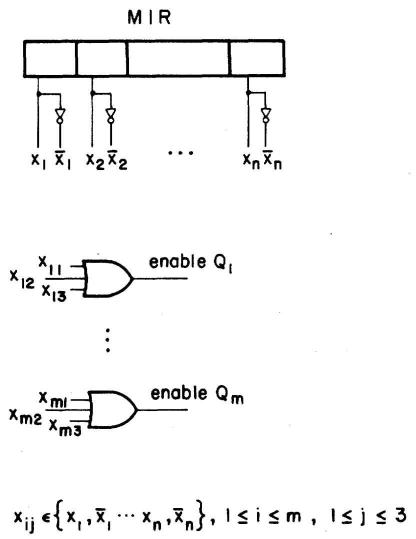
MIR
$x_1\ \bar{x}_1\ x_2\bar{x}_2\ \cdots\ x_n\bar{x}_n$
$x_{11}$
$x_{12}$
$x_{13}$
enable $Q_1$
$x_{m1}$
$x_{m2}$
$x_{m3}$
enable $Q_m$
$x_{ij} \in \{x_1, \bar{x}_1 \cdots x_n, \bar{x}_n\},\ 1 \le i \le m,\ 1 \le j \le 3$

Fig. 1. Generic machine built by construction algorithm.

that $BE$ is satisfiable if and only if there exists an encoding that enables $\{Q_1, \cdots, Q_m\}$.

*Proof:* Assume that $BE$ is satisfiable, and let $y_1, \cdots, y_n$ be a satisfying truth-value assignment to the variables $x_1, \cdots, x_n$ of $BE$. Then assign $y_i$ to MIR[$i$] ($1 \leqslant i \leqslant n$).

The converse argument is analogous. □

## VI. Summary

We have shown that the problem of deciding whether an arbitrary set of MO's can be encoded in a single MI is NP-complete. The proof relies on the use of an MIR encoding scheme closely related to *indirect encoded control* or *bit steering,* in which multiple MIR fields enable MO's. The bit steering technique is used in the RCA Spectra 70, Honeywell H1700, and IBM 360 computers [2].

It should be noted that in the machine built by the construction algorithm, the execution of each MO was determined by the values of up to *three* single-bit MIR fields. The general problem of satisfiability is solvable in polynomial time when the number of variables in each clause is at most 2 [7]. Thus, it is easy to see that for *direct encoded control,* where MO's are enabled by a single field, or bit steering in which each MO is enabled by at most two fields, the problem can be solved efficiently. This provides some direction for the design of control word structures, a problem addressed, for example, in [6].

## Acknowledgment

The authors wish to thank the anonymous referees for their helpful comments which improved this correspondence and S. R. Vegdahl for simplifying the proof.

## References

[1] A. K. Agrawala and T. G. Rausher, *Foundations of Microprogramming: Architecture, Software, and Applications.* New York: Academic, 1976.
[2] M. Andrews, *Principles of Firmware Engineering in Microprogram Control.* Potomac, MD: Computer Science Press, 1980.
[3] U. Banerjee, S. Shen, D. J. Kuck, and R. A. Towle, "Time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Comput.,* vol. C-28, Sept. 1979.
[4] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd ACM Symp. Theory of Computing,* New York, NY, 1971, pp. 151–158.
[5] S. Dasgupta, "The organization of microprogram stores," *Computing Surveys,* vol. 11, pp. 39–65, Mar. 1979.
[6] D. J. DeWitt, "A machine-independent approach to the production of optimal horizontal microcode," Ph.D. dissertation, Dep. Comput. and Commun. Sci., Univ. Michigan, Ann Arbor, 1976.
[7] S. Even, A. Itai, and A. Shamir, "On the complexity of timetable and multicommodity flow problems," *SIAM J. Comput.,* vol. 5, pp. 691–703, 1976.
[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.
[9] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *ACM Comput. Surveys,* vol. 12, pp. 261–294, Sept. 1980.
[10] P. W. Mallett, "Methods of compacting microprograms," Ph. D. dissertation, Dep. Comput. Sci., Univ. Southwestern Louisiana, Lafayette, Dec. 1978.
[11] E. L. Robertson, "Microcode bit optimization is NP-complete," *IEEE Trans. Comput.,* vol. C-28, pp. 316–319, Apr. 1979.
[12] M. Sint, "MIDL—A microinstruction description language," in *Proc. 14th Microprogramming Workshop,* Chatham, MA, Oct. 1981, pp. 95–108.
[13] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *Proc. 15th Microprogramming Workshop,* Palo Alto, CA, Oct. 1982, pp. 125–133.
[14] ——, "Local code generation and compaction in optimizing microcode compilers," Ph. D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1982.

# Routing Algorithms for Cellular Interconnection Arrays

A. YAVUZ ORUC AND DEEPAK PRAKASH

***Abstract*—The paper describes an algebraic model which provides a means for realizing an arbitrary permutation through various cellular-array-type networks. The model views a cellular array as an ordered set of transposition maps where each transposition corresponds to a permutation cell of the array. A permutation realizable by such an array is then expressed as a composition of the transpositions where the rules for the composition are determined by the topology of the array.**

***Index Terms*—Cellular interconnection arrays, cycle, interconnection networks, monotone increasing factorization, permutation, transposition.**

## I. Introduction

This paper focuses on a class of interconnection networks referred to as cellular interconnection arrays [1], [2]. Typically, a cellular interconnection array is a geometric pattern of interconnected switching cells with identical switching capabilities. Kautz *et al.* utilized the elementary permutation cell shown in Fig. 1 to build a number of different cellular arrays [1]. These include triangular, diamond, rectangular, pruned rectangular, rhomboidal, square, and almost square arrays. It is known that these interconnection arrays are all rearrangeable, that is, each one can realize an arbitrary permutation of its inputs onto its outputs.

Almost all cellular interconnection arrays of $n$ inputs have $O(n^2)$ cost complexity, which compares infavorably to $O(n \log n)$ of multistage networks such as Baseline [3], Omega [4], and Indirect Binary Cube networks [5]. Nevertheless, the regularity of the structure of cellular interconnection arrays allows for a modular interconnection network synthesis and makes them attractive for VLSI implementation. In addition, cellular arrays are all one-pass networks, while the best known bound for baseline-type networks is two passes [6].

Manuscript received July 7, 1983; revised January 16, 1984.

The authors are with the Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12181.

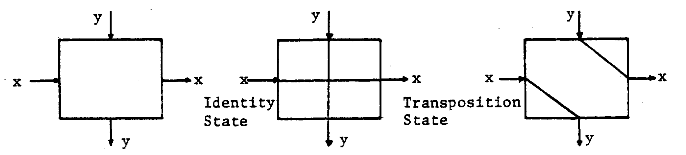

Fig. 1. The permutation cell.

An important task associated with cellular interconnection arrays is developing routing algorithms to realize arbitrary permutations by such arrays. This problem was addressed by Kautz *et al.* [1]. However, no formal algorithm has been provided. Here we present algorithms based on an algebraic representation of such arrays. A possible approach is to make use of the topological equivalence which exists between cellular permutation arrays and primitive sorting networks [7]. However, a sorting network requires more complex cells; each cell behaves as a comparator and an exchange switch. Moreover, a sorting network has a fixed interconnection structure which allows permuting only via data tagging. The algorithms provided here do not require these conditions.

The paper is organized as follows. In Section II, basic notation and algebraic preliminaries are presented. In Section III, we provide an algebraic modeling for triangular arrays, and using this model, we develop an algorithm to realize arbitrary permutations by such arrays. In Section IV, we consider diamond arrays and provide a routing algorithm for such arrays. In Section V, we discuss the modeling of other cellular arrays and the ways of setting them up to realize arbitrary permutations.

## II. Algebraic Preliminaries

In the following sections, we shall present a correspondence between a set of permutations and a cellular interconnection array. This section describes the notation and other algebraic facts to establish this correspondence.

Let $S$ be a set of $r$ symbols. A permutation $p$ on a set $S$ is a one-to-one mapping of $S$ onto itself. We shall write $(x)p = y$ to mean that the permutation $p$ moves $x$ to $y$ where $x, y \in S$. For convenience, the elements of $S$ are identified as integers $1, 2, \cdots, r$. The permutation $e$ defined by $(x)e = x$ for all $x \in S$ is called the *identity permutation*.

A common notation, called the cycle notation, will be used to represent permutations. A finite cycle $c = (x_1x_2 \cdots x_k)$ where $x_i \in S$, $i = 1, 2 \cdots k$ on $S$ is a permutation such that $(x_1)c = x_2, (x_2)c = x_3, \cdots, (x_k)c = x_1$. A cycle of $k$ symbols is called a *k-cycle*, and in particular, a cycle of two symbols is called a *transposition*. Two cycles are said to be *disjoint* if they do not have any elements in common. It is known that every permutation can be expressed as a product of disjoint cycles, and this product is unique up to the order of its factors.

The *composition* of two permutations $p, q$ on a set $S$, denoted $pq$, is also a permutation on $S$ defined by $(x)pq = ((x)p)q$ for each $x \in S$. The following facts for composing cycles follow directly from the definition of a composition map.

*Rule 1:* $(x_1x_2 \cdots x_{i-1}x_ix_{i+1} \cdots x_k) = (x_1x_2 \cdots x_{i-1}x_{i+1} \cdots x_k) \cdot (x_ix_{i+1})$.

*Rule 2:* $(x_1x_2 \cdots x_{i-1}x_ix_{i+1} \cdots x_j \cdots x_k) = (x_jx_{j+1} \cdots x_kx_1x_2 \cdots x_{i-1})(x_ix_{i+1} \cdots x_{j-1})(x_ix_j)$ where $j > i + 1$.

In what follows, we shall use these rules to realize arbitrary permutations by cellular arrays.

## III. Triangular Arrays

A cellular triangular permutation array of size $n$ (shown in Fig. 2 for $n = 8$) is a triangular array of $n(n - 1)/2$ interconnected permutation cells. A typical cell depicted in Fig. 1 is capable of mapping its inputs $x, y$ onto its outputs $x, y$ according to either of the two permutations indicated by the figure. It can be easily verified that a

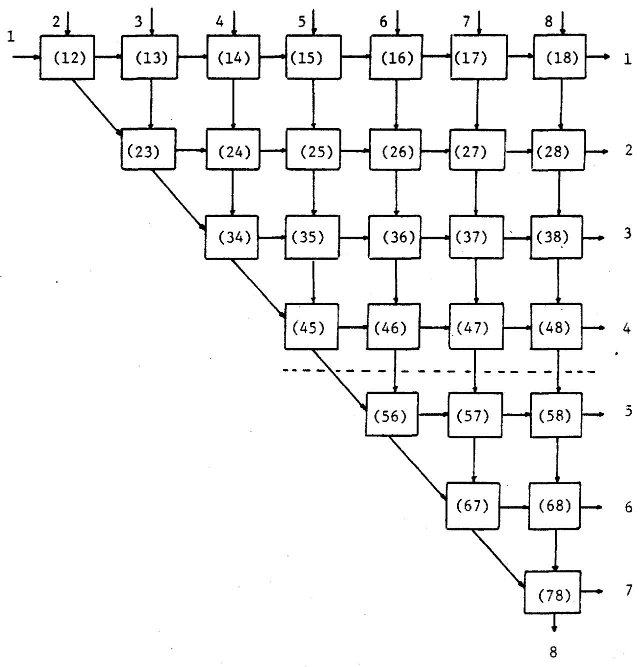

Fig. 2. Triangular array, $n = 8$.

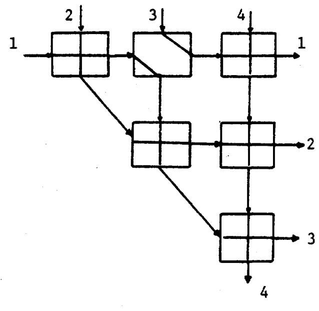

Fig. 3. A triangular array realizing (13).

triangular array such as the one shown in Fig. 3 realizes the transposition with which a permutation cell is associated if all the cells but that cell are set to the identity permutation. Thus, the triangular array of Fig. 3 realizes the transposition (1 3).

Given the above correspondence between a triangular array and a set of transpositions, we can develop an algorithm to realize arbitrary permutations. First we state the following definitions.

*Definition 1:* The transposition $(x\,y)$ associated with the cell $(x\,y)$ of a triangular permutation array where $x < y$ is said to precede the transposition $(w\,z)$ associated with the cell $(w\,z)$ of the same network where $w < z$ and is denoted as $(x\,y) \leq (w\,z)$ if either: a) $\{x, y\} \cap \{w, z\} = \emptyset$, or b) $y < z$, or c) $y = z$ and $x < w$.

For example, in Fig. 2, $(1\ 3) \leq (1\ 6)$ since $3 < 6$. Also, $(2\ 4) \leq (3\ 4)$ since $2 < 3$.

*Definition 2:* A product of transpositions $(x_1\ y_1)(x_2\ y_2) \cdots (x_r\ y_r)$ is said to be *monotone increasing* if $(x_i\ y_i) \leq (x_{i+1}\ y_{i+1})$ for all $i;\ 1 \leq i \leq r - 1$.

Clearly, a permutation of the inputs of a triangular permutation array is realizable by the array iff it can be factored into a monotone increasing product of transpositions. Moreover, since every triangular permutation array is rearrangeable, every permutation, and hence every cycle of the array, must have at least one factorization into a monotone increasing product of transpositions. Consider the network of Fig. 2 and the cycle (1 4 5 7 2). It can be factored as follows.

$$\begin{aligned}(1\ 4\ 5\ 7\ 2) &= (1\ 4\ 5\ 2)(2\ 7) && \text{Rule 1}\\ &= (1\ 4\ 2)(2\ 5)(2\ 7) && \text{Rule 1}\\ &= (1\ 2)(2\ 4)(2\ 5)(2\ 7) && \text{Rule 1.}\end{aligned}$$

It is easily seen that (1 2) (2 4) (2 5) (2 7) is a monotone increasing sequence ((1 2) $\leq$ (2 4) $\leq$ (2 5) $\leq$ (2 7)), and thus is realizable by the array of Fig. 2. The steps carried out in this example can be generalized into the following algorithm.

*Algorithm 1:* Let $S = \{x_1, x_2, \cdots, x_k\}$ and $c = (x_1x_2 \cdots x_k)$.

```
1) $p_1 := c, p_2 := e$.
2) Let $x_m = \max(S)$; that is, $x_m$ is the largest element in $S$.
3) Rewrite $p_1$ as $p_1 = (x_m x_{m+1} \cdots x_k x_1 x_2 \cdots x_{m-1})$.
4) Using Rule 1), let
       $p_1 = (x_{m+1} \cdots x_k x_1 x_2 \cdots x_{m-1})(x_{m+1} x_m)$
       $p_1 := (x_{m+1} \cdots x_k x_1 x_2 \cdots x_{m-1}); p_2 := (x_{m+1} x_m) \cdot p_2$.
5) $S = S - \{x_m\}$.
6) If $|S| > 2$, go to step 2).
7) $c := p_2$.
```

It is seen that the algorithm is based on the factorization of a cycle of $i$ symbols into a cycle of $i - 1$ symbols, and a transposition starting with $i = k$ and proceeding with $i = k - 1$, $i = k - 2, \cdots, i = 3$ with the provision that the factored transposition is defined over the largest symbol in the cycle and the one which is to the right of the largest symbol. This condition guarantees that the following two events occur simultaneously.

1) The algorithm always chooses the greatest transposition among all available ones.

2) The other cycle in the factorization no longer contains the largest symbol.

It then follows that the final factorization will be a monotone increasing product of transpositions. We conclude that given an arbitrary cycle, the algorithm always leads to a monotone increasing factorization of the cycle.

The algorithm described above can be used to realize an arbitrary permutation of $n$ symbols by a triangular array of size $n$. Recall that any permutation is a product of disjoint cycles, and the product is unique up to the order of its factors. Thus, if $p = c_1 c_2 \cdots c_k$ where $c_i$ are pairwise disjoint cycles, we use the algorithm for each cycle to obtain a monotone increasing factorization. Since the cycles do not have any element in common, the substitution of the factorizations into the product $c_1 c_2 \cdots c_2$ also yields a monotone increasing product. As an example, let $S = \{1, 2, \cdots 10\}$ and $p = c_1 c_2 c_3$ where

$$c_1 = (7\ 4\ 6), c_2 = (3\ 1\ 8\ 10), c_3 = (5\ 2\ 9).$$

Using the algorithm,

$$c_1 = (4\ 6)\ (4\ 7), c_2 = (1\ 3)\ (3\ 8)\ (3\ 10), c_3 = (2\ 5)\ (5\ 9).$$

Thus,

$$p = (4\ 6)\ (4\ 7)\ (1\ 3)\ (3\ 8)\ (3\ 10)\ (2\ 5)\ (5\ 9)$$

which is clearly a monotone increasing sequence of transpositions.

It is interesting to estimate the order of complexity of Algorithm 1. Let $p = c_1 c_2 \cdots c_k$ where $c_i$ is a cycle of $n_i$ symbols for $i = 1, 2 \cdots k$. It can be seen that the complexity of factorizing cycle $c_i$ is the same as the complexity of sorting a list of $n_i$ elements, which at best is $n_i \log_2 n_i$. Thus, factorizing $p$ into a monotone increasing sequence of transpositions requires a complexity of

$$\sum_{1 \leq i \leq k} n_i \log_2 n_i \leq n \log_2 n.$$

## IV. Diamond Arrays

Another type of cellular permutation array is a diamond array which is also capable of realizing all possible permutations of its inputs onto its outputs [1]. A diamond array of size eight is depicted in Fig. 4. The labels inside the cells are the transpositions with which they are associated, and the labels underneath each cell are the pairs of indexes where the first index in each pair designates the diagonal to which the cell belongs and the other index refers to the position of the cell in the diagonal. Observe that all possible transpositions of eight symbols are present on the array. However, unlike the case of a triangular array, it is not possible to order the transpositions of a diamond array with respect to its input symbols. Instead, we shall adopt an ordering which is based on the topology of the array.

*Definition 3:* The transposition $(x\ y)$ associated with the switching cell with index pair $(i, j)$ is said to precede the transposition $(w\ z)$ associated with the switching cell with index pair $(k, l)$, and is denoted as $(x\ y) \leq (w\ z)$ if either $\{x, y\} \cap \{w, z\} = \emptyset$ or $i \leq k$.

The definition provides the criterion for factorizing a permutation into a monotonic increasing product of transpositions. As an example, consider the realization of an arbitrary cycle $c$ by a diamond permutation array. Let $c = (1\ 5\ 7\ 2\ 4\ 8\ 3\ 6)$. It can be factorized as follows.

$$\begin{aligned}c &= (1\ 5\ 7\ 2\ 4\ 8\ 3\ 6)\\ &= (6\ 1\ 5)\ (7\ 2\ 4\ 8\ 3)\ (6\ 7) && \text{Rule 2}\\ &= (6\ 1\ 5)\ (3\ 7)\ (2\ 4\ 8)\ (2\ 3)\ (6\ 7) && \text{Rule 2}\\ &= (6\ 1\ 5)\ (3\ 7)\ (2\ 8)\ (4\ 8)\ (2\ 3)\ (6\ 7) && \text{Rule 1}\\ &= (5\ 6)\ (1\ 5)\ (3\ 7)\ (2\ 8)\ (4\ 8)\ (2\ 3)\ (6\ 7) && \text{Rule 1.}\end{aligned}$$

By inspection, the above factorization is monotone increasing, and hence $c$ is realizable by the diamond array shown in Fig. 4. The above example can be generalized into the following algorithm.

*Algorithm 2:* $c = (x_1 x_2 \cdots x_k), r := e$(identity permutation), stack = empty.

```
1) If $c$ is a transposition $r = c \cdot r$ and go to step 6).
2) Let $S$ be the set of all possible transpositions using the symbols
   that appear in the cycle $c$.
3) Let $t = (x_i x_j)$ where $i < j$ be the greatest transposition in
   the set $S$.
4) If $(i + 1 = j)$, then  $c = pt$ (by Rule 1)
                            so let $r := t \cdot r$ and $c := p$.
                     else  $c = qpt$ (by Rule 2).
                            so let $r := t \cdot r$ and $c := p$.
                            push $q$ onto stack
5) go to step 1).
6) If (stack is empty), then stop
                     else $c :=$ pop(stack) and go to step 1).
```

Algorithm 2 is based on the factorization of a cycle $c$ into either another cycle $p$ followed by a transposition $t$ or into two cycles $q$ and $p$ followed by $t$, with the provision that $t$ is the greatest among all possible transpositions of the symbols of $c$. Then $c$ is replaced by $p$ and $q$ is pushed into the stack if $c = qpt$. The algorithm halts after all the cycles pushed into the stack have been factorized into transpositions. Suppose that, after some number of iterations, $c = q_1 q_2 \cdots q_u q p t_1 t_2 \cdots t_v$ and $t_1 t_2 \cdots t_v$ is a monotone increasing sequence of transpositions. The fact that $t_1$ is greater than any transposition of the symbols of $p$ or symbols of $q$ follows directly from step 3) of the algorithm. On the other hand, it is also true that $t_1$ is greater than any transposition of the symbols of $q_i; 1 \leq i \leq u$ since $t$ and $q_i$ are disjoint for all $i; 1 \leq i \leq u$. Clearly, this argument inductively leads to the assertion that Algorithm 2 always yields a monotone increasing sequence of transpositions and halts after that.

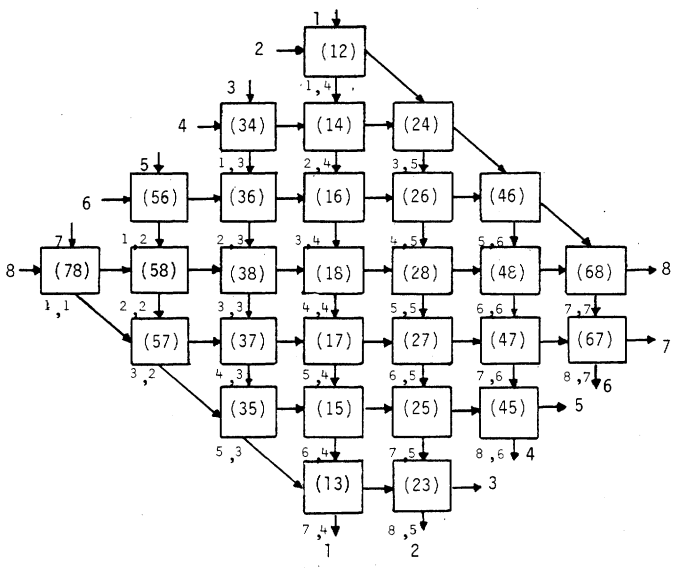
Fig. 4. Diamond array, $n = 8$.

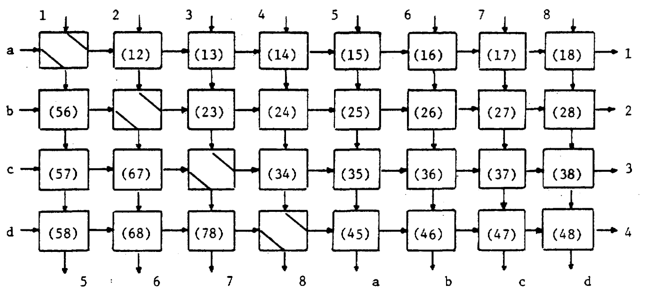
Fig. 5. Rectangular array, $n = 8$.

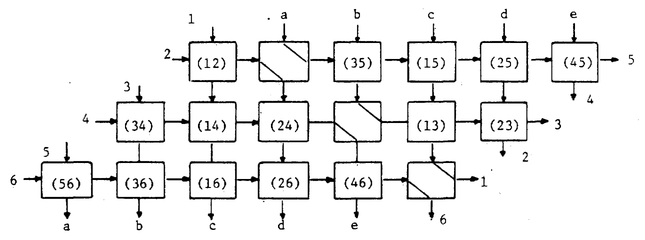
Fig. 6. Rhomboidal array, $n = 6$.

## V. Rectangular and Rhomboidal Arrays

The algorithms developed for triangular and diamond arrays can be used for rectangular and rhomboidal arrays. As Kautz *et al.* pointed out, a rectangular array of size $n$ is directly obtained from a triangular array of size $n$. As an example, consider the rectangular array of size eight shown in Fig. 5. It can be formed by first reflecting the triangle of cells below the dotted line in Fig. 2 about its hypotenuse, then placing this triangular subarray to the left of the remainder of the array, and inserting a diagonal of cells set to the identity permutation between them. It is seen from Figs. 2 and 5 that this transformation does not alter the ordering convention among the transpositions of the triangular array given in Definition 1. Thus, Algorithm 1 directly applies to rectangular arrays.

Kautz *et al.* also demonstrated that after transformation, a diamond array can be restructured as a rhomboidal array (Fig. 6). Although not detailed here, it can be shown that this transformation preserves the ordering in the sense of Definition 3 among the transpositions of the diamond array. Thus, Algorithm 2 can be used directly to realize arbitrary permutation on rhomboidal arrays.

## VI. Conclusions

The paper has introduced an algebraic representation for cellular permutation arrays. This representation has been shown to be a valuable tool for developing algorithms to realize arbitrary permutations by such interconnection arrays.

## References

[1] W. H. Kautz *et al.*, "Cellular interconnection arrays," *IEEE Trans. Comput.*, vol. C-17, pp. 443–451, May 1968.

[2] J. Gecsei, "Interconnection networks from three-state cells," *IEEE Trans. Comput.*, vol. C-26, pp. 705–711, Aug. 1977.

[3] C. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694–702, Aug. 1980.

[4] D. K. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145–1155, Dec. 1975.

[5] M. C. Pease, "The indirect binary $n$-cube multiprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 443–473, May 1976.

[6] C. Wu and T. Feng, "The reverse exchange network," *IEEE Trans. Comput.*, vol. C-29, pp. 801–810, Sept. 1980.

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