Sorting Binary Numbers in Hardware - A Novel Algorithm and its Implementation

Srikanth Alaparthi Kanupriya Gulati Sunil P. Khatri
Department of ECE, Texas A&M University, College Station TX 77843.

***Abstract*— This paper describes a novel algorithm for sorting binary numbers in hardware, along with a custom VLSI hardware design for the same. For sorting n, k-bit binary numbers, our proposed algorithm takes O$(n + 2k)$ time. Sorting is performed by assigning relative ranks to the input numbers. A rank matrix of size $n \times n$ is used to store ranks. Each row of the rank matrix corresponds to one of the n numbers, and it stores a single non-zero entry. The position of this entry represents the relative rank of the corresponding number. In the worst case, our algorithm requires *n*+2*k* clock cycles for assigning the final ranks. We start with a condition in which each number has an identical rank. In each of clock cycle, ranks are iteratively updated until the final ranks are determined after *n*+2*k* clock cycles. The proposed algorithm is implemented in a 65nm process, using a custom design approach to obtain a fast circuit. Our design is significantly faster than the fastest reported hardware sorting engine, with area performance which is superior for larger numbers.**

I. Introduction

Sorting is one of the key functions required for many applications such as decoders for digital communication, digital signal processing, VLSI CAD etc. As a consequence, there is tremendous interest in speeding up sorting in software as well as hardware. This paper focuses on an algorithm (and its implementation) for sorting in hardware. We assume that the problem consists of n unsigned numbers, each of which uses a k bit representation.

Our algorithm for sorting is based on ranking each of the n numbers relative to each other. A number p has a higher rank than another number q iff $p > q$. The ranks of the n numbers are stored in an $n \times n$ memory which we refer to as the *Rank Matrix*. The i^{th} row of this rank matrix represents the rank of the i^{th} number, and has 0's stored in all locations, except for one position which has a 1 stored in it. The position of this non-zero value corresponds to the rank of the i^{th} number. The initial state of the rank matrix has 1's in the least significant bit of each row, indicating that all ranks are identical. In each iteration, the ranks of the numbers are updated based on an inspection of a particular bit of each of the n numbers, starting with the MSB. We increase the rank of a number if the particular bit being inspected indicates that this number is larger than others (whose ranks are therefore not increased). During each iteration, special care is taken so that no *bubbles* (columns of all 0's) are introduced in the rank matrix. This is described in detail in the sequel. Each iteration of our algorithm is completed in one clock cycle. Therefore, after $n + 2k$ clock cycles, the n numbers are sorted, with the final state of the rank matrix indicating the relative rank of all the n numbers.

The implementation of our algorithm is performed in a 65nm PTM [1] process. We performed a custom design of the entire circuit, by conducting careful device sizing. The layout of the design was performed as well. Based on this, we estimated the area of the resulting design, as well as its delay (which was determined from HSPICE [2], with wiring parasitics accounted for). Our results demonstrate that in comparison with the best hardware sorting implementation to date [3], our approach is faster, with an area requirement which is superior to [3] for larger numbers.

The key contributions of this paper are:

- A novel algorithm to perform sorting, designed with a hardware implementation in mind, with a time complexity of O$(n + 2k)$.
- A custom VLSI implementation of this algorithm in a current process technology. The implementation includes layout generation. Delay numbers are obtained from HSPICE, and account for layout wiring parasitics. Area numbers are based on layout areas.
- Our delay and area characteristics improve upon the best known approach for hardware based sorting to date [3].

The rest of the paper is organized as follows. Section II discusses the previous work on sorting algorithms, with an emphasis on hardware based approaches. The details of our proposed algorithm are explained in Section III, along with an example. Section IV details the circuit implementation of the algorithm. Section V discusses the simulation methodology and the results obtained. In Section VI, we summarize the contributions of this paper and conclude.

II. Previous Work

Several past research efforts have aimed at improving the performance of sorting by implementing a variety of software based algorithms. Comparison based sorting algorithms (such as heap sort, merge sort, quick sort and insertion sort) iteratively compare the input numbers in order to sort them. Insertion sort and quick sort require O(n^2) time in the worst case. Heap sort and merge sort are optimal comparison sorts. They require O(*n log n*) time. It has been proven [4] that comparison based sorting algorithms have a worst-case runtime lower bound of O(*n log n*).

Some sorting algorithms assume that prior knowledge about the input numbers is available. For example, the bucket sort algorithm requires O(*n*) time for input numbers that are distributed uniformly in the interval [0,1). Counting sort assumes that input numbers are in the range {1,2,..M}, and can sort them in O(*n+M*) time. Similarly for sorting of *n*, *d*-digit numbers, radix sort requires O(*d(n+M)*) time, where M is the maximum possible value for *d*.

Radix sort and counting sort are suitable algorithms for sorting of binary numbers. For sorting of *n*, *k*-bit binary numbers, the radix sort takes O(*k(n+2)*) time. Counting sort can be modified to sort binary numbers in O(*nk*) time. However, it needs *2nk* memory read and write operations.

Although there are several sorting algorithms reported in the literature, their performance is discussed based on software implementations. The number of hardware implementations of sorting algorithms are limited.

The survey in [5] discusses the implementation of different sorting algorithms in hardware, and their analysis in terms of area and time complexity. The various algorithms are implemented on a parallel processing platform, and are compared in terms of time and memory requirements. In [6], a sorting algorithm is implemented using a pipelined bit-serial architecture. It requires (*n*+1)*k* clock cycles to sort a sequence of *n*, *k*-bit numbers. Although the clock speed is 240 MHz for sorting of 512, 16-bit numbers, the approach has a high latency of O(*k*(*n*+1)) cycles. The hardware/software co-design algorithm explained in [7] for sorting of arbitrary long integers uses standard radix sort recursively, by taking *p*-bits at a time out of the *k*-bits. The efforts in [8], [9], [10], [11] use a classic Batcher's network of comparison elements for sorting. The runtime complexity of sorting using Batcher's network of n processors is O$(\sqrt{n}\ log^2(n)\ log(k))$. Here, the depth of the sorting network is O$(log^2(n))$ [5], the time taken for comparison-exchange at any level is atleast O(*log(k)*) [5], [12] and the time taken to rearrange the data among the processors in preperation for the next comparison-exchange operation is O$(\sqrt{n})$ [5]. In contrast the runtime complexity of our approach is O(*n+2k*) which is superior to the complexity of Batcher's network for all pairs of (n,k) where $n > 2$ and $k > 1$.

A full-custom CMOS realization of a binary number sorting engine with linear area-time complexity is presented in [3]. The implementation is based on rank ordering and an efficient implementation of multi-input majority function. It is shown that the proposed sorting engine is capable of producing a fully sorted output vector set in (*n+k-1*) clock cycles. Also the area is linear in *n* and *k*. However as the area has a dependency on the input bit-width k, the implementation area can be high when sorting input numbers with $k > n$. However, the area of our proposed implementation is independent of *k* and so is particularly suitable for such cases. In this paper, we have compared our work with [3], and provide detailed results demonstrating that the implementation of our approach is superior to that of [3] in terms of area and delay, for a large set of k and n values.

III. Algorithm

Our algorithm for sorting of *n*, *k*-bit unsigned binary numbers is described by means of a representative example. Figure 1(a) shows a sample input matrix (referred to as M) consisting of 4, 4-bit binary numbers:{1011, 1010, 0110, 0001}. We are required to sort these numbers. The following terminology will be used in describing the algorithm. The *rank* of a number indicates the position of the number in the sorted order. The highest number has a rank n, while the lowest number has a rank 1, assuming that there are no duplicate numbers. To perform the sorting of *n* binary numbers, our algorithm uses a *rank matrix* of size *n*×*n*. Each entry of this matrix is referred to as a *cell*. The ranks in this matrix are iteratively updated based on a bit-wise inspection of the input matrix M, starting with the MSB of the input matrix. Figures 1(b) through (f) show the updated state of the rank matrix after each iteration.

At the start of the algorithm, all the numbers are awarded a rank of 1. In other words, the LSB column of the rank matrix initially consists of all 1's, as shown in Figure 1(b). In subsequent iterations, these 1's will be selectively shifted towards the left in the rank matrix (i.e. towards the MSB), until the state of the rank matrix represents the relative ranks of the input numbers at the end of the final iteration. A single left shift for a number is also referred to as a *promotion* of that number. Note that whenever a 1 value is shifted towards the left, the current cell value is reset to 0 while the cell to the left is set to 1. This guarantees that each row will have exactly one 1 value at any given point in time.

978-1-4244-3828-0/09/$25.00 ©2009 IEEE

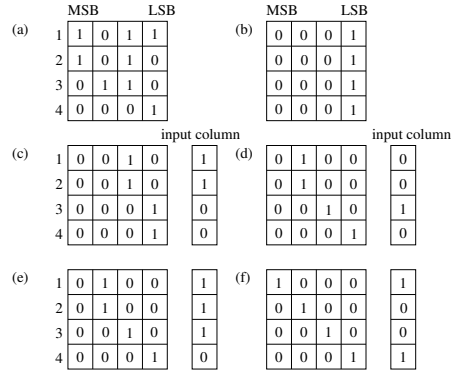

Fig. 1. (a) Input matrix M consisting of 4, 4-bit binary numbers (b) Initial rank matrix (c)-(f) Rank matrix status after each iteration

The input numbers from the input matrix M are accessed column-wise, starting with the MSB, and stored in a $n \times 1$ vector called the *input column*. In the i^{th} iteration, the i^{th} MSB column of the input matrix M is loaded into the *input column*. The rows in the rank matrix that correspond to the bits in the *input column* which are 1, will be shifted towards the left by one *cell*. As shown in Figure 1(c), after the first iteration, the 1's in the first two rows of the rank matrix are shifted towards the left due to the presence of 1's in the first two bits of the *input column*. At this point, the 1^{st} and 2^{nd} numbers have a higher rank than the 3^{rd} and 4^{th} numbers. We indicate this by saying that the 1^{st} (2^{nd}) number is *senior* to the 3^{rd} and 4^{th} numbers. We can say that the 3^{rd} (4^{th}) number is *junior* to the 1^{st} and 2^{nd} numbers. For the next iteration, the next column from the input matrix is loaded into the *input column* and the shifting process proceeds as above. However, while scheduling these shift operations, the algorithm takes care of the following special cases:

- Case 1: For any iteration, even if a particular row does not have a 1 in the corresponding bit of its *input column*, it still needs to be shifted towards the left if any of its juniors' rank is incremented in the same iteration. In other words, whenever a junior's rank is incremented, all it's seniors' ranks will be incremented as well, to maintain their seniority status. This is illustrated in Figure 1(d). In this case, even though the first two numbers have a 0 value in their input column, their rank is nevertheless incremented, since the third number (which is a junior) has a 1 value in its input column, and must therefore shift left.
- Case 2: During an iteration, if all the numbers with the same rank r in the rank matrix have a 1 in their corresponding *input column* bits, then none of the ranks are incremented. Also, if every rank needs to be incremented based on the above discussion, then the rank update can be voided. As shown in Figure 1(e), no shift is performed, since all the numbers with rank 2 and 3 (these are the third as well as the first and second numbers respectively) have a 1 in their corresponding input column bits. The rank matrix is therefore essentially the same as it was after the second iteration.

To summarize, there are two types of moves that are possible during any iteration:

- A *Self-move*: If a number's rank is incremented due to the presence of a 1 value in the corresponding bit of *input column*, the move is called *self-move*.
- A *Forced-move*: If a number's rank is incremented due to the fact that a junior number's rank is incremented, the move is called *forced-move*. A forced move is performed in order to maintain the seniority of the senior number.

At the end of k iterations, the rank matrix will contain the final (one-hot encoded) ranks of the input numbers. Figure 1(f) shows the final rank matrix after 4 iterations for our example. The first number is the largest as it has a 1 value in the MSB column of the rank matrix.

Theorem 3.1: The above rank update algorithm results in a correct sorted result after k iterations

Proof:

All numbers in our algorithm start out with identical initial ranks. Suppose two numbers a and b have the same rank after the $(j+1)^{th}$ column of the input matrix M has been processed. We refer to the MSB column of the input matrix as the k^{th} column. Suppose a has a 1 in its j^{th} bit position, then the rank of a will be incremented in the j^{th} iteration. Also suppose b has a 0 in its j^{th} bit position, then the rank of b will not be incremented. Since $a > b$, this behavior of the sorting algorithm results in correct operation.

After the j^{th} column is processed, suppose a number a is a senior of b. Each time b's rank is incremented, our algorithm increments a's rank as well (due to a forced move). Forced moves ensure that seniority, once established, is maintained for the rest of the run. Since $a > b$ has been established by processing the bits k through j, this condition cannot be reversed after processing the remaining $j-1$ bits. Hence the forced moves of the sorting algorithm results in correct operation.

Note that the operation of the algorithm under Case 2 above does not impact correctness – it merely avoids inserting a column of all 0's in the rank matrix, whenever there is at least one more significant column with at least one 1 value in it.

Therefore at the end of k iterations, the 1 entry in the left-most column (i.e. the MSB column) of the rank matrix identifies the largest number, the 1 entry in the next column to the right identifies the second largest number, and so on, resulting in a correct sorting operation. ■

Theorem 3.2: The size of the rank matrix is bounded by $n \times n$ for a input matrix M containing n input numbers

Proof: The number of rows of the rank matrix is equal to the n (the number of entries of M), by construction.

The number of columns of the rank matrix will be greater than *n* only if a column of all 0's is created during any iteration. A column of all 0's is formed when:

1) The ranks of all the numbers with the same rank r is incremented due to a self move, and
2) In the same iteration, none of the junior's ranks are incremented

However, our algorithm cancels the rank incrementation process in such a scenario, as discussed in Case 2. Thus, the algorithm never creates a column of all 0's.

Now assuming that each of the n numbers in the input matrix M are unique, we would require n columns in the rank matrix, which follows from the correctness of the algorithm (Theorem 3.1).

Hence the size of the rank matrix is bounded by $n \times n$. ■

Theorem 3.3: In any iteration, if the numbers have non-equal ranks, there are no forced moves iff there are no self moves.

Proof: Only if part: Without loss of generality, assume that at the start of an iteration, there is a senior-most number d in the rank matrix. This is valid since the numbers have non-equal ranks. The only time that this assumption is violated is when all the ranks are equal (which occurs, for example, in the first iteration). However, in this case, a forced move is not possible, since all ranks are equal. The absence of forced moves indicates that there is no need vacate a column in the rank matrix for a junior which is being promoted. In other words, no junior of the senior-most number d is promoted during this iteration.

Now if a self move increments the rank of any junior of d (and there are no forced moves), then the rank of d will equal the rank of its immediate junior(s). This is because in our algorithm, a column j with all 0's in the rank matrix will never be found, where there is at least one more significant column with at least one 1 value in it, due to the application of Case 2. Hence a self move without any forced moves causes a contradiction, since it violates the correctness of the algorithm.

If-part: Suppose there are no self moves but yet there is a forced move. Note that forced moves are only possible if the numbers have non-equal ranks. In this case, a column j of all 0's will be introduced in the rank matrix, when there is at least one more significant column than j with at least one 1 value in it in the rank matrix. This causes a contradiction, since the application of Case 2 specifically avoids the insertion of such a column of all 0's. ■

Note that Theorem 3.3 is not applicable at the start of the algorithm (since all numbers have equal ranks at the start of this iteration).

A. Complexity of the algorithm

As the width of the rank matrix is *n*, the largest number in the input matrix M has to have its rank move from the LSB column of the rank matrix to the MSB column of the rank matrix, requiring a total of *n*-1 shifts. So the complexity of the algorithm is O(*n*).

IV. Circuit Design

In this section, we describe our custom VLSI circuit implementation of the hardware sorting algorithm of Section III. Each cell of the rank matrix is implemented as a flip-flop surrounded by control and datapath logic, as shown in Figure 2. There are several global signals that are routed along rows and columns, through each cell. The cells in the LSB column of the rank matrix have flip-flops with an asynchronous set capability, and all the other cells have flip-flops with an asynchronous reset capability. The global reset signal, *GRESET*, is routed to all the set/reset pins of the cells. Therefore, when the *GRESET* signal is asserted, all the cells in the LSB column will have a value of 1, and all other cells will have a value of 0. The value stored in the flip-flop of the $(i,j)^{th}$ cell is referred to as *r(i,j)*. The input matrix M is stored in a memory which has a circular shifting capability. By appropriately shifting this memory, any input column (starting with the MSB column) is presented to the sorting circuit as required by the algorithm. At the start of each iteration, a circular shift operation is performed on the input matrix. After n iterations are completed the input matrix M has its original data. The current input column that is presented to the hardware sorting circuit, is referred to as m. The i^{th} bit of this input column is referred to as *m(i)*. In any iteration, when a flip-flop

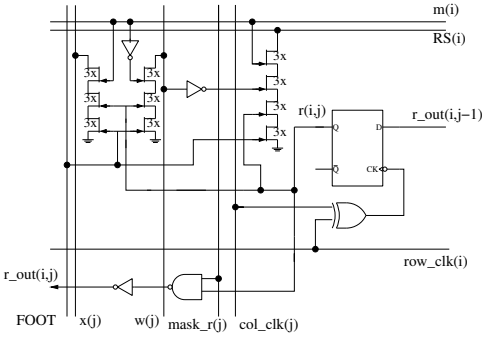

Fig. 2. Schematic of *cell(i,j)*

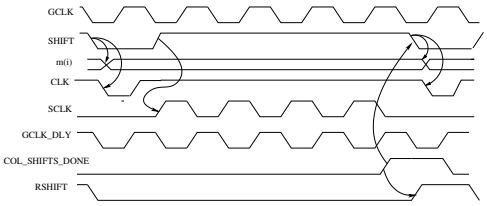

Fig. 3. Timing Diagram Showing all Control Signals during an Iteration

is clocked, its value is shifted towards the left. The clock to the flip-flop in the $(i,j)^{th}$ cell is derived from *row_clk(i)* and *col_clk(j)* signals. The *row_clk()* signals perform self-moves, while the *col_clk()* signals perform forced moves. Row indices are i, and column indices are j. The generation of these shift signals is explained in the sequel.

The timing of the signals in our implementation is described in Figure 3. Each iteration consists of inspecting a column m of the input and performing appropriate forced moves (via the *col_clk()* signals) and / or self moves (via the *row_clk()* signals). A free-running clock called $GCLK$ times all the operations. An iteration is initiated by the *SHIFT* signal falling. This causes a new input column m to be driven out to the hardware shifting engine, as well as a signal CLK to fall for one cycle. The global wires are pre-charged during the low phase of the *CLK* signal, and are evaluated down by the NMOS logic that is present in each cell, during the high or evaluation phase of the *CLK* signal. Once these signals are evaluated, they are latched at the rising edge of the *SHIFT* signal and are held for rest of the current iteration. An appropriate number of shift clock *SCLK* pulses are now created, and these are used to generate an appropriate number of *col_clk()* signals for forced moves in the current iteration. When all the forced moves have been completed, a single *row_clk()* pulse is generated from *SCLK*. The completion of the column shifts is indicated by a *COL_SHIFT_DONE* signal. After the row shift (self move) is done, a new iteration is initiated, by means of a new *SHIFT* pulse being derived.

A. Computation of Required Global Signals

As shown in Figure 3, the falling edge of *SHIFT* indicates the start of a new iteration and *m(i)*'s are updated. The circuit then computes the following signals at the start of the iteration, to determine the cells that will be shifted during the current iteration. Recall that the i index corresponds to rows, and j corresponds to columns.

$$x(j) = \sum_i r(i,j) \cdot m(i) \tag{1}$$

$$w(j) = \sum_i r(i,j) \cdot \overline{m(i)} \tag{2}$$

$$SL(j) = x(j) \cdot w(j) \tag{3}$$

$$RS(i) = \sum_j r(i,j) \cdot m(i) \cdot w(j) \tag{4}$$

In the above equations, the summation indicates the logical OR, and the $\cdot$ indicates the logical AND operation. *x(j)*, *w(j)* and *SL(j)* are column signals whereas *RS(i)* is a row signal. *x(j)*, which is computed using Equation 1, indicates that at least one of the cells in the j^{th} column *will shift* due to self move, since *r(i,j)* = 1 and *m(i)* = 1. *w(j)*, computed using Equation 2, indicates that at least one cell in the j^{th} column has *r(i,j)* = 1 and *m(i)* = 0. In other words, at least one of the cells that has *r(i,j)* = 1 *will not be shifted* during the current iteration since *m(i)* = 0. If *w(j)* = 0, every cell in the j^{th} column has *r(i,j)m(i)* = X1 or 0X, where X represents a don't care bit, so none of the ranks will be shifted. This is because if $r(i,j)$ = 0, there is no need to shift and if all *m(i)* in the current iteration are 1, again there is no need to shift (as discussed in Section III Case 2). $SL(j)$, computed using Equation 3, is the signal that combines the information from $x(j)$ and $w(j)$. If $SL(j)$ = 0, then we have $x(j)$ = 0 (i.e. no cells move) or $w(j)$ = 0 (i.e. all cells move, but in this case, a column of all 0's will be created, so we can as well not shift any cells). In other words, $SL(j)$ = 1 indicates the condition that the column must be shifted.

RS(i), the row-wide signal computed using Equation 4, indicates the necessity of a row-shift (or self move). A row shift is necessitated only when there is a 1 value in $m(i)$, and $w(j)$ = 1 for the particular j for which $r(i,j)$ = 1.

B. Forced moves or Column shifts

After the evaluation of the global signals, we first execute the *forced moves*. The number of 1's in *SL(j)* array gives the required number of column shifts. If *SL(j)* = 1, the columns *j*+1, *j*+2, ..., *n*-1 have to shift once towards the left. The shifts are executed starting from the MSB bit of the *SL(j)* array. After any column j (whose $SL(j)$ entry is a 1) is shifted, the corresponding $SL(j)$ value is cleared. The circuit first checks whether all *SL(j+1), SL(j+2), ..., SL(n)* are 0. A precharged NOR gate is used to generate a signal called *SCLKEN(j)*, which is the logical OR of *SL(j+1), SL(j+2), ... , SL(n)*. If *SCLKEN(j)* = 0 and *SL(j)* = 1, then the columns *j*+1 to *n*-1 should be shifted once towards the left.

SCLK is the clock signal that is used to perform the column shifts. On the falling edge of *SCLK* the column shifts are executed. As shown in Figure 3, *SCLK* is enabled only during the high phase of *SHIFT*. From the *SCLK* signal, appropriate column shifting clocks are generated during the column shifting phase, and a row shifting clock is generated in the row shifting phase.

If the j^{th} column is to shift during an iteration, the signal *col_shift(j)* is asserted high. col_shift(j) is shifted and ANDed to obtain a singe pulse. This is ANDed with *SCLK* and *COL_SHIFT_DONE*, and negated to obtain *col_clk(j)*, which is the required clock signal for the shift operation of column j. Due to space constraints, only the generation of the *col_clk(0)* signal is shown in Figure 4, highlighted with a similarly labeled circle.

In any iteration, consider a case where the j^{th} column is not required to shift and $(j+1)^{th}$ column is required to shift. The signal *mask_r(j)* is used to mask the output of the flops of the j^{th} column. As the flops in $(j+1)^{th}$ column are clocked, they sample their input, *r(i,j)*. But since the j^{th} column is not shifting, the output of flops (of the the j^{th} column) are ANDed using *mask_r(j)* signal, as shown in Figure 2.

Once the *SL(j)* signal is used to cause a column shift of the columns from *j*+1 to *n*-1, the *SL(j)* is reset to 0. The *CLR_SL(0)* signal shown in Figure 4 (highlighted with a similarly labeled oval) makes SL(j) = 0 on the falling edge of *SCLK*. If all *SL* values are zero, then it implies that all the forced moves are finished. The condition of all *SL*'s being zero is detected by the *col_shift(0)* signal. At this point, a row shift is initiated.

C. Self Moves or Row shifts

Once the column shifts are done (indicated by the *col_shift(0)* signal), a signal called *RSHIFT* is asserted high. During the high phase of *RSHIFT*, based on the value of *RSF(i)*, the i^{th} row is clocked, which shifts the entire row towards left. Note that the *RSF(i)* signal is the same as the *RS(i)* signal, except that it is registered. The rows with an *RSF(i)* value = 0 will not be shifted. All the row shifts are done in parallel, in a single clock cycle. Once column and row shifts are completed for any iteration, the next iteration is started automatically.

D. Generation of SHIFT and RSHIFT signal

After the forced (column) moves are executed, *RSHIFT* is asserted high to enable self (row) moves. *RSHIFT* is generated by latching the *col_shift(0)* signal. In Figure 4 the schematic for generation of the *RSHIFT* signal is

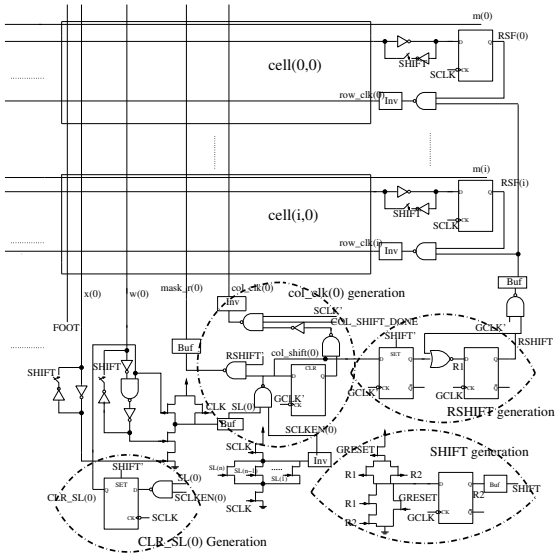

Fig. 4. Top-level Schematic

highlighted with an oval labeled as such. In the same clock transition, *SHIFT* signal is asserted low, initiating a new iteration. The *RSHIFT* and *SHIFT* signal transitions happen at the falling edge of *GCLK*, as shown in Figure 3. The circuit for generating *SHIFT* is also shown in Figure 4, which is implemented using complex CMOS gate and a flip-flop. The falling edge of *SHIFT* is used to load the next set of *m(i)* values. The self moves of one iteration and the loading of *m(i)* values, precharging and evaluation of global signals for the next iteration are done during the same clock cycle, as indicated in Figure 3.

E. Detecting all $SL(j)$ = 0 at the Start of an Iteration

Consider a case where all $SL(j)$ values are 0 at the start of an iteration. This indicates that no forced moves are needed. Then, according to *Theorem 3.3*, no self moves are required during this iteration. The condition of all *SL*'s being zero is detected by the *col_shift(0)* signal. Once this condition is detected, *SHIFT* is asserted low at the next falling edge of *GCLK*, to enable the next iteration. This is implemented in order to avoid wasted clock cycles.

V. Methodology and Results

We implemented our hardware based sorting approach in HSPICE, using a 65nm PTM [1] technology. A PERL [13] script is used to generate the HSPICE deck, with a user specified number of input numbers n and also the bit-width k of the numbers. A set of n circular shift registers are implemented to store the input numbers and drive them out column-wise (MSB first) to the sorting circuit. The circular shift registers are shifted on the falling edge of *SHIFT*. The entire circuit was designed in a custom VLSI design approach, with custom sizing of each device and driver. Global wires are optimally buffered based on their load (which depends on n and k). This buffering is performed by the PERL script.

We performed the layout of our design as well. The layout of a single cell is shown in Figure 5. We use this layout information to determine the length (and hence the parasitic capacitance and resistance) of each of the global wires. The global wires, *x(j), w(j), col-clk(j), foot* and *mask-r(j)* are routed across the columns and *row-clk(i), m(i)* and *RS(i)* are routed across the rows. Further, these wires are modeled in HSPICE using the parasitics extracted from the layout. These wiring parasitics are implemented in a distributed manner in the design.

The width of rank matrix is *n* - the number of inputs. The rank of the largest number has to be shifted from the LSB column to the MSB column in the rank matrix. This requires a total of *n*-1 shifts. In our design, two extra clock cycles are needed for each iteration: one for the precharging and evaluation of global signals, and another for switching from column shift mode to row shift mode. In addition, one clock cycle is needed for the global reset operation using the *GRESET* signal. So for *k* iterations, *n*+2*k* clock cycles are needed for assigning ranks to *n* *k*-bit binary numbers.

Simulations were performed for different combinations of *n*,*k*. Table I shows the achieved clock speed and power dissipation for different combinations of n and k. Note that for all the simulated examples, our design operates at above 600MHz, with a low power consumption. The simulations were verified for correctness for a large number of test cases, which exercised each of the special features of our algorithm.

Input (n, k)	CLK (ns)	Power (mW)	Leakage Power (μW)
(5,8)	1.0	0.93	29
(5,16)	1.0	1.00	30
(5,32)	1.0	1.20	29
(8,8)	1.1	1.36	66
(8,16)	1.1	1.50	65
(8,32)	1.2	1.58	51
(12,8)	1.4	2.99	165
(12,16)	1.4	3.24	148
(16,8)	1.6	3.48	388
(16,16)	1.6	3.47	365

TABLE I
SIMULATION RESULTS FOR DIFFERENT PAIR OF (n, k) VALUES

Our implementation is also compared with that of [3], which is the best known hardware sorting approach to date. The area and delay comparison plots are shown in Figures 6 and 7 respectively. The delay of [3] is dominated by an adder which is required to add n single bit operands. We determined the number of stages required by such an adder, by using an approach similar to a column compression operation in a parallel prefix multiplier. We verified that the number of stages we came up with matched that reported in [3]. Since the delay of [3] is computed in terms of the number of two-input gate delay equivalents, our delays was also represented in this manner. We took the critical circuit path of our design, and computed its delay in terms of the number of 2-input gate delay equivalents. Both these delays were computed as functions of n and k. For the area of the approach of [3], we computed the area of the each rank order filter (ROF) cell required, and added this to the area of the n input single bit adder. These areas were computed in terms of transistor counts. Our area was also computed in terms of transistor counts, in terms of n and k.

The area (delay) ratio of our implementation to the implementation in [3] is calculated for different (*n*,*k*) combinations. The portion of the curve for which each ratio is below '1' corresponds to (*n*,*k*) values for which our design is superior to that of [3]. For a particular *n*, we note that as *k* increases, the area ratio decreases. For example, for *n* = 16 and *k* = 32, the area and delay ratios are below 1. For sort 16 32-bit numbers (a common configuration), our implementation is better than [3]. Similarly, Figure 6 shows that our approach is superior to that of [3] for most values of (*n*,*k*).

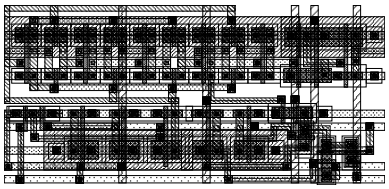

Fig. 5. Layout of a single cell (dimensions: 206λ x 96λ)

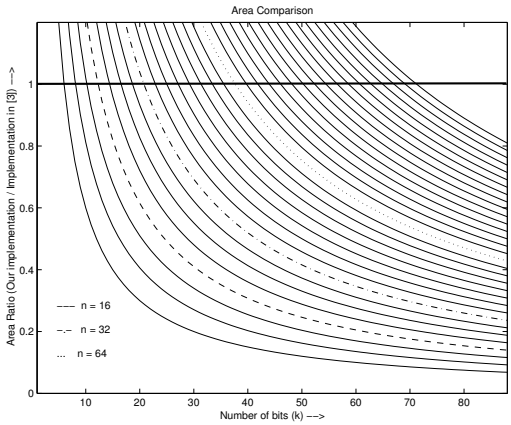

Fig. 6. Ratio of Area of our Implementation to Area of [3] (n varies from 4 (bottom) to 128 (top) with increments of 4)

VI. Conclusions

This paper presents a novel algorithm for sorting unsigned binary numbers, along with its custom VLSI implementation. In the worst case, our implementation completes sorting in *n*+2*k* clock cycles. Compared to the best known approach [3], our approach is superior in delay for most (n, k) values, and in terms of area, our approach outperforms [3] for larger k values.

References

[1] PTM, http://www.eas.asu.edu/∼ptm.
[2] I. Meta-Software, "HSPICE user's manual," Campbell, CA.
[3] T.Demirci, I.Hatirnaz, and Y.Leblebici, "Full-custom CMOS realization of a high-performance binary sorting engine with linear area-time complexity," in *International Symposium on Circuits and Signals*, May 2003, pp. 453 – 456.
[4] T. Cormen, C. L. R. Rivest, and S. Clifford, *Introduction to Algorithms*, 2nd ed. Boston, MA: McGraw-Hill Book Company, 2001.
[5] C.D.Thompson, "The VLSI complexity of sorting," *IEEE Transactions on Computers*, vol. 32, pp. 1171–1184, Dec 1983.
[6] M.Afghahi, "A 512 16-b bit-serial sorter chip," *IEEE Transactions on Solid State Circuits*, vol. 26, no. 10, pp. 1452 – 1457, Oct 1991.
[7] S. W. Cheng, "Arbitrary long digit integer sorter HW/SW co-design," in *Proceedings of the ASP-DAC*, Jan 2003, pp. 538–543.
[8] G.M.Blair, "Low cost sorting circuit for VLSI," *IEEE Transactions on Circuits Systems*, vol. 43, no. 6, pp. 515–516, Jun 1996.
[9] C.Y.Huang, G.J.Yu, and B.D.Liu, "A hardware design approach for merge-sorting network," *IEEE International Symposium on Circuits and Signals*, vol. 4, pp. 534–537, May 2001.
[10] T.Nakatani, S.T.Huang, B.W.Ardet, and S.K.Tripathi, "K-way bitonic sort," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 283–287, Feb 1989.
[11] K.E.Batcher, "Sorting networks and their applications," in *Proceedings of AFIPS Spring Joint Computer Conference*, Apr 1968, pp. 307–314.
[12] H. Sun, J. Cai, W. He, and M. Gao, "Logarithmic complexity implmentation for integer comparator," in *IEEE Int. Conf. Neural Networks and Signal Processing*, Dec 2003, pp. 328–330.
[13] L. Wall and R. Schwartz, *Programming perl*. O'Reilly and Associates, Inc., 1992.

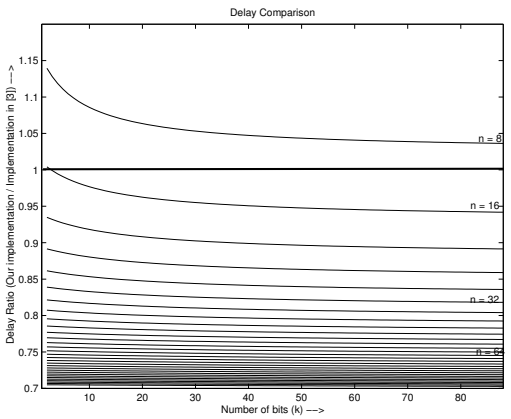

Fig. 7. Ratio of Delay of our Implementation to the Delay of [3] (n varies from 4 to 128 with increments of 4)