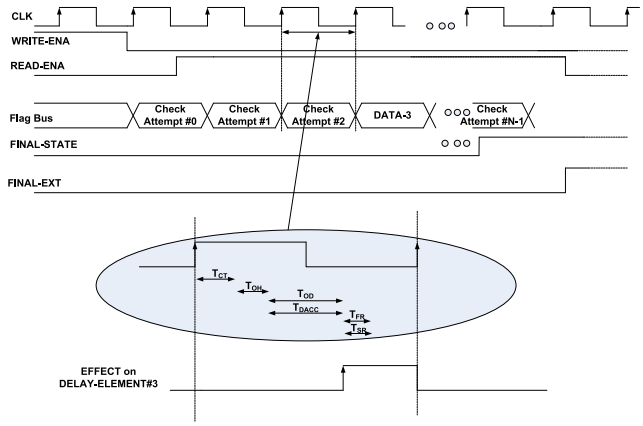

Fig. 10. Timing diagraph for our sorting algorithm's read-sort phase.

simple DFF register of size k-bit. This operation is equivalent to the recording of the element in the transposed matrix in our algorithm (Section III). Simultaneously, the one-hot decoder enables an associated register in another register array—the flag register (FR_i) array—which records the number of occurrences of this element in the input set. For each occurrence of a duplicated element, the associated flag register is triggered, and the occurrence is recorded by incrementing the register's stored value using a 10-bit incrementor. This operation is equivalent to having multiple "1s" for repeated elements in a row in the transpose matrix (Section III).

All input elements follow the same sequential operation at every rising clock edge. Fig. 7 illustrates a detailed block diagram of the write-evaluate phase's data path, which shows the input bus and all control signals that are fed from the control unit (Section V-B). Fig. 8 depicts the associated timing diagram, which shows the detailed streamlined sequential timing for the write-evaluate phase. In this diagram, the *START-EXT* signal indicates the beginning of a new block of $N = 2^K k$-bit input elements, which arrive sequentially on each clock cycle. The *START-EXT* signal consecutively triggers several intermediate signals in the write-evaluate data path's circuit. First, the reset signal *RES* is asserted high for one clock cycle to initialize all registers (omitted from Fig. 7 for figure clarity). Next, the *WRITE-ENA* signal is used to direct the input data to the one-hot decoder, and enable the clocking source for the order and flag register arrays, which are actually gated by another AND-gate that comes from the one-hot decoder.

Following the timing diagram in Fig. 8, the write-evaluate cycle time requires time for the one-hot decoder (T_{oh}), time for the order and flag registers' access times, (T_{or}) and (T_{fr}), respectively, and time for the flag register increment (T_{acc}). The total write-evaluate phase's cycle time ($T_{\text{write}-\text{cycle}}$) is

$$T_{\text{write-cycle}} = T_{\text{oh}} + T_{\text{or}} + T_{\text{acc}} + T_{\text{fr}}. \tag{9}$$

The delay element's components have no influence on the write-evaluate cycle time since these components only change the duty cycle while preserving the cycle time. All of the registers (order and flag) are structured in parallel, such that the access times to the registers are on the order of fractions of a nano-second. Additionally, the simple incrementor is less than a nano-second time scale since the bit-width is only k-bits. One incrementor is shared for all flag registers since only one element is input per clock cycle.

A parallel counter in the control unit (Section V-B) controls the end of the write-evaluate phase when the counter's value reaches the maximum number possible inputted elements (i.e., $N = 2^{\text{k}}$). Even though the input set may contain less than the maximum number of elements, assuming that the input set is full realizes the simplisity of the read-sort phase's operation. The control unit asserts the *READ-ENA* signal and deasserts the *WRITE-ENA* signal when the write-evaluate phase completes, which enables the read-sort phase on the next clock edge. The write-evaluate phase requires a fixed N clock cycles since the phase always iterates for the maximum number of potential input elements.

2) Read-Sort Phase: Fig. 9 illustrates a detailed block diagram of the read-sort phase's data path, which comprises of a k-bit sorted shift register (SR_i) array of size N that stores the elements in their final sorted order, and a k-bit PC that indexes into the order register array to process each element in turn. The element ordering, ascending or descending, is user-specified, and can be controlled by either left- or right-shifting in the elements. A one-detector circuit detects if the flag register value is "1" or not, and a decrementor circuit subtracts a "1" from the flag register, the result of which is stored back in to the flag register, when processing replicated elements. In this figure, the write-evaluate phase's data path components that are used in the read-sort phase are encompassed in the dashed lines.

The read-sort phase begins after the *WRITE-ENA* signal is deasserted and the *READ-ENA* signal is asserted, which sends the *PC*'s value to the one-hot decoder at each new read-sort clock cycle. The one-hot decoder converts this counter value to the value's one-hot representation, which enables the associated order and flag registers to read/release the registers' values, and the order register's value is stored into the sorted register array if-and-only-if that element's flag register value is greater than "0," meaning there was at least one occurrence of that input element. The one-detector evaluates the flag register value to control whether or not the element is stored

in the sorted register array. If the flag register records a value equal to or greater than "1," the associated element should be stored in the sorted register array a number of times equal to the flag register's value. The case is simple when the flag register value is "1," which is detected by the one-detector. To avoid complex comparison units (i.e., equal to or greater than "1"), detecting values greater than "1" can be easily determined using the decrementor's carry out single. Thus, if the one-detector's evaluation is false (i.e., "0" is the one-detector's decision output), but when decrementing the flag register's value, the resulting carry out flag is "0," this means that the flag register's value was greater than "1." In both cases, the input element should be stored into the sorted register array. Indexing to the next input element is inhibited by disabling the *PC*'s increment, which allows the replicated element to be stored in the sorted register array until the flag register value reaches "0." Otherwise, the flag register's value is "0," the element is not in the input set, and thus is not stored into the sorted register array, and the *PC* is incremented.

The read-sort cycle time can be divided into three cases based on the flag register's value. For clarity, these cases will be described with references to the example in Fig. 1 and the discussion of the structure in Section III. In case one, the flag register's value is "0" (i.e., the element is not in the binary matrix), and thus, this element is not stored in the sorted register array, and the *PC* is incremented (i.e., proceed to the next row in the transpose matrix). The timing of the read-sort cycle ($T_{\text{read}-\text{cycle}}$) in case one is the sum of the *PC*'s increment (T_{PC}), the one-hot decoder's (T_{OH}), and the one-detector's (T_{OD}) delays

$$T_{\text{read}-\text{cycle}} = T_{\text{PC}} + T_{\text{OH}} + T_{\text{OD}}. \tag{10}$$

We can see that the one-detector and decrementor both operate concurrently with the flag register value's evaluation.

In case two, the flag register's value is "1," meaning that the element is in the input set once, and thus this element is read from the order register using the one-hot decoder and a tri-state buffer at the register's output, the element is stored in the sorted register array, and the *PC* is incremented. As with case one, a flag register value of "0" and "1" both require one clock cycle. The timing of the read-sort cycle ($T_{read-cycle}$) in this case is the sum of the *PC*'s increment (T_{PC}), the one-hot decoder's (T_{OH}), the one-detector's (T_{OD}), and the sorted register array's (T_{SR}) delays

$$T_{\text{read}-\text{cycle}} = T_{\text{PC}} + T_{\text{OH}} + T_{\text{OD}} + T_{\text{SR}}. \tag{11}$$

In case three, the flag register's value is greater than "1" (i.e., the element's corresponding row in the transpose matrix contains more than one "1"). Similar to case two, this element is stored into the sorted register array, but in this case, the flag register is also decremented. The *PC*'s increment is disabled until the element's flag register reaches "1," signaling that all occurrences of the element have been stored into the sorted output array. The timing of the read-sort cycle ($T_{\text{read}-\text{cycle}}$) in this case is the sum of the *PC*'s increment (T_{PC}), the one-hot decoder's (T_{OH}), the decrementor's (T_{DA}), and the flag register array's (T_{FR}) delays

$$T_{\text{read-cycle}} = T_{\text{PC}} + T_{\text{OH}} + T_{\text{DA}} + T_{\text{FR}}. \tag{12}$$

Fig. 10 shows the timing diagram for the read-sort phase for all three cases, where the circled area shows the clock cycle operations for case two and three. Case three is assumed to be the worst case due to the decrementor's delay, which has more delay than the one-detector delay (T_{OD}) as given in case 2.

The additional required logic gates' delays, such as the XOR gate, tri-state buffer, and AND gates, are not included in the above delay equations since these gates require only fractions of nano-seconds. Additionally, delay buffer #3 (Fig. 9) has no effect on the read-sort cycle time since this delay element is only used for maintaining the setup-hold time between the clock (*CLK*) and the element being stored in the sorted register array.

Case three represents the worst case, upper bound sorting time when the input element set contans N occurances of the same element (i.e., one row in the transpose matrix has all "1" values, while all other rows have all "0" values). The corresponding flag register's value for this element is "N," while all other flag registers' values are "0." Our algorithm requires $N-1$ cycles to check all flag register values (i.e., all transpose matrix rows), even though all values are "0," and N cycles to output the single replicated element N times into the sorted register array. Therfore, the total number of clock cycles are $2N-1$ plus one cycle for reset, resulting in a total worst case, upper bound of $2N$.

The best case, lower bound occurs when all elements in the input set are distinct (i.e., every transpose matrix row contains either a single "1" or no "1s," case one and case two, respectively). During the read-sort phase, each cycle either stores one element or nothing, respectively, to the sorted register array, which requires N clock cycles to sort N elements.

On average and in most general cases, the input set will contain a mixture of distinct and repeated elements, and the actual sorting time will fall between the upper and lower bounds. Considering both the write-evaluate and read-sort phases, the required number of clock cycles ranges from $2N$ *to* $3N$ to sort the input elements, with the addition of the one clock cycle for reset and one clock cycle for the control switch between the write-evaluate and read-sort phases.

B. Control Unit Operation

The control unit receives input signals from the data path and outputs the appropriate control signals back to the data path. The control unit also receives the external and hand-shaking components' signals in order to interface with the external components that are using the sorting hardware, and synchronizes the complete sorting operation. There are several methods for designing the control unit [54], [55], and prior work on sorting hardware typically found it sufficient to present only the data path design and no detail on the control logic [2], [34]–[45]. However, in our work, we present the complete control unit design in order to provide a holistic sorting implementation with all signals, which alleviates any discrepancy between the control and data path units. Additionally, our inclusion of the control unit's design shows the simplicity of our sorting hardware, with the control unit using a small number of gates and is scalable and easily

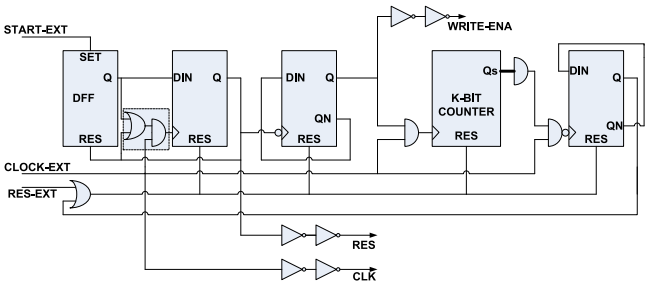

Fig. 11. Control unit diagram for the write-evaluate unit.

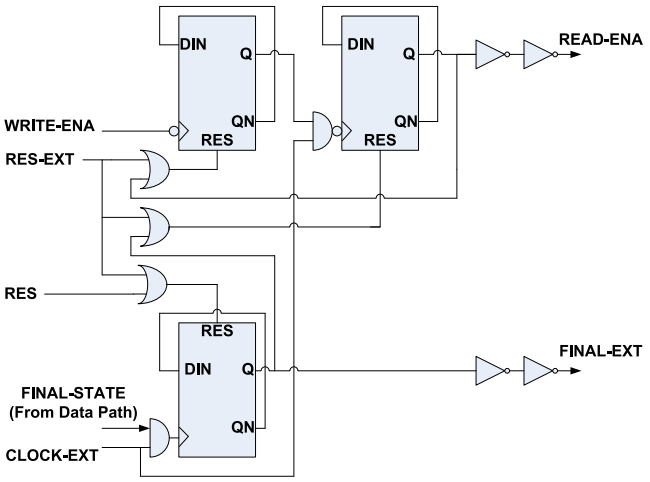

Fig. 12. Control unit diagram for the read-sort unit.

reconfigurable to different data types and sizes. We note that further area optimization can easily be achieved by reusing components for many handshaking controls with the data path unit, however, without loss of generality and for an easier conceptual explanation, we describe the control unit without shared components. In regards to timing and power, most of the components in the control unit are fast, and respond within the DFF access time delay. Additionally, most of the DFFs are clock-gated with an enable signal to minimize the DFFs' switching activities upon needed, thus reducing the overall circuit's power consumption.

Collectively, Figs. 11 and 12 depict the complete block diagrams for the control unit. For ease of explanation, the control unit divides the control logic structure into the write-evaluate and read-sort phases' controls, respectively, however, physically the control units share common components, such as the clock and the reset-initialization block.

The write-evaluate control circuitry (Fig. 11) is derived from the write-evaluate timing diagram (Fig. 8) and receives as input the external signals *CLOCK-EXT*, *RES-EXT*, and *START-EXT*. These signals control the sorting of the input bus elements, such that the data path generates the outputted sorted elements on the output bus and signals the end of sorting by asserting the *FINAL-EXT* signal. The internal reset-initialization block is triggered by the *START-EXT* signal, which in turn asserts the *RES* signal for one clock cycle. This complete clock cycle ensures that the reset-initialized components receive the asserted *RES* signal for long enough to ensure state initialization in the components, regardless of the underlying technology and fan-out interconnect. Several reset signals are branched and routed to different components in order to minimize the effective load on the *RES* signal. Additionally, the clock tree is designed in order to balance the clock edges across the components and preserve the setup-hold time margins, the details of which have been omitted in this figure for figure clarity.

All input and output signals are associated with appropriately-sized drivers to minimize the resistor-capacitive load on the input signals, and ensures that the signals propagate quickly enough and at full-swing with an appropriate signal slew-rate. We refer the reader to [53] for further details on load balancing and using appropriately-sized drivers. Asserting the *RES* signal (after *START-EXT* is asserted) for one clock cycle begins initializing the master-slave DFF structure for further operations. Subsequently, de-asserting the *RES* signal triggers asserting the *WRITE-ENA* signal for the complete write-evaluate phase. Once the control unit's *PC* reaches the saturated state $N = 2^K$, all input elements have been processed, which indicates the end of the write-evaluate phase. The *WRITE-ENA* signal is de-asserted and the *READ-ENA* signal is asserted on the next *CLK* edge, as illustrated in the timing diagram in Fig. 8.

The read-sort phase's control unit's circuitry (Fig. 12) is derived from the read-sort timing diagram (Fig. 10). The *READ-ENA* signal is asserted one clock cycle after the *WRITE-ENA* is de-asserted. At this point, the data path's *PC* is enabled and activates the one-hot decoder, order register array, flag register array, and one-detector. When the data path's *PC* saturates (i.e., all order and flag register values have been evaluated), the data path asserts the *FINAL-STATE* signal that drives the control unit. The control unit deasserts the *READ-ENA* signal and asserts the *FINAL-EXT* signal indicating that sorting is complete. The *FINAL-STATE* signal indicates that all rows in the transpose matrix have been scanned and mapped to the sorted array register.

The synchronization of these operations are inherent-by-design using DFFs with a SET and RESET structure, as given in [59]. The complete control unit only requires seven DFFs for controlling the continuous sorting of input elements. The simplicity of our control unit circuitry design is due to the continuous forward-flowing data through the data path and results in simple timing, which is amenable to efficient circuit design structures.

VI. Simulations and Results

Without loss of generality and for comparison purposes, we implemented, tested, and verified our sorting algorithm and hardware architecture using a sample system with $N = 1024$ input data elements, which is similar to many prior hardware sorting integrated circuits (ICs) [2], [37]–[45], [47]–[49]. We architected our proposed comparison-free sorting hardware at the CMOS transistor level using 90-nm Taiwan Semiconductor Manufacturing Company (TSMC) technology with a 1 V power supply [56]. We gathered timing delay values, total power consumption, and total transistor counts using HSPICE simulations [57].

TABLE I
COMPONENT TIME DELAYS AND TRANSISTOR COUNTS ASSUMING 90-nm TECHNOLOGY

Component	Size	Delay	Transistor Count
Binary order registers	1024 registers 10 DFFs per register	0.14 ns	204800
Binary flag registers	1024 registers 10 DFFs per register	0.14 ns	204800
Serial registers	1024 registers 10 DFFs per register	0.14 ns Shift one element from one register to another	204800
One-Hot Decoder	10-to-1024 5 levels	0.68 ns	5456
One-detector	10-bit input 1-bit output 2 levels	0.26 ns	24
Incrementor Decrementor	10-bit input 10-bit output	0.37 ns	200
Tri-State Buffer (D_S)	10 buffers per register	0.14 ns	10*1024*4 *2=81920
Parallel Counter	10-bit parallel-in 10-bit parallel-out	0.167 ns	330*2= 660
Control Logic	7 DFF	0.14 ns	7*24=168

The one-hot decoder, which converts the 10-bit input bus binary representation to the 1024-bit one-hot weight representation, uses a four-input fan-in NAND logic gate with a five-level hierarchical structure, resulting in a timing delay of $T_{\text{OH}} = 0.688$ ns. The order and flag registers are comprised of ten parallel DFFs, such that the register access time can be approximated using a single DFF access time of $T_{\text{DFF}} = 0.14$ ns. Similarly, the tri-state buffer and multiplexer are approximated as the same delay as the DFF access time $T_{\text{TB}} = T_{\text{MUX}} = T_{\text{DFF}}$.

The one-detector uses a parallel prefix-tree structure of four-input OR-gates, which take as input 10 bits and activates a two-level output, resulting in a timing delay of $T_{\text{OD}} = 0.26$ ns. The data path's 10-bit *PC* is implemented based on state-look ahead logic [58], giving a timing delay to the next state of approximately 0.167 ns. The incrementor/decrementor circuit takes a 10-bit input bus and add/subtract a "1," giving a timing delay of approximately 0.37 ns.

Table I summarizes all of the components' delay times and associated transistor counts. These results, combined with (9)–(12), show that the write-evaluate phase's clock cycle time is $CLK_W < 2$ ns and the read-sort phase's clock cycle time is $CLK_R < 2$ ns. These timings result in an approximate

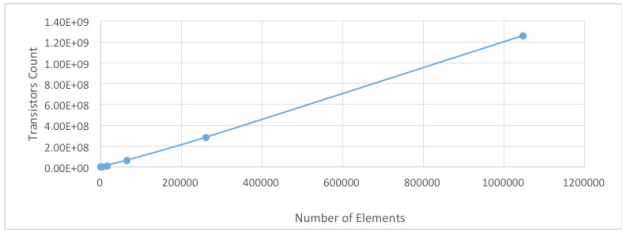

Fig. 13. Transistor counts for the order, flag, and sorted register arrays as compared number of elements.

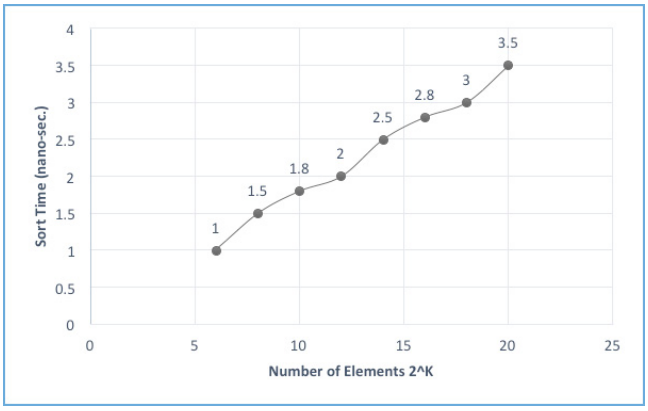

Fig. 14. Clock cycle time as compared to bus width.

conservative clock cycle frequency of 500 MHz, and the total power consumption given the technology factor at this frequency is 1.6 mW. Sorting 1024 elements requires a total number of clock cycles ranging from $3 \times 1024 = 3076$ to $2 \times 1024 = 2048$, depending on the number of duplicated input elements, resulting in a total time (for our clock speed of 500 MHz) of approximately 4–6 μs. Additionally, the total transistor count is less than 750 000 to sort 1024 elements.

Our design alleviates complex components such as memory and pipelining structures, which are considered in hardware designs as the bottleneck for performance and power consumption [13]. The only design bottleneck with respect to performance is the one-hot decoder; however, an optimized version of this component could be used [51], [52]. Since our focus is to architect a holistic circuit design, rather than optimizing special components and leveraging advanced CMOS technologies, we consider the integration of these optimizations as orthagonal to our design.

Fig. 13 shows how the transistor count scales as compared to the number of data elements for the order, flag, and sorted register arrays since these structures dominate the transistor count. These results show that our design's transistor growth rate is linear, with a small increase in the slope rate of less than six, giving a linear complexity ratio of O(N) with respect to transistor count.

Fig. 14 shows sorting speed in clock cycle time as compared to the number of data elements $N = 2^K$ for a k-bit bus. Our results ignore the interconnect parasitic values and the required buffering sizes, and focus only on our design's components' delays. Using the access delay times reported in Table I and (12) for upper bound limits on maximum frequency, and assume the worst case data distribution (all N elements are repeated), Fig. 14 shows a linear complexity of O(N) for

TABLE II
SORTING COMPUTATION TIME FOR AN INPUT SET OF 1024 ELEMENTS

ALGORITHM	PLATFORM	AVERAGE SORTING TIME (μsec)	NAME
REF [5]	SINGLE-CORE CPU	708	QUICK SORT
REF [15]	MANY-CORE CPU	300	AA-SORT
REF [19]	SINGLE-CORE GPU	100	MIN-MAX Butterfly
REF [20]	MANY-CORE GPU	37	PARALLLEL MERGING
PROPOSED	ASIC	6	COMPARISON-FREE

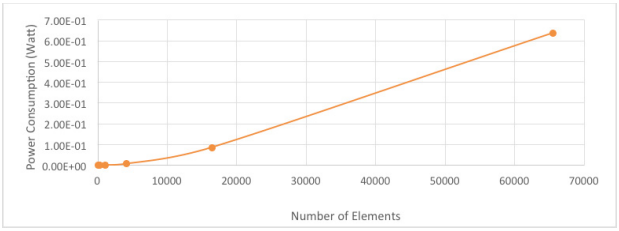

Fig. 15. Power consumption as compared to number of data elements.

end-to-end execution time for our sorting design with a small growth rate less than 1.5. This small rate is due to using basic registers (flag, order, and sorted registers) that access the bus in parallel.

The power consumption is relative to the switching activity and the transistors' static leakage. To reduce power consumption, our design's datapath and control units' components are gated with enable signals to restrict activity to only the components operational periods. The write-evaluate and read-sort phases each activate two register arrays: the order and flag register arrays, and the flag and sorted register arrays, respectively. Therfore, during the write-evaluate phase, the sorted register array is shut off, and in the read-sort phase, the order register array is shut off. All other components operate in both phases, therefore the phases' consume approximately equal power. Fig. 15 shows our design's power consumption as compared to the number of data elements and assuming a 500 MHz running frequncy. The operating frequency limits are evaluation to a maximum of $N = 2^{16}$ data elements, since larger sizes would require slower a slower clock frequency. Our design's power consumption shows a linear complexity of O(N) for a number of data elements less than 2^{16} with a growth rate of about 6.4.

Overall, our design shows a linear growth rate O(N) with respect to total transistor count, end-to-end execution time, and power consumption. This is in contrast to other work's [2], [35], [41], [48] that report a linear complexity of O(N), but the growth rate is usually in the order of greater than 100.

We also compare our design with data reported in literature for related CPU and GPU sorting algorithms [5], [15], [19], [20]. Table II reports the execution time for sorting 1024 elements using both single- and multicore CPUs and GPUs not considering the the front-end memory initialization time and the back-end memory merging time; just only the computation time. These results show that our design is even faster than prior algorithms who effectively harness the computing resources, to the best of our knowledge.

For general purposes, we have compared our sorting design with prior work with respect to hardware complexity and

TABLE III
COMPARISON BETWEEN PRIOR WORK AND OUR PROPOSED SORTING DESIGN

Sorting Design	Cycle Ranges	Complexity & Power Ratio
[44]	Not linear: for N<200, $2N$ cycles + initialization time	Comparisons, pipelining, memory, parallelisms, large power ratio
[26]	Not linear, for N<100, $3N$ to $4N$ cycles	Comparisons, pipelining, memory, parallelisms, large power ratio
[45]	(N+K-1) cycles + initialization time + output resorting time	Comparisons, pipelining, memory, large power ratio
[47]	Not linear, (N+2K) cycles + initialization time	Comparisons, memory, pipelining, moderate power ratio
[31]	$4N$ to $5N$ cycles + initialization time	Comparisons, pipelining, memory, parallelisms, large power ratio
Proposed	$2N$ to $3N$ cycles	No comparison, no memory, no pipelining, low power ratio

sorting performance in number of clock cycles. These comparisons are independent of technology factors in order to avoid uncertainty with respect to different technology scale comparisons and technology simulation environments, which makes the comparison fair because technology circuit implementations can vary greatly, ranging from different FPGA varieties/families to custom application specified integrated circuits using CMOS, NMOS, PMOS, Domino, pass-transistor logic families, and many others [53]. These implementation specifics have a large influence on the design performance and design cost, which may result in unrealistic or inaccurate conclusions. Therefore, we compare our design with prior designs with respect to common features for sorting hardware design circuit architectures, such as the number of cycles with respect to number of input elements, design structure of the data path and control units that leads to scalability and flexibility for different applications, and finally, the design computation complexity and data movement directions, which impact the design cost and power factor. These types of comparisons provide a larger evaluation picture considering the huge number of sorting hardware designs.

Table III compares our design with prior hardware sorting algorithms that have a single computing engine and several sorting partitions that require merging small sorted partitions

TABLE IV
COMPARISON WITH RECENT FPGA SORTING ALGORITHMS: SPIRAL [47] AND RESOLVE [48]

N Elements	64			1,024			16,384		
FPGA	Trans. Count	BRAM	Time	Trans. Count	BRAM	Time	Trans. Count	BRAM	Time
Spiral_SN1	2,382,300	10	0.5	35,553,750	162	8.2	-	-	-
Spiral_SN2_1	1,150,450	5	3.2	6,011,850	45	82	$1.0X10^8$	964	1835
Spiral_SN2_S	2,401,400	10	0.5	17,202,850	125	8.2	$17.8X10^8$	1395	131
Spiral_SN5	4,126,300	18	0.5	31,058,900	225	8.2	-	-	-
Resolve	1,123,400	2	0.54	2,599,900	7	8.3	16434350	70	131.1
Comp.-Free	23,040	-	0.4	614,400	-	6.1	13762560	-	98.3

to obtain the final sorted output. We evaluated the designs based on the number of clock cycles required to sort an input set of size N. This evaluation illustrates the complexity scaling of our simple forward data flowing design for increasing bit-widths as compared to the prior methods that merge the datapath and control units' functionalities within the parallel computing cells, memory, and comparison circuitry, all of which usually dictate the circuit's design complexity (number of transistors), runtime complexity (number of cycles to sort N elements), and power. Dividing computing cells that integrate the datapath with the control unit usually requires two operations: element evaluation and result updating, which requires repeating evaluation decisions. Furthermore, prior rank-based designs required repeated ALU computations within the SRAM or memory array, which is usually characterized as being time consuming.

For additional comparison, we evaluate the data reported in [49], which presents recent work on hardware sorting algorithms implemened on the Xilinx FPGA xc7vx690tffg1761-2 using 32-bit fixed point operations and running at a frequency of 125 MHz. Table IV shows the overall transistor counts, required number of BRAMs, and sorting time in microseconds. These compared designs show a linear increase in the FF/LUT count with respect to the number of elements, however the BRAM requirements do not scale linearly. Since memory devices introduce performance bottlenecks, this results in the non-linear execution time and non-linear transistor count.

With respect to all evaluated results, our comparison-free sorting design provides an efficient linear scalability of O(N). Our design uses simple registers (flag, order, and sorted registers) that are accessed on both the rising and falling clock edges, and simple standard CMOS components with a forward flowing data movement architecture. Even though our design shows a linear performance cost of O(N), our hardware design is recommended for data element set sizes of less than 2^{16} due to practical integration into large computing IC devices (e.g., graphics engines, routers, grid controllers.), where the sorting hardware accounts for no more than 10% of the IC's characteristics (power and area).

VII. CONCLUSION

In this paper, we proposed a novel mathematical comparison-free sorting algorithm and associated hardware implementation. Our sorting design exhibits linear complexity O(N) with respect to the sorting speed, transistor count, and power consumption. This linear growth is with respect to the number of elements N for $N = 2^K$ where K is the bit width of the input data. The slope of the linear growth rate is small, with a growth rate of approximately 6 for the transistor count and power consumption, and 1.5 for the sorting speed.

The order complexity and growth rates are due to simple basic circuit components that alleviate the need for SRAM-based memory and pipelining complexity. Our mathematically-simple algorithm streamlines the sorting operation in one forward flowing direction rather than using compare operations and frequent data movement between the storage and computational units, as with other sorting algorithms. Our design uses simple standard library components including registers, a one-hot decoder, a one detector, an incrementer/decrementer, and a *PC*, combined with a simple control unit that contains a small amount of delay logic.

Our design is at least 6× faster than software parallel algorithms that harness powerful computing resources for input data set sizes in the small-to-moderate range up to 2^{16}. Additionally, our hardware design's performance is approximately 1.5× better as compared to other optimized hardware-based hybrid sorting designs in terms of transistor count and design scalability, number of clock cycles and critical path delay, and power consumption. Thus, our design is suitable for most IC systems that require sorting algorithms as part of their computational operations.

Our results show that our comparison-free sorting CMOS hardware can sort N unsigned integer elements from end-to-end with any input data set distribution within $2N$ to $3N$ clock cycles (lower and upper bounds, respectively) at a clock frequency of 0.5 GHz using a 90-nm TSMC technology with a 1 V power supply and a power consumption of 1.6 mW for $N = 1024$ elements.

Future work includes leveraging our sorting algorithm for commercial parallel processing computing power, such as GPUs and parallel processing machines, in order to further improve large-scale sorting, and thus, further enhance embedded sorting for big data applications.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming*. Reading, MA, USA: Addison-Wesley, Mar. 2011.

[2] Y. Bang and S. Q. Zheng, "A simple and efficient VLSI sorting architecture," in *Proc. 37th Midwest Symp. Circuits Syst.*, vol. 1. 1994, pp. 70–73.

[3] T. Leighton, Y. Ma, and C. G. Plaxton, "Breaking the$\Theta(n \log^2 n)$ barrier for sorting with faults," *J. Comput. Syst. Sci.*, vol. 54, no. 2, pp. 265–304, 1997.

[4] Y. Han, "Deterministic sorting in O(n log log n) time and linear space," *J. Algorithms*, vol. 50, no. 1, pp. 96–105, 2004.

[5] C. Canaan, M. S. Garai, and M. Daya, "Popular sorting algorithms," *World Appl. Programm.*, vol. 1, no. 1, pp. 62–71, Apr. 2011.

[6] L. M. Busse, M. H. Chehreghani, and J. M. Buhmann, "The information content in sorting algorithms," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2012, pp. 2746–2750.

[7] R. Zhang, X. Wei, and T. Watanabe, "A sorting-based IO connection assignment for flip-chip designs," in *Proc. IEEE 10th Int. Conf. ASIC (ASICON)*, Oct. 2013, pp. 1–4.

[8] D. Fuguo, "Several incomplete sort algorithms for getting the median value," *Int. J. Digital Content Technol. Appl.*, vol. 4, no. 8, pp. 193–198, Nov. 2010.

[9] W. Jianping, Y. Yutang, L. Lin, H. Bingquan, and G. Tao, "High-speed FPGA-based SOPC application for currency sorting system," in *Proc. 10th Int. Conf. Electron. Meas. Instrum. (ICEMI)*, Aug. 2011, pp. 85–89.

[10] R. Meolic, "Demonstration of sorting algorithms on mobile platforms," in *Proc. CSEDU*, 2013, pp. 136–141.

[11] F.-C. Leu, Y.-T. Tsai, and C. Y. Tang, "An efficient external sorting algorithm," *Inf. Process. Lett.*, vol. 75, pp. 159–163, Sep. 2000.

[12] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," in *Proc. 8th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, Jan. 1997, pp. 360–369.

[13] L. Xiao, X. Zhang, and S. A. Kubricht, "Improving memory performance of sorting algorithms," *J. Experim. Algorithmic*, vol. 5, no. 3, pp. 1–20, 2000.

[14] P. Sareen, "Comparison of sorting algorithms (on the basis of average case)," *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 3, pp. 522–532, Mar. 2013.

[15] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-SORT: A new parallel sorting algorithm for multi-core SIMD processors," in *Proc. 16th Int. Conf. Parallel Archit. Compil. Techn. (PACT)*, 2007, pp. 189–198.

[16] V. Kundeti and S. Rajasekaran, "Efficient out-of-core sorting algorithms for the parallel disks model," *J. Parallel Distrib. Comput.*, vol. 71, no. 11, pp. 1427–1433, 2011.

[17] G. Capannini, F. Silvestri, and R. Baraglia, "Sorting on GPUs for large scale datasets: A thorough comparison," *Int. Process. Manage.*, vol. 48, no. 5, pp. 903–917, 2012.

[18] D. Cederman and P. Tsigas, "GPU-Quicksort: A practical quicksort algorithm for graphics processors," *ACM J. Experim. Algorithmics (JEA)*, vol. 14, Dec. 2009, Art. no. 4.

[19] B. Jan, B. Montrucchio, C. Ragusa, F. G. Ghan, and O. Khan, "Fast parallel sorting algorithms on GPUs," *Int. J. Distrib. Parallel Syst.*, vol. 3, no. 6, pp. 107–118, Nov. 2012.

[20] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. 23rd IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–10.

[21] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the 'best' sorting algorithm from optimal energy consumption," in *Proc. ICSOFT*, vol. 2. 2009, pp. 199–206.

[22] A. D. Mishra and D. Garg, "Selection of best sorting algorithm," *Int. J. Intell. Inf. Process.*, vol. 2, no. 2, pp. 363–368, Jul./Dec. 2008.

[23] T.-C. Lin, C.-C. Kuo, Y.-H. Hsieh, and B.-F. Wang, "Efficient algorithms for the inverse sorting problem with bound constraints under the $l\infty$-norm and the Hamming distance," *J. Comput. Syst. Sci.*, vol. 75, no. 8, pp. 451–464, 2009.

[24] F. Henglein, "What is a sorting function?" *J. Logic Algebraic Programm.*, vol. 78, no. 7, pp. 552–572, Aug./Sep. 2009.

[25] E. Mumolo, G. Capello, and M. Nolich, "VHDL design of a scalable VLSI sorting device based on pipelined computation," *J. Comput. Inf. Technol.*, vol. 12, no. 1, pp. 1–14, 2004.

[26] E. Herruzo, G. Ruiz, J. I. Benavides, and O. Plata, "A new parallel sorting algorithm based on odd-even mergesort," in *Proc. 15th EUROMICRO Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Feb. 2007, pp. 18–22.

[27] M. Thorup, "Randomized sorting in O(n log log n) time and linear space using addition, shift, and bit-wise Boolean operations," *J. Algorithms*, vol. 42, no. 2, pp. 205–230, Feb. 2002.

[28] M. Afghahi, "A 512 16-b bit-serial sorter chip," *IEEE J. Solid-State Circuits*, vol. 26, no. 10, pp. 1452–1457, Oct. 1991.

[29] J.-T. Yan, "An improved optimal algorithm for bubble-sorting-based non-Manhattan channel routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 2, pp. 163–171, Feb. 1999.

[30] L. Skliarova, D. Mihhailov, V. Sklyarov, and A. Sudnitson, "Implementation of sorting algorithms in reconfigurable hardware," in *Proc. 16th IEEE Medit. Electrotech. Conf. (MELECON)*, Mar. 2012, pp. 107–110.

[31] N. Tabrizi and N. Bagherzadeh, "An ASIC design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip," in *Proc. IEEE 6th Int. Conf. ASIC (ASICON)*, Oct. 2005, pp. 46–49.

[32] H. Schröder, "VLSI-sorting evaluated under the linear model," *J. Complex.*, vol. 4, no. 4, pp. 330–355, Dec. 1988.

[33] H.-S. Yu, J.-Y. Lee, and J.-D. Cho, "A fast VLSI implementation of sorting algorithm for standard median filters," in *Proc. 12th Annu. IEEE Int. ASIC/SOC Conf.*, Sep. 1999, pp. 387–390.

[34] G. Campobello and M. Russo, "A scalable VLSI speed/area tunable sorting network," *J. Syst. Archit.*, vol. 52, no. 10, pp. 589–602, Oct. 2006.

[35] W. Zhou, Z. Cai, R. Ding, C. Gong, and D. Liu, "Efficient sorting design on a novel embedded parallel computing architecture with unique memory access," *Comput. Elect. Eng.*, vol. 39, no. 7, pp. 2100–2111, Oct. 2013.

[36] V. Sklyarov, "FPGA-based implementation of recursive algorithms," *Microprocess. Microsyst.*, vol. 28, nos. 5–6, pp. 197–211, Aug. 2004.

[37] R. Lin and S. Olariu, "Efficient VLSI architectures for Columnsort," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 1, pp. 135–138, Mar. 1999.

[38] S. W. Moore and B. T. Graham, "Tagged up/down sorter—A hardware priority queue," *Comput. J.*, vol. 38, no. 9, pp. 695–703, Sep. 1995.

[39] G. V. Russo and M. Russo, "A novel class of sorting networks," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 43, no. 7, pp. 544–552, Jul. 1996.

[40] S. Dong, X. Wang, and X. Wang, "A novel high-speed parallel scheme for data sorting algorithm based on FPGA," in *Proc. IEEE 2nd Int. Congr. Image Signal Process. (CISP)*, Oct. 2009, pp. 1–4.

[41] A. Szél and B. Fehér, "Efficient sorting architectures in FPGA," in *Proc. Int. Carpathian Control Conf. (ICCC)*, May 2006, pp. 1–4.

[42] 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.

[43] T. Demirci, I. Hatirnaz, and Y. Leblebici, "Full-custom CMOS realization of a high-performance binary sorting engine with linear area-time complexity," in *Proc. Int. Symp. Circuits Syst. (ISCAS)*, vol. 5. May 2003, pp. V453–V456.

[44] 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.*, Mar. 2007, pp. 431–436.

[45] S. Alaparthi, K. Gulati, and S. P. Khatri, "Sorting binary numbers in hardware—A novel algorithm and its implementation," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2009, pp. 2225–2228.

[46] J. F. Hughes *et al.*, *Computer Graphics: Principles and Practice*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2014.

[47] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate (FPGA)*, Monterey, CA, USA, Feb. 2015, pp. 240–249.

[48] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Trans. Design Autom. Electron. Syst.*, vol. 21, no. 4, May 2016, Art. no. 55.

[49] J. Matai *et al.*, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate (FPGA)*, Monterey, CA, USA, Feb. 2016, pp. 195–204.

[50] *Sorting Algorithms Animations*, accessed on 2017. [Online]. Available: https://www.toptal.com/developers/sorting-algorithms

[51] (2010). *Cadence Online Documentation*. [Online]. Available: http://www.cadence.com

[52] (2015). *Synopsys Online Documentation*. [Online]. [Online]. Available: http://www.synopsys.com

[53] J. P. Uyemura, *CMOS Logic Circuit Design*. Norwell, MA, USA: Kluwer, 1999.

[54] J. P. Hayes, *Computer Architecture and Organization*, 2rd ed. New York, NY, USA: McGraw-Hill, 1994.

[55] S. Lee, *Advanced Digital Logic Design Using VHDL, State Machines, and Synthesis for FPGA's*. Luton, U.K.: Thomson Holidays, 2006.

[56] Taiwan Semiconductor Manufacturing Corporation. *90 nm CMOS ASIC Process Digests*, 2005.

[57] Synopsys. (2010). *HSPICE*. [Online]. Available: http://www.synopsys.com
[58] S. Abdel-Hafeez and A. Gordon-Ross, "A gigahertz digital CMOS divide-by-N frequency divider based on a state look-ahead structure," *J. Circuits, Syst. Signal Process.*, vol. 30, no. 6, pp. 1549–1572, 2011.
[59] V. Stojanovic and V. G. Oklobdzija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems," *IEEE J. Solid-State Circuits*, vol. 34, no. 4, pp. 536–548, Apr. 1999.

Saleh Abdel-Hafeez (M'01) received the B.S.E.E., M.S.E.E., and Ph.D. degrees in computer engineering from the USA with a specialization of very large scale integration (VLSI) design.

In 1997, he joined S3 Inc., Huntsville, AL, USA, as a member of their technical staff, where he was involved in the IC circuit design related to cache memory, digital I/O, and ADCs. He was the Chairman of Computer Engineering Department. He is currently an Associate Professor with the College of Computer and Information Technology, Jordan University of Science and Technology, Irbid, Jordan. He holds three patents (6, 265, 509; 6, 356, 509; 20040211982A1) in the field of IC design. His current research interests include circuits and architectures for low-power and high-performance VLSI.

Ann Gordon-Ross (M'00) received the B.S. and Ph.D. degrees in computer science and engineering from the University of California, Riverside, CA, USA, in 2000 and 2007, respectively.

She is currently an Associate Professor of Electrical and Computer Engineering with the University of Florida, Gainesville, FL, USA, where she is a member of the NSF Center for High Performance Reconfigurable Computing (CHREC). She is very active in promoting diversity in STEM fields. Her current research interests include embedded systems, computer architecture, low-power design, reconfigurable computing, dynamic optimizations, hardware design, real-time systems, and multicore platforms.

Dr. Gordon-Ross is the Faculty Advisor for the Women in Electrical and Computer Engineering and the Phi Sigma Rho National Society for Women in Engineering and Engineering Technology, and she is an active member of the Women in Engineering ProActive Network. She received the CAREER award from the National Science Foundation in 2010, the Best Paper Awards at the Great Lakes Symposium on VLSI in 2010 and the IARIA International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies in 2010, and the Best Ph.D. Forum Award at the IEEE Computer Society Annual Symposium on VLSI in 2014. She has been a Guest Speaker and has organized several international workshops/conferences on this topic, and participates in outreach programs at local K-12 schools.