Proceedings of the Fourth IASTED International Conference
CIRCUITS, SIGNALS, AND SYSTEMS
November 20–22, 2006, San Francisco, CA, USA
ISBN Hardcopy: 0-88986-605-8 / CD: 0-88986-607-4

A PARALLEL SORT ENGINE WITH DYNAMIC MEMORY FOR A MULTIPROCESSOR-ON-A-CHIP

Nozar Tabrizi
Department of Electrical and Computer Engineering
Kettering University
Flint, MI, 48504, USA
ntabrizi@kettering.edu

Nader Bagherzadeh
Department of Electrical Eng and Computer Science
University of California, Irvine
Irvine, CA 92697, USA
nader@uci.edu

ABSTRACT: We propose a custom-designed alternative to a memory system (generated by a memory generator) used in a 4K-word sorting accelerator which improves area efficiency by some 20%. We also show how the control unit is dramatically simplified with this new memory comparing with the sophisticated memory controller in the previous version. Furthermore, since the memory introduced here is custom designed, its size is tailored to any specific need.

KEY WORDS: ASIC design, dynamic memory cells, network-on-a-chip, sorting accelerator, VLSI.

1. Introduction

We have recently developed MaRS [1] a reconfigurable heterogeneous computing engine, targeting multimedia data processing and wireless communication. To further support database systems, IP routing, bio informatics and cognitive-processing-based applications, integrating an efficient sorter onto the architecture is inevitable. Single-PE-based sorters suffer from significant performance overhead, while multiple-PE-based sorters (still with inefficient performance) occupy unacceptable die area. As a first attempt towards a high-performance area-efficient sorter, we introduced a novel sort algorithm and then developed an ASIC sorting accelerator (SA) for MaRS (or similar architectures) in [2] where we used memories generated by the Artisan Register-File Generator [3]. Now in this paper we propose a custom-designed alternative to that memory improving area efficiency by some 20%. We also show how the control unit is dramatically simplified with this new memory. Furthermore, since the memory introduced here is custom designed, its size is tailored to any specific need, compared to memory banks generated by a memory generator which come in only some fixed sizes. And finally, this solution makes the design more affordable, as no memory generator is required anymore.

The backbone of MaRS is a 2D mesh network of 32-bit processing elements (PEs) interconnected through 12 communication channels, as shown in Figure 1. (For now ignore the SE which has been plugged into the network.) Each PE is comprised of a *router* and a RISC processor (or *execution unit* (EU)). The router is in charge of allocating proper communication channels at run time to the incoming data-transfer requests, hence providing them with dynamically-allocated communication paths between neighboring PEs, and eventually facilitating a wormhole-routed network for MaRS. Every channel is comprised of a FIFO buffer, and the corresponding data bus. On the other hand each EU, in addition to the traditional instruction set, features three network-specific instructions which let a PE communicate with other PEs (and the memory stations, which comprise the second layer of memory hierarchy). MaRS supports point-to-point (one-to-one) single transactions, and also point-to-point and multicast (one-to-many) block transfers, initiated by any of the PEs (or the surrounding memory stations). In this architecture the group processors may bind together an arbitrary number of PEs as a macro-pipeline stage. Then several macro-pipeline stages may operate at the same time executing different kernels concurrently, as shown in Figure 1; hence tailoring the system to the intended application.

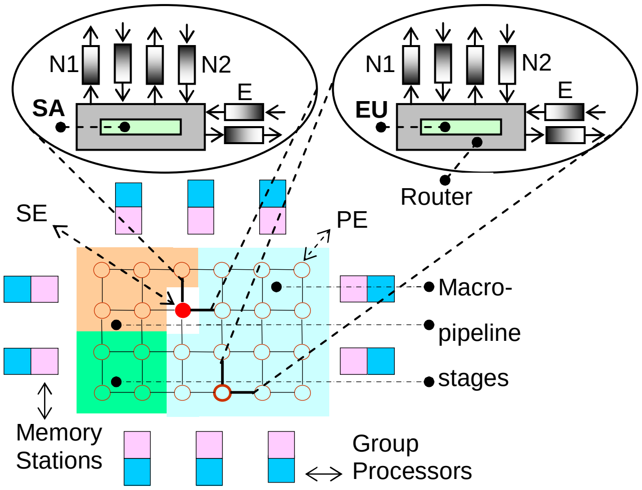

Figure 1. Heterogeneous MaRS with a 3-stage macro-pipeline, and one SA

In general, two different scenarios exist for hardware sorters:
1) The keys are all available to the sorter in a local memory before the sort procedure begins; this is the situation in some works such as [4] and [5].

2) The keys are sent sequentially from a key generator (source node) to the sorter (in a distributed system). This is what happens in our work and also some other works such as the one reported in [6].

These two scenarios of course need different algorithms for proper implementations. We utilize a novel pipelined and parallel sort algorithm consistent with the MaRS (or similar) architecture, to develop an ASIC sorting accelerator (SA) for MaRS. An SA can then be attached to a router, resulting in a sorting element (SE). An SE may replace any PE and be plugged into the network as illustrated in Figure 1. Each added SE can provide any PE (i.e. the key generator) on the network with the sorting service through the same communication protocol that is utilized for inter-PE communication, while the SE remains transparent to the ongoing network traffic. A data block is delivered to an added SE through a block transfer initiated by the requesting PE, the keys are sorted and then the sorted data leave the SE and reach the requesting PE through a similar procedure.

The rest of this paper is organized as follows. Some background work is reviewed in Section 2. Section 3 reviews our algorithm. The new memory is presented in Section 4. Section 5 is the conclusion.

2. Background Work

The well-known bubble sort algorithm [7] is inherently sequential, as consecutive comparisons have one shared operand. To overcome this shortcoming Baudet and Stevenson [8] use the serial odd-even transposition sort algorithm [7], lending itself to parallelization. This algorithm uses alternate odd and even phases to compare a_i and a_{i+1}, where i is odd and even in odd and even phases, respectively, requiring n phases of alternatively $n/2$ and $n/2 -1$ or a total of $n^2/2 – n/2$ comparisons, where n is the number of keys.

Bentley and Kung introduce the basic tree sort algorithm [9]. Starting with the well-known *match binary tree*, it is first modified, so that the losers are not removed from the tree anymore; they stay and continue to compete with their new competitors, if any. It takes the first winner log n steps to traverse the tree and reach the root. Henceforth empty nodes accept data form their nonempty children and the process continues so that at every other step the next element reaches the root in the specified order.

Batcher in his pioneering work [10] has introduced two iterative sorting networks, *odd-even* and *bitonic*, with the time and logic complexity of $\log^2 n$ and $n\log^2 n$, respectively. Several parallel sorting algorithms [11] [12] [13] [14] [15] have then been developed based on Batcher's work.

Stone [15] has modified Batcher's bitonic sort and proposed a slower sort algorithm but with reusable processing elements in which n/2 two-input sorters may be reused repeatedly to sort an array of n elements in $\log^2 n$ cycles.

In [14], Lee et al. have improved the time complexity of Stone's algorithm from $\log^2 n$ to log n.(log n +1)/2, but with additional logic.

Lee et al. [6] have proposed a parallel bubble sorter which requires n/2 identical compare/steer units stacked on each other, where n is the number of keys.

Olariu et al [4][5] have addressed sorting of a matrix of keys using a sorting device based on Batcher's algorithm, but now the number of keys can be much larger than the I/O size of the sorting device. They assume that all N keys are available in p (partially filled) memory banks, driving a p-sorter or a p-size sorting network, before sorting begins. After some calls to the sorter and some matrix manipulations the entries in the p memory banks are eventually sorted and left in the same p memory banks. This architecture can provide a good basis for an on-chip tightly coupled coprocessor. Interested readers may refer to [16] for a taxonomy of parallel sorting.

3. Our Algorithm

Our pipelined sorting algorithm introduced in [2] is illustrated in Figure 2 with an 8-leave sort tree and non-ascending sort order. In general input data enter the sort tree n key at-a-time through the leaves (where n is the number of leaves), and move toward the root, where the sorted keys appear from. The sort tree is a balanced and complete tree with n-1 decision (non-leaf) nodes and a depth of log n. Each decision node takes two keys, compares them and selects the greater one, resulting in only n + log n iterations to sort an array of n keys.

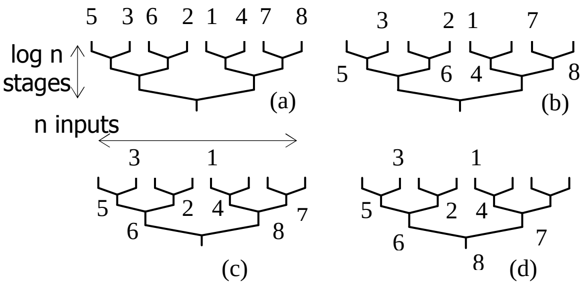

Figure 2. An 8-input pipelined sorting tree in the first four sorting states

When a key moves ahead its (previous) location becomes empty. Therefore, when a leaf becomes empty it will remain empty for the rest of the sort procedure. In each iteration an entry sitting on a node (next to its competitor, i.e. its sibling) traverses the tree one level forward if it wins (i.e. it is greater than its competitor) and the destination (i.e. the parent node) is either already empty or is becoming empty in the current iteration. The latter lets a chain of keys move forward together during the same iteration. The proposed algorithm is pipelined due to this simultaneous data movement in a chain of locations. As an example simultaneous moves of keys 7 and 8 are shown in Figures 2*b* and 2*c*. Notice that an empty competitor always loses; and a node with two empty children becomes and remains empty if it wins.

Before the first winner reaches the root the tree is filled one row at a time through *m* pipelines (necessarily shorter than log *n* +1) operating in parallel, where *m* is the width of the row to be filled next. For example, the two moves corresponding to (2, 6) and (7, 8) in Figure 2 during the iteration leading to (c) illustrates the presence of two pipelines (each 2 stages deep) in this iteration. Upon the arrival of the first winner at the root the number of pipelines is reduced to one.

Figure 3 shows the internal architecture of one decision (or compare/select) node of the sort tree. The two input keys are compared in a 32-bit comparator and then the greatest one is selected and saved in an output register which is concatenated with a valid bit to signify whether or not the corresponding register (node) is empty.

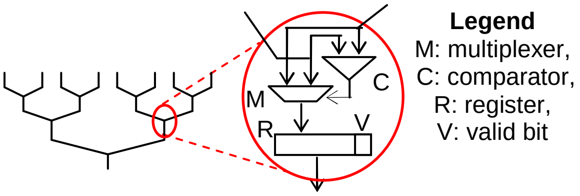

Figure 3. Internal architecture of one decision node in the sort/merge tree

Figure 4 shows a complete block diagram for the whole sorter. This block diagram is comprised of the following 3 blocks:

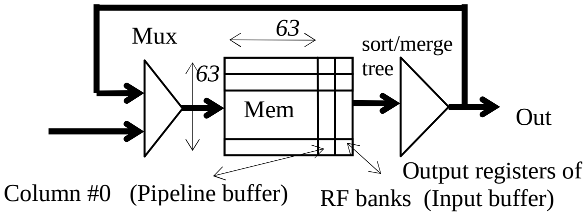
Mux
63
63
Mem
sort/merge tree
Out
Output registers of RF banks (Input buffer)
Column #0 (Pipeline buffer)

Figure 4. A block diagram of the sorter

1- The input multiplexer or Mux for short. This is a 32-bit 2-input multiplexer to let either a key from the key generator or the output of the sort/merge tree reach the Mem.

2- The sort/merge tree: This block has a tree structure with 63 leaves. A small-scale tree with only 8 leaves is illustrated in Figure 3. This figure also shows the internal architecture of each individual node of this (and any-size) tree.

3- Mem: In [2] Mem is made up of 63 two-port (one read and one write) register files (RFs), each 64 words long. These RFs have been generated by the Artisan Register File Generator. The RFs are synchronous, so that the content of an addressed location in an RF is loaded into an extra (output) register during a read cycle. This output register may be imagined as the 65th storage cell in a 64-word RF used in this SA. Write operations are also synchronous, so that the value on the data bus is transferred into the location addressed by the value present on the 6-bit address bus, when an active edge of the clock signal is applied.

We have used these 63 output registers as an input buffer (or parallel-in/parallel-out register) for the sort tree. Additionally, we have used column # 0 (comprised of locations # 0 of the 63 RFs) as a serial-in/parallel-out (SIPO) shift register, as shown in Figure 4. The remaining 63 words in each RF are used as a 63-word serial-in/serial-out shift register (SISO) as explained below.

The input keys are received one word at-a-time (from the sending PE) and injected to the serial-to-parallel converter mentioned above. Then every 63 keys are applied to the input buffer which is considered the first stage of the sort tree. The keys undergo the sort procedure in groups of 63 and then are laid into the memory horizontally. In other words every 63 consecutive sorted keys are seated in the corresponding SISO. In the second phase 63 sorted sub-arrays (each 63 words long) are merged through the same tree. That is why we call this tree sort/merge tree.

4. Dynamic Memory for Sorter

The 8-transistor memory cells employed in the register files (generated by the Artisan Register File Generator) are static. However, since the lifetime of the keys stored in the memory is deterministic, dynamic storage cells with no refreshing requirement can also be used, resulting in a denser design for the RFs. The only cost for this improvement is an upper limit on the size of the memory, hence the sorter.

Let’s assume that the parallel-to-serial converter, the pipeline buffer and all SISOs each are *p* words long. In order to determine the upper bound mentioned above we need to figure out the lifetime of the keys staying in the memory. By lifetime we mean the time interval between consecutive write and read operations performed for that specific key. In our algorithm there are two different lifetimes for each key. The first one (LF1) corresponds to the first write into the pipeline buffer and the following read operation. In this write/read process the pipeline buffer is in fact a serial to parallel converter, in which *p* keys are written one at a time, but are read out all together; hence the lifetime of a key is

$LF1 = p - i + 1$

where *i* is the key’s order number. The worst case happens when $i = 1$

$$LF1_{worst} = p \text{ cycles} \qquad (1)$$

The second lifetime corresponds to the second write operation in the sort stage (write from the tree back to the memory), and the read operation in the merge stage. Figure 5 shows a register located on arbitrary coordinates, (*x*, *y*), in the memory array. We now determine the lifetime of the key corresponding to this register.

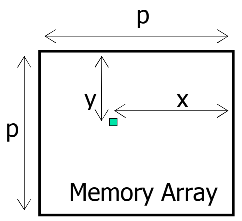

Figure 5. A register located on arbitrary coordinates, (*x*, *y*)

LF2: No of cycles to wait in memory:
LF2 = $p - x + (p - y).p + x$
LF2 = $p^2 + p - y.p$
$LF2_{worst}$ = LF2 (@ y = 1)

$$LF2_{worst} = p^2 \text{ cycles} \qquad (2)$$

The effective lifetime: LF = max ($LF1_{worst}$, $LF2_{worst}$) where $LF1_{worst}$ and $LF2_{worst}$ are given in (1) and (2), respectively.

Therefore,

LF = p^2 cycles

Max waiting time in memory: WT_{worst} = LF. T_{clk} where T_{clk} is the clock period.

$WT_{worst} = p^2 . T_{clk}$

WT_{worst} has to be less than the lifetime of electrical charge stored on the storage capacitance of the dynamic cell or the data may be lost. Assuming the charge lifetime of 1 msec and a 4 nsec (T_{clk}) clock signal, *p* is worked out equal to 500, resulting in some 240 K words as the upper limit for the memory size.

Figure 6 shows a sorter with a 4 x 4 x 1 dynamic memory for illustrations purposes. The same architecture can be expanded to any size as long as the upper limit of the memory is not violated. Our current design is based on a 16 x 16 x 32-bit dynamic memory. As shown in Figure 6 each memory cell is comprised of two inverters and two transmission gates wired up as a dynamic master-slave FF. The keys are injected to a serial-to-parallel shift register (SIPO), and then every 4 consecutive keys are applied to the input buffer which is in fact the parasitic capacitance of the corresponding input of the sort/merge tree. Therefore, ClkT is asserted every 4 clock cycles. The keys then undergo the sort procedure in groups of 4 and eventually are laid into the four dynamic shift registers (SISOs) shown in Figure 6. In other words, every 4 consecutive sorted keys are seated in the corresponding SISO. In the second (merge) phase four sorted sub-arrays (each 4 words long) are merged through the same tree. The sorted keys now leave the sort engine for the requesting PE. During the merge phase, ClkT is completely off, isolating the SIPO from the tree, but now the signal called phase 2 is asserted to let the merge tree be available to the SISOs.

Figure 7 shows how simple is the control unit of this new (16 x 16) memory system. The same concept can be generalized to control larger memories as well. In Figure 7*a* phase signals are generated by a timer. Remember that during phase 1 keys are injected into the SIPO and then are delivered to the sort tree 16 keys at-a-time. In the meanwhile the sorted keys coming out of the sort tree are seated in the 16 SISOs, so that at the end of phase 1 each SISO contains 16 sorted keys. ClkT is generated in Figure 7*b*. This signal delivers the contends of the SIPO to the sort tree by enabling 16 transmission gates every 16 cycles, i.e. ClkT is supposed to be asserted every 16 cycles while the sorter is in Phase 1. Care must be taken to keep clock skew within a safe range. In Figure 7*c* we see how 16 different clock signals are generated for 16 SISOs. In phase 1, the first 16 pulses (after the first sorted

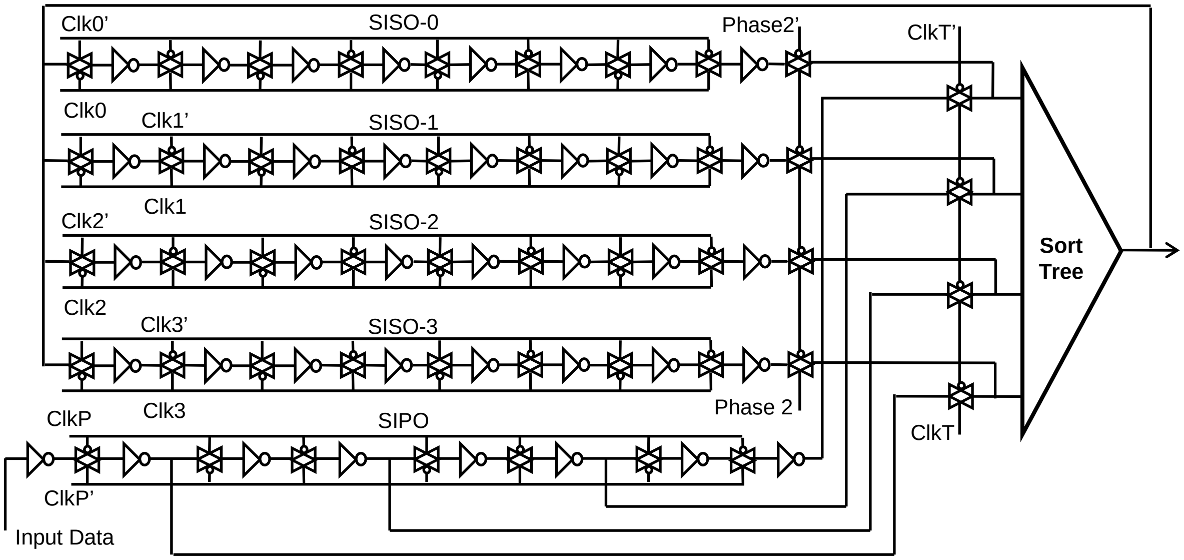

Figure 6. Logic diagram for a 4x4x1 sorter

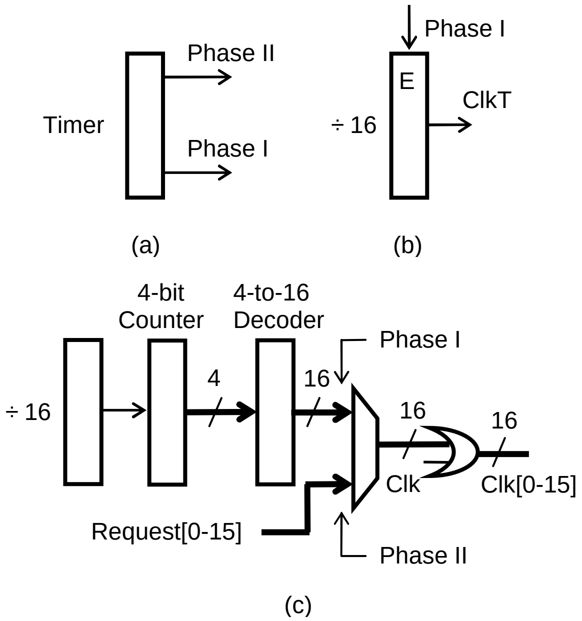

Figure 7. Control unit: (a) phase signals, (b) ClkT, (c) Clk0 to Clk15

key appears at the output of the tree, which is signified by an asserted valid signal, not shown in this figure) are applied to the first SISO, the second 16 pulses are applied to the second SISO and so on. Remember that there is a valid bit attached to every word. A word with an asserted valid bit means a valid key, otherwise the corresponding word would be considered invalid. On the other hand, in the merge phase an SISO is clocked only if a request is received from the merge tree, signifying that the top of the SISO has been consumed and now the next key should become available. The master enable/reset signal has not been shown in these figures.

Figure 8 shows a single memory cell (master-slave FF) and its layout. We have used magic layout editor [17] and spice [18] as a circuit simulator in this work. As shown in Figure 8 a single cell is 46 λ x 42 λ in area, and it can be stacked both vertically (after being properly flipped) and horizontally to create as long and/or as wide dynamic registers as needed, with some possible clock drivers. Based on this layout and considering 2λ = 0.13 micron (for the underling technology) the total area for a 63 x 64 x 32-bit memory is estimated 1.2 mm^2. On the other hand, we have used Cadence BuildGates Extreme (BGX) version v5.0-s006 synthesis tool, along with the TSMC 0.13 micron Low Voltage Process OverDriven (CL013LVOD) typical technology to synthesize one SA. The wire delay model is tsmc13_wl10. The register file banks and the corresponding technology file (TLF v4.1) have been generated by the Artisan Register File Generator. With a 4ns clock signal, the whole area is estimated 2 mm^2 out of which 1.6 mm^2 (approximately) are consumed by the 63 RFs. Now by replacing the register files with our custom designed shift registers the total area is expected to be around 1.6 mm^2 which shows some 20% area improvement. Area improvement is expected to be even more if the static registers inside the sort tree are replaced with dynamic registers similar to what we have used in the sort memory.

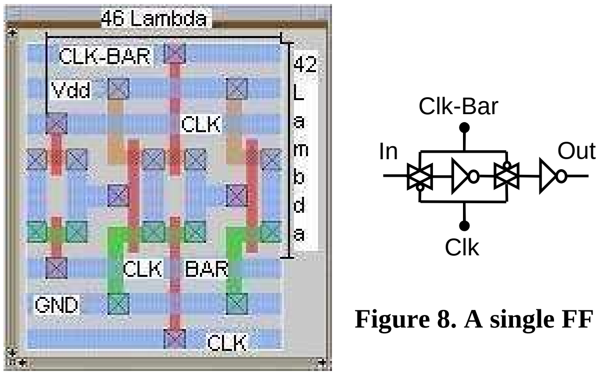

Figure 8. A single FF

5. Conclusion

In this paper we proposed a custom-designed alternative to a memory system used in a 4K-word sort engine which improves area efficiency by some 20%. We also showed how the control unit is significantly simplified with this new memory. Furthermore, since the memory introduced here is custom designed, its size is tailored to any specific need, compared to memory banks generated by a memory generator which come in only some fixed sizes. We have not performed power analysis for this memory yet. How ever, we are expecting some improvement in power consumption as well.

References

[1] Tabrizi N., Bagherzadeh N., Kamalizad A. and Du H. “MaRS: A Macro-pipelined Reconfigurable System,” *in proceedings of the ACM International Conference on Computing Frontiers (CF’04)*, pp. 343-349, Ischia, Italy, 14-16 April 2004.

[2] Tabrizi N. and Bagherzadeh N. “An ASIC Design of a Novel Pipelined and Parallel Sorting *Accelerator* for a Multiprocessor-on-a-Chip,” *the 6th International Conference on ASIC (ASICON’05)*, Vol. 1, pp. 44-47, Shanghai**,** China, October 24-27, 2005.

[3] http://www.artisan.com/

[4] Olariu, S., Pinotti, S. M. and Zheng, S. Q. “An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device,” *IEEE Transactions on Computers*, vol.49, no. 12, pp. 1310-1324, Dec 2000.

[5] Olariu, S., Pinotti, S. M. and Zheng, S. Q. “How to Sort *N* items using a Sorting Network of Fixed I/O size,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp 487-499, May 1999.

[6] Lee, D. T., Chang, H., and Wong, C. K., “An On-Chip Compare/Steer Bubble Sorter,” *IEEE Transaction on Computers*, Vol. C-30, No. 6, pp. 396-405, June 1981.

[7] Knuth, D. E. "Sorting and searching. In The Art of Computer Programming," Reading, Mass.: Addison-Wesley May 1997.

[8] Baudet, G., and Stevenson, D., "Optimal sorting algorithms for parallel computers," *IEEE Transactions on Computer*, C-27, 1, Jan. 1978.

[9] Bentley, J. L., and Kung, H. T., "A tree machine for searching problems," *In Proceedings of the 1979 International Conference on Parallel Processing*, Aug. 1979.

[10] Batcher, K. E. "Sorting networks and their applications," In Proceedings *of the 1968 Spring Joint Computer Conference*, Vol. 32., AFIPS Press, Reston, Va., pp. 307-314, Atlantic City, N.J., Apr. 30-May 2, 1968.

[11] Chien, M. and Oru, A., "Adaptive Binary Sorting Schemes and Associated Interconnection Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 6, June 1994.

[12] Kumar, M. and Hirschberg, D. "An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes," *IEEE Transactions on Computers*, vol. 32, no. 3, pp. 254-264, Mar. 1983.

[13] Lee, J. D., "Design of General-Purpose Bitonic Sorting Algorithms with a Fixed Number of Processors for Shared-Memory Parallel Computers," Computer Systems and Theory, vol. 26, no. 1, pp. 33-42, 1999.

[14] Lee, J. and Batcher K. E., "Minimizing Communication in the Bitonic Sort," *IEEE Transactions on Parallel and Distributed Systems*, Vol 11, No 5, pp 459-474, 2000.

[15] Stone, H. S., "Parallel processing with the perfect shuffle," *IEEE Transactions on Computers*, C-20, 2, Feb. 1971.

[16] Bitton, D., DeWitt, D, J., Hsiao, D. K., and Menon, J., "A Taxonomy of Parallel Sorting," *Computing Surveys*, Vol. 16, No. 3, pp. 287-318, September 1984.

[17] http://vlsi.cornell.edu/magic/

[18] http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/MANUALS/spice3.html