THE VLSI COMPLEXITY OF SORTING

by

C. D. Thompson

Memorandum No. UCB/ERL M82/5

14 February 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley, CA 94720

THE VLSI COMPLEXITY OF SORTING

by

C. D. Thompson

Memorandum No. UCB/ERL M82/5

14 February 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The VLSI Complexity of Sorting

C. D. Thompson

Division of Computer Science
U. C. Berkeley
Berkeley, CA 94720

0. 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 comsume less power. Other adjustments to the model make it possible to compare pipelined and non-pipelined designs.

Using the new model, this paper briefly describes thirteen different designs for VLSI sorters. (None of these sorters are 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, $\Omega(N^2)$.

Key words and phrases: parallel algorithms, area-time complexity, VLSI, sorting, bubble sort, bitonic sort, heapsort, shuffle-exchange network, mesh-connected computers.

1. 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 [Knu 73, p.3]. This is probably no longer the case, given the large number of

This work was supported in part by the National Science Foundation under Grant ECS-8110684 and by the U.S. Army Research Office under Grant OAAG29-78-G-0167.

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 [Knu 73, 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 inter-gate wiring as by its gates themselves. This technological novelty makes it appropriate to re-evaluate 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 [Tho 80a], 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 non-pipelined 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 AT^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 AT^2 product of less than $\Omega(N^2)$ [Vui 80], these designs are area- and time- optimal to within logarithmic factors.

A number of different models for VLSI have been proposed in the past few years [B&K 81, C&M 81, K&Z 81, Tho 80a, Tho 80b, Vui 80]. 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 [Tho 80a] or at only one time and place [B&K 81]. Savage [Sav 81] has characterized these as the "semelocal" 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, the semelocal assumption is relaxed by allowing a chip to access each input value from several different I/O memories. The intent is to allow redundant input codes: if each input bit appears in k places, a chip's area*time^2 performance may be enhanced by a factor of k^2 [K&Z 81].

Additionally, the new model is not semelective, for it allows multiple accesses to problem inputs, outputs, and intermediate results. Here, the intent is to model the use of off-chip RAM storage; the area of the RAM is *not* included in the total area of the sorting circuit. This omission clarifies the area*time^2 tradeoff for sorting circuits, since RAM area is involved in an entirely different form of tradeoff. (The recent work of Hong and Kung [H&K 81] indicates that a (time * lg space) tradeoff may describe how local memory affects the speed of a sorting circuit with fixed I/O bandwidth.) Leaving RAM area out of the new model permits the analysis of sublinear size circuits. 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 a "processing" circuit.

Other authors have used non-semelective models, although none has elaborated quite so much on the idea. Lipton and Sedgewick [L&S 81] point out that the "standard" AT^2 lower bound proofs do not depend on semelective assumptions. Hong [Hon 81] defines a non-semelective 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 [L&S 81] point out another form of input 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 might be constrained to appear at I/O port #1, regardless of the problem inputs. This natural restriction begs an important theoretical question: what is the hardest part of the sorting problem? Is it determining the rank order of the inputs, or is it permuting the inputs into sorted order, given their ranks? (A solution to the first subproblem is all that is required of a "where-oblivious" [L&S 81] sorting circuit.) I hope to be able to evaluate the VLSI complexity of these two subproblems in the near future; if you are interested in this problem area, please give me a call. For the purposes of this paper, however, the sorting problem will be assumed to include both "ranking" and "permuting" of the input values.

The catalog of input restrictions is not yet complete. In both Vuillemin's [Vui 80] and Thompson's [Tho 80b] 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 3.10 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 $\Omega(N^2)$ lower bound on AT^2 performance until the definitions of area and time are adjusted.

In the new model, the area performance of a design is its "area per problem," equal to its total area divided by its degree of concurrency. 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, with no correction factor for its concurrency.

The time performance of a design is newly defined as the delay between the presentation of one set of problem inputs and the production of the outputs for that problem. The old definition of time performance was the rate at which a design accepted input bits. It is easy to see that two sorting chips have twice the time performance of one, under the old definition. They also have twice the area. Under the new definition, area and time performances are not affected when a design is replicated.

The old and new definitions of area and time can be contrasted by analyzing the combined sorting performance of N independent serial processors. As will be shown in Section 3.1, each one of these processors has an area of $O(lg\ N)$ and each can solve one sorting problem every $O(N\ lg^2N)$ time units. A collection of N processors would thus consume input data at the rate of one bit every $O(lg\ N)$

time units. Their total area is $O(N \lg N)$, yielding an "impossibly good" area*time^2 performance of $O(N \lg^3 N)$ under the old definitions of area and time. Under the new definitions, the total area per problem is just $O(\lg N)$ and the solution delay is $O(N \lg^2 N)$, so the AT^2 performance is $O(N^2 \lg^5 N)$.

This paper is organized in the following fashion: Section 2 discusses the new VLSI model of computation, then defines it precisely; Section 3 sketches thirteen different designs for VLSI sorters and analyzes the area-time performance of each; Section 4 compares the performances of each of the designs, with some discussion of the "constant factors" ignored by the asymptotic model; and Section 5 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 3 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.

2. 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 [M&C 80], 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 $\mu m^2 = 1.5 * 10^{-6}\ cm^2$; eventually, it may be as small as .4 μm^2 [M&C 80, 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 to 50 ns; and this may decrease to as little as 2 to 4 ns [M&C 80]. For the superconducting technology of Josephson junctions, a clock period of 1 to 3 ns is achievable today, using a process for which the area unit is 25 μm^2 [Ket 80].

The speed of a VLSI circuit may be adversely affected by the presence of very long wires, unless special measures are taken. In many MOS 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 cross-chip communication, 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 [M&C 80, p. 14]. The use of these driver circuits is reflected in the VLSI model's "logarithmic delay" assumption, that a length-k wire has $O(lg\ k)$ delay. Each stage of a driver's amplifier chain is individually clocked, so that the driver behaves like an $O(lg\ k)$-bit shift register. Note that this design for long-wire drivers achieves unit bandwidth. Every wire, even the longest one, has a throughput of one bit per time unit.

The logarithmic delay assumption is used here because it leads to realistic circuit designs and time bounds. As it turns out, the time bounds obtained for VLSI sorting under this assumption are no 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). In the circuits of Section 3, the delay of the drivers is overlapped with the delays of comparison operations. The sole effect of the logarithmic delay assumption is thus to ensure that the VLSI designer strives for such an overlap.

It may be argued that the logarithmic delay assumption is too severe or too lenient, depending on the technology. The former is currently the case in the I^2L and Josephson junction processes [Eva 79, Ket 80]. As of now, both are really unit-delay technologies; no drivers are needed for cross-chip communication. However, the results of this paper still apply if the drivers are omitted from the circuit constructions of Section 3.

It seems unlikely that the logarithmic delay assumption will ever be too lenient on synchronous MOS circuits. Seitz [Sei 79] 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 assumption -- the true delay would best be modeled as logarithmic plus a small constant. Modelling delay as a linear function of distance, as suggested by Chazelle and Monier [C&M 80], 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 may become invalid. 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 to 3 *ns* clock [Ket 80]. 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.

Before leaving the subject of wire delay, it should be noted that the model of this paper makes provision for the "self-timed" regime predicted by [Sei 79]. It may eventually become very difficult to guarantee that all portions of a VLSI circuit get a clock signal with the correct frequency and/or phase. Fortunately, it is feasible to have the long-wire drivers include timing information with the data being transmitted, so that special "receiver" circuits can resynchronize the data with respect to the local version of the clock. Also, single-stage, unit-delay "repeater" circuits can be used to avoid driver delays in the interconnection networks of Sections 3.8 and 3.13.

Thus far in the discussion, only "standard features" have been introduced to the VLSI model. The interested reader is referred to [Tho 80a] for more details on the practical significance of the model, and to [Sav 79] for an excellent introduction to the theoretical aspects of VLSI modelling.

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 small 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. If the access pattern is more complex, a random access memory (RAM) is used. Such a memory has an access time of $O(lg\ k)$ [M&C 80, 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. In particular, it might seem more realistic to involve several I/O ports in a word-parallel memory interface. Such interfaces are not defined here, in an effort to keep the paper's model as simple and general as possible. In any event, a parallel interface could be "simulated" with several serial interfaces at little or no cost in 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. However, a few restrictions must be placed on their usage, 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.) All I/O ports connecting to a single memory must be physically adjacent to each other in the chip layout, to avoid any possibility of "rats-nest" wiring to the memory chips.

The model makes as 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 already assumed for 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 4 to use more realistic estimates of I/O costs when evaluating Section 3's constructions.

The complete model of VLSI computation is summarized in the following list of assumptions.

Assumption 1: Embedding.

a. Wires are one unit wide.

b. Two wires may cross over each other at right angles (in one unit square).

c. A logic node occupies $O(1)$ area. It has $O(1)$ input wires and $O(1)$ output wires, none of which are more than $O(1)$ units long.

d. Each logic node belongs to a self-timed region. All wires connecting to a logic node lie entirely within its self-timed region.

e. A self-timed region may be as much as $O(lg\ N)$ units wide or long.

f. A driver node with an output wire of length k can be laid out in an $O(1)$-by-$O(k)$ unit rectangle. Its input wire is $O(1)$ units long. The output wire may pass through any number of self-timed regions before it connects to the input of a repeater or receiver node.

g. A receiver node occupies $O(1)$ area. Its output wire is $O(1)$ units long. Its input wire may be of any length.

h. A repeater node with $O(1)$ output wires of length at most k can be laid out in an $O(1)$-by-$O(k)$ unit rectangle. Its input wire is at least $k/2$ units long. (Note that the driver node of Assumption 1f can be constructed from a chain of $lg\ k$ repeater nodes, each twice the size of the previous one.)

i. An I/O memory and its associated I/O ports occupy $O(1)$ area. Each I/O port has one input wire and one output wire, each of $O(1)$ length.

j. Two nodes may cross over each other at right angles, but their intersection area does not count toward the required area for either node. (The crossover region is full of wires, so there is no room for transistors.)

Assumption 2: Problem definition.

a. A chip has degree of concurrency p if it solves p 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. $M = N^{1+\varepsilon}$, for some fixed $\varepsilon > 0$. Furthermore, a nearly non-redundant code must be used, so that input and output values are represented as $O(lg\ N)$ bit words. (This assumption makes it possible to express area and time bounds in terms of N alone.)

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 at most one bit of information in a unit of time.

b. Logic nodes, repeater nodes, and receiver nodes have $O(1)$ delay.

c. The driver node for a wire of length k has $O(lg\ k)$ delay.

Assumption 4: Transmission functions.

a. A deterministic finite-state automaton (FSA) is associated with each node. The "state" of a node is a bit vector encoding the current state of its FSA. There is a fixed mapping between the (single-bit) signals appearing on the input and output wires of a node, and the inputs and outputs of its FSA.

b. The state of a node is changed every time unit, *i.e.* its FSA undergoes one state transition per time unit.

c. Logic nodes, repeater nodes, and receiver nodes are limited to $O(1)$ bits of state.

d. Driver nodes have $O(lg\ k)$ bits of state, one bit for each stage in their amplification chain.

e. The state vector of a "k-bit" I/O memory contains one bit for each of its assigned problem input and output bits. The assignment of problem bits to memories is one-to-one and is not data-dependent.

f. There are $O(lg\ k)$ bits in the state vector of each I/O port attached to a k-bit memory; at most $O(1)$ I/O ports can be attached to each memory. The state vectors are used to address specific memory bits, as explained in Assumptions 4g and 4h. Two different ports may not access the same bit simultaneously.

g. "RAM-type" k-bit I/O ports run a memory cycle every $O(lg\ k)$ time units. During the first $lg\ k$ time units of a cycle, the port receives a bit-serial address on its input wire. The next input signal is interpreted as a read/write indicator. If a write cycle is indicated, the following input signal is written into the addressed bit. During the last time unit of a memory cycle, the value of the addressed bit is available on the I/O port's output wire.

h. "Shift-register-type" I/O ports run a memory cycle every $O(1)$ time units. During the first time unit of a cycle, the value of the currently-addressed data bit is available on the port's output wire. In the last time unit of a cycle, the signal appearing on the port's input wire is written into this data bit, then the port's address register is incremented (mod k).

Assumption 5: Area, time performance.

a. The total area 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 p. See Assumption 2a.

c. The time performance T of a chip is the average number of time units it takes to solve any one of its p problem instances.

3. 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 can be built of $O(1)$ gates [Mor 79] in $O(1)$ 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 Figure 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 array for $O(lg\ N)$-bit words complete comparison-exchange operations in $T = O(lg\ N)$ time units. The total area of the array as drawn is $O(lg^2 N)$, and its concurrency is $lg\ N$, giving it an area performance of $A = O(lg\ N)$. However, in most applications the square boxes can be deleted, since the input and output data is already "staggered." The total area of the circles is only $O(lg\ N)$, so that the area performance of the pipelined comparison-exchange module can be as good as that of the non-pipelined module, $A = O(1)$.

A third building block is the programmed control unit, or 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 $O(lg\ N)$ 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 $O(lg\ N)$ 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 is probably no reason to use a parallel processor with a

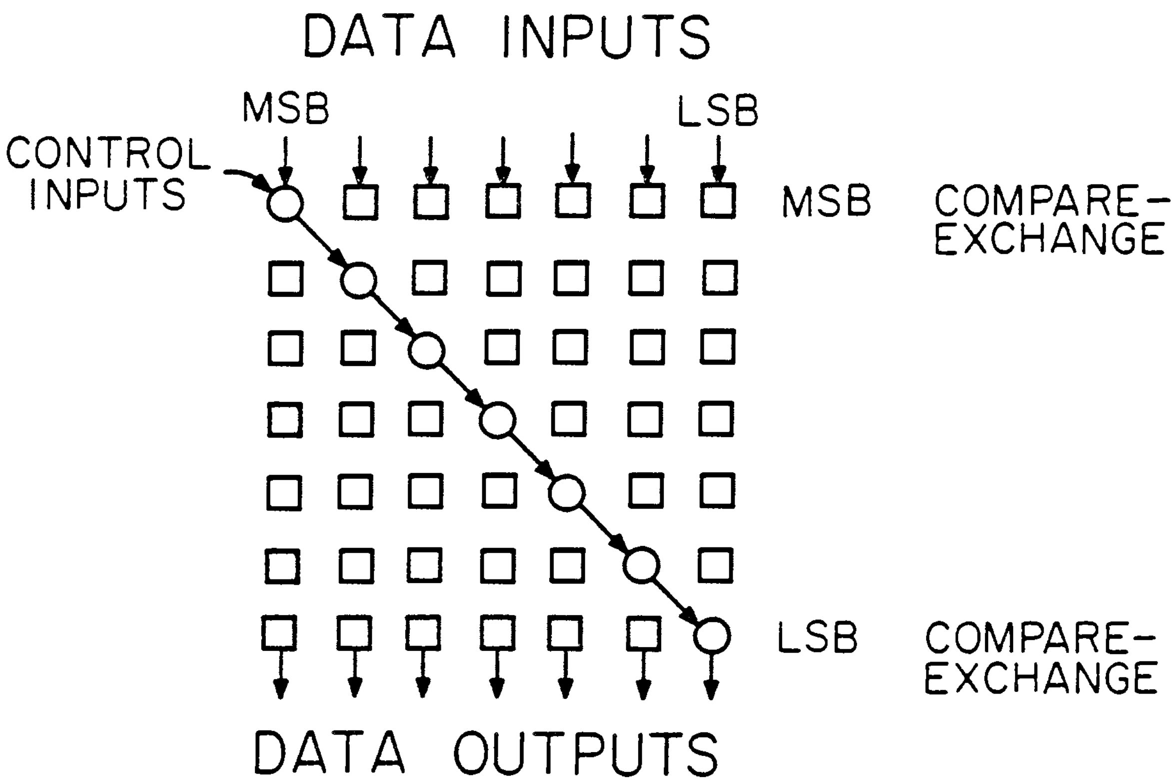

Figure 1: A pipelined comparison-exchange module.

random-access memory, under the model of this paper, since the delay of a serial processor matches the delay of a RAM -- any further processor speed is useless.

A word-parallel processor can perform one comparison-exchange operation every $O(1)$ 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.

3.1. Uniprocessor Heapsort

This is the smallest sorter imaginable. It has one bit-serial processor running a standard heapsort algorithm [Knu 73, 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 $T = O(N\ lg^2 N)$ units of time. The area performance of this design is $A = O(lg\ N)$.

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

3.2. $(lg\ N)$-processor Heapsort

Heapsort can be parallelized on a linear array of $lg\ N$ bit-serial processors, one for each level of the heap [Arm 78] (see Figure 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 handles one data element, the smallest one. The kth processor $(0 \le k < lg\ N)$ handles 2^k elements. Total sorting time is $T = O(N\ lg\ N)$, and the total area is $A = O(lg^2 N)$.

3.3. $(1+lg\ N)$-processor Mergesort

The mergesort algorithm, like the heapsort, fits quite nicely on about $lg\ N$ processors [Tod 78]. 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 Figure 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 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_0 is a special case. It performs an especially simple computation. 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

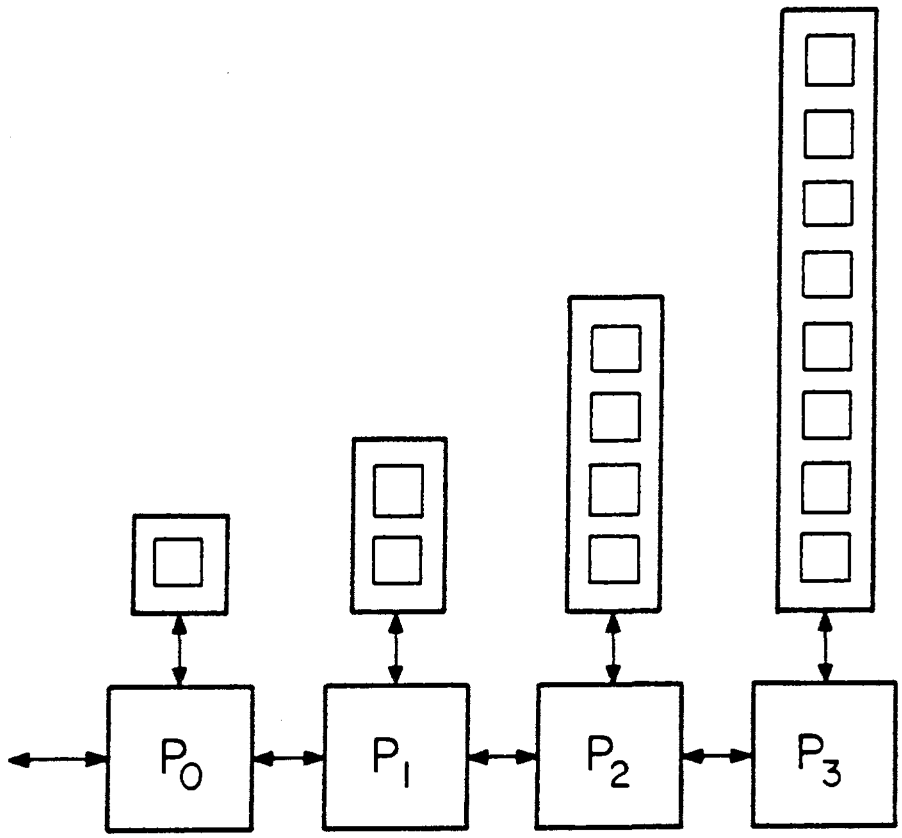

Figure 2: The $(lg\ N)$-processor heapsort for $N=16$.

sorted lists of length 2 by processor P_1.

The FIFO queues must be implemented with random-access memory circuits, since fixed length shift-register memories will not work properly. First of all, the FIFOs are variable in length, so special arrangements (and extra time) would be necessary to "jump over" unused shift-register cells. Furthermore, elements are not extracted from the FIFOs at a uniform rate. A "pop" operation occurs on a FIFO queue only when the element at the head of the queue "wins" a comparison, and is sent on to the next processor in the chain.

Since random-access memory must be used, the best achievable data rate through a processor is one element every $O(lg\ N)$ time units. This allows just enough time for one bit-serial random access and one bit-serial comparison.

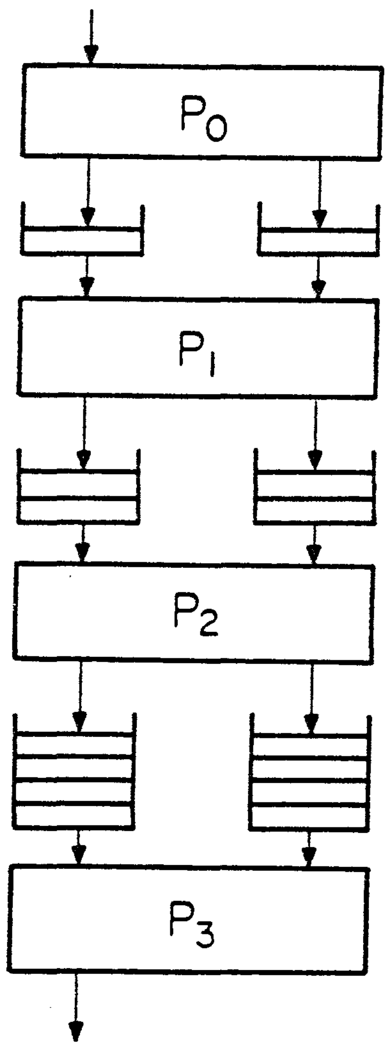

Figure 3: The $(1+lg\ N)$-processor mergesort for $N=8$.

(The modules can not be pipelined easily, since there is no way of knowing which data elements should next be compared until the previous comparison is completed and the appropriate FIFO is popped.)

The time performance of the mergesorter is limited by the data rate of its individual processors. It takes $O(N\ lg\ N)$ 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 $T = O(N\ lg\ N)$. The total 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.

3.4. $(lg\ N)$-processor Bitonic Sort

Superficially, this design is very similar to the previous one. Both designs contain about $lg\ N$ processors, and the kth processor has about 2^k words of data in a FIFO queue. However, in this case, the k processors are connected in a "ring" rather than in a simple linear array. Also, the bitonic sort processors do not require random access into their I/O memories: a simple 2^k-word FIFO queue is all that is required. See Figure 4. (This arrangement has been called a "cascade" when used to perform the FFT [Des 80].)

The processors execute a bitonic sorting algorithm [Knu 73, p. 237]. For the purposes of this paper, this algorithm can be described as $lg\ N$ iterations of: a distance-1 operation, followed by a distance-2 operation, ..., followed by a distance-2^k operation, ..., followed by a distance-$N/2$ operation. The total number of operations is clearly $lg^2 N$.

A distance-2^k operation is either a no-op or a comparison-exchange between all pairs of data whose indices differ only in the kth bit. Somewhat fewer than half of the operations are no-ops.

Processor P_k $(0 \le k < lg\ N)$ performs only distance-2^k operations, pairing up the appropriate data elements using a 2^k-word FIFO queue. Three patterns of data flow are sufficient. When it uses pattern number 1, processor P_k places the elements it receives from processor P_{k-1} onto the back of its FIFO queue, and sends 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 larger of the two to processor P_{k+1} and placing the smaller on the back of its queue. Pattern number 3 is the same as pattern number two, except that the smaller of the two elements is sent to the next processor and the larger is placed on the back of the queue.

The only remaining problem is to define the control algorithm for setting the pattern numbers. In general, processor P_k executes pattern number 1 on the first kN elements it encounters. This corresponds to k iterations of the "no-op" operation on N elements. Next, it alternately fills its queue with new elements (by executing 2^k instances of pattern number 1) then performs comparison-exchanges (by executing 2^k instances of pattern number 2 or 3). It executes pattern 2 on the second half of the j-th set of 2^{k+1} elements during the i-th iteration of its "distance-2^k" operation on the N data elements $(0 \le j < (lg\ N)/(k+1),\ 0 \le i < lg\ N)$ whenever $j\ DIV\ 2^i$ is odd; otherwise it executes pattern 3 on the second half of this set of elements.

The control algorithm is thus rather simple to program. Processor P_k requires three counters. The first one counts the iteration number i, incrementing after every N elements. Pattern number 1 is selected whenever $i<k$. The second counter keeps track of the sets of 2^{k+1} elements; while the top bit of this counter is zero, pattern number 1 is selected. The third counter (for j, above) is incremented whenever the second counter overflows; as long as pattern number 1 hasn't been selected by the first two counters, the i-th bit of this third counter determines which of patterns 2 and 3 should be selected.

The total area of the design is $A = O(lg^2N)$, since there are $lg\ N$ processors of $lg\ N$ area each. If bit-serial processors are used, a distance-2^k operation takes $O(N\ lg\ N)$ time. The processors can all work on different operations simultaneously, so that the $lg\ N$ iterations in a complete bitonic sort take just $T = O(N\ lg^2N)$ time.

The area*time^2 performance of the design may be improved by using word-parallel processors. Now each operation requires only $O(N)$ time, if $O(lg\ N)$ communication lines are provided between processors. Total time is $T = O(N\ lg\ N)$; total area is still $A = O(lg^2N)$. Note that this parallelized design requires $O(lg^2N)$ 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 [CLW 80]. Their algorithm has an inferior time performance to the one described above, since only one of their processors is active at any time.)

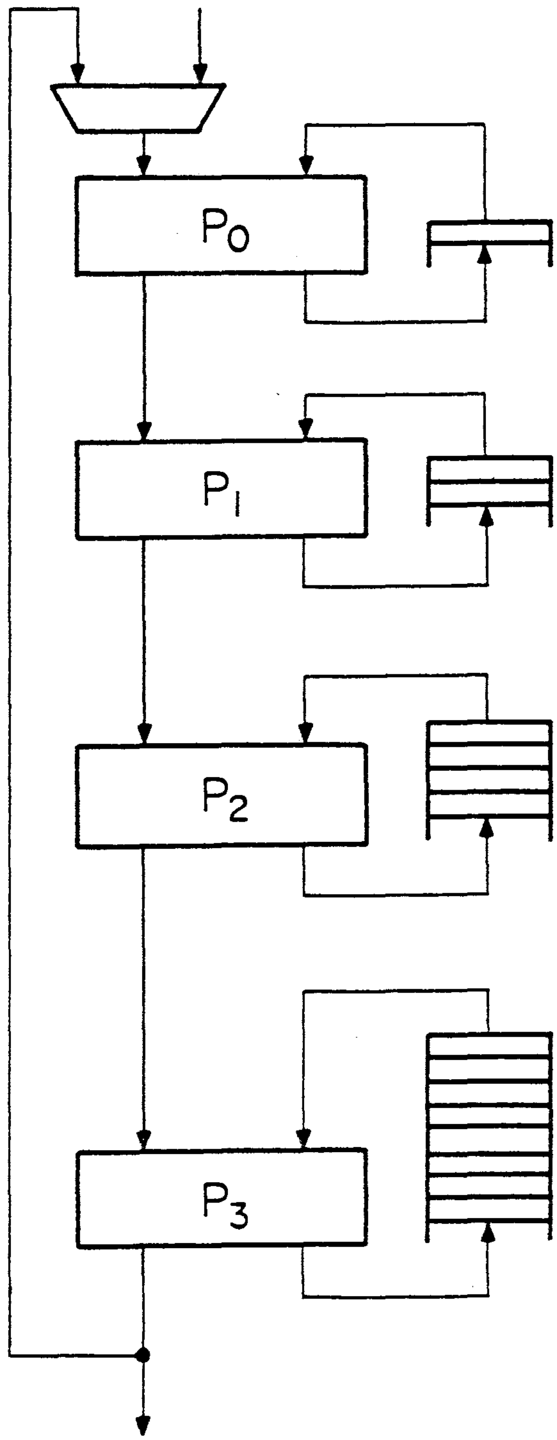

Figure 4: The (lg N)-processor bitonic sorter for N=16.

3.5. $O(lg^2 N)$-processor Bitonic Sort

This design "unrolls" the $lg\ N$-processor bitonic sort, so that each processor is responsible for only one distance-2^k 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 Figure 5. 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-2^j operation is implemented with 2^j words of FIFO storage, and since all but k of the distance-$N/2^k$ operations are no-ops, the total storage is $\Sigma(Nk/2^k)$, or a little less than $2N$ words. The problem solution time is proportional to the length of this pipeline, or $T = O(N)$ 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 area*time^2 performance of this design is a factor of $lg\ N$ better than that of the previous design. To better 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 $lg\ N$ $(lg\ N)$-processor bitonic sorters. Both have the same amount of total area, and both solve $lg\ N$ sorting problems in $O(N\ lg\ N)$ time (if word-parallel processors are used). However, the $O(lg^2 N)$-processor implementation solves each sorting problem with logarithmically less delay.

3.6. $\sqrt{N\ lg\ N}$-processor Bitonic Sort

Chung, Luccio, and Wong have recently proposed implementing a bitonic sort on $\sqrt{N\ lg\ N}$ processors in a linear array [CLW 80]. 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 somewhat less than $O(N)$ time, if it uses word-parallel processors. Working together, the entire array performs an N-element sort in $T = O(N)$ and $A = O(\sqrt{N\ lg^3 N})$.

According to the model of this paper, this approach is highly non-optimal in an AT^2 sense. It is no faster, but much larger, than the $lg^2 N$-processor bitonic sorting design.

3.7. $(N/2)$-processor Bubble Sort

The familiar bubble sorting algorithm can be fully parallelized on a linear array of $N/2$ bit-serial (or pipelined) comparison-exchange modules [Muk 72, CLW 80]. 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 zeroes, then serially loaded with N data elements through the left-most module. If it is then "flushed out" by loading maximal elements through the left-most module, the N data elements will emerge from the left-most module in $O(N)$ comparison times.

The total area of the $N/2$-processor bubble sorter is $A = O(N\ lg\ N)$. When bit-serial modules are used, each comparison takes $O(lg\ N)$ time, so $T = O(N\ lg\ N)$. The time performance is improved if word-parallel modules are used: in this case, $T = O(N)$. Even so, the combined area*time^2 performance of the design remains dismal. According to the $AT^2 = \Omega(N^2)$ lower bound, a sorter with $O(N\ lg\ N)$ area should sort in about $O(\sqrt{N})$ time. The following subsection describes a sorter that nearly achieves this bound.

3.8. 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 [N&S 79, Tho 80a]. 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 $T = O(\sqrt{N})$.

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. The driver nodes of Assumptions 1f and 3c take $O(lglg\ N)$ time to amplify a signal for a wire of this length. However, once a signal has been amplified, it can travel from one processor to the next along a chain of repeater nodes (Assumption 1h). Using this approach, cross-chip communication takes only

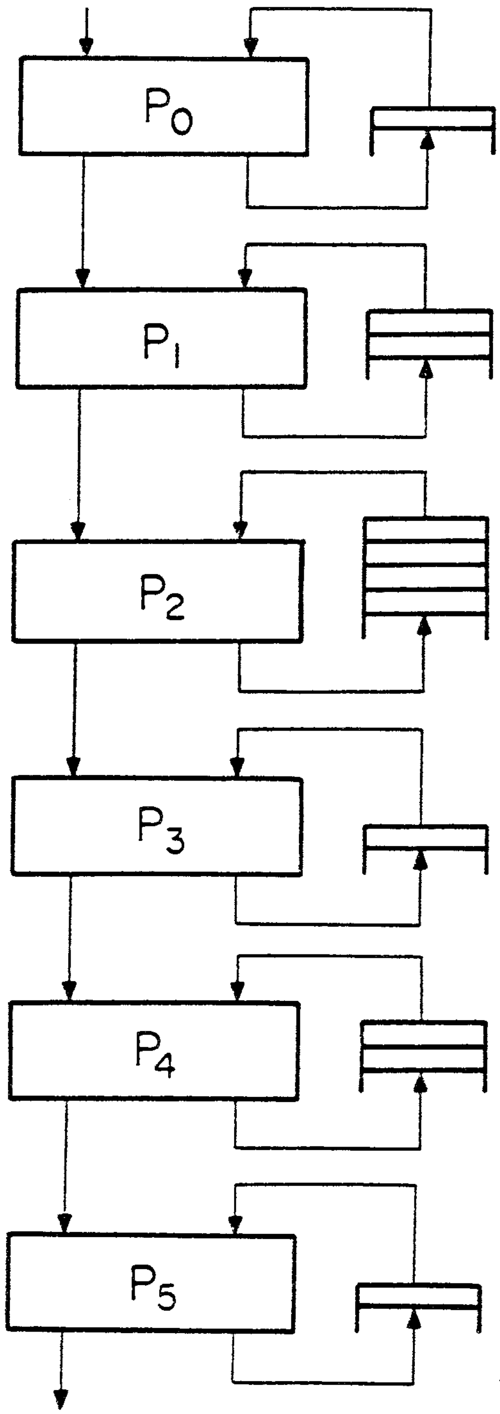

Figure 5: The $(1/2)(lg\ N)(1+lg\ N)$-processor bitonic sorter for $N=8$.

$O(\sqrt{N})$ time on an N-element mesh. (An earlier version of the N-processor bitonic sort on the mesh [Tho 80a] had a poorer time performance because it used driver nodes for all interprocessor communication.)

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, but the word-parallel data paths between neighboring processors require more room in the asymptotic limit.

(A more complicated sorting algorithm [Tho 77] will run on the mesh in the same asymptotic time and area. For large N, it may have a constant factor advantage in time but a constant factor disadvantage in area, due to the larger program required to control the processors.)

3.9. 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 [Sto 71]. If bit-serial interconnections are used, the $O(N\, lg^2 N)$ comparison-exchanges in the bitonic sort take a total of $T = O(lg^3 N)$ time, since $N/2$ comparisons can be done at a time. The long-wire drivers and the comparison-exchanges introduce approximately equal delays into the computation.

Given that this design sorts so quickly, it should not be surprising that it requires a lot of area. Otherwise, it would be a counterexample to the $AT^2 = \Omega(N^2 lg^2 N)$ lower bound. An asymptotically optimal embedding of the shuffle-exchange graph has been recently obtained [KLLM 81, Leig 81a]. It requires area $A = O(N^2/\, lg^2 N)$.

The AT^2 performance of this design is a little sub-optimal. Seemingly, the bitonic sorting algorithm makes too many long-distance data movements on the shuffle-exchange network. Some of these were avoided in the N-processor mesh design, because the "no-ops" were deleted. Perhaps the shuffle-exchange sorting design can be made more efficient by a similar trick.

3.10. N-processor Bitonic Sort on CCC

Preparata and Vuillemin [P&V 79] have shown that their "cube-connected cycles" interconnection pattern can run the bitonic sort algorithm as efficiently as the shuffle-exchange pattern: $A = O(N^2/\, lg^2 N)$, $T = O(lg^3 N)$. 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.

3.11. $(N\ lg^2 N)$-processor Bitonic Sort

Batcher's bitonic sorting network [Muk 72, Knu 73, 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-2^k operation" takes up a 2^k by N area of the chip; the total area occupied by the network is $N\ \Sigma(Nk/2^k) = O(N^2)$, since there are k distance-$N/2^k$ comparison-exchange operations.

The network is naturally pipelined with a concurrency of $lg^2 N$, since there is about a word of storage in the long-wire drivers associated with most of the $O(lg^2 N)$ rows of comparators. Its area performance is thus its total area divided by its concurrency, $A = O(N^2/lg^2 N)$. The delay experienced by each problem is $T = O(lg^3 N)$. Note that this area-time performance is the same as that of the two previous designs. The additional processors work on different problem instances, so they cannot speed up the computation of any single instance.

An improvement can be made to the construction outlined above, leading to a better AT^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. Now the comparators for the distance-2^k operation will each occupy $O(2^k)$ area. (The transistors in the comparators should all be oriented in the same direction, so that they have more current-driving capability when the comparator is "stretched" to fill an $O(1)$-by-$O(2^k)$ rectangle.) The total area is still $O(N^2)$: the enlarged comparators take up the space occupied by the long-wire drivers in the original construction. However, bits now travel from one row of comparators to the next in $O(1)$ time. The network has total delay $T = O(lg^2 N)$, concurrency $O(lg\ N)$, and area performance $O(N^2/lg\ N)$.

Vaughn Pratt recently pointed out to the author that shellsort can be implemented on either $(lg^2 N)$ or $(N\ lg^2 N)$ processors with the same area*time^2 performance as the bitonic sort.

3.12. N^2-processor 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 [Knu 73, 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 $T = O(N)$, 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, $A = O(N\ lg\ N)$. Its time performance worsens, becoming $T = O(N\ lg\ N)$.

3.13. (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 nodes of the "row trees" provide 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 node in two orthogonal trees.

This network has been called by various names, including the "Orthogonal Tree Network" [Nat 81] and the "Mesh of Trees" [Leig 81a, Leig 81b]. Figure 6 illustrates it for the case $N=16$.

A brute-force sorting algorithm can be implemented on this network, as pointed out by the authors cited above. (Muller and Preparata [M&P 75] 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 nodes of a row tree. The inputs are then broadcast 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 $input[j]$.

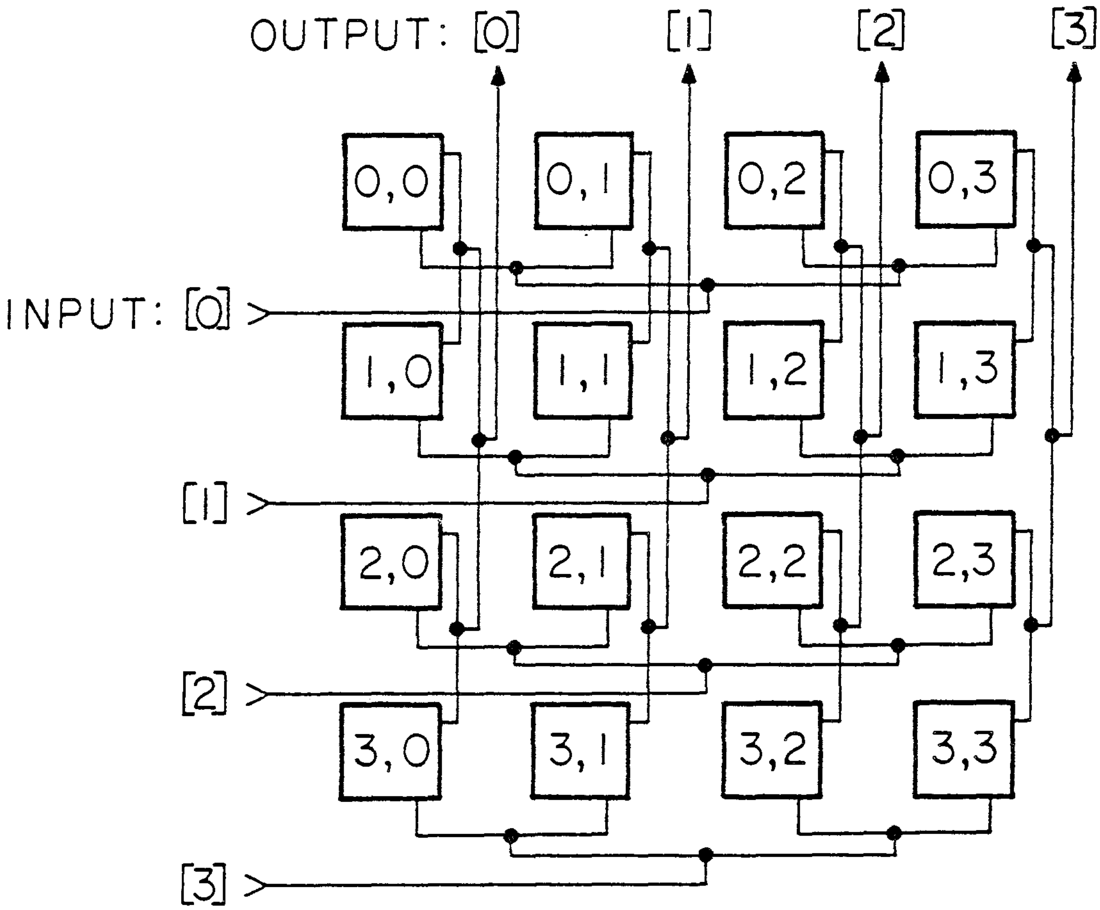

Figure 6: The Orthogonal Tree Network for $N=16$.

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 nodes of the row trees. Finally processor (i,j) sends up the value of *input*$[i]$ through its column tree if *rank*$[i] = j$. (This operation is called a "selection" in the discussion immediately below.) The sorted values are now available at the roots of the column trees.

It remains to establish the area and time complexity of this sorting procedure. Since broadcast, summation, and selection operations are involved on trees with N leaves, the best possible time performance is $T = O(lg\ N)$. This performance is in fact achievable, but only with careful design.

Observe that in Figure 6 the wires connecting nodes 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. This means that scaled repeater nodes (Assumptions 1h and 4c) should be used to form the internal nodes of row and column trees. The repeater node on the kth level of a tree ($k = 0$ for the root) contains $O(1)$ gates scaled up to occupy an $O(1)$-by-$O(N/2^k)$-unit area. See Figure 7.

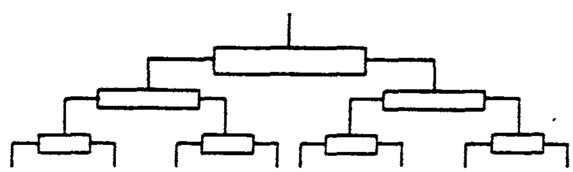

Figure 7: A row tree, internal nodes drawn to scale.

The functionality of the repeater nodes must be such to enable it to perform the tree operations alluded to above: broadcast, summation, and selection. These operations produce and consume $lg\ N$ bit numbers; but each repeater node contains only $O(1)$ gates. The solution to this design problem is to use bit-serial logic.

The summation operation is performed by having the internal nodes of the tree act compute one bit of the sum during each clock period. Each node adds up the two bits sent up to it by its children, adds the result to the contents of a local one-bit register "S" ($S = 0$, initially), and sends the carry bit up to its ancestor. The sum bit is retained in each node's "S" register for a fixed amount of time, as described below. After $lg\ N$ time periods have elapsed, all the carries have rippled up to the root node of the row tree, so that register "S" of this node contains the most significant bit of the sum.

The other sum bits are computed by a similar process. The first wave of carries was initiated by the processors at the leaves of the tree, when they sent up a 0 or a 1 as the result of their comparison. Assume this occurs at time $T=0$. A second wave of carries can be initiated at time $T=2$ by the immediate ancestors of the leaf processors, if they send up the contents of their "S" register. Eventually, this will result in the root node being able to evaluate the second-most significant bit of the sum. (As a matter of fact, this bit will be available just one time unit after the most significant bit was computed.) In general, the kth-most significant bit of the sum is generated at the root of the tree at time $k+lg\ N$, as a result of the "S" register bits being released at time $T=2k$ by the internal nodes at a distance k from the leaf processors.

Of course, the root nodes of the row trees do not have room to store the entire $lg\ N$ bits of a sum, for they have only $O(1)$ gates. Fortunately, they are not required to do so. The purpose of the summation operation is to set the stage for a "selection" operation on the column trees. The sum bits can thus be broadcast down through the row trees, as they are computed, so that each processor is informed of the rank of its input element.

The circuitry for "selection" of one element from a column tree is quite straightforward. Barring logic errors, only one processor in each column tree will attempt to send its input value up to the root. It can do so in a bit-serial fashion; the other processors can send up zeros; and the internal nodes can compute the "or" of their two inputs.

The entire sorting algorithm can thus be performed in $O(lg\ N)$ time on N^2 leaf processors of $O(lg\ N)$ area, interconnected by orthogonal trees formed from about $N\ lg\ N$ internal nodes of $O(1)$ gates apiece. Because the internal nodes must be enlarged as they near the root, each tree occupies $O(N\ lg\ N)$ area. The total area of the circuit is thus $A = O(N^2 lg^2 N)$. Or is it?

The observant reader will have already noticed that the long repeater nodes of Figure 7 will intersect each other in Figure 6, when both column and row trees are considered. This means that the transistors in the repeater nodes can not be laid out in a contiguous region. In fact, the root nodes of the trees will have $O(N)$ units of transistor area which is intersected by $O(N\ lg\ N)$ wires in the repeater nodes of the orthogonal trees. With this many crossings, it is inevitable that individual transistors in the repeater nodes will have to be "broken up" as they are enlarged for the higher reaches of the tree. The "broken" transistors will work in parallel to provide the required increase in current. (Unfortunately, this fragmentation brings us closer to the limits of the "logarithmic rule" for wire delay. The period of the system clock will undoubtably be determined by the performance of these broken transistors. Therefore the true time performance of this design will be somewhat worse than the $O(lg\ N)$ figure claimed above, for the length of the clock period will increase as N increases.)

Tom Leighton recently pointed out that it is not really necessary to have the long repeater nodes intersect each other in an $O(N^2 lg^2 N)$-area layout of the orthogonal tree network. However, it is necessary to have $\Omega(N^2 lg^2 N)$ wire crossings in any layout of this network.

4. Comparison of the Designs

The area and time performance of the thirteen sorting circuits is summarized in Table 1, below. The entries in this table give the area and time performance of each of the designs of Section 3. As defined in Section 2, the area performance of a design is its "processing" area divided by its concurrency. This metric is an indication of the power consumed per sorting problem. The time performance can be summarized as the elapsed time between the first input to the circuit and the last output from the chip for each sorting problem.

Table 1 shows that nearly all of the designs considered in this paper are within a factor of $O(lg^k N)$ of being optimal in an area*time^2 sense. The sole exceptions are the bubble sorters and the $\sqrt{N\ lg\ N}$-processor bitonic sorter.

Table 2 contains additional information about the sorters. 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 area required by each design, broken down into "processor" and "memory" categories. One bit of memory occupies one unit of area; processor area is more difficult to characterize -- see Section 2. (For the purposes of this paper, processing circuitry communicates with memory circuitry only through I/O ports, as described in Assumptions 4g and 4h.)

The final column of Table 2 lists the processor-memory bandwidth in bits/(unit time). These entries are also equal to the number of I/O ports

Design	Area Perf.	Time Perf.	Area*Time^2
(Lower bound)	-	-	$\Omega(N^2)$
Uniprocessor	$lg\ N$	$N\ lg^2 N$	$N^2\ lg^5 N$
$lg\ N$ - proc. heapsort	$lg^2 N$	$N\ lg\ N$	$N^2\ lg^4 N$
$lg\ N$ - proc. mergesort	$lg^2 N$	$N\ lg\ N$	$N^2\ lg^4 N$
$lg\ N$ - proc. bitonic	$lg^2 N$	$N\ lg\ N$	$N^2\ lg^4 N$
$lg^2 N$ - proc. bitonic	$lg^3 N$	N	$N^2\ lg^3 N$
$\sqrt{N\ lg\ N}$ - proc. bitonic	$\sqrt{N\ lg^3 N}$	N	$N^3\ lg^{3/2} N$
$N/2$ - proc. bubble	$N\ lg\ N$	N	$N^3\ lg\ N$
N - proc. bitonic, mesh	$N\ lg^2 N$	$\sqrt{N}$	$N^2\ lg^2 N$
N - proc. bitonic, S-E	$N^2/\ lg^2 N$	$lg^3 N$	$N^2\ lg^4 N$
N - proc. bitonic, CCC	$N^2/\ lg^2 N$	$lg^3 N$	$N^2\ lg^4 N$
$N\ lg^2 N$ - proc. bitonic	$N^2/\ lg\ N$	$lg^2 N$	$N^2\ lg^3 N$
N^2 - proc. bubble	$N\ lg\ N$	N	$N^3\ lg\ N$
N^2 - proc. rank sort	$N^2 lg^2 N$	$lg\ N$	$N^2 lg^4 N$

Table 1: Area-time bounds for the sorting problem.

required by each design. Since problem inputs and outputs must pass through the processor-memory interface, it should not be surprising that the designs with high concurrency and/or good time performance must have a large processor-memory bandwidth. What is a bit surprising is that all designs are within a factor of $lg\ N$ of making optimal use of their I/O ports. All keep their I/O ports busy all the time, and all produce at least one bit of problem output for every $lg\ N$ accesses to the bits in their memory circuits. Another interpretation of the processor-memory bandwidth figures is that no design makes tremendous use of its memory for temporary storage.

Of course, a sorting circuit should not be selected just because it is asymptotically optimal. A circuit designer 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 area*time^2 optimal if it is programmed to use

Design	Concurrency	Total Area		I/O B.W.
		P	M	
Uniprocessor	1	$lg\ N$	$N\ lg\ N$	1
$lg\ N$ - proc. heapsort	1	$lg^2 N$	$N\ lg\ N$	$lg\ N$
$lg\ N$ - proc. mergesort	1	$lg^2 N$	$N\ lg\ N$	$lg\ N$
$lg\ N$ - proc. bitonic	1	$lg^2 N$	$N\ lg\ N$	$lg^2 N$
$lg^2 N$ - proc. bitonic	2	$lg^3 N$	$N\ lg\ N$	$lg^3 N$
$\sqrt{N\ lg\ N}$ - proc. bitonic	1	$N\ lg\ N$	$N\ lg\ N$	$lg\ N$
$N/2$ - proc. bubble	1	$N\ lg\ N$	$N\ lg\ N$	$lg\ N$
N - proc. bitonic, mesh	1	$N\ lg^2 N$	$N\ lg\ N$	$\sqrt{N}\ lgN$
N - proc. bitonic, S-E	1	$N^2/lg^2 N$	$N\ lg\ N$	$N/lg^2 N$
N - proc. bitonic, CCC	1	$N^2/lg^2 N$	$N\ lg\ N$	$N/lg^2 N$
$N\ lg^2 N$ - proc. bitonic	$lg\ N$	N^2	$N\ lg^2 N$	N
N^2 - proc. bubble	$N/lg\ N$	N^2	N^2	N
N^2 - proc. rank sort	1	$N^2 lg^2 N$	$N\ lg\ N$	N

Table 2: Other performance measures.

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 2) to $lg\ N$ bits per time unit.

The $(lg\ N)$-processor bitonic design has about 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 2 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 1 or Table 2. 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 area*time^2 performance becomes noticable. For example, when $N = 256$, the $(lg^2 N)$-processor'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 1 and 2 are variants on a fully-parallelized bitonic sort. The shuffle-exchange processor has a slight area advantage over the CCC processor, because of its simpler control algorithm. However, the CCC is a somewhat more regular interconnection pattern, so that it 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 a CCC 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 $AT^2 = \Omega(N^2)$. 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.

5. 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 [Tho 80a, pp. 36-38], but it is not known whether this bound is achievable. (Some results on

k-level embedding have been obtained recently [Ros 81].)

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 $AT^2 = O(N^2 lg^2 N)$, achieved by the N-processor bitonic sort on a mesh. The best lower bound is $\Omega(N^2)$ [Vui 80], 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 [Tho 80a]. (The current model allows the bits of each input value to be read in through different ports.) It seems probable that the $AT^2 = \Omega(N^2 lg^2 N)$ result for word-oriented 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^2 performance better that that of the bitonic sort on the mesh.

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. For example, equality testing is very easy if one only requires that the answer be "probably" correct [Yao 79, L&S 81].

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.

References

[Arm 78] Philip K. Armstrong, *U.S. Patent 4131947*, issued December 26, 1978.

[B&K 81] R. P. Brent and H. T. Kung, "The Area-Time Complexity of Binary Multiplication," *JACM Vol. 28*, No. 3, pp. 521-534, July 1981.

[C&M 81] B. Chazelle and L. Monier, "Towards More Realistic Models of Computation for VLSI," *Proc. 11th Annual ACM Symp. on Theory of Computing*, pp. 209-213, April 1979.

[CLW 80] Kin-Man Chung, Fabrizio Luccio, and C. K. Wong, "On the Complexity of Sorting in Magnetic Bubble Memory Systems," *IEEE-TC Vol. C-29*, No. 7, pp. 553-562, July 1980.

[Des 80] A. Despain, "Very Fast Fourier Transform Algorithms for Hardware Implementation," *IEEE-TC Vol. C-28*, No. 5, pp. 333-341, May 1979.

[Eva 79] S. A. Evans, "Scaling I^2L for VLSI," *IEEE Journal of Solid-State Circuits, Vol. SC-14*, No. 2, pp. 318-326, April 1979.

[Hon 81] J-W Hong, "On Similarity and Duality of Computation," unpublished manuscript, Peking Municipal Computing Center, China.

[H&K 81] J-W Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th Annual ACM Symp. on Theory of Computing*, pp. 326-333, May 1981.

[Ket 80] M. B. Ketchen, "AC Powered Josephson Miniature System," *1980 Int'l Conf. on Circuits and Computers*, IEEE Computer Society, pp. 874-877, October 1980.

[KLLM 81] D. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller, "New Layouts for the Shuffle-Exchange Graph," Extended Abstract, MIT Applied Mathematics Dept., 1981.

[Knu 73] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1973.

[K&Z 81] Zvi M. Kedem and Alessandro Zorat, "Replication of Inputs May Save Computational Resources in VLSI," *Proc. 22nd Symp. on the Foundations of Computer Science*, IEEE Computer Society, October 1981.

[Leig 81a] F. T. Leighton, *Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI*, Ph.D. Thesis, Department of Mathematics, MIT, August 1981.

[Leig 81b] F. T. Leighton, "New Lower Bound Techniques for VLSI," *Proc. 22nd Symp. on the Foundations of Computer Science*, IEEE Computer Society, October 1981.

[Leis 81] C. E. Leiserson, *Area-Efficient VLSI Computation*, Ph.D. Thesis, Department of Computer Science, C-MU, October 1981.

[L&S 81] Richard J. Lipton and Robert Sedgewick, "Lower Bounds for VLSI," *Proc. 13th Annual ACM Symp. on Theory of Computing*, pp. 300-307, May 1981.

[M&C 80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Mor 79] Hans P. Moravec, "Fully Interconnecting Multiple Computers with Pipelined Sorting Nets," *IEEE-TC Vol. C-28*, No. 10, pp. 795-798, October 1979.

[Muk 72] Amar Mukhopadhyay and Tadao Ichikawa, "An n-Step Parallel Sorting Machine," *TR 72-03*, University of Iowa Department of Computer Science, 1972.

[M&P 75] D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," *JACM Vol. 22*, No. 2, pp. 195-201, April 1975.

[N&S 79] David Nassimi and Sartaj Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE-TC Vol. C-27*, No. 1, pp. 2-7, January 1979.

[Nat 81] Dhruva Nath, S.N. Maheshwari, P.C.P. Bhatt, "Efficient VLSI Networks for Parallel Processing based on Orthogonal Trees," manuscript, Electrical Engineering Dept., Indian Institute of Technology, New Delhi 110016, India, 1981.

[P&V 79] F. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *20th Annual Symp. on Foundations of Computer Science*, IEEE Computer Society, pp. 140-147, October 1979.

[P&V 80] F. Preparata and J. Vuillemin, "Area-Time Optimal VLSI Networks for Multiplying Matrices," *Info. Proc. Letters Vol. 11*, No. 2, pp. 77-80, October 1980.

[Ros 81a] Rosenberg, Arnold L., "Three-Dimensional VLSI, I: a case study," *IBM Report RC-8745*, IBM T.J. Watson Research Center, 1981. (Portions appear in *Proc. CMU Conf. on VLSI*, pp. 69-79, October 1981.)

[Sav 79] J. Savage, "Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," *TR-CS-50*, Brown University Dept. of Computer Science, August 1979.

[Sav 81] J. Savage, "Planar Circuit Complexity and the Performance of VLSI Algorithms," *TR-CS-68*, Brown University Dept. of Computer Science, July 1981. (Portions appear in *CMU Conf. on VLSI*, pp. 61-68, October 1981.)

[Sei 79] C. L. Seitz, "Self-timed VLSI Systems," *Proc. Caltech Conf. on VLSI*, Caltech Computer Science Dept., pp. 345-356, January 1979.

[Sto 71] H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE-TC Vol. C-20*, No. 2, pp. 153-161, February 1971.

[Tho 77] C. D. Thompson, "Sorting on a Mesh-connected Parallel Computer," *CACM Vol. 20*, p. 263-271, April 1977.

[Tho 80a] C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis, Computer Science Department, C-MU, August 1980.

[Tho 80b] C. D. Thompson, "Fourier Transforms in VLSI," *UCB/ERL M80/51*, Department of Electrical Engineering and Computer Science, U.C. Berkeley, October 1980.

[Tod 78] Todd, S., "Algorithm and Hardware for a Merge Sort Using Multiple Processors," *IBM Journal of Research and Development Vol. 22*, No. 5, pp. 509-517, September 1978.

[Vui 80] J. Vuillemin, "A Combinatorial Limit to the Computing Power of VLSI Circuits," *Proc. 21st Symp. on the Foundations of Computer Science*, IEEE Computer Society, pp. 294-300, October 1980.

[Yao 79] A. C. Yao, "Some Complexity Questions Related to Distributive Comptuting," *Proc. 11th Annual ACM Symp. on Theory of Computing*, pp. 209-213, May 1979.

[Yao 81] A. C. Yao, "The Entropic Limits of VLSI Computations," *Proc. 13th Annual ACM Symp. on Theory of Computing*, pp.308-311, May 1981.