The VLSI Complexity of Sorting

CLARK D. THOMPSON

***Abstract*—The area–time complexity of sorting is analyzed under an updated model of VLSI computation. The new model makes a distinction between "processing" circuits and "memory" circuits; the latter are less important since they are denser and consume less power. Other adjustments to the model make it possible to compare pipelined and nonpipelined designs.**

Using the new model, this paper briefly describes thirteen different designs for VLSI sorters. (None of these sorters is new, but few have been laid out or analyzed in a VLSI model.) The thirteen sorting circuits are used to document the existence of an area $*$ time^2 tradeoff for the sorting problem. The smallest circuit is only large enough to store a few elements at a time; it is, of course, rather slow at sorting. The largest design solves an N-element sorting problem in only $O(\lg N)$ clock cycles. The area $*$ time^2 performance figure for all but three of the designs is close to the theoretical minimum value $\Omega(N^2 \lg N)$.

***Index Terms*—Area–time complexity, bitonic sort, bubble sort, heapsort, mesh-connected computers, parallel algorithms, shuffle-exchange network, sorting, VLSI, VLSI sorter.**

I. Introduction

SORTING has attracted a great deal of attention over the past few decades of computer science research. It is easy to see why: sorting is a theoretically interesting problem with a great deal of practical significance. As many as a quarter of the world's computing cycles were once devoted to sorting [19, p. 3]. This is probably no longer the case, given the large number of microprocessors running dedicated control tasks. Nonetheless, sorting and other information-shuffling techniques are of great importance in the rapidly growing database industry.

The sorting problem can be defined as the rearrangement of N input values so that they are in ascending order. This paper examines the complexity of the sorting problem, assuming it is to be solved on a VLSI chip. Much is already known about sorting on other types of computational structures [19, pp. 1–388], and much of this knowledge is applicable to VLSI sorting. However, VLSI is a novel computing medium in at least one respect: the size of a circuit is determined as much by its intergate wiring as by its gates themselves. This technological novelty makes it appropriate to reevaluate sorting circuits and algorithms in the context of a "VLSI model of computation."

Using a VLSI model, it is possible to demonstrate the existence of an area $*$ time^2 tradeoff for sorting circuits. A preliminary study of this tradeoff is contained in the author's Ph.D. dissertation [42], in which two sorting circuits were analyzed. This paper analyzes eleven additional designs under an updated model of VLSI computation. The updated model has the advantage of allowing fair comparisons between pipelined and nonpipelined designs.

None of the sorting circuits in this paper is new, since all are based on commonly known serial algorithms. All have been proposed before for hardware implementation. However, this is the first time that most of these circuits have been analyzed for their area and time complexity in a VLSI implementation. Ten of the sorters will be seen to have an area $*$ time^2 performance in the range $O(N^2 \lg^2 N)$ to $O(N^2 \lg^5 N)$. Since it is impossible for any design to have an area $*$ time^2 product of less than $\Omega(N^2 \lg N)$ [44], these designs are area- and time-optimal to within logarithmic factors.[1]

A number of different models for VLSI have been proposed in the past few years [5], [7], [16], [25], [37], [42], [43], [46]. They differ chiefly in their treatment of chip I/O, placing various restrictions on the way in which a chip accesses its input. Typically, each input value must enter the chip at only one place [42] or at only one time and place [5]. Savage [36] has characterized these as the "unilocal" and "semelective" assumptions, respectively.

The model of this paper builds on its predecessors, removing as many restrictions on chip I/O as possible. Following Kedem and Zorat [16], the unilocal assumption could be relaxed by allowing a chip to access each input value from k different I/O memories. Kedem's proof suggests that clever use of such "multilocal" inputs could improve a chip's area $*$ time^2 performance by a factor of k^2. Unfortunately, there seem to be neither interesting sorting circuits that take advantage of multilocal data, nor examples of naturally occurring multilocal inputs. Thus, the new model retains the unilocal assumption.

The semelective assumption is much less justifiable than the unilocal assumption. It is perfectly feasible to design a chip that makes multiple accesses to problem inputs, outputs, and intermediate results contained in off-chip memory. In a break from previous practice in theoretical VLSI models, the area of this off-chip memory is not included in the total area of a chip. This serves to clarify the area $*$ time^2 tradeoff for sorting circuits; memory area seems to be involved in a (lg area) $*$ time tradeoff, at least for circuits with fixed I/O bandwidth and small amounts of memory [13]. Leaving memory area out of the new model permits the analysis of sublinear size circuits.

Manuscript received March 15, 1982; revised June 10, 1982. This work was supported by the National Science Foundation under Grant ECS-81-10684.

The author is with the Computer Science Division, University of California, Berkeley, CA 94720.

[1] Knuth's notation for the base-two logarithm $\lg \triangleq \log_2$ is used throughout this paper. See [20] for standard definitions of order notation for lower ($\Omega(\,)$), exact ($\Theta(\,)$), and upper ($O(\,)$) bounds.

0018-9340/83/1200-1171$01.00

It also makes the model's area measure more sensitive to the power consumption of a circuit, since memory cells have a low duty cycle and generally consume much less power per unit area than do "processing" circuits.

Other authors have used nonsemelective models, although none has elaborated quite so much on the idea. Lipton and Sedgewick [25] point out that the "standard" AT^2 lower bound proofs do not depend on semelective assumptions. Hong [14] defines a nonsemelective model of VLSI with a space-time behavior which is polynomially equivalent to that of eleven other models of computation. His equivalence proofs depend upon the fact that VLSI wiring rules can cause at most a quadratic increase in the size of a zero-width-wire circuit. Unfortunately, Hong's transformation does not necessarily generate optimal VLSI circuits from optimal zero-width-wire circuits, since a quadratic factor cannot be ignored when "easy" functions like sorting are being studied.

Lipton and Sedgewick [25] point out another form of I/O restriction, one that is not removed in this paper's model. In most situations it is natural to restrict one's attention to circuits which produce their outputs at fixed locations. For example, the most significant bit of the largest output value in a "where-oblivious" circuit might be constrained to appear at I/O port 1, regardless of the problem inputs.

Adoption of the where-oblivious restriction begs an important theoretical question: what does "sorting" mean? Is it determining the rank order of the inputs? Is it permuting the inputs into sorted order, given their ranks? Or does it involve both ranking and permuting? The where-oblivious restriction adopted above implies an affirmative answer to the last question.

From a practical point of view, a sorting circuit should be required to rank and permute its inputs. It is possible to conceive of uses for circuits that can only rank or can only permute, but most applications require both. For example, consider the problem of removing the duplicate records that are frequently produced by projection operations on a relational database. The duplicates can be detected and purged in a straightforward fashion once the records are permuted into rank order (on any remaining key), thereby saving space and time in later database operations.

A historical argument can also be made in favor of requiring a "sorting" circuit to rank and permute its data. The original meaning of "sorting" is "separating or arranging according to class or kind" [19, p. 1], a process that would seem to involve both classification and movement. Thus, for practical and historical reasons, the interesting study of "ranking circuits" is left to other papers and other authors. Only (ranking and permuting) = "sorting" circuits are studied here.

The catalog of I/O restrictions is not yet complete. In both Vuillemin's [46] and Thompson's [43] models of pipelined VLSI computation, analogous inputs and outputs for different problems must be accessed through identical I/O ports. For example, input 1 of problem 2 must enter the chip at the same place as input 1 of problem 1. While this seems to be a natural assumption for a pipelined chip, it leads to a number of misleading conclusions about the optimality of highly concurrent designs. For instance, the highly parallelized bubble sort design of Section III-L is nearly area * time^2 optimal under the old models, but it is significantly suboptimal under the model of this paper.

When the restriction on pipelined chip inputs is removed, it becomes impossible to prove an $\Omega(N^2 \lg N)$ lower bound on area * time^2 performance until the definitions of area and time are adjusted.

In the new model, the area performance A of a design is its "area per problem," equal to its total processing area $A_{\text{processing}}$ divided by its degree of concurrency c. Thus, it does not matter how many copies of a chip are being considered as a single design: doubling the number of chips doubles both its concurrency and its total area, leaving its area performance invariant. The old definition of area performance was the total area of a design (including its "memory area") with no correction factor for its concurrency.

The time performance of a design can be measured in a number of ways. Vuillemin [46] concentrates on the data rate D (or I/O bandwidth), a measure of a circuit's throughput. While this is an important parameter, a circuit's data rate is not a very useful definition of time performance under this paper's unrestricted I/O model. A design consisting of two identical sorting chips would have twice the data rate of either of its chips considered separately. As with the area measure discussed above, it would be possible to "normalize" a circuit's data rate by dividing by its concurrency. The resulting measure defies easy interpretation; fortunately, a better measure is available.

Vuillemin [46] also considers the period T_p between successive presentations of complete sets of problem inputs. This measure is closely related to a circuit's data rate D, for it is numerically equal to the problem size (in bits) divided by D. As such, it suffers from similar difficulties of interpretation when inputs are not assumed to form a single stream.

The time measure adopted here is the delay T_d between the presentation of one set of problem inputs and the production of the outputs from that problem. This measure is obviously unaffected by replication: two sorting chips have the same delay as one. In defense of the delay measure, it can be argued that a design's delay is a more fundamental limitation than its data rate or period. As indicated above, the latter measures can always be improved by replicating the design and splitting its I/O stream. Also note that an upper bound result on T_d implies an upper bound result on T_p. That is, a circuit's period need be no larger than its delay, since idle time serves no useful purpose in this paper's model of computation.

As Vuillemin [46] notes, any lower bound on circuit area in terms of its period T_p or data rate D immediately implies a similar bound in terms of its delay T_d. By this reasoning, T_p is the stronger time measure for lower bounds, just as T_d is the stronger time measure for upper bounds. Vuillemin's $AT_p^2 = \Omega(N^2)$ lower bound on the complexity of sorting N numbers thus implies an analogous result for the T_d metric. His proof must also be adapted to this paper's metric of area performance and to its less-restricted model of I/O behavior. However, one can do better than an $AT_d^2 = \Omega(N^2)$ result. Vuillemin's "transitivity" argument for sorting is somewhat weak in the sense that it only measures the complexity of permuting the

least significant bit of each input word. By considering $\epsilon \lg N$ of the least significant bits, it is possible to show that $AT_p^2 = \Omega(N^2 \lg N)$ for the problem of sorting N words of $(1 + \epsilon)\lg N$ bits each [44].

There is reason to believe that an even stronger lower bound is obtainable. An $AT_d^2 = \Omega(N^2 \lg^2 N)$ result has been shown for a slightly more restricted model of I/O, in which all the bits of each input value must be read through the same I/O port [42]. (In the present model, bit 0 of an input could be read on the other side of the chip from bit 1 of that input.) It is difficult to imagine how a circuit could take advantage of such a "nonlocalized" I/O pattern and thus subvert the $\Omega(N^2 \lg^2 N)$ lower bound. Indeed, all of the circuits presented in this paper access the bits of each input value in a localized fashion, and thus none do better than $AT_d^2 = O(N^2 \lg^2 N)$.

This paper is organized in the following fashion: Section II discusses the new VLSI model of computation, then defines it precisely; Section III sketches thirteen different designs for VLSI sorters and analyzes the area–time performance of each; Section IV compares the performances of each of the designs, with some discussion of the "constant factors" ignored by the asymptotic model; and Section V concludes the paper with a list of some of the open issues in VLSI complexity theory.

In an attempt to keep the paper to a reasonable length, the constructions of Section III are described as briefly as possible. Readers wishing to "fill in the details" will have to follow the references, where applicable, and then exercise their own ingenuity. This is a regrettable situation, but an inevitable one since there is no accepted "high-level design language" for VLSI.

II. Model of VLSI Computation

In all theoretical models of VLSI, circuits are made of two basic components: wires and gates. A gate is a localized set of transistors, or other switching elements, which perform a simple logical function. For example, a gate may be a "j–k flip-flop" or a "three input NAND." Wires serve to carry signals from the output of one gate to the input of another.

Two parameters of a VLSI circuit are of vital importance, its size and its speed. Since VLSI is essentially two-dimensional, the size of a circuit is best expressed in terms of its area. Sufficient area must be provided in a circuit layout for each gate and each wire. Gates are not allowed to overlap each other at all, and only two (or perhaps three) wires can pass over the same point.

A convenient unit of area is the square of the minimum separation between parallel wires. In the terminology of [26], this paper's unit of area is equal to $(4\ \lambda)^2$, where λ is a constant determined by the processing technology. Each unit of area thus contains one, two, or three overlapping wires; or else it contains a fraction of a gate. The actual size of this area unit becomes smaller as technology improves. In 1978, it was typically 150 $\mu m^2 = 1.5 \times 10^{-6}$ cm^2; eventually, it may be as small as 0.4 μm^2 [26, p. 35].

The speed of a synchronous VLSI circuit can be measured by the number of clock pulses it takes to complete its computation. Once again, the actual size of this time unit is a technological variable. In 1978, a typical MOS clock period was 30–50 ns; and this may decrease to as little as 2–4 ns [26]. For the superconducting technology of Josephson junctions, a clock period of 1–3 ns is achievable today, using a process for which the area unit is 25 μm^2 [17].

The speed of a VLSI circuit may be adversely affected by the presence of a very long wires, unless special measures are taken. In many VLSI processes, a minimum-sized transistor cannot send a signal from one end of the chip to the other in one clock period. To accomplish such unit-delay cross-chip communication, and to achieve large fanouts, special "driver" circuits are employed. These drivers amplify the current of the signal; $O(\lg k)$ stages of amplification are required to drive a length-k wire [26, p. 14] or to drive k inputs. The use of $O(\lg k)$-stage drivers is reflected in the VLSI model's loading rules, as formalized in Assumption 1f). Under these rules, each stage of a driver has twice as many unit-area gates as the preceding one. The gates are wired in parallel, so each stage has twice the current sourcing (or sinking) capability of its predecessor. Furthermore, the stages are individually clocked. Thus, the driver behaves like an $O(\lg k)$-bit shift register with unit bandwidth and unit-power input requirements. Every wire, even the longest one, had a throughput of one bit per time unit; however, a length-k wire has $T_d = O(\lg k)$ delay. As argued in [43], the area of the long-wire driver circuits can be ignored (up to constant factors), as long as there is room to lay out the wires they drive.

Assumption 1f (the "logarithmic delay assumption") is used here because it leads to realistic circuit designs and time bounds, as well as to an interesting theoretical question. As it turns out, the time bounds obtained for VLSI sorting under this assumption are rarely different from the ones that would be obtained under a "unit-delay" assumption (in which each gate is able to transmit its output all the way across the circuit, in one clock period). The only exceptions might be the highly parallelized versions of the long-wire networks discussed in Sections III-J and III-K. Driver delays will exceed single-bit comparison delays in these networks unless one is able to lay them out with the smallest possible maximum wire-lengths. This is an attractive open problem, at least for the N-vertex shuffle-exchange network of Section III-J: can it be embedded with a maximum wire length of only $O(N/\lg^2 N)$?

From a practical standpoint, it may be argued that the logarithmic delay assumption is either too severe or too lenient, depending on the technology. The former is currently the case in the I^{2}L and Josephson junction processes [11], [17]. As of now, both are really unit-delay technologies. Minimum-sized gates can drive wires the entire length of a chip without significantly increasing logic delay. However, the results of this paper still apply if the drivers are omitted from the circuit constructions of Section III.

It seems unlikely that the logarithmic delay assumption will ever be too lenient on synchronous MOS circuits. Seitz [38] projects a signal transmission velocity of (1 cm)/(3 ns) in a fully developed MOS technology. This means that a cross-chip communication will only take a few clock periods, even if the "chip" is as large as a present-day "wafer." In other words, the time performance of the fully developed MOS technology is only slightly overestimated by the logarithmic delay assump-

tion. The true delay would best be modeled as logarithmic plus a small constant. Modeling delay as a linear function of distance, as suggested by Chazelle and Monier [7], would greatly exaggerate the importance of delay in the determination of the speed of such circuits.

If circuits ever become much faster or much larger than envisioned today, the logarithmic delay assumption should be modified. As a case in point, consider the Josephson junction circuit assemblies currently built by IBM. They are 10 cm on a side, and they run on a 1–3 ns clock [17]. The wires in these circuits are superconductors, but, of course, they cannot transmit information at a velocity greater than (a fraction of) the speed of light. Right now, the clock frequency and circuit dimensions are just small enough to allow a signal to propagate from one side of the circuit to the other in one clock period. Any increase in either speed or size would make this impossible. The computational limitations of such enhanced (and hypothetical) technologies could be analyzed under Chazelle and Monier's linear delay assumption.

Thus far in the discussion, only "standard features" have been introduced to the VLSI model. The interested reader is referred to [42] for more details on the practical significance of the model, and to [37] for an excellent introduction to the theoretical aspects of VLSI modeling.

As noted in the introduction, a major distinction between the model of this paper and most previous VLSI models is the way in which it treats "I/O memory." Here, only a nominal area charge is made for the memory used to store problem inputs and outputs, even if this memory is also used for the storage of intermediate results.

In the new model, each input and output bit is assigned a place in a k-bit "I/O memory" attached to one or more "I/O ports." Two types of access to the I/O memory are distinguished. If the bits are accessed in a fixed order, the I/O memory is organized as a shift register and accessed in $O(1)$ time per bit. If the access pattern is more complex, a random access memory (RAM) is used. Such a memory has an access delay of $O(\lg k)$ [26, p. 321]. The random access time covers both the internal delays of the memory circuit as well as the time it takes the I/O port to transmit (serially) $O(\lg k)$ address bits to the RAM.

This paper's serial I/O interfaces may seem a bit artificial, since typical RAM's are accessed with word-parallel address and data lines. More careful consideration reveals that any such word-parallel RAM interface could be simulated with several serial interfaces at no cost in asymptotic area and time.

Allowing more than one I/O port to connect to a single I/O memory makes it easy to model the use of multiport memory chips. Their usage must be restricted, however, to remove the (theoretical) temptation to use multiport memories and printed-circuit board wiring as a means of avoiding on-chip wiring. (Note that a two-port memory provides a communication channel between its two I/O ports, eliminating any need for an on-chip wire between them.) The restriction is that all I/O ports connecting to a single memory must be physically adjacent to each other in the chip layout. This avoids any possibility of "rat's-nest" wiring to the memory chips, making the model essentially unilocal rather than multilocal [36] in its I/O assumptions.

The model makes a few assumptions as possible about the actual location of the I/O memory circuitry, even though this can have a large effect on system timing. If the memory is placed on a different chip from the processing circuitry, its access time is considerably increased. Fortunately, this will not always invalidate the model's timing assumptions. The $O(\lg k)$ delay of a k-bit RAM will dominate the delay of an off-chip driver, if k is large enough. Alternatively, if k is small, it should be relatively easy to locate the RAM on the processor chip. As for off-chip "shift register" I/O memories, there should be no particular difficulty in implementing these in such a way that one input or output event can happen every $O(1)$ time units.

As indicated above, time charges for off-chip I/O are problematical and may be underestimated in the current model. Area charges for I/O are also troublesome. Here, I/O ports are assumed to have $O(1)$ area even though they are obviously much larger than a unit-area wire crossing or an $O(1)$ area gate. It is also assumed that a design can have an unlimited number of I/O ports. In reality, chips are limited to one or two hundred pins, and each pin should be considered a major expense (in terms of manufacturing, reliability, and circuit board wiring costs). An attempt is made in Section IV to use more realistic estimates of I/O costs when evaluating the constructions in Section III.

The complete model of VLSI computation is summarized below.

Assumption 1—Embedding:

a) A VLSI circuit consists of wires and nodes embedded in the Euclidean plane. In graph-theoretic terms, wires are hyperedges joining two or more nodes. These hyperedges are embedded as a tree of (two-ended, straight-line) wire segments and arbitrarily positioned "fan-out points."

b) Wire segments have unit width; the length of a wire is the total length of all its wire segments.

c) At most two wires may cross over each other at any point in the plane.

d) A node occupies $O(1)$ area. Thus, a node has at most $O(1)$ input wires and $O(1)$ output wires. (In general, a node can implement any Boolean function that is computable by a constant number of TTL gates. Hence, an "and gate" or a "j-k flip-flop" is represented by a single node: see Assumption 4.)

e) Wires may not cross over nodes, nor can nodes cross over nodes.

f) Unboundedly long wires and large fan-outs are permissible under the following loading rule. A length-k wire may serve as the input wire for n nodes only if it is the output wire for at least $c_w k + c_g n$ nodes. (In electrical terms, the output nodes work in parallel as current sources or sinks for the capacitive load represented by a wire. The "loading constants" c_w, c_g are always less than one—otherwise it would be impossible to connect two nodes together—but their precise values are technology-dependent.)

Assumption 2—Problem Definition:

a) A chip has degree of concurrency c if it solves c problem instances simultaneously.

b) Each of the N input variables in a problem instance takes on one of M different values with equal likelihood.

c) $M = N^{1+\epsilon}$, for some fixed $\epsilon > 0$. Furthermore, a nearly

nonredundant code is used, so that each input and output value is represented as a word with $\approx(1+\epsilon)(\lg N) = \Theta(\lg N)$ bits. (This assumption makes it possible to express area and time bounds in terms of N alone. It also seems to be required in the proof of a strong lower bound on the area $*$ time^2 complexity of the sorting problem [44].)

d) The output values of a problem instance are a permutation of its input values into increasing order.

Assumption 3—Timing:

a) Wires have unit bandwidth. They carry a one-bit "signal" in each unit of time.

b) Nodes have $O(1)$ delay. (This assumption, while realistic, is theoretically redundant in view of Assumption 3a).)

Assumption 4—Transmission Functions:

a) A transmission function is associated with each node, defining how its outputs and internal state react to the signals on its input wires. More precisely, the "state" of a node is a bit-vector that is updated every time unit according to some fixed function of the signals on its input wires. Similarly, the signals appearing on the output wires of a node are some fixed function of its current state. (With this definition, a node is seen to have the functionality of a finite state automaton of the Moore variety.)

b) If outputs from two or more nodes are connected to the same wire, their output signals must never disagree. To ensure this, the nodes must have identical transmission functions and be connected to the same input wires. (A weaker version of this assumption allows "or-tying" of output wires [44].)

c) Nodes fall into three classes: logic nodes, I/O ports, and I/O memories. I/O memories are further classified as either "RAM-type" or "shift-register-type" memories.

d) I/O memories may not be connected directly to logic nodes.

e) Logic nodes and I/O ports are limited to $O(1)$ bits of state.

f) A $(k_1 \times k_2)$-bit I/O memory contains $k_1k_2 + \lg k_1k_2$ bits of state. An "address register" is formed of $\lg k_1k_2$ bits of state. The other k_1k_2 bits in the state vector are "data."

g) Each problem input bit is assigned to a fixed (i.e., problem-independent) position in some I/O memory's data area. Problem inputs are initially available only at these positions. At the beginning of a computation, all other state bits are initialized to fixed, problem-independent values.

h) Each problem output bit is assigned to a fixed position in an I/O memory. At the completion of a computation, the memory data corresponding to the output bits must have the values defined in Assumption 2. (Note that Assumptions 4a), 4b), 4g), and 4h) make the model deterministic, where-oblivious [25], and unilocal [36].)

i) I/O ports connected to RAM-type $(k_1 \times k_2)$-bit memories can run a memory cycle every $O(k_2 + \lg k_1)$ time units. (These memory cycles are allocated on a first-come, first-serve basis among the $O(1)$ ports connected to a single memory.) During the first $\lg k_1$ time units of a cycle, the port receives a bit-serial "word address" on an input wire. The next input signal is interpreted as a read/write indicator. If a write cycle is indicated, the following k_2 input signals are written into the address word in the memory. Alternatively, during the last k_2 time units of a read cycle, the value of the addressed word appears in bit-serial format on the I/O port's output wire.

j) I/O ports connected to shift-register-type $(k_1 \times k_2)$-bit memories can run a memory cycle every $O(k_2)$ time units. As in the case of a RAM-type memory, cycle requests are served in a FIFO basis. During the first (third, fifth, etc.) time unit of a cycle, the value of the currently addressed data bit is available on the port's output wire. During even-numbered time units of a cycle, the signal appearing on the port's input wire is written into the addressed data bit, and the memory's address register is incremented mod k_1k_2.

Assumption 5—Area, Time Performance:

a) The total processing area $A_{\text{processing}}$ of a chip is the number of unit squares in the smallest enclosing rectangle.

b) The area performance A of a chip is its total area divided by its degree of concurrency c.

c) The total time T_{total} is the average number of time units that elapse between the beginning and end of a computation on c problem instances.

d) The time performance T_d of a chip is the average (over all problem instances) of the number of time units between the first and last memory cycles accessing the data in a particular problem instance.

e) The period T_p of a chip is equal to T_{total} divided by c. Note that $T_p \le T_d \le T_{\text{total}}$, since a chip with $T_p > T_d$ must be "wasting time" between successive problem instances, and thus could be redesigned to have $T_p = T_d$. Also note that if $c = 1$, then $T_p = T_d = T_{\text{total}}$.

f) The I/O bandwidth of a sorting chip is the (average) total number of bits read or written into its I/O memories, divided by T_{total}.

III. Circuit Constructions

This section presents thirteen constructions for sorting chips. Each will be briefly described in its own subsection. First, however, we present a few useful building blocks.

A serial comparison-exchange module (a "comparator") can be built of $O(1)$ gates [27] in $A = O(1)$ area. It has two bit-serial data inputs, A and B, and two bit-serial data outputs, $\max(A, B)$ and $\min(A, B)$. These inputs and outputs are serialized in a binary code, most-significant bit first.

In some applications, two control lines are added to the comparison-exchange module. The four control states are 1) unconditionally "pass-through" the two inputs; 2) unconditionally "swap" the two inputs; 3) send the larger of its two inputs to output 1; 4) send the smaller of its two inputs to output 1. These more complex modules can still fit in $O(1)$ area and produce two output bits every $O(1)$ time units.

Comparison-exchange modules may be pipelined, as illustrated in Fig. 1 for the case of seven-bit words. Pairs of input values enter the module from the top, and move downwards through the array at the rate of one row per time unit. In each row, the circular element performs a comparison-exchange operation on one bit of the inputs; the square elements pass their inputs through unchanged. Information about the "direction" of the comparison-exchange for each pair of input values travels diagonally through the array, from one circle to the next.

A pipelined comparison-exchange module for $O(\lg N)$-bit

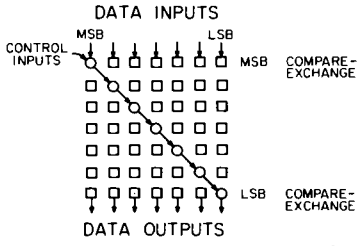

Fig. 1. A pipelined comparison-exchange module.

words can do a complete comparison-exchange operation every $T_p = O(1)$ time units with a delay of $T_d = O(\lg N)$ time units. The total area of the comparator as drawn is $O(\lg^2 N)$, and its concurrency is $\lg N$, giving it an area performance of $A = O(\lg N)$. In most applications the square boxes can be deleted, since the input and output data are already "staggered." This reduces $A_{\text{processing}}$ to $O(\lg N)$, giving the pipelined comparator an area performance of $A = O(1)$. Note that this is identical to the area performance of the nonpipelined comparator.

A third building block is the programmed control unit (PCU). A PCU is used to generate a large number of control signals from a very small area. In the constructions below, entire sorting algorithms are encoded into $O(1)$ PCU instructions. Each instruction is $O(\lg N)$ bits long, and executes in $T_d = O(\lg N)$ time units. The instruction set includes branches, arithmetic operations (shifts, adds, and negations), tests, and register-register moves. A PCU has $O(1)$ different registers. One of these registers is connected to the control lines of a comparison-exchange module. Another register is used to generate address and control signals for any I/O ports in the vicinity.

In the constructions below, the term "bit-serial processor" is used to denote the combination of a PCU, $O(1)$ I/O ports, and a bit-serial comparison-exchange module. Each processor can fit into an $O(1)$-by-$O(\lg N)$ unit rectangle, and can perform one comparison-exchange operation every $T_p = T_d = O(\lg N)$ time units.

"Word-parallel processors" are used to augment the performance of some of the designs. A word-parallel processor is constructed from a PCU, a pipelined comparison-exchange module, and $O(\lg N)$ I/O ports connected to shift-register memories. (There seems to be no reason to use a parallel processor with a random-access memory, since the delay of a serial processor matches the delay of an $O(N)$-word RAM.)

A word-parallel processor can perform one comparison-exchange operation every $T_p = O(1)$ time units, for its inputs are easily "staggered" in the manner required by its pipelined comparison-exchange module. Finally, a word-parallel processor fits into an $O(1)$-by-$O(\lg N)$ units rectangle. It thus occupies the same area as does a serial processor, to within constant factors.

Now we are ready to examine sorting circuits for VLSI. The designs are presented in order of increasing parallelism.

A. Uniprocessor Heapsort

This is the smallest sorter imaginable. It has one bit-serial processor running a standard heapsort algorithm [19, pp. 145–149] on N words of data. Each comparison-exchange and each "random" access to the input data takes $O(\lg N)$ time, so a complete heapsort takes $T_d = T_p = O(N \lg^2 N)$ units of time. The area performance of this design is $A = O(\lg N)$.

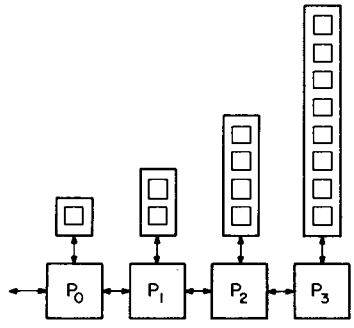

Fig. 2. The $(\lg N)$-processor heapsort for $N = 16$.

Other fast sorting algorithms, such as mergesort or quicksort, could be used in a uniprocessor design. However, none would yield a better AT_d^2 performance, since all require $O(N \lg N)$ random accesses to the processor's I/O memory.

B. $(\lg N)$-Processor Heapsort

Heapsort can be parallelized on a linear array of $\lg N$ bit-serial processors, one for each level of the heap [2], [40] (see Fig. 2). The heap operations are pipelined; during an insertion (or deletion) a data element moves down (or up) the heap by one level every $O(\lg N)$ time units. The processor at the top of the heap stores one data element, the smallest value that has been encountered. The kth processor ($0 \le k < \lg N$) handles 2^k elements, storing them in a $(2^k \times \lg N)$-bit RAM. Total sorting time is $T_d = T_p = O(N \lg N)$, and the area is $A = O(\lg^2 N)$.

C. $(1 + \lg N)$-Processor Mergesort

The mergesort algorithm, like the heapsort, fits quite nicely on about $\lg N$ processors [45]. Two variable-length FIFO queues are associated with each processor; processor P_k ($0 \le k \le \lg N$) has two 2^k-word queues attached to its output lines.

Referring to Fig. 3, processor $P_k (k > 0)$ merges sorted lists of length 2^{k-1} into sorted lists of length 2^k. It does this by placing the smaller of the elements at the head of its two input queues onto the tail of one of its output queues. Once an entire output list of 2^k elements is complete, the processor starts filling its other output queue. This process repeats as long as inputs are presented to the chip.

Processor P_o is a special case. It merely "splits" its input stream into two, placing alternate elements onto its left-hand and right-hand output queues. These elements should be considered sorted lists of length 1, since they are "merged" into sorted lists of length 2 by processor P_1.

To achieve maximal performance, it is tempting to use pipelined, word-parallel processors. Unfortunately, it seems to be impossible to use these efficiently. There is no way to decide which data elements should next be entered into the pipeline until the previous comparison is complete and the appropriate queue is popped. Thus bit-serial processors are used in this paper's mergesort design.

The FIFO queues between processors are most easily built of RAM memory. A 2^k-word RAM's $O(k)$ access time is fast

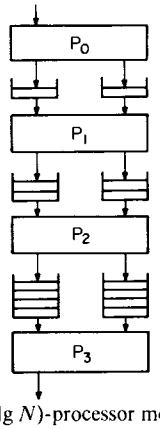

Fig. 3. The $(1 + \lg N)$-processor mergesort for $N = 8$.

enough to keep up with a bit-serial processor working on $O(\lg N)$-bit words, since $k \leq \lg N$. (Alternatively, as pointed out by an ingenious referee, each variable-length 2^k-word FIFO could be implemented with two 2^k-word shift registers. Processor P_{i+1} can empty one of the shift registers while P_i fills up the other one. Even though it requires twice as much memory, the referee's idea is advantageous if shift register memory cells are much smaller or cheaper than RAM memory cells. The idea does, however, have a constant factor disadvantage in time since it introduces an additional $O(2^i)$ delay between P_i's output stream and P_{i+1}'s input stream.)

The time performance of the mergesorter is limited by the data rate of its individual processors. It takes $T_d = O(N \lg N)$ time units for all the input elements to clear the first processor, and another $O(N \lg N)$ time units for the elements to percolate through the $O(N)$ words of internal FIFO storage. Total time for a sort is thus $T_d = O(N \lg N)$. The area of the design is $A = O(\lg^2 N)$, since each of the $1 + \lg N$ processors fits into an $O(\lg N)$ area rectangle.

D. $(\lg N)$-Processor Bitonic Sort

Superficially, this design is very similar to the previous two in that it uses $O(\lg N)$ processors with geometrically varying memory sizes. In this case, processor $P_k (0 \leq k < \lg N)$ has an auxiliary $(N/2^{k+1})$-word fixed-length FIFO queue, as illustrated in Fig. 4. (If the feedback path from $P_{\lg N-1}$ to P_0 were deleted, Fig. 4 would be identical to the "cascade" design for pipelined FFT computation [9].)

The processors execute a bitonic sorting algorithm [19, p. 237]. For the purposes of this paper, the bitonic sort algorithm can be described as $\lg N$ "global iterations." Each global iteration consists of $\log N$ "distance-$(N/2^{k+1})$ operations" on the N input values, in the following order: a distance-$N/2$ operation, a distance-$N/4$ operation, . . . , a distance-2 operation, and finally a distance-1 operation. As indicated by the repeated use of the index k, processor P_k is responsible for performing all distance-$(N/2^{k+1})$ operations. With this in mind, a global iteration is seen to be one complete pass of the data around the ring of processors in Fig. 4.

The "distances" in the "operations" refer to the natural indexing of data values within the bitonic sorter. Initially, input x_i enters processor P_0 just before input x_{i+1}. Using its $N/2$-word FIFO queue, P_0 is able to compare input $x_{N/2+i}$ with input x_i, and to exchange these values (if necessary) before

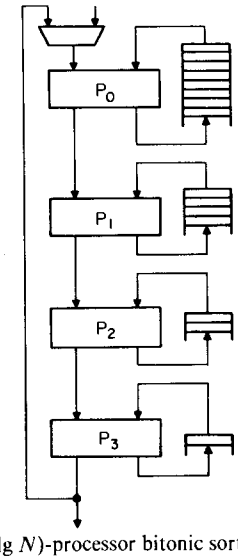

Fig. 4. The $(\lg N)$-processor bitonic sorter for $N = 16$.

passing a naturally indexed sequence $(x'_0, x'_1, \cdots)$ to processor P_1. In other words, processor P_k does compare-exchange operations on data values whose indexes differ only in the kth most significant bit. These index pairs are of the form $(i, N/2^{k+1} + i)$.

Finally, the "direction" of each processor's comparison-exchange operations is not fixed during the course of the bitonic sort algorithm. Sometimes x_i and $x_{N/2^{k+1}+i}$ should be interchanged if $x_i > x_{N/2^{k+1}+i}$, sometimes they should be interchanged if $x_i < x_{N/2^{k+1}+i}$, and sometimes they need not be interchanged under any circumstances (this last is a "no-op" distance-$(N/2^{k+1})$ operation). These three possibilities are reflected in three patterns of data flow through individual processors. When it uses pattern number 1, processor P_k performs a "no-op" by placing the elements it receives from processor P_{k-1} onto the back of its FIFO queue, and sending the elements that come off the front of its queue to processor P_{k+1}. In pattern number 2, processor P_k does a comparison-exchange on the element at the front of its queue and the element it receives from processor P_{k-1}, sending the smaller of the two to processor P_{k+1} and placing the larger on the back of its queue. Pattern number 3 is the same as pattern number 2, except that the larger of the two elements is sent to the next processor and the smaller is placed on the back of the queue. The complete bitonic sort algorithm is summarized in Fig. 5.

Interpreting the algorithm of Fig. 5, processor P_k executes pattern number 1 on the first $(\lg N - k - 1)N$ elements it encounters. This corresponds to $(\lg N - k - 1)$ "global" iterations of the outermost loop in Fig. 5. Processor P_k becomes active in its reordering of the data stream only during the last $k + 1$ global iterations. It alternately fills its queue with new elements (executing $N/2^{k+1}$ instances of pattern number 1) and performs comparison-exchanges of the queue data with the incoming data (executing $N/2^{k+1}$ instances of pattern number 2 or 3).

The conditional expressions of statements 7 and 8 may be implemented with three counters (for g, i, and j) in each processor. The DIV and MOD operations merely select one bit of these counters to control the pattern of data flow. Thus, the bitonic sort can be performed on $\lg N$ bit-serial processors of $O(\lg N)$ area, for a total area of $A = O(\lg^2 N)$. It takes $T_p =$

```
1.  FOR g ← 0 TO lg N − 1 DO
2.    FOR j ← 0 TO 2^t−1 DO
3.      FOR i ← 0 TO N/2^(t+1)−1 DO   /* Fill up FIFO with new data */
4.        {execute pattern 1};
5.      OD;
6.      FOR i ← 0 TO N/2^(t+1)−1 DO  /* Perform comparison-exchanges as necessary */
7.        IF g < lg N − k − 1 THEN {execute pattern 1}
8.        ELSEIF ((j DIV (2^(t+k+1)/N)) MOD 2) = 0 THEN {execute pattern 2}
9.        ELSE {execute pattern 3} FI;
10.     OD;
11.   OD;
12. OD.
```

Fig. 5. The bitonic sort algorithm executed by processor P_k of Fig. 4.

$O(\lg N)$ time for a data element to pass through a bit-serial processor, so that each "global iteration" takes $O(N \lg N)$ times. Total time for a bitonic sort on bit-serial processors is thus $T_d = O(N \lg^2 N)$.

The area $*$ time^2 performance of the design may be improved by using word-parallel processors. Now each global iteration requires only $O(N)$ time, if $O(\lg N)$ communication lines are provided between processors. Total time is $T_d = O(N \lg N)$; total area is still $A = O(\lg^2 N)$. Note that this parallelized design requires $O(\lg^2 N)$ I/O ports, in order to provide sufficient memory bandwidth to the FIFO queues. Also, portions of the control algorithm will have to be hard-wired, so that the three counters described above can be incremented in $O(1)$ time.

It is interesting that the lg N processor heapsorter has exactly the same area and time performance as the lg N processor bitonic sorter, even though the heapsorter does not use parallelized comparators. The heapsort algorithm requires each of the lg N processors to make "random accesses" to their local memory. The extra time taken by these slower accesses is exactly balanced by the greater number of comparison-exchange operations required by the bitonic sorting algorithm.

(Chung, Luccio, and Wong have also proposed a lg N-processor bitonic sort for a magnetic bubble memory system [8]. Their algorithm has an inferior time performance to the one described above, since only one of their processors is active at any time.)

E. $O(\lg^2 N)$-Processor Bitonic Sort

This design "unrolls" the lg N-processor bitonic sort, so that each processor is responsible for only one distance-$N/2^{k+1}$ operation. Since about half of the $\lg^2 N$ operations of the bitonic sort algorithm are no-ops, only about $(1/2)\lg^2 N$ processors are required in this version of the algorithm; see Fig. 6. Each processor fits in an $O(1)$-by-$O(\lg N)$ unit rectangle, so the entire design occupies $O(\lg^3 N)$ area.

A surprisingly large amount of time and FIFO storage area is saved by eliminating the no-ops when "unrolling" the bitonic sort on lg N processors. Since a distance-$N/2^{k+1}$ operation is implemented with $N/2^{k+1}$ words of FIFO storage, and since all but $k+1$ of the distance-$(N/2^{k+1})$ operations are no-ops, the total storage is $\Sigma(k+1)(N/2^{k+1})$, or a little less than $2N$ words. The problem solution time is proportional to the length

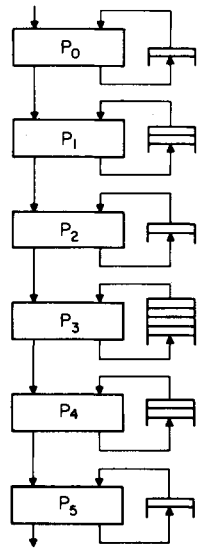

Fig. 6. The $(1/2)(\lg N)(1 + \lg N)$-processor bitonic sorter for $N = 8$.

of this pipeline, or $T_d = O(N)$ if word-parallel processors are used. The area performance is half of its total area, $A = O(\lg^3 N)$, because the pipeline stores two problems at a time.

The AT_d^2 performance of this design is a factor of lg N better than that of the previous design. To understand this phenomenon, it is helpful to compare the performance of one $O(\lg^2 N)$-processor bitonic sorter with that of a collection of lg N identical (lg N)-processor bitonic sorters. Both have the same amount of total area, and both solve lg N sorting problems in $T_{\text{total}} = O(N \lg N)$ time (if word-parallel processors are used). Due to the elimination of the no-ops, however, the $O(\lg^2 N)$-processor implementation solves each sorting problem with logarithmically less delay.

F. $\sqrt{N \lg N}$-Processor Bitonic Sort

Chung, Luccio, and Wong have recently proposed implementing a bitonic sort on $\sqrt{N \lg N}$ processors in a linear array [8]. Here, each processor has $\sqrt{N/\lg N}$ words of shift register storage. It can run a serial bubble sort algorithm on its local store in $T_d = O(N/\lg N)$ time, if it uses word-parallel processors. Working together, the entire array performs an N-element sort in $T_d = O(N)$ and $A = O(\sqrt{N \lg^3 N})$. The total area increases to $A = O(N \lg N)$ if the shift registers are made of logic nodes rather than I/O memories, in order to decrease the circuit's unreasonably large I/O bandwidth.

According to the model of this paper, this approach is highly nonoptimal in an AT_d^2 sense. It is no faster, but much larger, than the $O(\lg^2 N)$-processor bitonic sorting design.

G. $(N/2)$-Comparable Bubble Sort

As noted by a number of researchers, the bubble sorting algorithm can be fully parallelized on a linear array of $N/2$ bit-serial (or pipelined) comparison-exchange modules [3], [6], [8], [12], [15], [24], [28], [29]. Each module performs the following simple computation: Of the two data elements it receives from its left- and right-hand neighbors, it sends the smaller to the left and the larger to the right. The array can be

initialized in parallel with zeros, then serially loaded with N data elements through the leftmost module. If it is then "flushed out" by loading maximal elements through the leftmost module, the N data elements will emerge from the leftmost module in $O(N)$ comparison times.

The area of the $N/2$-comparator bubble sorter is $A = O(N \lg N)$, since each comparator occupies $O(\lg N)$ area. When bit-serial modules are used, each comparison takes $O(\lg N)$ time, so $T_d = O(N \lg N)$. Word-parallel modules improve the bubble-sorter's delay to $T_d = O(N)$. Even so, its AT_d^2 performance remains dismal. According to the $AT_d^2 = \Omega(N^2 \lg N)$ lower bound, a sorter with $O(N \lg N)$ processing area should sort in about $T_d = O(\sqrt{N})$ time.

There are at least three other ways of sorting N numbers on $O(N)$ processors with similar area-time performance figures. Heapsort can be run on a balanced binary tree of N bit-serial processors [26, pp. 297–299]. This tree structure can also be used to perform the broadcast operations required in recently proposed implementations of enumeration sort [49] and radix sort [10], [47] on N processors. If built from bit-serial processors, these designs would have $A = O(N \lg N)$, $T_d = O(N \lg N)$. It might be possible to use pipelined comparators and word-parallel communication lines efficiently in these designs, although it is not obvious how this can be done. In any event, the AT_d^2 performance of these designs is intrinsically cubic (instead of quadratic) in N, making them highly nonoptimal for large sorting problems.

H. N-Processor Bitonic Sort on Mesh

The bitonic sort can be adapted to run very efficiently on N bit-serial processors connected in a square mesh [31], [42]. Word-parallel connections are used in the mesh in order to speed up the movement of data over long distances.

The operation of this algorithm is rather complicated and will not be explained here. It is sufficient to know that the $O(N \lg^2 N)$ comparison-exchanges in the bitonic sort require a total of $O(\lg^3 N)$ of the N processors' time. However, it can take as much as $O(\sqrt{N})$ time to rearrange the data among the processors in preparation for the next comparison-exchange step. Fortunately, only a few of the comparison-exchange operations take this amount of time, so that the total time to sort N elements is only $T_d = O(\sqrt{N})$.

To achieve the time bound asserted above, it is necessary to move words of data from one processor to the next in $O(1)$ time. This is a little difficult to arrange, since the wires between neighboring processors are $O(\lg N)$ units long. According to the model's loading rules [see Assumption 1f)], $T_d = O(\lg\lg N)$ time is required to amplify a signal for a wire of this length. Once a signal has been applied, it can be received, gated, and retransmitted in $T_d = O(1)$ time by $O(\lg N)$ identical, unit-sized nodes working in concert.

The total area of the design is $A = O(N \lg^2 N)$. Note that the N processors take up only $O(N \lg N)$ area. The word-parallel data paths (and their high-power drivers) require more room than the processors themselves, in the asymptotic limit.

(Batcher's odd-even sorting algorithm [21] or a merge sort algorithm [41] may have constant factor advantages for some N. Aside from these constant factors, both algorithms can be implemented on the mesh with the same AT_d^2 performance as that derived above for Batcher's bitonic sort.)

I. N-Processor Bitonic Sort on Shuffle-Exchange Net

Stone notes that the bitonic sort is easily adapted to run on N bit-serial processors interconnected in the shuffle-exchange pattern [39]. If bit-serial interconnections are used, the $O(N \lg^2 N)$ comparison-exchanges in the bitonic sort take a total of $T_d = O(\lg^3 N)$ time.

Given that this design sorts so quickly, it should not be surprising that it cannot be laid out in a small amount of area. Indeed, it is possible to prove that the smallest layout for the shuffle-exchange graph occupies $\Omega(N^2/\lg^2 N)$ area [42]. An embedding of this size has been recently obtained [18], [23]. This embedding can be "stretched" in the vertical direction in $O(\lg N)$ places to make room for N bit-serial processors, each occupying an $O(1)$-by-$O(\lg N)$ rectangle. The modified embedding has area $A = O(N^2/\lg^2 N)$, and its longest wires are of length $O(N/\lg N)$.

The AT_d^2 performance of this design is a little suboptimal. One might attempt to improve it by adding parallelism to the interprocessor communication lines. If, for example, k wires were laid down for every edge in the shuffle-exchange graph, the resulting network could conceivably sort k times faster. Unfortunately, network area would increase by a factor of k^2, leaving the AT_d^2 performance figure unchanged.

(To achieve the factor of k speedup alluded to in the previous paragraph, the processors would have to be fitted with parallel comparison-exchange modules. More significantly, the size of each processor would have to be increased, so that it could drive its output wires with only $O(\lg N)/d$ delay. Thus, it seems that the maximum-possible speedup factor of $k = \Theta(\lg N)$ would be very difficult to achieve in a layout $\Theta(\lg N)$ times as large: there would only be $O(N)$ area available for each of the N processors; each of the $\Theta(\lg N)$ gates in each processor could be implemented with at most $O(N/\lg N)$ unit-sized nodes; the longest wires in a Leighton-style layout would have length $O(N)$; the long-wire driver delay would be $O(\lg\lg N)$; the time per comparison-exchange would be dominated by these long-wire delays; and the resulting circuit would sort in $T_d = O(\lg^2 N \lg\lg N)$ instead of $O(\lg^2 N)$. The problem with this construction is that no one knows how—or whether it is possible—to lay out the N-node shuffle-exchange graph using edges of length at most $O(N/\lg^2 N)$ [23].)

J. N-Processor Bitonic Sort on the PCCC

Preparata and Vuillemin [33] have shown that their "cube-connected cycles" (CCC) interconnection pattern can run the bitonic sort algorithm as efficiently as the shuffle-exchange pattern; $A = O(N^2/\lg^2 N)$ and $T_d = O(\lg^3 N)$. Their network has the advantage of having a simple, asymptotically optimal, layout; the asymptotically optimal layout for the shuffle-exchange graph is much less uniform. On the other

hand, the bitonic sort algorithm for the CCC is somewhat more complicated than the bitonic sort on the shuffle-exchange.

Quite recently, the CCC design has been improved to achieve AT_d^2 optimality over a wide range of areas and times. The new construction is called the "pleated cube-connected cycles" (PCCC) [4]. The idea behind the construction is to provide more "short" wires for the shorter-distance operations of the bitonic sort algorithm (see Section III-D). The less-frequent long-distance operations are performed more slowly over fewer "long" wires. As the average wire length of a PCCC increases, its area increases but its sorting time decreases. The result is a family of networks whose areas range from $A = O(N \lg N)$ to $A = O(N^2/\lg^4 N)$. Each network sorts in AT_d^2-optimal time: from $T_d = O(\sqrt{N \lg N})$ to $T_d = O(\lg^3 N)$.

The times quoted for the PCCC are for bit-serial implementations. Conceivably, parallel PCCC networks could be devised to run a bitonic sort in $A = O(N^2/\lg^2 N)$ and $T_d = O(\lg^2 N)$. Any such construction would be complicated by the problem of wire delays that was encountered in the attempt in Section III-I to speed up the shuffle-exchange network.

(Another approach to improving the CCC's performance is suggested by Reif and Valiant [34]. They have devised a probabilistic algorithm that would run in $T_d = O(\lg^2 N)$ time on the $A = O(N^2/\lg^2 N)$ CCC, if each processor had access to a random number generator. It would be interesting to know if this time performance is achievable by a deterministic algorithm, in keeping with this paper's model of computation.)

K. ($N \lg^2 N$)-Comparator Bitonic Sort

Batcher's bitonic sorting network [19, p. 237] can be laid out explicitly on a VLSI chip. Each of the $(1/2)(\lg^2 N + \lg N)$ parallel comparison-exchange operations is implemented by a row of $N/2$ bit-serial comparison-exchange modules. The bit-serial interconnections between the rows of comparators require more room than the comparators themselves, at least asymptotically. The wiring in front of the comparators doing a "distance-$(N/2^{k+1})$ operation" takes up an $O(N/2^{k+1})$ by $O(N)$ area of the chip. (As in Section III-D, $0 \le k < \lg N$.) The total area occupied by the network is thus $\Sigma(k+1)$-$(N^2/2^{k+1}) = O(N^2)$, since there are $k+1$ distance-$(N/2^{k+1})$ comparison-exchange operations.

The total delay through the network is $T_d = \Theta(\lg^3 N)$, for each of the $\Theta(\lg^2 N)$ rows adds $\Theta(\lg N)$ delay. Since there are no feedback paths, the network is easily pipelined with a concurrency of $\Theta(\lg^2 N)$. The area performance is the total area divided by the concurrency $A = O(N^2/\lg^2 N)$.

An improvement can be made to the construction outlined above, leading to a better AT_d^2 performance. There is no need for multiple stages of amplification at the outputs of the bit-serial comparators if the comparators themselves are "scaled up" to match the length of their output wires. One such comparator for a distance-$(N/2^{k+1})$ operation occupies $O(N/2^{k+1})$ area. (An $O(\lg N)$-stage driver is needed at the output of each distance-(1) comparator to amplify its signal for the following scaled-up comparator.) The total area is still $O(N^2)$: the comparators now take up space asymptotically equal to that of the long-wire drivers in the original construction. Sorting delay is decreased since bits travel from one row of comparators to the next in $T_d = O(1)$ time. The improved network has total delay $T_d = O(\lg^2 N)$, concurrency $O(\lg N)$, and area performance $A = O(N^2/\lg N)$.

Pratt has pointed out that shellsort [19, p. 84] can be implemented on either $(\lg^2 N)$ or $(N \lg^2 N)$ processors with the same AT_d^2 performance as the bitonic sort.

As this article was going to press, an $O(N \lg N)$-comparator, $O(\lg N)$-depth sorting network was reported [1]. Implemented in a fully parallel fashion, it could be built of $O(\lg N)$ rows of $N/2$ comparators each. The connections between the rows are "expander graphs" with an average wire length of $\Theta(N)$. Total area is $O(N^2 \lg N)$. Sorting delay need only be $T_d = O(\lg N)$ if each comparator is "scaled-up" by a factor of N. Thus, $AT_d^2 = O(N^2 \lg^3 N)$, an asymptotic improvement over the $O(N \lg^2 N)$-comparator bitonic sorter. This asymptotic improvement is nonetheless a constant-factor disaster, for there are an astronomically large number of rows in the currently proposed "$O(\lg N)$-depth" construction.

L. N^2-Comparator Bubble Sort

A final attempt can be made to optimize the bubble sort for VLSI, providing a different comparison-exchange module for each of the N^2 comparisons in a bubble sort on N elements [19, p. 224]. The resulting network is not very impressive. If built from bit-serial comparators, it occupies $O(N^2)$ area. Total delay through the network is $T_d = O(N)$, and $N/\lg N$ problem instances will fit in it at any given time. Its area performance is its total area divided by its concurrency, $A = O(N \lg N)$. Note that the same time performance can be obtained in a small fraction of this area with the $\lg^2 N$-processor bitonic sorting design.

When built of pipelined comparison-exchange modules, the N^2-processor bubble sorter occupies a total of $N^2 \lg^2 N$ area. Its concurrency increases to about $N \lg N$, giving it the same area performance as before, $A = O(N \lg N)$. Strangely enough, its time performance worsens, becoming $T_d = O(N \lg N)$. The reason for this anomalous behavior is that the bit-serial implementation is already fully pipelined.

M. (N^2)-Processor Rank Sort

Consider a square array of N^2 processors, interconnected in the following peculiar way. The N processors in each row are the leaves of a balanced binary tree; the internal vertices of the "row trees" provide bit-serial communication paths between the root of the tree and its N "leaf" processors. Similarly, a "column tree" provides connections between the N processors in each column of the array. Each processor is thus a leaf vertex in two orthogonal trees.

This network has been called by various names, including the "orthogonal tree network" [32] and the "mesh of trees" [22], [23]. Fig. 7 illustrates it for the case $N = 4$.

A brute-force sorting algorithm can be implemented on this network, as pointed out by the authors cited above. (Muller and Preparata [30] describe this algorithm without reference to the natural shape of the orthogonal tree network.) Each of the N inputs to a sorting problem can be presented to one of the root vertices of a row tree. The inputs are then broadcast

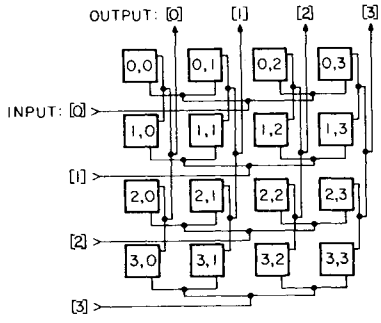

Fig. 7. The orthogonal tree network for $N = 4$.

TABLE I
AREA–TIME BOUNDS FOR THE SORTING PROBLEM

Design	Area Perf. (A)	Time Perf. (T_d)	AT_d^2
Lower bound	-	-	$\Omega(N^2 \lg N)$
1. Uniprocessor (s.)	$\lg N$	$N \lg^2 N$	$N^2 \lg^5 N$
2. $\lg N$ - proc. heapsort (s.)	$\lg^2 N$	$N \lg N$	$N^2 \lg^4 N$
3. $\lg N$ - proc. mergesort (s.)	$\lg^2 N$	$N \lg N$	$N^2 \lg^4 N$
4. $\lg N$ - proc. bitonic (p.)	$\lg^2 N$	$N \lg N$	$N^2 \lg^4 N$
5. $\lg^2 N$ - proc. bitonic (p.)	$\lg^3 N$	N	$N^2 \lg^3 N$
6. $\sqrt{N \lg N}$ - proc. bitonic (p.)	$N \lg N$	N	$N^2 \lg N$
7. $N/2$ - comp. bubble (p.)	$N \lg N$	N	$N^2 \lg N$
8. N - proc. bitonic, mesh (s.)	$N \lg^2 N$	$\sqrt{N}$	$N^2 \lg^2 N$
9. N - proc. bitonic, S-E (s.)	$N^2/\lg^2 N$	$\lg^3 N$	$N^2 \lg^4 N$
10. N - proc. bitonic, PCCC (s.)	$N^2 \lg^2 N / T_d^2$	$\lg^3 N \le T_d \le \sqrt{N \lg N}$	$N^2 \lg^2 N$
11. $N \lg^2 N$ - comp. bitonic (p.)	$N^2/\lg N$	$\lg^2 N$	$N^2 \lg^3 N$
12. N^2 - comp. bubble (s.)	$N \lg N$	N	$N^2 \lg N$
13. N^2 - proc. rank sort (s.)	$N^2 \lg^2 N$	$\lg N$	$N^2 \lg^4 N$

to the leaves of the tree, so that each processor in a row has a copy of that row's input. Next, the column trees come into play: the jth leaf processor of the jth column tree sends a copy of its input to its root. This value is "broadcast" downwards through the column trees, so that processor (i, j) now contains copies of two input values, $input[i]$ and $input[j]$.

The next step in the sorting algorithm is to compute the ranks of the inputs. The ith row tree evaluates the rank of input i by "summing" the results of comparing $input[i]$ with $input[j]$. To be more specific, processor (i, j) compares its two input values, sending a "1" up through its row tree if $input[i] > input[j]$ or if $((input[i] = input[j])$ and $(i > j))$. These values are summed by the row trees. A moment's reflection should convince the reader that the sum of the values in row i is the rank of $input[i]$, with ties being broken by the $i > j$ calculation. The root of each row tree will have a different integer from the range of possible ranks, $[0, N - 1]$.

The input ranks are next broadcast to the leaf vertices of the row trees. Finally, processor (i, j) sends up ("selects") the value of $input[i]$ through its column tree if $rank[i] = j$. The sorted values are now available at the roots of the column trees.

It is a straightforward exercise to verify that the internal vertices can perform the broadcast, summation, and selection operations in a bit-serial fashion with $O(1)$ bits of storage and $O(1)$ logic devices.

It remains to establish the area and time complexity of this sorting procedure. Since broadcast, summation, and selection operations are performed on trees with N leaves, the best possible time performance is $T_d = O(\lg N)$. Also, as proved in [23], the minimum possible area for the orthogonal tree network is $O(N^2 \lg^2 N)$. These performance figures are in fact achievable, but only with careful design.

Observe that in Fig. 7 the wires connecting vertices in the row and column trees are not all of the same length. The closer one gets to the root, the longer the wires; the wires double in length from one level of the tree to the next. Thus, each logic device in an internal vertex on level k should be built of $O((N \lg N)/2^k)$ unit-sized nodes, enabling the vertex to drive its $O((N \lg N)/2^k)$-length wires at unit delay, as required. Since there are $O(1)$ logic devices in an internal vertex, and since there are 2^k vertices at level k, each level fits into the $O(1)$-by-$O(N \lg N)$ rectangle available to it in the "obvious" $O(N^2 \lg^2 N)$-area layout suggested by Fig. 7.

IV. COMPARISON OF THE DESIGNS

The area and time performance of the thirteen sorting circuits is summarized in Table I below. For easy reference, the designs are numbered from 1 to 13 in correspondence with Sections III-A–III-M in which they are discussed. The "s" or "p" in the design's name indicates whether it is implemented with serial or pipelined comparators. The "area performance" column gives each design's processing area divided by its concurrency, in accordance with the definition in Section II. This metric is an indication of the power consumed per sorting problem. A design's "time performance," as formally defined in Section II, can be characterized as the elapsed time between the first input to the circuit and the last output from the chip for each sorting problem.

Table I shows that nearly all the designs considered in this paper are within a factor of $O(\lg^k N)$ of being optimal in an AT_d^2 sense. The sole exceptions are the bubble sorters and the $\sqrt{N \lg N}$-processor bitonic sorter.

Table II contains additional information about the sorters. (The PCCC design appears twice in this table, on lines 10a and 10b, in its slowest and fastest forms. Intermediate figures are possible: see Section III-J.) The first entry for each design shows its concurrency, defined as the number of sorting problems that should be solved simultaneously to achieve the maximum possible area performance. The second and third entries indicate the total "processing" and "memory" area required in a system containing that design. (One bit of RAM or shift-register memory occupies one unit of area.) The final entry is the design's processor-memory bandwidth in bits/(unit time). This entry is also equal to the number of I/O ports required on the design's "processing" chip, since all the designs keep all their ports busy essentially all the time.

Thus far in the paper, A_{memory} and I/O bandwidth have not been considered explicitly. Furthermore, $A_{\text{processing}}$ has largely ignored in deference to the theoretically "correct" area performance measure $A = A_{\text{processing}}/c$. From a systems perspective, however, the measures of Table II are more important than those of Table I. A design will occupy circuit board area proportional to $A_{\text{processing}} + A_{\text{memory}}$, and its "processing"

TABLE II
OTHER PERFORMANCE MEASURES

Design	Concurrency	$A_{processing}$	A_{memory}	I/O B.W.
1. Uniprocessor	1	$\lg N$	$N \lg N$	1
2. $\lg N$ - proc. heapsort	1	$\lg^2 N$	$N \lg N$	$\lg N$
3. $\lg N$ - proc. mergesort	1	$\lg^2 N$	$N \lg N$	$\lg N$
4. $\lg N$ - proc. bitonic	1	$\lg^2 N$	$N \lg N$	$\lg^2 N$
5. $\lg^2 N$ - proc. bitonic	2	$\lg^3 N$	$N \lg N$	$\lg^3 N$
6. $\sqrt{N \lg N}$ - proc. bitonic	1	$N \lg N$	$N \lg N$	$\lg N$
7. $N/2$ - proc. bubble	1	$N \lg N$	$N \lg N$	$\lg N$
8. N - proc. bitonic, mesh	1	$N \lg^2 N$	$N \lg N$	$\sqrt{N} \lg N$
9. N - proc. bitonic, S-E	1	$N^2/\lg^2 N$	$N \lg N$	$N/\lg^2 N$
10a. N - proc. bitonic, PCCC	1	$N \lg N$	$N \lg N$	$\sqrt{N \lg N}$
10b. N - proc. bitonic, PCCC	1	$N^2/\lg^4 N$	$N \lg N$	$N/\lg^2 N$
11. $N \lg^2 N$ - proc. bitonic	$\lg N$	N^2	$N \lg^2 N$	N
12. N^2 - proc. bubble	$N/\lg N$	N^2	N^2	N
13. N^2 - proc. rank sort	1	$N^2 \lg^2 N$	$N \lg N$	N

portion will have to have enough pins to support its I/O bandwidth.

Of course, a sorting circuit should not be selected just because it is asymptotically optimal. A digital engineer is interested only in actual speeds and sizes. Although the model of computation of this paper is not exact enough to permit such analyses, some statements can be made about the relative sizes and speeds of the designs.

The smallest design is clearly the $O(\lg N)$ area uniprocessor. Somewhat surprisingly, this design is nearly AT_d^2 optimal if it is programmed to use any of the $O(N \lg N)$-step serial algorithms.

If more sorting speed is desired, the ($\lg N$)-processor heapsort design becomes attractive. It requires almost exactly $\lg N$ times as much area as the uniprocessor design, since the processors and programs for the two designs are very similar. The design has the smallest possible delay of any sorter that receives its inputs in a single bit-serial stream, since the first output is available immediately after the last input has been received. (The $N/2$-processor bubble sorter is the only other design considered in this paper that has this property. All others introduce at least an $O(\lg N)$ delay between the last input time and the first output time.)

A major drawback of ($\lg N$)-processor heapsorter is that it requires $\lg N$ independently addressable memories, one for each processor. The total memory-processor bandwidth increases proportionately (see Table II) to $\lg N$ bits per time unit.

The ($\lg N$)-processor bitonic design has been the same area and time performance as the ($\lg N$)-processor heapsort design. The former has the advantage of a slightly simpler control algorithm, and it uses the simpler shift-register type of I/O memory; the latter uses a more efficient sorting algorithm and hence less memory bandwidth.

The ($\lg^2 N$)-processor bitonic sorter is smaller than either of the ($\lg N$) processor designs, for moderately sized N. Its control algorithm is extremely simple, so that a "processor" is not much more than a comparison-exchange module. Its major drawback is that it makes continuous use of $(1/2) * (\lg N) * (\lg N + 1)$ word-parallel shift-register memories, of various sizes.

The ($\sqrt{N \lg N}$)-processor bitonic sorter has been entered in Table II with a total area of $O(N \lg N)$, so that there is room on the chip for all of its temporary storage registers. Otherwise, it would require $\sqrt{N \lg N}$ separate I/O memories. It has the same speed and a somewhat better I/O bandwidth than the ($\lg^2 N$)-processor bitonic sorter just discussed. However, the latter's shift registers could also be placed on the same chip as its processing circuitry, equalizing the I/O bandwidth for the two designs. When "constant factors" are taken into consideration, the ($\sqrt{N \lg N}$)-processor design is clearly much larger than the ($\lg^2 N$)-processor design, because it has more processors and a much more complicated control algorithm.

The ($N/2$)-processor bubble sorter has a couple of significant advantages that are not revealed in either Table I or Table II. Its comparators need very little in the way of control hardware, so that at least for small N, it occupies less area than any of the preceding designs. Also, it can be used as a "self-sorting memory," performing insertions and deletions on-line. (The uniprocessor and the ($\lg N$)-processor heapsorter can also be used in this fashion.) However, for even moderately sized N, the bubble sorter's horrible AT_d^2 performance becomes noticeable. For example, when $N = 256$, the ($\lg^2 N$)-processor bitonic sorter's 36 comparators and 491 words of storage probably occupy less room than the 128 comparators in a bubble sorter. Nonetheless, the bubble sorter always maintains about a 2:1 delay advantage over the ($\lg^2 N$)-processor bitonic sorter, when similar comparators are used.

The N-processor mesh-type bitonic sorter is the first design to solve a sorting problem in sublinear time. Unfortunately, it occupies a lot of area. Each of its processors must run a complicated sorting algorithm, reshuffling the data among themselves after every comparison-exchange operation. Its I/O bandwidth must also be large, since it solves sorting problems so rapidly. However, constant factor improvements may be made to its area and bandwidth figures by reprogramming the processors so that each handles several data elements at a time. Also, large area and bandwidth are not always significant problems: in an existing mesh-connected multiprocessor, the N processors are already in place and the I/O data may be produced and consumed by local application routines.

The next three designs in Tables I and II are variants on a fully parallelized bitonic sort. The shuffle-exchange sorter has a slight area advantage over the CCC or PCCC sorter because of its simpler control algorithm. However, the CCC and PCCC are somewhat more regular interconnection patterns, so that they may be easier to wire up in practice. Both designs are smaller in total asymptotic area than the ($N \lg^2 N$)-processor bitonic sorter, which solves $\lg N$ problems at a time. Nonetheless, the control structure of this last design is so simple that, as a rough guess, it takes less area than the others for all $N < 2^{20}$. (Of course, if a shuffle-exchange or PCCC processor has already been built, the additional area cost for programming the sorting algorithm is very small.)

There seems to be little to recommend the N^2-processor bubble sorter. It has the same I/O bandwidth, a bit more total area, and a much worse time performance than the ($N \lg^2 N$)-processor bitonic sorter.

Finally, the N^2-processor rank sorter can be characterized

as being larger but not all that much faster than the N- and (N lg N)-processor bitonic sorters. Its chief interest is theoretical: it sorts in a minimal number, $O(\lg N)$, of gate delays. No other sorting circuit of equivalent time performance could possibly beat its area performance by more than a logarithmic factor or two, considering the theoretical limit of $AT_d^2 = \Omega(N^2 \lg)$. However, it remains an open question whether it is possible to build a $T = O(\lg N)$ sorter that occupies even a little less area.

V. Closing Remarks

At the time of this writing, there are a number of important open questions in VLSI complexity theory. A simply stated but seemingly perplexing problem is to find out how much area can be saved when additional "layers" of wiring are made available by technological advances. It is known that a k-level embedding can be no smaller than $1/k^2$ of the area of a two-level embedding [42, pp. 36–38], but it is not known whether this bound is achievable. (Some results on k-level embedding have been obtained recently [35].)

A second problem is to derive matching upper and lower bounds for the area * time^2 complexity of the sorting problem. The best upper bound is $AT_d^2 = O(N^2 \lg^2 N)$, achieved by the N-processor bitonic sort on a mesh and by the PCCC. As discussed in Section I, the best lower bound is $AT_p^2 = \Omega(N^2 \lg N)$ [44], which leaves a gap of $O(\lg N)$. The gap can be closed by adding the assumption that all lg N bits of each input value are read in through a single I/O port [42]. (The current model allows the bits of each input value to be read in through different ports.) It seems probable that the $AT_p^2 = \Omega(N^2 \lg^2 N)$ result for word-oriented, "localized" I/O can be extended to handle the less restrictive model of this paper. On the other hand, it is conceivable that such a bound is impossible because of the existence of some yet-to-be-discovered sorting circuit with an AT_d^2 performance better than that of the bitonic sort on the mesh or PCCC.

A third problem is to evaluate separately the VLSI complexity of the "ranking" and "permutation" subproblems of the sorting problem, as discussed in Section I. Current lower bounds for the sorting problem can be viewed as arguments about the data flow required in a permuter; they seem to say little about the problem of ranking the data. Lower bounds on ranking would presumably be more subtle and give more insight into how "control information" must flow in a sorting circuit.

Another set of problems is opened up by the fact that the area * time^2 bounds are affected greatly by nondeterministic, stochastic, or probabilistic assumptions in the model. My intuition is that the VLSI complexity of sorting is not sensitive to such assumptions, but it would be nice to be sure of this. Counterintuitive results have already been proved: equality testing is very easy if the answer need only be "probably" correct [25], [48].

A final and very important problem in VLSI theory is the development of a stable model. Currently there are almost as many models as papers. If this trend continues, results in the area will become difficult to report and describe. However, it is far from settled whether wire delays should be treated as being linear or logarithmic in wire length, and the costs of off-chip communication remain unknown.

Acknowledgment

Referee 1's detailed and thought-provoking report is gratefully acknowledged.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," in *Proc. 15th Annu. ACM Symp. Theory Comput.*, Apr. 1983, pp. 1–9.
[2] P. K. Armstrong, U.S. Patent 4 131 947, Dec. 26, 1978.
[3] P. K. Armstrong and M. Rem, "A serial sorting machine," *Comput. Elect. Eng.*, Pergamon, vol. 9, Mar. 1982.
[4] G. Bilardi, M. Pracchi, and F. P. Preparata, "A critique of network speed in VLSI models of computation," *IEEE J. Solid-State Circuits*, vol. SC-17, pp. 696–702, Aug. 1982.
[5] R. Brent and H. T. Kung, "The area-time complexity of binary multiplication," *J. Ass. Comput. Mach.*, vol. 28, pp. 521–534, July 1981.
[6] T. C. Chan, K. P. Eswaren, V. Y. Lum, and C. Tung, "Simplified odd-even sort using multiple shift-register loops," *Int. J. Comput. Inform. Sci.*, vol. 7, pp. 295–314, Sept. 1978.
[7] B. Chazelle and L. Monier, "Towards more realistic models of computation for VLSI," in *Proc. 11th Annu. ACM Symp. Theory Comput.*, Apr. 1979, pp. 209–213.
[8] K.-M. Chung, F. Luccio, and C. K. Wong, "On the complexity of sorting in magnetic bubble memory systems," *IEEE Trans. Comput.*, vol. C-29, pp. 553–562, July 1980.
[9] A. Despain, "Very fast Fourier transform algorithms for hardware implementation," *IEEE Trans. Comput.*, vol. C-28, pp. 333–341, May 1979.
[10] Y. Dohi, A. Suzuki, and N. Matsui, "Hardware sorter and its application to data base machine," in *Proc. 9th Annu. Symp. Comput. Arch. (ACM SIGARCH Newsletter)*, vol. 10, Apr. 1982, pp. 218–225.
[11] S. A. Evans, "Scaling I^2L for VLSI," *IEEE J. Solid-State Circuits*, vol. SC-14, pp. 318–326, Apr. 1979.
[12] L. J. Guibas and F. M. Liang, "Systolic stacks, queues, and counters," in *Proc. 1982 Conf. Advanced Res. VLSI*, Massachusetts Inst. Technol., Cambridge, Jan. 1982, pp. 155–164.
[13] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, May 1981, pp. 326–333.
[14] J.-W. Hong, "On similarity and duality of computation," Peking Munic. Computing Center, Peking, People's Repub. China, 1981, unpublished.
[15] G. Kedem, "A first in, first out and a priority queue," Dep. Comput. Sci., Univ. Rochester, Rochester, NY, Tech. Rep. 90, Mar. 1981.
[16] Z. M. Kedem and A. Zorat, "Replication of inputs may save computational resources in VLSI," in *Proc. 22nd Symp. Found. Comput. Sci.*, IEEE Comput. Soc., Oct. 1981.
[17] M. B. Ketchen, "AC powered Josephson miniature system," in *Proc. 1980 Int. Conf. Circuits Comput.*, IEEE Comput. Soc., Oct. 1980, pp. 874–877.
[18] D. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller, "New layouts for the shuffle-exchange graph," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, May 1981, pp. 334–341.
[19] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[20] ——, "Bit omicron and big omega and big theta," *SIGACT News*, vol. 8, pp. 18–24, Apr.–June 1976.
[21] M. Kumar and D. S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," *IEEE Trans. Comput.*, vol. C-32, pp. 254–264, Mar. 1983.
[22] F. T. Leighton, "New lower bound techniques for VLSI," in *Proc. 22nd Symp. Found. Comput. Sci.*, IEEE Comput. Soc., Oct. 1981.
[23] ——, "Layouts for the shuffle-exchange graph and lower bound techniques for VLSI," Ph.D. dissertation MIT/LCS/TR-724, M.I.T. Lab. for Comput. Sci., Massachusetts Inst. Technol., Cambridge, June 1982.
[24] C. E. Leiserson, "Area-efficient VLSI computation," Ph.D. dissertation CMU-CS-82-108, Comput. Sci. Dep., Carnegie-Mellon Univ., Pittsburgh, PA, Oct. 1981.

[25] R. J. Lipton and R. Sedgewick, "Lower bounds for VLSI," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, May 1981, pp. 300–307.
[26] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[27] H. P. Moravec, "Fully interconnecting multiple computers with pipelined sorting nets," *IEEE Trans. Comput.*, vol. C-28, pp. 795–798, Oct. 1979.
[28] A. Mukhopadhyay and T. Ichikawa, "An n-step parallel sorting machine," Dep. Comput. Sci., Univ. Iowa, Iowa City, Tech. Rep. 72-03, 1972.
[29] A. Mukhopadhyay, "WEAVESORT—A new sorting algorithm for VLSI," Univ. Central Florida, Orlando, Tech. Rep. 53-81, 1981.
[30] D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and for switching," *J. Ass. Comput. Mach.*, vol. 22, pp. 195–201, Apr. 1975.
[31] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol. C-28, pp. 2–7, Jan. 1979.
[32] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees," Dep. Elec. Eng., Indian Inst. Technol., New Delhi, India, 1981, unpublished.
[33] F. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," in *Proc. 20th Annu. Symp. Found. Comput. Sci.*, IEEE Comput. Soc., Oct. 1979, pp. 140–147.
[34] J. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks," in *Proc. 15th Annu. ACM Symp. Theory Comput.*, Apr. 1983, pp. 10–17.
[35] A. L. Rosenberg, "Three-dimensional VLSI, I: A case study," in *Proc. CMU Conf. VLSI,"* Comput. Sci. Press, Oct. 1981, pp. 69–79.
[36] J. Savage, "Planar circuit complexity and the performance of VLSI algorithms," in *VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds. Woodland Hills, CA: Comput. Sci. Press, Oct. 1981.
[37] ——, "Area-time tradeoffs for matrix multiplication and related problems in VLSI models," *J. Comput. Syst. Sci.*, vol. 22, pp. 230–242, Apr. 1981.
[38] C. L. Seitz, "Self-timed VLSI systems," in *Proc. Caltech Conf VLSI*, Dep. Comput. Sci., California Inst. Technol., Pasadena, Jan. 1979, pp. 345–356.
[39] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.
[40] Y. Tanaka, Y. Nozaka, and A. Masuyama, "Pipeline searching and sorting modules as components of data flow database computer," in *Proc. Int. Fed. Inform. Processing*, Oct. 1980, pp. 427–432.
[41] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 263–271, Apr. 1977.
[42] C. D. Thompson, "A complexity theory for VLSI," Ph. D. dissertation CMU-CS-80-140, Comput. Sci. Dep., Carnegie-Mellon Univ., Pittsburgh, PA, Aug. 1980.
[43] ——, "Fourier transforms in VLSI," *IEEE Trans. Comput.*, vol. C-32, pp. 1047–1057, Nov. 1983.
[44] C. D. Thompson and D. Angluin, "On AT^2 lower bounds for sorting," unpublished manuscript, May 1983.
[45] S. Todd, "Algorithm and hardware for a merge sort using multiple processors," *IBM J. Res. Develop.*, vol. 22, pp. 509–517, Sept. 1978.
[46] J. Vuillemin, "A combinational limit to the computing power of VLSI circuits," *IEEE Trans. Comput.*, vol. C-32, pp. 294–300, Mar. 1983.
[47] L. E. Winslow and Y.-C. Chow, "Parallel sorting machines: Their speed and efficiency," in *Proc. AFIPS 1981 Nat. Comput. Conf.*, Fall 1981, pp. 163–165.
[48] A. C. Yao, "Some complexity questions related to distributive computing," in *Proc. 11th Annu. ACM Symp. Theory Comput.*, May 1979, pp. 209–213.
[49] H. Yasuura, N. Takagi, and S. Yajima, "The parallel enumeration sorting scheme for VLSI," Dep. Inform. Sci., Kyoto University, Kyoto, Japan, Yajima Lab. Res. Rep. ER-81-03, Apr. 1982.

Clark D. Thompson, for a photograph and biography, see p. 1057 of the November issue of this TRANSACTIONS.