G. Miranker
L. Tang
C. K. Wong

A "Zero-Time" VLSI Sorter

A hardware sorter suitable for VLSI implementation is proposed. It operates in a parallel and pipelined fashion, with the actual sorting time absorbed by the input/output time. A detailed VLSI implementation is described which has a very favorable device count compared to existing static RAM.

1. Introduction

Sorting is one of the most important operations in data processing. It is estimated that in data processing centers over 25 percent of CPU time is devoted to sorting [1]. Many sequential and parallel sorting algorithms have been proposed and studied [2–13]. Implementation of various sorting algorithms in different hardware structures has also been investigated [3, 4, 6, 11, 13–17].

In this paper, we describe a sorter in which the sorting time is completely overlapped with the input/output time. It has complete parallel operation and processes data in a pipelined fashion. It can sort in both ascending and descending order and can overlap the sorting time of two consecutive input sequences. Because of the regularity of its structure, it is most suitable for VLSI implementation. A detailed implementation is presented to illustrate the basic principle. Further optimization in various aspects of the design is clearly possible.

2. Principle

The sorter consists basically of a linear array of $n/2$ cells (we assume n is even), each of which can store two items of the sequence to be sorted (Fig. 1). There is only one connection between a cell and its upper and its lower neighbor cell. After comparison, one of the two items goes to the next neighbor cell through this connection. Since the data flow is the same for all cells at any given time, this removed item occupies the space newly created in the next cell. (The removed item at the bottom cell goes out of the array in a downward data flow while the item at the top cell goes out of the array in an upward data flow.) The initial sequence is entered into the sorter one item at each step. After the last item has been entered, the data flow direction is reversed, and the sorted sequence is then extracted as output, also serially. Each step, executed synchronously and simultaneously by all the cells, has two phases:

1. Compare: The two items in each and every cell are compared to each other,
2. Transfer: Subject to the result of the comparison, the desired sorting order (ascending or descending), and the sorting state (input or output), one or the other of the two items is transferred to a neighbor cell and the original cell receives an item from the other neighbor cell.

The sorter not only processes the items of a given sequence in a pipelined fashion, but can also sort different sequences in a pipelined way (provided that some extra hardware is added to the sorter), i.e., while one sorted sequence is being produced as output, a new sequence can be entered as input at the same time from the other end of the sorter. In this way, the I/O time of the sequence is completely absorbed by the sorting time needed by another sequence.

Figure 2 shows an example of the sorting of a sequence in ascending order. Here "∞" represents the largest item possible. At the input stage the larger of the two items in each cell is transferred down, while at the output stage the

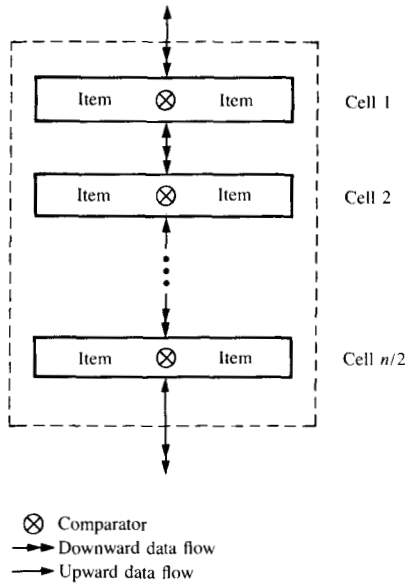


Figure 1 Block diagram of the sorter.

smaller of the two is transferred up. Note that at the end of the input stage (step 6), the smallest item must be in the top cell, the second smallest must be in either the top or the second cell. In general, the ith smallest item must be in one of the top i cells. This is why the output sequence is sorted.

The same principle applies to the descending sort; we have only to replace "∞" by "$-\infty$," the smallest item, and interchange larger and smaller. It is shown later that it is not necessary to flood the sorter initially with either "∞" or "$-\infty$." (See Fig. 14, shown later.)

Let A, B be the two items stored in a cell. Let $M = \max(A, B)$, $m = \min(A, B)$. If we consider the sorting of an isolated sequence, and the sequence is entered through and extracted from the top (top sequence), the specific action in the transfer phase can be summarized as shown in Table 1.

If the sequence is entered through and extracted from the bottom port of the sorter (bottom sequence), the situation would be as reflected in Table 2. A fact to be noted is that the roles of M and m are interchanged when we consider a descending as opposed to an ascending sort.

When we overlap the output of a sequence with the input of another, it is clear from Tables 1 and 2 that the transfer actions are different for the two sequences. For example, for an ascending sort, in the upward movement, we have $m\uparrow$ for the output (top) sequence and $M\uparrow$ for the input (bottom) sequence. That is, in each cell containing two items from the

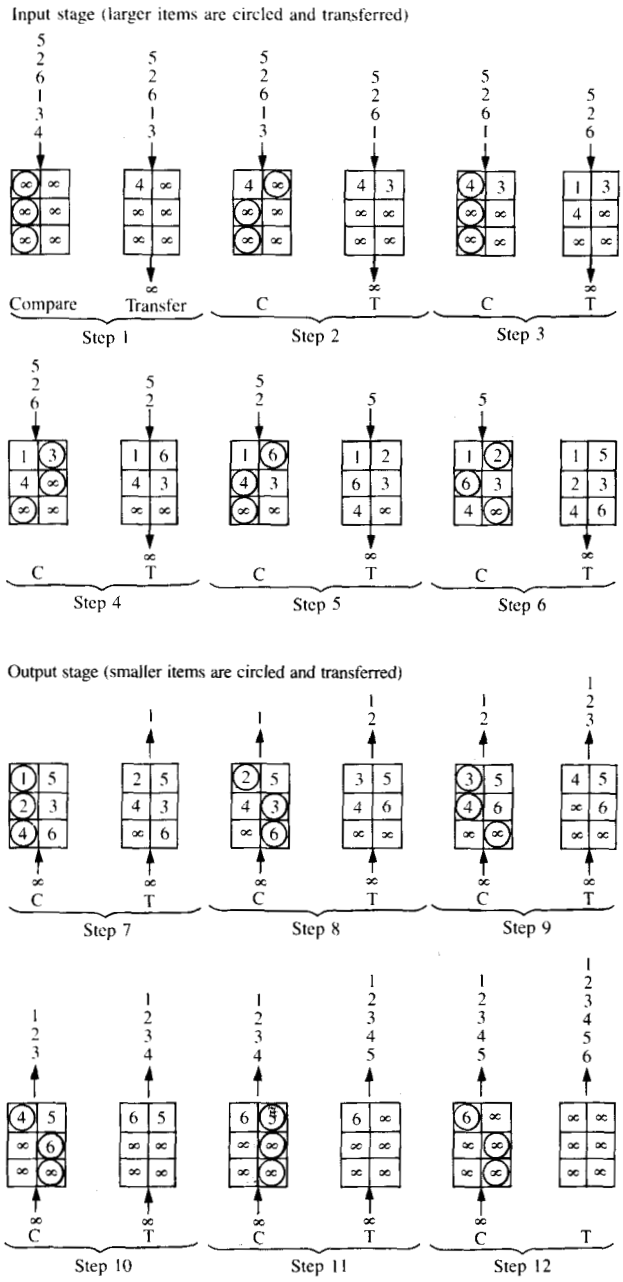


Figure 2 Example of ascending sort of a single top sequence.

Table 1 Transfer actions for a single top sequence.

Sort order \ *Stage*	*Input (down)*	*Output (up)*
Ascending	M moves down to next cell ($M\downarrow$)	m moves up to next cell ($m\uparrow$)
Descending	m moves down to next cell ($m\downarrow$)	M moves up to next cell ($M\uparrow$)

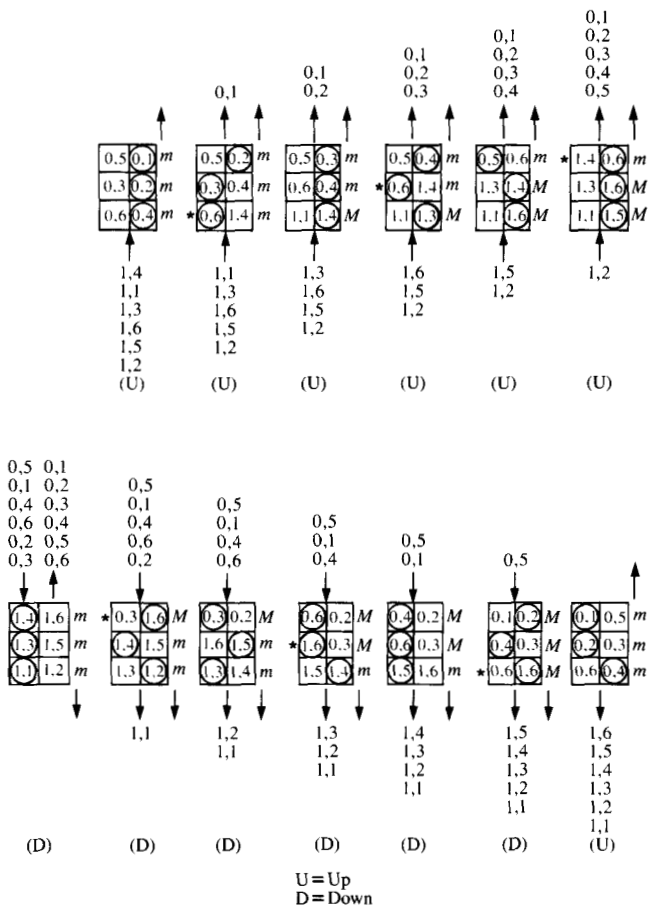


Figure 3 Example of sorting overlapping sequences.

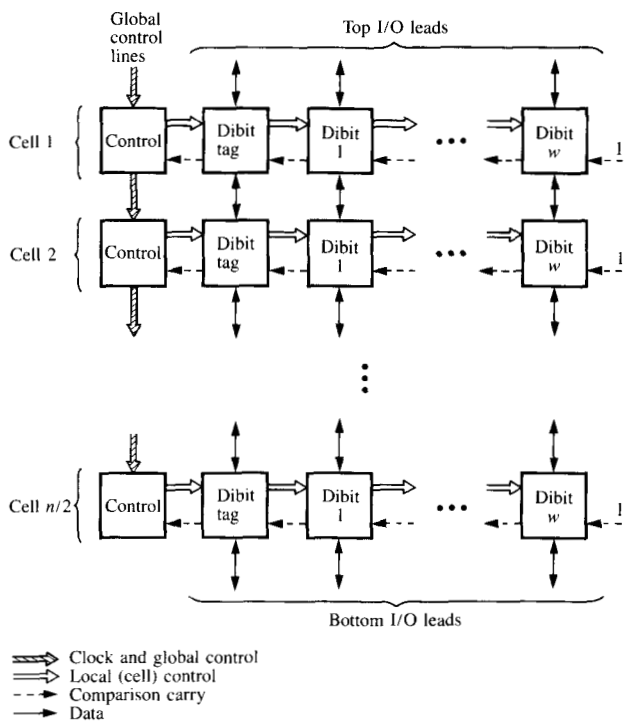


Figure 4 Overall topological layout of sorter.

Table 2 Transfer actions for a single bottom sequence.

Sort order \ *Stage*	*Input (up)*	*Output (down)*
Ascending	$M\uparrow$	$m\downarrow$
Descending	$m\uparrow$	$M\downarrow$

Table 3 Transfer actions for overlapping sequences.

Data movement \ *Tag bits*	*0*	*0*	*1*	*1*	*0*	*1*
Downward	$M\downarrow$	$(m\downarrow)$	$m\downarrow$	$(M\downarrow)$	$M\downarrow$	$(M\downarrow)$
Upward	$m\uparrow$	$(M\uparrow)$	$M\uparrow$	$(m\uparrow)$	$m\uparrow$	$(m\uparrow)$

top sequence, we promote the minimum of the two upwards ($m\uparrow$), while in each cell containing two items from the bottom sequence, we promote the maximum of the two upwards ($M\uparrow$). This is because the top sequence is in its output mode, and items should go out in ascending order, while the bottom sequence is still in its input mode, and larger items should be pushed up to the top so that later in the output mode the items of this sequence can come out (from the bottom) in ascending order.

For a cell containing an item from each sequence, we want to promote the item from the top sequence up, whatever the relative magnitudes of the two items may be. Thus we attach a flag to each item when it is entered: "0" ("1") to items in the top (bottom) sequence. This flag is considered part of the item, in the comparison as well as in the transfer. Consequently, for a cell containing items from both sequences, we simply promote the minimum of the two up ($m\uparrow$). Thus, we obtain Table 3 on transfer actions. The parenthesized entries correspond to the descending sort.

The third column represents the frontier cell between the two sequences. If we include the tag bit as the most significant bit of the items for the purpose of comparison, the item from a bottom sequence with tag bit = 1 is always M and the two sequences are always kept separate. An example of the sorting with the added tag bits is shown in Fig. 3.

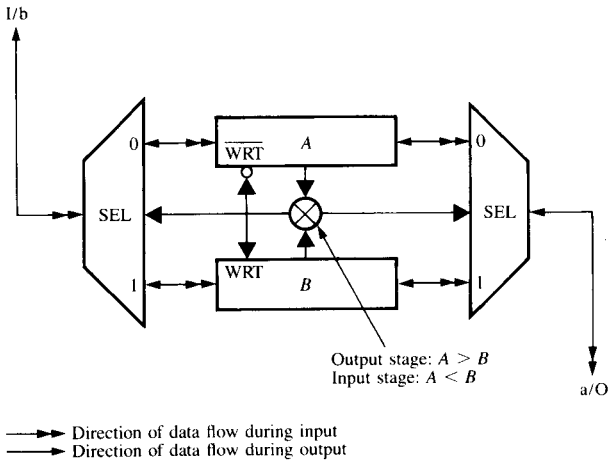


Figure 5 Block diagram of a dibit cell.

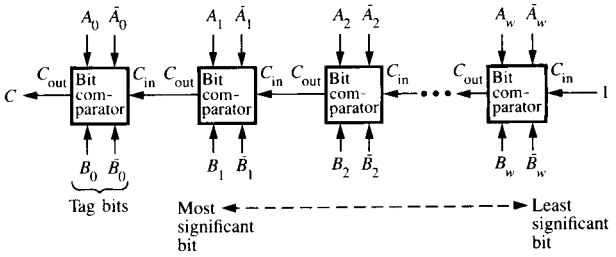


Figure 6 Overall comparator structure.

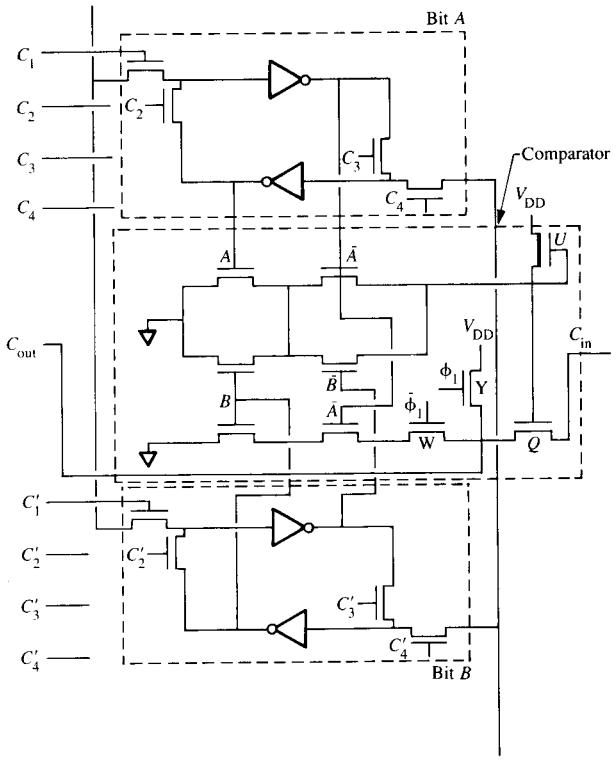


Figure 7 Circuit schematic of a dibit cell.

It should be emphasized once again that the sorter can be used to sort one ascending and one descending sequence with the existing flags but without the complexity of the moving M/m boundary.

3. Logic design

Throughout this paper, the cell array of the sorter is represented vertically. Each cell, containing two w-bit items, is a horizontal linear array (row) of w "dibit" cells. The overall topological layout is shown in Fig. 4. In an actual physical layout, a carpenter folding [18] of the cell array might be needed to obtain a more square-shaped chip.

Dibit cell Each such cell is a compare/steer unit for two bits, one from each of the two items A and B, representing the same bit position. For simplicity, these bits are referred to as bit A and bit B, respectively. Figure 5 is the block diagram of a dibit cell. In downward (upward) movements, after comparison, one of the two bits is shifted out on line a (b) to the next (previous) cell, while a bit from the previous (next) cell is being shifted in on line I (O). In this figure, the terms "input" and "output" refer to a top sequence, and the controls are indicated for an ascending sort. For example, at the input stage, if $A < B$, then the signal from the comparator is 1, which sets off the selector (SEL), allowing bit B to go down line a and a new bit to come in from I. The case $A = B$ also generates signal 1.

The comparators of the dibit cells in a cell row are chained as in Fig. 6. C is the comparison result of items A and B, i.e., $C = 1$ if item $A \geq$ item B and $C = 0$ otherwise. The comparison carry chain is precharged during clock phase ϕ_1 (gates W and Y in Fig. 7).

A circuit schematic of a dibit cell is shown in Fig. 7. The precharged carry-propagate-type comparator is shown together with the two bit cells. It should be noted that every bit cell of item A (B) in a cell row is controlled by the same four signals C_1, C_2, C_3, and C_4 (C'_1, C'_2, C'_3, and C'_4), so that all the bits of an item are recycled or shifted at the same time.

Since the comparator circuit in Fig. 7 is very important for implementation, we give a more detailed explanation of its action here. The comparison result is as follows: If bit A

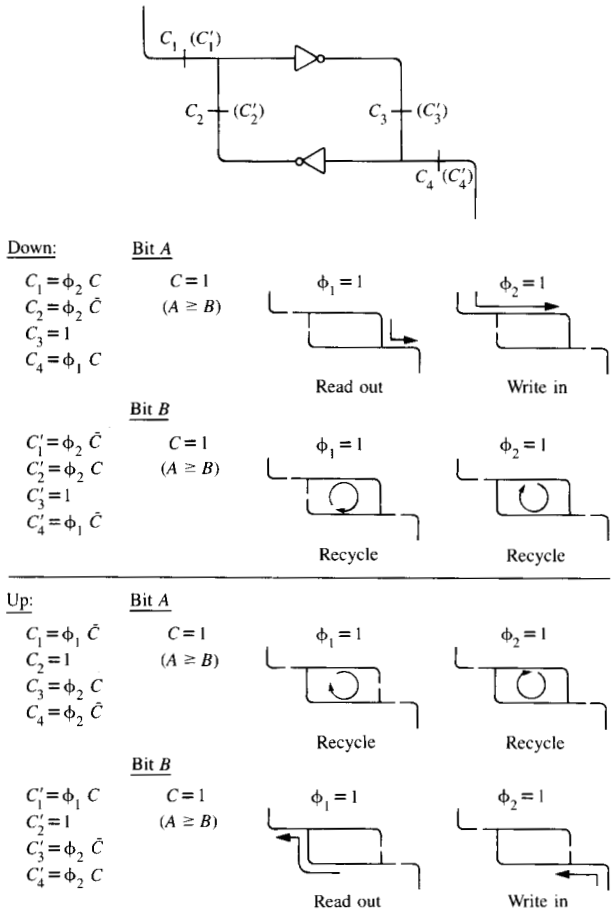


Figure 8 Required gatings for the $A \geq B$.

$>$ bit B, then $C_{out} = 1$; if bit $A <$ bit B, then $C_{out} = 0$; and if bit $A =$ bit B, then $C_{out} = C_{in}$. To verify this, we just consider the case when $A = 1$, $B = 0$. Other cases are similar. $A = 1$, $B = 0$ implies that gate U is grounded; thus the signal at U is 1. The U, Q pair form an inverter. Consequently, the signal at Q is 0. On the other hand C_{out} is precharged to 1 at $\phi_1 = 1$. Now that $B = 0$, $\overline{A} = 0$, $Q = 0$, we have $C_{out} = 1$ whatever the value of C_{in} may be.

The other parts (i.e., the bit cells) of Fig. 7 are explained in the next paragraph.

Control To illustrate, let us consider an ascending sort with a top sequence. Each cell is a two-inverter loop controlled by four gates using a two-nonoverlapping-phase clock. Note that for the global control of the sorter, we need one extra clock phase, in which one can change from the up to the down phase, or the down to the up. It can also be used for initialization. But more importantly, it is needed to make sure that a racing condition does not occur. (See the section on timing.) The required gatings for different situations with $A \geq B$ (i.e., comparison result $C = 1$) are shown in Fig. 8.

More specifically, $C = 1(A \geq B)$ in the down phase generates the following signals:

$C_1 = \phi_2$,

$C_2 = 0$,

$C_3 = 1$,

$C_4 = \phi_1$

for bit A and

$C_1' = 0$,

$C_2' = \phi_2$,

$C_3' = 1$,

$C_4' = 0$

for bit B. Thus, at $\phi_1 = 1$, only gates C_3, C_4 are connected for bit A and only gate C_3' is connected for bit B, corresponding to reading out bit A and recycling bit B. At $\phi_2 = 1$, gates C_1, C_3 are connected for bit A and gates C_2', C_3' are connected for bit B, corresponding to writing in a new bit to replace bit A and continuing recycling bit B. The up phase is similar. In the case of $A < B$, just interchange the gatings for A and B. The boolean expressions for generating the correct signals in all situations are as follows:

$$C_1 = \phi_2 Ia + \phi_1 \bar{I}\bar{a}, \quad C_1' = \phi_2 I\bar{a} + \phi_1 \bar{I}a;$$
$$C_2 = \phi_2 I\bar{a} + \bar{I}, \quad C_2' = \phi_2 Ia + \bar{I};$$
$$C_3 = I + \phi_2 \bar{I}a, \quad C_3' = I + \phi_2 \bar{I}\bar{a};$$
$$C_4 = \phi_1 Ia + \phi_2 \bar{I}\bar{a}, \quad C_4' = \phi_1 I\bar{a} + \phi_2 \bar{I}a.$$

$I = 1(0)$ indicates the downward (upward) movement; a is the boolean variable which takes opposite values (0 and 1) in opposite situations:

- Ascending ($Opt = 0$) versus descending sort ($Opt = 1$),
- Top ($SR = 0$) versus bottom sequences ($SR = 1$),
- And $A \geq B$ (comparison carry $C = 1$) versus $A < B$ ($C = 0$).

It follows that a is the exclusive-OR of C, SR, and Opt, i.e.,

Opt \ SR	0	1
0	C	$\overline{C}$
1	$\overline{C}$	C

See Fig. 9 for the circuit schematic of the cell control.

To have homogeneous and regular cells, we have avoided the explicit use of the tag bit combination to distinguish top

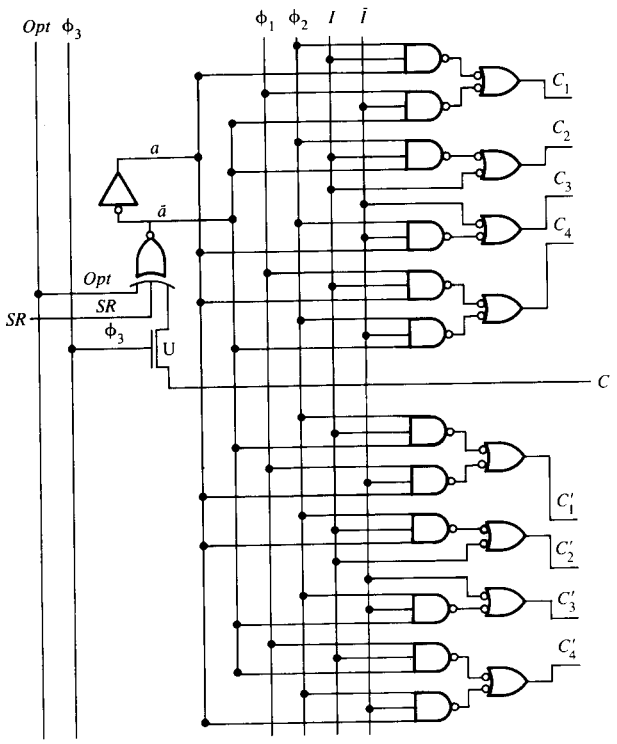


Figure 9 Circuit schematic of the cell control.

Table 4 Transfer actions and corresponding shift register control for overlapping sequences.

	Ascending			*Descending*		
Tag bits	00	11	01	00	11	01
Down	$M\downarrow$	$m\downarrow$	$M\downarrow$	$m\downarrow$	$M\downarrow$	$M\downarrow$
Up	$m\uparrow$	$M\uparrow$	$m\uparrow$	$M\uparrow$	$m\uparrow$	$m\uparrow$
SR	0	1	0	0	1	1

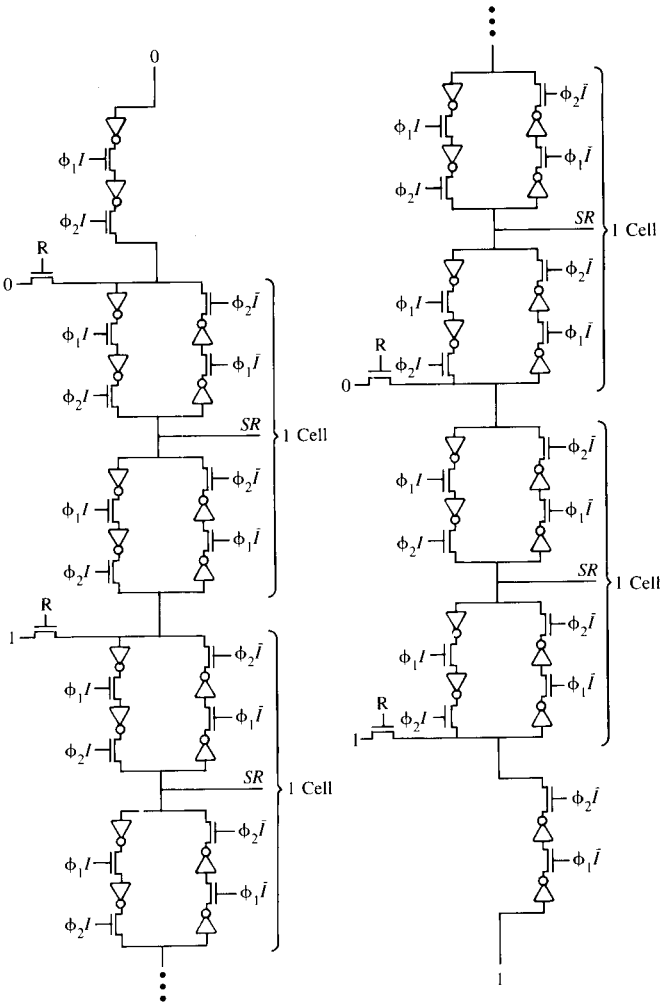


Figure 10 Bidirectional double shift-register chain to control the transfer action.

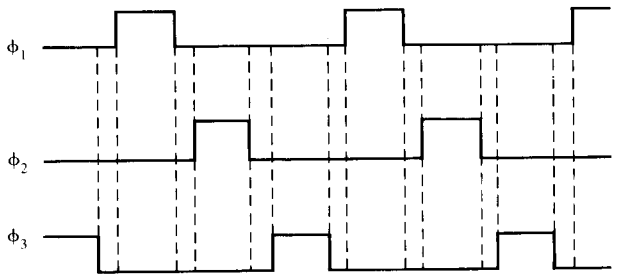
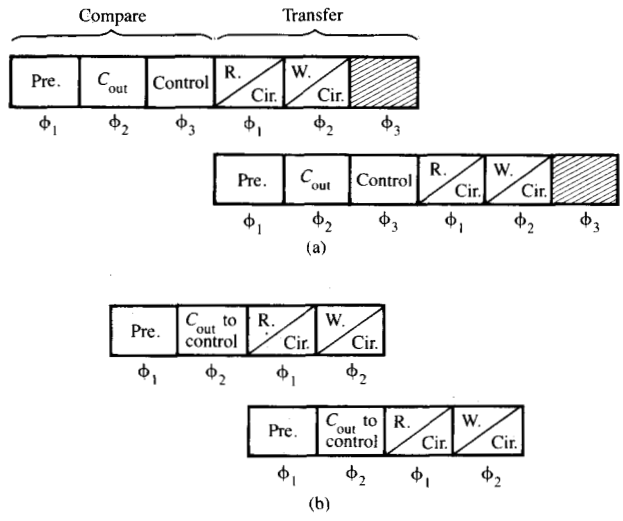


Figure 11 Three-phase clock.

and bottom sequences (Table 3); instead we have a bidirectional double shift-register chain,whose contents move up and down in synchrony with those of the cells and whose output at each level is taken to be SR, as shown in Fig. 10, so that an item of a top (bottom) sequence is always chaperoned by $SR = 0$ (1). A slight complication occurs at the frontier. The desired transfer action then is shown in Table 4. The reader can easily check from Fig. 10 that the two extra unidirectional shift registers at the two ends are needed to fulfill the requirement of the third column in both ascending and descending sort.

4. Timing

As mentioned in the previous section, we use a three-nonoverlapping-phase clock, as shown in Fig. 11. During phase ϕ_1, the transfer bit is read out from cell (i), while the other bit is recycled and the comparison carry chain precharged [Fig. 12(a)]. During ϕ_2, the transfer bit is written



Pre. = Precharging the comparison carry line.
C_{out} = Compute C_{out} in the comparators for bit pairs, obtaining C.
Control = C is fed into the control circuit of C_1, C_2, C_3, C_4, and C_1', C_2', C_3', C_4'.
R. = Read the transfer bit out to the next cell (down or up).
W. = Write in the transfer bit from the other next cell (up or down).
Cir. = The stay bit is recycled in the cell.

Figure 12 (a) Action taken at different clock phases. (b) Racing condition would occur without phase 3.

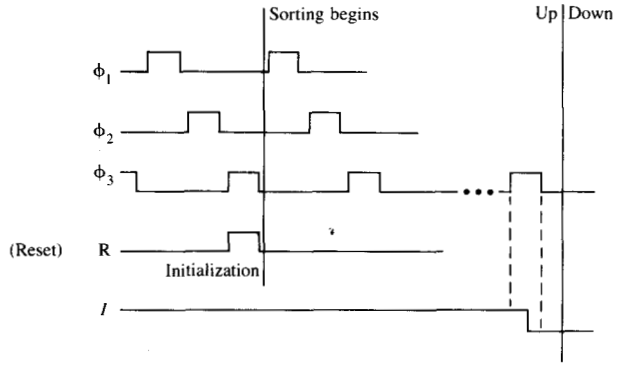
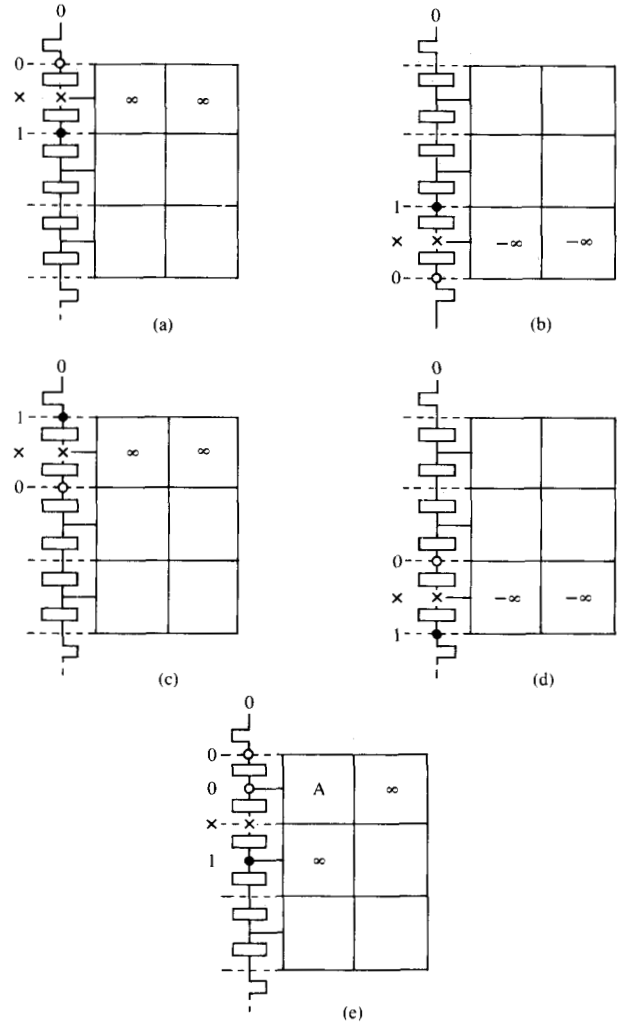


Figure 13 Action taken at phase 3.

into the next cell ($i + 1$ or $i - 1$), while the other bit is making a full recycle and the comparison is taking place. At ϕ_3, the comparison result signal is fed into the control circuit of each cell.

In addition, phase ϕ_3 is needed (see Fig. 13),

1. For the transition from the up to down and down to up stages,



○ = zero, × = don't care, ● = 1, A = item from sequence

Figure 14 Initialization for different sorting situations. (a) Initialization 1 for ascending sort ($t = 0$). (b) Initialization 2 for ascending sort ($t = 0$). (c) Initialization 2 for descending sort ($t = 0$). (d) Initialization 1 for descending sort ($t = 0$). (e) Sorting from configuration (a) at $t = 1+$ (one clock cycle afterward).

2. For the initialization,
3. And to avoid a racing condition in the loop of the comparator, control, and bit cell.

Specifically, at $\phi_2 = 1$, we obtain the value of C (see Fig. 9), and it goes to the control at $\phi_3 = 1$. At the next $\phi_1 = 1$, bit A and bit B begin their transfer phase while the C line (or the C_{out} line in the individual bit comparators) is precharged. [See Fig. 12(a).] At $\phi_2 = 1$, a new bit is written in (the other has been circulating) and compared with the other bit. Without ϕ_3, we would have the situation in Fig. 12(b). Thus, at the second $\phi_2 = 1$, while writing out according to a previous set of signals C_i, C_i', $i = 1, 2, 3, 4$, a comparison

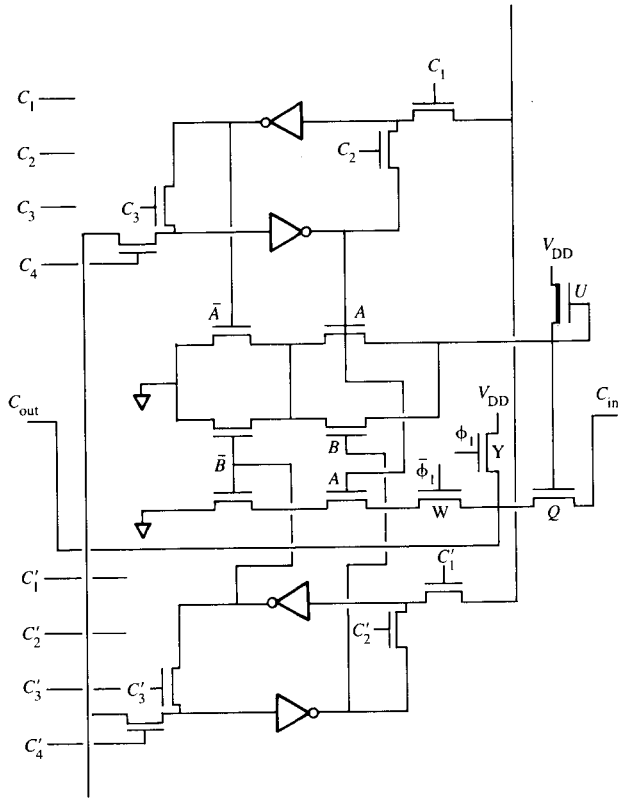


Figure 15 Circuit schematic of a dibit cell with comparison done on complemented bits.

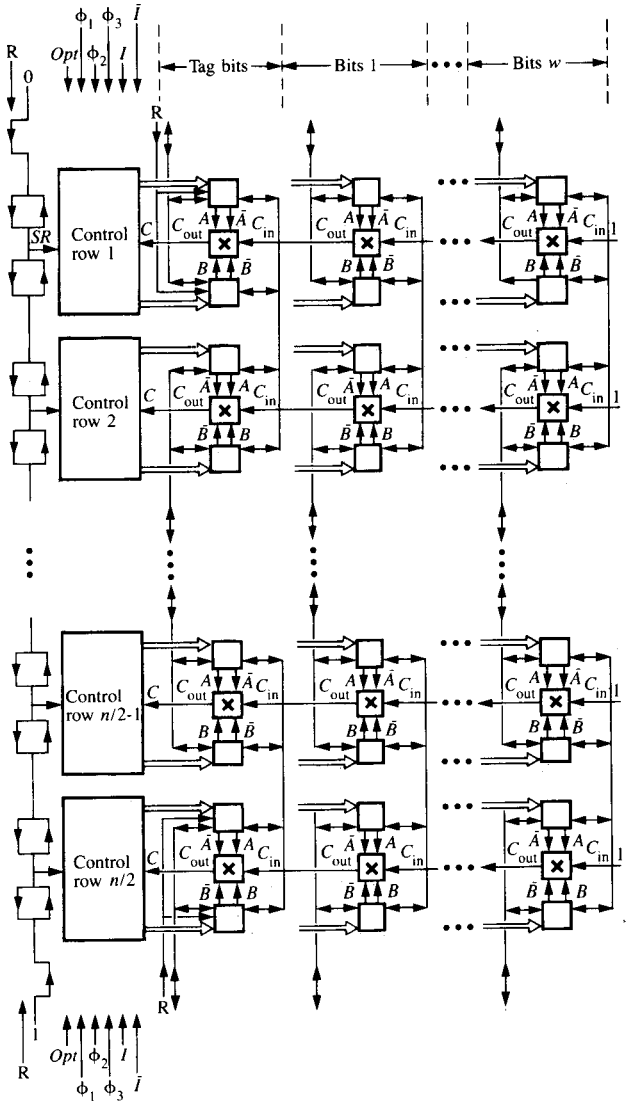


Figure 16 Overall sorter structure.

would be made, C_{out} (or C) would go to the control, and a new set of signals $\hat{C}_i$, $\hat{C}'_i$, $i = 1, 2, 3, 4$, would be generated. A racing condition would have occurred.

5. Initialization

Before the beginning of a sort, instead of initializing all the cells with "∞" or "$-\infty$," it is necessary only to fill in the two border cells with tags, distinct from the tags of the sequence coming in, together with appropriate setting of the comparison shift registers as in Fig. 14. Recall that top (bottom) sequences have tag bit "0" ("1"). So here "∞" ("$-\infty$") represents any number with tag bit "1" ("0"). It could be easily checked from Table 4, and, e.g., Fig. 14(e) that these initializations are indeed adequate.

All the initial values are injected into the sorter during clock phase ϕ_3.

6. Concluding remarks

1. The circuits are drawn up as if the wires connecting dibit cells of rows i and $i + 1$ have enough capacitance to store the transfer bit. If they do not, it would be a simple matter to add to them connection inverters. Without the inverters, comparisons on adjacent row cells must be implemented differently. Indeed, as can be seen in Fig. 6, a bit leaving a cell is in complemented form in comparison to when it was entered. Therefore, to produce the same comparison carry output we need to invert the roles of A and $\bar{A}$, and also B and $\bar{B}$, as in Fig. 15. A redrawn global block diagram is shown in Fig. 16, where the alternation between adjacent rows is clearly indicated. Note also that an even number of rows is recommended so that data are entered and extracted in "true" form. (Otherwise either the top or bottom would be in "false," i.e., negated form.)
2. For our implementation (Fig. 6) we have a device count of 26 for a dibit cell, i.e., 13 per bit versus 6 in today's 16K static RAM. So a sorter chip would very likely have a

capacity of up to 8K bits or 256 32-bit cells. The sorter can be trivially extended to handle key/pointer pairs by simply omitting the compare logic on the portion of the storage cell associated with the pointer. (Then it will require only eight devices per pointer bit.)

3. We can use the sorter to merge two sorted strings by repeatedly passing them through the sorter in an appropriate way. For example, the generalized odd-even merge algorithm described in [6, p. 241, Exercise 38] can be employed for this purpose. Reference [6] has a detailed description and timing analysis of this merging algorithm.

References

1. D. E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley Publishing Co., Reading, MA, 1973.
2. K. E. Batcher, "Sorting Networks and their Applications," *AFIPS Conf. Proc.* **32,** 307–314 (1968).
3. T. C. Chen, V. Y. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files," *Proceedings of the Fourth International Conference on Very Large Data Bases*, ACM, New York, September 1978, pp. 312–318.
4. K. Chung, F. Luccio, and C. K. Wong, "On the Complexity of Sorting in Magnetic Bubble Memory Systems," *IEEE Trans. Computers* **C-29,** 553–563 (July 1980).
5. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," *Commun. ACM* **21,** 657–661 (August 1978).
6. D. T. Lee, H. Chang, and C. K. Wong, "An On-Chip Compare/Steer Bubble Sorter," *IEEE Trans. Computers* **C-30,** 396–405 (June 1981).
7. H. Lorin, *Sorting and Sort Systems*, Addison-Wesley Publishing Co., Reading, MA, 1975.
8. D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," *J. ACM* **22,** 195–201 (April 1975).
9. D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Computers* **C-28,** 2–7 (January 1979).
10. F. P. Preparata, "New Parallel-Sorting Schemes," *IEEE Trans. Computers* **C-27,** 669–673 (July 1978).
11. Y. Tanaka, Y. Nozaka, and A. Masuyama, "Pipelined Searching and Sorting Modules as Components of a Data Flow Database Computer," *Proc. IFIP '80*, October 1980, pp. 427–432.
12. C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Commun. ACM* **20,** 263–271 (April 1977).
13. H. Yasuura, N. Takagi, and S. Najima, "The Parallel Enumeration Sorting Scheme for VLSI," *IEEE Trans. Computers* (to appear).
14. M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *IEEE Computer* **13,** 26–40 (January 1980).
15. H. T. Kung, "The Structure of Parallel Algorithms," *Advances in Computers*, Vol. 19, Academic Press, Inc., New York, 1980, pp. 65–112.
16. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Co., Reading, MA, 1980.
17. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *IEEE Trans. Computers* **C-28,** 384–394 (June 1979).
18. C. E. Leiserson, "Area-Efficient Graph Layouts (for VLSI)," *Proc. 21st Annual Sym. on Foundations of Computer Science*, IEEE, New York, Oct. 13–15, 1980, pp. 270–281.

Received April 13, 1982; revised October 7, 1982

Glen S. Miranker *Valid Logic Systems, Inc., 650 North Mary Avenue, Sunnyvale, California 94086.* Dr. Miranker is an engineering group leader at Valid Logic Systems. From 1979 to 1981, he was with IBM at the Thomas J. Watson Research Center in Yorktown Heights, New York. While there he worked on the simulation engine and single chip 801. He also taught VLSI design for the IBM Education Department at the Research Center. From 1980 to 1981 he was an adjunct professor at Columbia University, New York. Dr. Miranker received a B.S. in computer sciences in 1975 from Yale University, New Haven, Connecticut, an M.S. in 1977 and a Ph.D. in 1979, both from the Massachusetts Institute of Technology, Cambridge. He was a Hertz scholar while at college. Dr. Miranker is a member of the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, and Tau Beta Phi.

Luong Tang *Switchco Inc., 329 Alfred Avenue, Teaneck, New Jersey 07666.* Mr. Tang has been development manager at Switchco since March 1982. He is also a Ph.D. candidate in the Electrical Engineering Department at Columbia University, New York. His principal interest is in computer communications. This work was pursued while on summer leave at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. Mr. Tang received his Engineer Diploma from the Ecole Supérieure d'Electricité in France in 1972.

Chak-Kuen Wong *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Wong joined IBM in 1969 as a member of the Computer Sciences Department at the Thomas J. Watson Research Center. His current interests include VLSI design algorithms, abstract and concrete computational complexity theory, optimization problems related to data allocation, magnetic bubble memory structures, theory of fuzzy sets, and satellite switching/time domain multiple-access systems. Dr. Wong received a B.A. in mathematics from the University of Hong Kong in 1965 and an M.A. and a Ph.D. in mathematics from Columbia University, New York, in 1966 and 1970, respectively. For the academic year 1972 to 1973, he was a Visiting Associate Professor of Computer Science in the Department of Computer Science at the University of Illinois, Urbana. For the academic year 1978 to 1979, he was a Visiting Professor of Computer Science in the Department of Electrical Engineering and Computer Science at Columbia University, New York. Dr. Wong received an IBM Outstanding Invention Award in 1971 for a new family of sorting methods and two IBM Invention Achievement Awards. He holds three U.S. patents and is currently writing a book on algorithms in mass storage systems. Dr. Wong is a member of the Association for Computing Machinery and the New York Academy of Sciences, and a senior member of the Institute of Electrical and Electronics Engineers. He is also an editor of the *IEEE Transactions on Computers*, an Advisory Editor of the international journal *Fuzzy Sets and Systems*, and a Foreign Editor of the Chinese journal *Fuzzy Mathematics*.