Data-Driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter

Behrooz Parhami, *Fellow*, *IEEE*, and Ding-Ming Kwai

Abstract—We present a strategy for designing stable insertion sorters based on linear arrays with data-driven control. The novelty of our approach lies in each data item carrying a control tag to specify how it is to be operated upon by a receiving cell and in performing two parallel comparisons within each cell. To assure first-in/first-out handling of equal key values, some data items must be marked to reflect their past histories. Such marking is conveniently carried out by modifying the data item's control tag. It is the combination of the above features that allows us to derive the first single-cycle priority queue that operates in fully pipelined mode, with no broadcasting of data values or control signals. By performing more than two parallel comparisons in each cell, the VLSI implementation cost of our stable sorter can be reduced. We show that highly cost-effective designs can be obtained by selecting an optimal cell size in terms of the number of comparators it contains.

Index Terms—Data-driven architectures, distributed control, FIFO, linear processor arrays, priority queue, stable sorting, tagged insertion sorter, VLSI.

1 Introduction

Hardware sorting capability is required in many applications, including large data routing switches and priority queues. A sorter deals with a data structure composed of a list of n records, each of which has a key whose value dictates its position in the eventual ordering.

The problem of sorting in nondecreasing order may be stated as follows. Given a list of n records with key values $x_1, x_2, \cdots, x_n$, rearrange them into a new list with keys $y_1 = x_{k_1}, y_2 = x_{k_2}, \cdots, y_n = x_{k_n}$, such that $y_1 \le y_2 \le \cdots \le y_n$ and, for each pair of consecutive outputs satisfying $y_i = y_{i+1}$, we have $k_i < k_{i+1}$. In other words, we impose the first-in/first-out (FIFO) order on the set of records that have equal key values. In the literature on sorting, such an algorithm is referred to as a stable sorting algorithm.

Even though theoretical log-depth and somewhat more practical $\log^2$-depth sorting networks are widely known [13], most hardware implementations of sorting are based on a linear array structure in view of its simplicity and ease of VLSI realization [16]. Pipelined schemes, such as rebound sorting [2], [4], [5], up/down sorting [8], enumeration sorting (via counting) [15], and balanced heap sorting [9], have been proposed on linear arrays. To implement a stable sorting algorithm, however, it is often assumed that an entry number or input order index is included with each record. This index expands the range of key values by a factor of n, thus leading to added cell complexity and poor scalability.

- *B. Parhami is with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560. E-mail: parhami@ece.ucsb.edu.*
- *D.-M. Kwai is with Worldwide Semiconductor Corp., Hsinchu, Taiwan.*

Manuscript received 18 July 1996.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100244.

The tagged up/down sorter [12] uses a single-bit tag to mark a displaced record; when set, this tag bit indicates that the record is to be returned to its proper FIFO order. However, because the up/down sorter performs sorting during both insertion and extraction, it does not provide ready access to the record with the largest key value in the list. Thus, when used as a priority queue, the tagged up/down sorter lacks the overwriting capability viz. the ability to discard the lowest-priority event to make room for a higher-priority one [14].

To allow overwriting, the sorter has to perform sorting only during insertion. Moreover, if interleaving of insertion and extraction is to be possible, input and output must be through one end of the linear array, leading to bidirectional data flow. To ensure an appropriate data flow commensurate with insert and extract operations, each cell needs a certain amount of control or decision-making capability.

Previous work has suggested that control signals and data values can be broadcast to all cells, making the sorter behave like a dynamically partitioned shift register [7]. The simplicity of the control circuitry in such broadcast-based designs is due to the use of buses which establish global communications paths in the linear array.

To ensure regular control flow in a sorter, one might pipeline the control signals along with the data values. In the resulting data-driven control scheme, each inserted data item carries a control tag indicating its state and how it is to be operated upon by the receiving cell. This scheme yields a simple and regular cell structure with only local communication [3], [11], thus limiting the global connections to clock and power supply.

In this paper, we present such a data-driven control for the stable insertion sorter that supports a very high clock rate and unlimited scalability. By pipelining the control signals, which take the form of tags attached to the data values, cell operations can be orchestrated in a totally distributed manner.

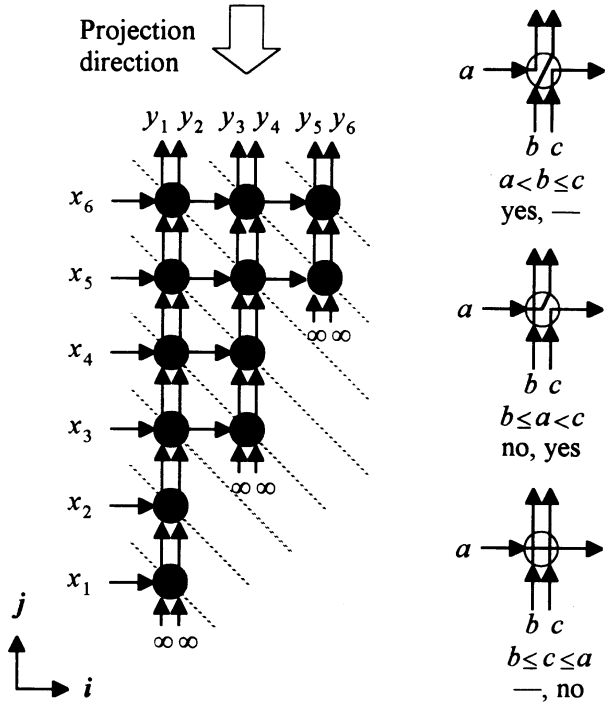


Fig. 1. Dependence graph for insertion sorting with two comparisons per node. Also shown are the node's three routing states based on the outcomes of $a < b$ and $a < c$ comparisons $b \le c$.

The control tags also provide a convenient mechanism for carrying abbreviated record histories that enable efficient implementation of stable sorting or FIFO order. It is a combination of the preceding features that allows us to obtain a single-cycle priority queue that operates in fully pipelined mode with neither data nor control broadcasting while also providing the overwriting capability.

Our presentation is organized as follows. Section 2 introduces a tagged insertion sorter with data-driven control in which each cell performs two key comparisons in parallel. Section 3 deals with the case when the tagged insertion sorted is used as a priority queue and addresses issues of FIFO ordering and exception handling. In Section 4, we generalize our design by allowing each cell to do more than two comparisons at the same time, discussing cost-effectiveness and optimality issues. Section 5 contains our conclusions.

2 Tagged Insertion Sorter

2.1 Deriving the Insertion Sorter

The Insertion sorting algorithm can be represented by the dependence graph of Fig. 1, where each node represents the operation of comparing the horizontally arriving datum a with the vertically arriving data b and c (∞ denotes the largest possible key value). The dashed equi-temporal lines denote a scheduling scheme in which time is partitioned into discrete steps by a global clock. Such a synchronous mode of operation with regular data flow is commonly known as systolic computation [6].

A linear array and its corresponding cell structure can be obtained by projecting the dependence graph of Fig. 1 in the j direction. The resulting array, with input and output at its left end, is depicted in Fig. 2. The nodes located at the same i coordinate are mapped to a single cell whose operations are regulated by a global clock according to the given schedule. Each cell is connected to its left and right neighboring cells, except that the rightmost cell has no right neighbor (its right input is externally supplied) and the leftmost cell serves as an input/output point for the entire network.

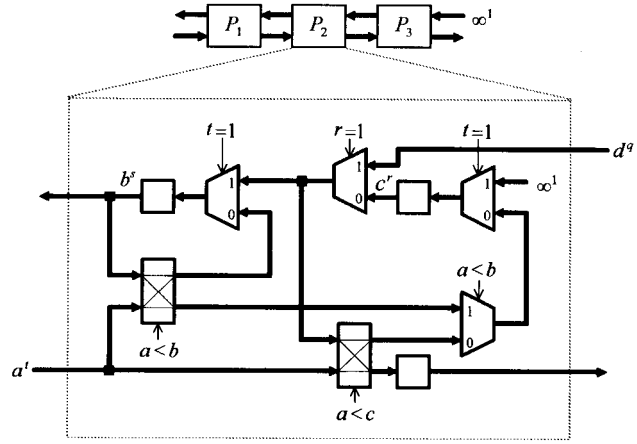


Fig. 2. A three-cell insertion sorter and its cell structure.

Given a and $b \le c$, the rearrangement of a, b, and c in nondecreasing order can be accomplished as follows. Depending on the outcomes of the two comparisons or tests $a < b$ and $a < c$, the rearrangement yields one of the following three orders, as shown in Fig. 1:

$a < b \le c$ Comparison outcomes: (yes, —)

$b \le a < c$ Comparison outcomes: (no, yes)

$b \le c \le a$ Comparison outcomes: (—, no)

Note that the smallest value is $\min(a, b)$ and the largest value is $\max(a, c)$. Clearly, the two comparisons needed to distinguish which of the three cases above holds can be performed in parallel. In Fig. 2, the multiplexer controlled by the comparison outcome $a < b$ is used to select the median value from $\max(a, b)$ and $\min(a, c)$. The right-to-left data flow in the linear array is needed to permit data output in sorting or extraction for priority queue operation. The roles of the tag bits t, r, and q are explained in Section 2.2.

2.2 Data-Driven Control Scheme

We assume that the end of insertion is signified by presenting a null record having the key value ∞ to the linear array. To each real or null record we attach a tag of 0 for insertion and 1 for extraction. The tag moves along with data and the receiving cell interprets it as an instruction. We use the notation a^t to denote a data item with key a and tag t. An input a^0 to a cell indicates that it is to be inserted and instructs the receiving cell to compare a with its stored values b and c. The outcomes of the comparisons $a < b$ and $a < c$ then direct the crossbar switches and multiplexers to order a, b, and c, passing the largest to the right.

A null record ∞^1 is input immediately after the insertion of the last normal data record to begin the extraction phase. According to the schedule in Fig. 1, the sorted data items

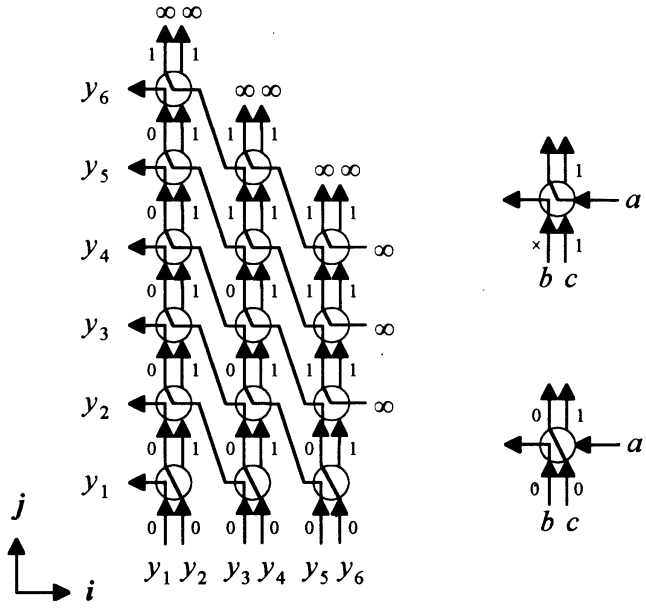


Fig. 3. Dependence graph for extraction from the tagged insertion sorter. Also shown are the node's two routing states based on the tag values marked on the vertical edges.

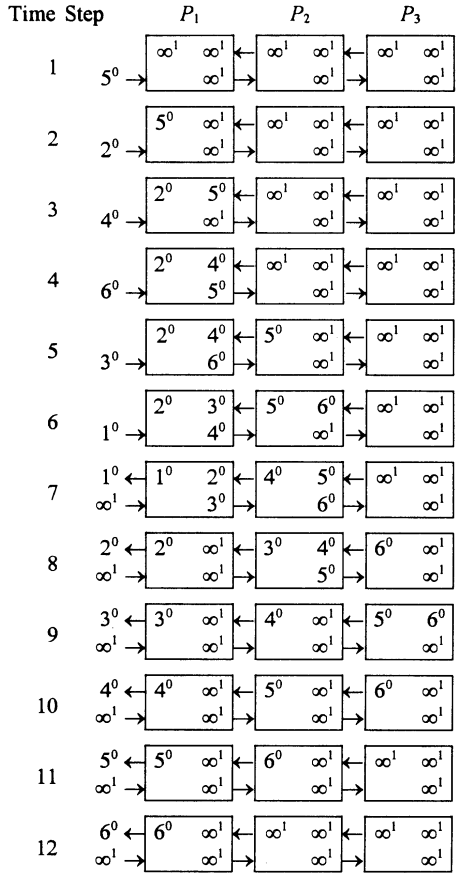


Fig. 4. An example of the insertion sorting algorithms on a three-cell linear array. Control tags are shown as superscripts.

y_{2i-1} and y_{2i} are stored in the cell P_i at time step $n + i - 1$, where $1 \le i \le \lceil n/2 \rceil$. When the null record ∞^1 arrives at the cell P_i at time step $n + i$, the values y_{2i-1} and y_{2i} held there are shifted to the left and the vacancy created by the removal of y_{2i} is filled by ∞^1.

At the next time step, y_{2i+1} is ready to be extracted; P_i fetches y_{2i+1} in synchrony with the next cell P_{i+1} shifting it to the left. The sorted data items emerge from the left end of the array in consecutive time steps. At the end of extraction, all registers are restored to their initial states. Fig. 3 depicts the dependence graph for the extraction phase. Note that the two routing states are conveniently specified by a control tag r which instructs the multiplexer in Fig. 2 to select c^r or d^q.

As an example, Fig. 4 illustrates the behavior of the linear array of Fig. 2 when sorting the sequence of numbers 5, 2, 4, 6, 3, 1. For n insert and extract operations, the control tags attached to the inputs consist of n zeros followed by n ones.

3 Priority Queue Operation

If the tagged insertion sorter is to be used as a priority queue, the cell design must be modified to handle some situations that would not arise for sorting, where all inputs are presented before extraction takes place. Since sorting can be viewed as a special case of priority queue operation, the resulting design can be used for either purpose.

We note that when an insertion occurs after an extraction, at most one normal data item is stored in the cell. Thus, we let the cell fetch the normal data item from the next cell (which pushes it out at the same time) for comparison with the incoming data. To handle the situation when neither insertion nor extraction is to be performed, we expand the tag to two bits, assigning 00 for insertion, 10 for no operation, and 11 for extraction. Insertion of ∞^{10} thus will leave the contents of the priority queue unchanged. Fig. 5 provides an example of interleaved insertions and extractions in a priority queue.

3.1 Ensuring FIFO Order

The control scheme discussed so far does not guarantee FIFO ordering. This can be shown through an example. Suppose that the data items 1, 3_1, 3_2, 2 are inserted into the array, where the subscripts of equal values indicate their input order. During insertion, 1 and 3_1 will be stored in the first cell P_1 until the smaller input 2 arrives and pushes 3_1 to the right. The data item 3_1 then lags one step behind 3_2 which has been stored in the second cell P_2. Because a cell always passes on any data item that is greater than or equal to its stored data item, 3_2 will continue to occupy the position where 3_1 should stay.

Note that forcing a swap in the case of equal values (i.e., changing the swapping condition from $<$ to $\le$) does not solve the problem, since it erroneously causes 3_2 to displace 3_1, initially stored in P_1, pushing the earlier item to the right. One way around this problem is to associate an entry order index with each data item. However, this approach in effect expands the range of key values by a factor equal to the sorter size n. This is undesirable since it increases the cell complexity and limits the size of the array which is otherwise readily expandable to arbitrarily large sizes.

A more efficient solution is to utilize the unassigned tag value 01 to designate a displaced item for enforcing the FIFO order; when an input value equal to a stored value

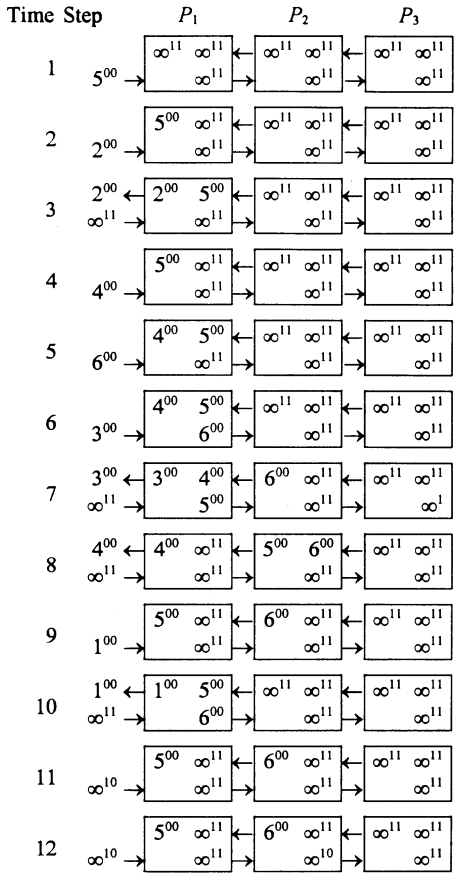


Fig. 5. An example of priority queue operation. Control tags are shown as superscripts.

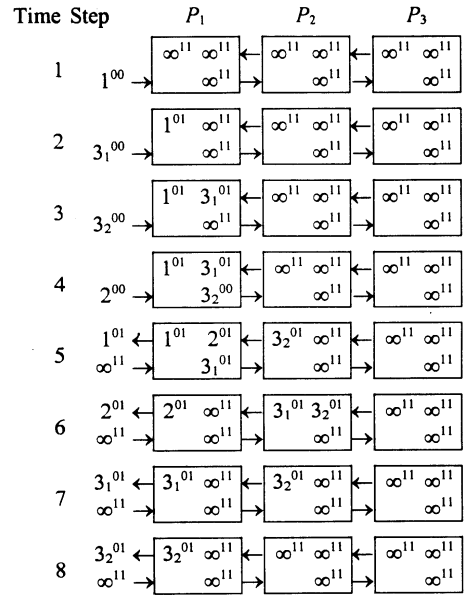


Fig. 6. Example showing the use of the tag value 01 to ensure FIFO ordering. Control tags are shown as superscripts and arrival order of equal values as subscripts.

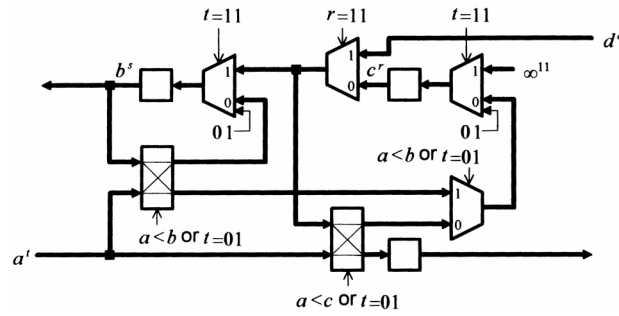


Fig. 7. The modified cell structure with data records carrying two-bit control tags to allow the incorporation of idle cycles and FIFO order.

enters a cell, the stored value is kept and the new one is forwarded to the right because of its later arrival time, but for a displaced value that was originally stored to the left of the cell, the opposite must occur. Fig. 6 shows an example for the priority queue operation using this modified scheme for the control tags.

Each inserted data item initially carries a 00 tag. When the item is stored in a cell, its tag changes to 01. This change needs no extra hardware; the tags sent from the crossbar switches to the b^s and c^r registers are simply tied to 01. When a data item tagged with 01 comes in, the cell is forced to swap and send its current data item to the right. Note that in this case the comparisons are redundant as the incoming value is always compared to a larger value or to an equal value with a later arrival time. Fig. 7 shows the structure of the modified cell that enforces the FIFO order.

3.2 Overflow and Underflow Detection

When more than n data items are inserted into the sorter, some items will be pushed out from the right end of the array. Such a discarded data item is relevant only if it carries a tag of 00 or 01. Thus, a data item tagged with 0x (i.e., with the first tag bit equal to 0) emerging from the right end of the array indicates an overflow. If the first tag bit of the emerging data item is 1, the data item is a null input, and the sorter capacity has not been exceeded.

Equally simple is underflow detection by examining the tags of extracted data items. Underflow occurs when more data items are extracted from the array than previously inserted into it. We note that a normal output must carry a tag of 01, implying that the data item has been placed in sorted order. Thus, for an extracted data item, a control tag 0x indicates normal output and 1x indicates an underflow condition.

Fig. 8 shows the complete interpretation of the two-bit tags and their corresponding cell actions for ease of reference.

4 Design Trade-Offs and Optimality

The preceding design can be generalized to allow more than two parallel comparisons in each cell. In the generalized version, each of the $\lceil n/k \rceil$ cells contains k comparators and crossbars, $k + 1$ registers, and $2k$ multiplexers. Fig. 9 shows an example with $k = 3$ comparisons per cell.

Thus, the total component count is reduced as we increase k, leading to more compact designs. On the other hand, segmenting the design into a smaller number of more complex cells leads to longer wires and propagation delays. In the extreme case of $k = n$, all n comparisons are performed in parallel; inputs are broadcast to n comparators and crossbars and the control signals to $2n$ multiplexers, leading to a shift-register-like design [7]. An inserted data

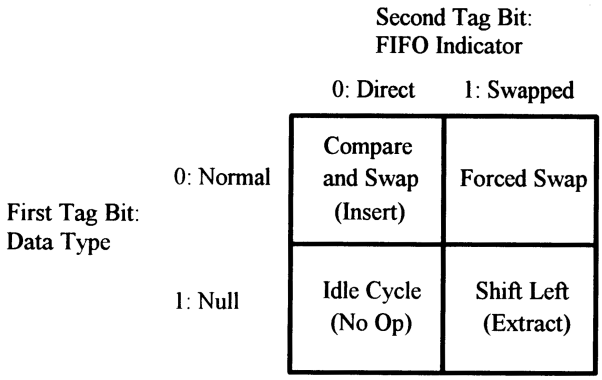


Fig. 8. Cell instruction encoded in two control tag bits.

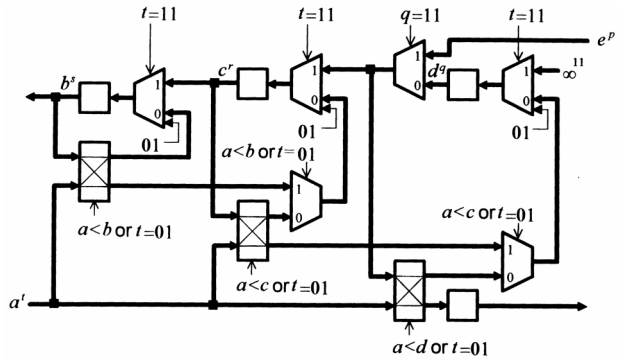


Fig. 9. The structure of a cell that performs three comparisons in parallel.

item is simultaneously compared to all the stored values, shifting all larger data items to the right. Extraction simply shifts all the data items to the left.

To estimate the propagation delay for the various designs, we adopt a simple and well-known model [10]. We assume the use of square areas for the layout of cells and the entire array, making the longest wire length proportional to $\sqrt{k}$. The propagation delay Δ through a VLSI metal wire [3] is given by the equation

$$\Delta = c_1 + c_2 \ln k + c_3 k$$

with c_1, c_2, and c_3 being technology-dependent constants [10]. We have derived these constants for the following analysis using a 0.8 μm CMOS double-metal technology with the second layer of metal for interconnections.

Fig. 10 shows the decrease in layout area and increase in clock period of the broadcast design with $k = n$ compared to the fully pipelined design with $k = 2$. The clock paths are assumed to be evenly distributed on the chip; thus, the effect of clock skew is not accounted for in deriving the cost and delay estimates.

With smaller cells, the wires running across cells become shorter and the linear array can operate at a higher clock frequency. The fully pipelined design uses $\lceil n/2 \rceil - 1$ more registers than the broadcast design. These registers act as repeaters, limiting the propagation delay of data and control signals and allowing a higher operating frequency. The area overhead due to the replication of registers is further discussed below.

Fig. 10 indicates that as the sorter size n increases, the cost reduction allowed by the broadcast design approaches

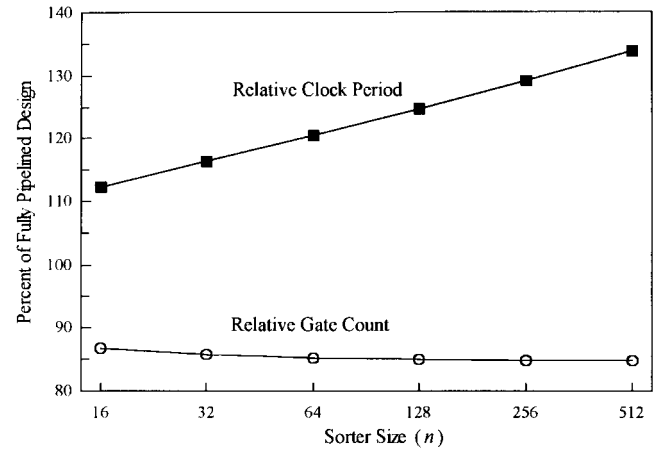


Fig. 10. The percent increase in clock period and percent decrease in area of the broadcast design ($k = n$) with respect to the fully pipelined design ($k = 2$).

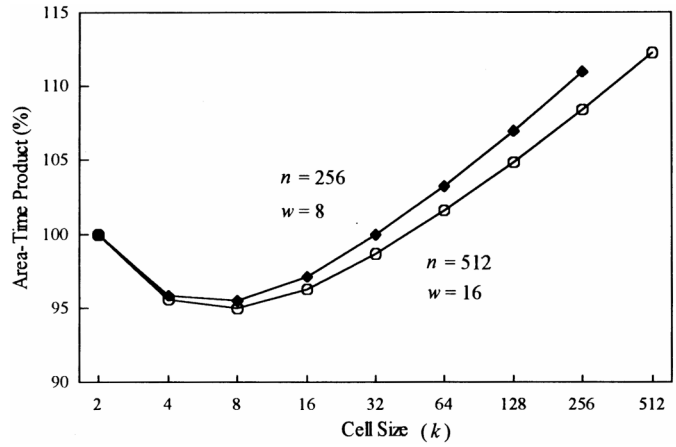


Fig. 11. The cost-effectiveness of designs with different cell sizes relative to the fully pipelined design ($k = 2$).

a constant factor. This result can be justified by an approximate analysis of VLSI layout area. for $(2w + 2)$-bit inputs (w bits for key, w bits for data, and 2 bits for tag), a comparator/crossbar switch is about four times as large, and a register is about three times as large as a multiplexer. The ratio of the layout areas between the two designs, when the sorter size n is large, becomes:

$$\frac{4n + 3(n+1) + 2n}{4n + 3(3n/2) + 2n} \approx 85.7\%.$$

The cost ratio depicted in Fig. 10 is slightly smaller than this value because the tag decoding logic also has to be replicated in each cell.

Segmenting the design into larger cells can be seen as trading off speed for more compact VLSI layout. Since all the designs achieve the total latency of $2n$ clock cycles for sorting n records, performance is determined solely by the operating frequency. We can use the area-time product to measure the cost-effectiveness of designs as we vary the segmentation parameter k.

We want to know the value of k that leads to the most cost-effective design. Fig. 11 shows that the design with eight comparators per cell yields the lowest area-time product. We

note that the optimum also depends on several other factors, including clock skew, and device characteristics [1], which are not considered in this paper. For example, repeating the analysis for 0.5 μm, CMOS shows that the optimum moves to a smaller cell size.

5 Conclusion

We have proposed a data-driven control scheme to solve the control flow problems in a stable insertion sorter. In our sorters, the control signals are pipelined together with the data signals in the form of tags attached to the data values. Our designs use only local intercell communications, thus limiting the global connections to clock and power supply. Such fully pipelined designs support higher operating speeds, are truly scalable, and lead to cost-effective VLSI implementations. Similar pipelined data-driven designs can be applied to a wide array of other computational problems.

Given that no data value or control information is broadcast to the cells, it is possible to convert our designs to ones with asynchronous control, thus eliminating the need for clock distribution. This would ease the area overhead of clock distribution and the speed penalty imposed by clock skew. As feature size in VLSI circuits continues to shrink, asynchronous designs will become more attractive not just from the standpoint of speed, but also with regard to power consumption.

We showed that two tag bits are sufficient for correct operation (including FIFO ordering and overflow/underflow detection), while providing high operating speed and unlimited scalability via the elimination of broadcasting and long wires. FIFO ordering is ensured by proper manipulation of control tags. Exception handling is accomplished by examining tags, rather than by the more complex process of counting the number of data items inserted into, or extracted from, the sorter.

Each cell in the sorter thus obtained compares its input with two stored data items and operates in a single cycle for insert or extract operation. As far as we know, our design is the first single-cycle priority queue that operates in fully pipelined mode with no broadcasting. Our design was shown to be generalizable to the case where more than two comparisons are performed in each cell. Increasing the degree of parallelism (reducing the pipeline depth) leads to more compact designs at the expense of longer propagation delays. Our study of cost/performance trade-offs has revealed the existence of an optimal cell size that leads to the most cost-effective design.

References

[1] M. Afghahi and C. Svensson, "Performance of Synchronous and Asynchronous Schemes for VLSI Systems," *IEEE Trans. Computers*, vol. 41, no. 7, pp. 858-872, July 1992.

[2] B. Ahn and J.M. Murray, "A Pipelined, Expandable VLSI Sorting Engine Implemented in CMOS Technology," *Proc. IEEE Int'l Conf. Circuits and Systems*, pp. 134-137, Portland, Ore., May 1989.

[3] D. Audet, Y. Savaria, and N. Arel, "Pipelined Communications in Large VLSI/ULSI Systems," *IEEE Trans. VLSI Systems*, vol. 2, pp. 1-10, Mar. 1994.

[4] T.C. Chen, V.Y. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files," *Proc. Fourth Int'l Conf. Very Large Databases*, pp. 312-318, 1978.

[5] S. Karthik, I. de Souza, J.T. Rahmeh, and J.A. Abraham, "Interlocking Scheme for Micropipelines: Application to a Self-Timed Rebound Sorter," *Proc. IEEE Int'l Conf. Computer Design*, pp. 393-396, Cambridge, Mass., Oct. 1991.

[6] S.Y. Kung, *VLSI Array Processors*. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[7] C.-Y. Lee and J.-M. Tsai, "A Shift Register Architecture for High-Speed Data Sorting," *J. VLSI Signal Processing*, vol. 11, pp. 273-280, Nov. 1995.

[8] D.T. Lee, H. Chang, and C.K. Wong, "An On-Chip Compare/Steer Bubble Sorter," *IEEE Trans. Computers*, vol. 30, no. 6, pp. 396-405, June 1981.

[9] Y.-C. Lin, "On Balancing Sorting on a Linear Array," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 5, pp. 566-571, May 1993.

[10] C. Mead and M. Rem, "Minimizing Propagation Delays in VLSI," *IEEE J. Solid-State Circuits*, vol. 17, pp. 773-775, Aug. 1982.

[11] J.D. Meindl, "Gigascale Integration: Is the Sky the Limit?" *IEEE Circuits and Devices*, vol. 12, pp. 19-24, 32, Nov. 1996.

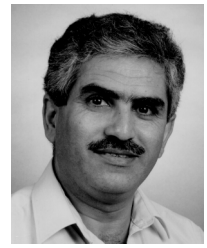
[12] S.W. Moore and B.T. Graham, "Tagged Up/Down Sorter—A Hardware Priority Queue," *The Computer J.*, vol. 38, pp. 695-703, Sept. 1995.

[13] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures*. Plenum Publishing Corp., to appear, 1999.

[14] D. Picker and R.D. Fellman, "A VLSI Priority Packet Queue with Inheritance and Overwrite," *IEEE Trans. VLSI Systems*, vol. 3, pp. 245-253, June 1995.

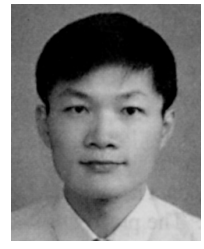
[15] H. Yasuura, N. Takagi, and S. Yajima, "The Parallel Enumeration Sorting Scheme for VLSI," *IEEE Trans. Computers*, vol. 31, no. 12, pp. 1,192-1,201, Dec. 1982.

[16] Y. Zhang and S.Q. Zheng, "Design and Analysis of a Systolic Sorting Architecture," *Proc. Seventh IEEE Symp. Parallel and Distributed Processing*, pp. 652-659, Oct. 1995.



Behrooz Parhami received his PhD in computer science from the University of California, Los Angeles, in 1973. Presently, he is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. His research deals with parallel architectures and algorithms, computer arithmetic, and reliable computing. In his previous position with Sharif University of Technology in Tehran, Iran (1974-1988), he was also involved in the areas of educational planning, curriculum development, standardization efforts, technology transfer, and various editorial responsibilities, including a five year term as editor of *Computer Report*, a Farsi-language computing periodical. His technical publications include more than 170 papers in journals and international conferences, a Farsi-language textbook, and an English/Farsi glossary of computing terms. Two textbooks on computer arithmetic (Oxford) and parallel processing (Plenum) are forthcoming in 1999.

Dr. Parhami is a fellow of both the IEEE and the British Computer Society, a member of the ACM, and a distinguished member of the Informatics Society of Iran, for which he served as a founding member and president during 1979-1984. He also served as chairman of the IEEE Iran Section (1977-1986) and received the IEEE Centennial Medal in 1984.



Ding-Ming Kwai received the BS and MS degrees in Taiwan from the National Cheng Kung University, Tainan, and the National Chiao Tung University, Hsinchu, in 1987 and 1989, respectively, and the PhD degree from the University of California, Santa Barbara, in 1997. He was with the Chung Cheng Institute of Technology, Taoyuan, Taiwan, as a reserve officer during 1989-1991 and with the Hualon Microelectronics Corporation, Hsinchu, Taiwan, as a design engineer during 1991-1993. Dr. Kwai is currently with Worldwide Semiconductor Corporation, Hsinchu, Taiwan. His research interests include parallel processing, VLSI architectures, and fault-tolerant computing.