Computing on an Iterative Mesh with One-Way Data Flow

Anwer Z. Kotob

Carla D. Savage

Center for Communications and Signal Processing
Department of Computer Science
North Carolina State University
Raleigh, NC 27695

CCSP-TR-88/10

January 1988

Abstract

In this paper, we describe a generic procedure to transform any algorithm for a cellular mesh with two-way data flow between adjacent cells into an algorithm for an iterative mesh model with only one-way data flow between adjacent cells. We discuss the practical implications of this result for two-dimensional signal processing and the relationship to some open theoretical questions.

Index Terms

mappings between array models of computation, mesh of processors, one-way data flow, real-time architectures, systolic arrays

1. Introduction:

In this paper, we describe a generic procedure to transform any algorithm for a cellular mesh with two-way data flow between adjacent cells into an algorithm for an iterative mesh model with only one-way data flow between adjacent cells [Figure 1]. Precisely, we show how a two-way $m \times n$ cellular mesh, computing for t time units, can be simulated by a one-way iterative mesh with t columns of $m+1$ rows each: input columns enter the iterative mesh in successive time units and, after a delay of $3t$ time units, output columns leave the mesh in successive time units [Figure 2]. Further, successive problem instances can be pipelined, allowing the possibility of processing a sequence of input arrays (e.g. images) at transmission rates. We discuss the practical implications of this result for two-dimensional signal processing and the relationship to some open theoretical questions.

We use the term *mesh* to refer to a two-dimensional array of processing elements, each connected to its four nearest neighbors. Operation of the mesh is described in terms of a *cell program* which defines the state of a cell at time t as a function of the states of the cell and its neighbors at the previous time unit [Figure 3]. If the mesh has $two-way\ data\ flow$, the neighbors of a cell are those cells above, below, left, and right [Figure 3(a)]. In a mesh with $one-way\ data\ flow$, a cell has only the cells above and to the left as neighbors [Figure 3(b)]. We assume that input and output for a mesh computation each

consist of an $m \times n$ array of values. In this paper, we call a computation on an $m \times n$ mesh *cellular* if for $1 \leq i \leq m$, $1 \leq j \leq n$, cell (i,j) of the array stores input item (i,j) at the beginning of the computation and holds output item (i,j) at the end of the computation [Figure 1(a)]. During the computation, there is no I/O. We will call a mesh computation *iterative* if the input enters the mesh, one column per time unit, at the leftmost column of the mesh and the output leaves the mesh one column per time unit, at the rightmost column of the mesh [Figure 1(b)].

In the case of linear arrays, several researchers have made the observation that a two-way linear cellular array, computing for t time units, can be simulated by a one-way linear iterative array with $O(n+t)$ cells [2, 6]. For the case of the two-dimensional mesh, the only related work of which we are aware is that of [2] which simulates a two-way cellular mesh by a one-way circular *cellular* mesh. Simulation by a one-way *iterative* array uses a different technique and appears to have different applications.

In Section 2, a description is given of the simulation of a two-way cellular mesh by a one-way iterative mesh. Applications and observations are discussed in Section 3.

2. The Transformation:

In this section we give an informal description of the simulation of a two-way cellular mesh, P, by a one-way iterative mesh, Q. For convenience in describing the procedure, we assume that P is surrounded by a border of constant values, as shown in Figure 7(a), so that the cell program, f, of P is a function of the four neighbors of cell (i,j) even when i=1,m or j=1,n. (For an $m \times n$ array, P, this will require that each column of Q use $m+3$ cells rather than $m+1$ as stated in Figure 2.)

Columns of the initial states of P enter iterative array Q, one column per time unit. Let $P_t(i,j)$ denote the state of cell (i,j) of P at time t. In the simulation, a cell of Q waits until it has enough information to compute $P_t(i,j)$ for some i, j, and t. Enough information is saved by the cell so that with its new input at the next time unit, $P_t(i,j+1)$ can be computed.

Precisely, we can show that *all* computations performed in array P at time t are performed in column t of Q by cells (t+2, t) ... (t+n+1, t). Further, column t of Q computes the j^{th} column of P at time $3t+j$. If array P computes for a total of t^* time units, it follows that the final states of P, P_{t^*}, leave column t^* of Q in successive time units $3t^*+1$ through $3t^*+n$, one column of P_{t^*} per time unit.

In Figure 4 we show the cell program, g, for the one-way iterative array, Q, simulating the two-way cellular array P with cell program f. Each cell of

Q holds four values C, L, R, and D, not including the output value it computes. Figure 5 illustrates the flow of data in Q described by the cell program g of Figure 4. We omit the details of the formal proof that array Q with cell program g correctly simulates array P with cell program f which verifies that all boundary cases do work correctly.

Figure 6 illustrates how and when a neighborhood of cell (i,j) in array P at time t will show up in array Q. Figure 7 shows the one-way iterative array simulation of a 3×3 two-way cellular array, P, computing for 3 time units. In this figure we assume A_i, B_i, C_i, ..., G_i, H_i, I_i represent the contents of P at time i for $0 \leq i \leq 3$.

3. Applications and Observations:

We begin this section with the disclaimer that the transformation of Section 2 in no sense guarantees optimal one-way solutions on the iterative mesh. With knowledge of the problem at hand, data stored in the simulating array could be substantially pruned. Or, knowing the problem at hand and knowing that a one-way iterative array solution exists, a designer may start from scratch to devise a clever solution, entirely unrelated to the transformation.

On the other hand, the transformation described in Section 2 proves that any problem whose solution can be *described* as a cellular mesh computation can be solved on an iterative mesh with one-way data flow and further that successive problem instances (separated by two columns of "null" values) can be pipelined. This suggests a natural approach to processing sequences of two-dimensional signals at transmission rates.

Image Processing:

Consider the application of processing 512×512 images transmitted at video rates: 30 frames per second. A local image operation such as *median smoothing* might be described by the following cell program for a cellular mesh P: "replace the value of cell (pixel) (i,j) by the median of its current value and those of its nearest neighbors". The cells of the mesh would execute their program a small number of times k, depending on the size of the neighborhood over which smoothing is to occur. However, rather than perform this computation on a 512×512 cellular mesh, the transformation described in Section 2 shows that the computation can be simulated on an iterative mesh Q of k columns, each with 513 cells, which is a more reasonable number of processors for small k.

3. Applications and Observations:

We begin this section with the disclaimer that the transformation of Section 2 in no sense guarantees optimal one-way solutions on the iterative mesh. With knowledge of the problem at hand, data stored in the simulating array could be substantially pruned. Or, knowing the problem at hand and knowing that a one-way iterative array solution exists, a designer may start from scratch to devise a clever solution, entirely unrelated to the transformation.

On the other hand, the transformation described in Section 2 proves that any problem whose solution can be *described* as a cellular mesh computation can be solved on an iterative mesh with one-way data flow and further that successive problem instances (separated by two columns of "null" values) can be pipelined. This suggests a natural approach to processing sequences of two-dimensional signals at transmission rates.

Image Processing:

Consider the application of processing 512 × 512 images transmitted at video rates: 30 frames per second. A local image operation such as *median smoothing* might be described by the following cell program for a cellular mesh P: "replace the value of cell (pixel) (i,j) by the median of its current value and those of its nearest neighbors". The cells of the mesh would execute their program a small number of times k, depending on the size of the neighborhood over which smoothing is to occur. However, rather than perform this

computation on a 512 × 512 cellular mesh, the transformation described in Section 2 shows that the computation can be simulated on an iterative mesh Q of k columns, each with 513 cells, which is a more reasonable number of processors for small k. Successive images can be pipelined through Q and in order to meet transmission rates, it must be possible to execute the cell program of Q within 1 sec./(30*512) ≈ 70 microseconds, which is very reasonable, especially if the program is implemented in hardware. For global operations, e.g. region labeling, we retain the advantage that an iterative array can process at transmission rates, still allowing a generous 70 microseconds to execute the cell program. However, global operations require time at least c*512 on the cellular mesh and therefore the iterative mesh simulating the cellular computation will require at least c*512 *columns*.

So, even global operations on images can be performed at transmission rates on an iterative mesh *if* enough cells are available.

Recirculating Output:

Assume that P is an $m \times n$ two-way cellular mesh with cell program f. Let g be the cell program for the one-way iterative mesh simulation of P which is described in Section 2. Let R be a one-way iterative mesh with cell program g and with k columns, each with $m+1$ cells. The output of R can be recirculated to follow the input of R and flow through R again for further processing. Recirculating the data $c-1$ times through R has the same effect as passing the

input only one time through an iterative array R' which has the same cell program as R, but which has ck columns of $m+1$ cells each. This in turn gives the same result as the array P executing cell program f for ck time units. As a result, then, a computation on a two-way cellular array, P, which requires t^* time units can be simulated by a one-way iterative array R with $k < t$ columns, of $m+1$ cells each, by recirculating the output of R back through R for $\lceil (t^*/k) \rceil - 1$ times *as long as the following condition is met:* For all i,j: $i \leq i \leq m, 1 \leq j \leq n$, $P_{t^*}(i,j) = P_{k\lceil (t^*/k) \rceil}(i,j)$. If k divides t, this will certainly be true. Otherwise, the condition says that if P computes for longer than necessary, it still gets the same result.

Thus, whenever the condition can be satisfied, solving a problem on a one-way iterative array with too few columns can be handled in a natural way by recirculating output, thereby avoiding the issue of problem decomposition.

Sorting:

It was first established by Thompson & Kung [14] that sorting $m*n$ numbers could be performed by an $m \times n$ two-way cellular mesh in time $O(m+n)$ using, for example, the snake ordering on the mesh. Although the algorithm of [14] was described recursively, the recursion can be "unwound" to uncover a cell program, f, which, when executed repeatedly, for $O(m+n)$ steps, will sort the array. Once the function f is obtained, the generic procedure of Section 2 can be applied to yield the cell program g for sorting on a

one-way iterative mesh. What we found most intriguing was the idea that there might be a more natural way to sort on a one-way iterative array (using at most $O(m+n)$ columns of $m+1$ cells each). This is related to the open question of whether or not it is possible to sort on an $m \times n$ cellular mesh in time $o(m*n)$ (i.e., time strictly smaller, asymptotically, than $m*n$) using only local operations, as we discuss below.

In the past few years several more algorithms have been discovered for sorting on an $m \times n$ cellular mesh in time $O(m+n)$ [12, 9]. With some work, each of these algorithms can be expressed as a cell program, f, which, when executed by each cell in the array for $O(m+n)$ steps, will sort the array. However, each of these algorithms uses a *global* strategy to sort the array. For example to implement a substep such as "sort all rows", the number of times a particular cell compares with its row neighbors depends on the size of the array. Note that in the cellular array there is no global control: all control is encapsulated within the cell state and the cell program. Thus, for example, the cell program for the Thompson and Kung sorting algorithm must tell the cell how to keep track of enough information so that the cell knows when it is to be shuffling or merging, swapping or comparing. In a local sorting algorithm, the neighbor or neighbors to which a cell compares and the action taken as a result of that comparison is independent of the size of the array. Even the *shearsort* [13] algorithm for sorting in time $O((n+m)\ log(n+m))$ is not local by our cri-

terion.

In [7] Kosaraju proposed a local sorting scheme which was shown in [8] to be at least $\Omega(n^2)$ for an $n \times n$ mesh. We proposed a local sorting scheme, *wavesort*, which we conjectured to be linear, but experimental evidence indicates that it is nonlinear, and probably quadratic. No local sorting algorithm which is faster than quadratic has been found. On the other hand no one has been able to show that local sorting requires more than linear time.

Aside from the theoretical interest, the existence of a linear time local sort would mean that if the entire cell program were hardwired, two $m \times n$ meshes could be combined to sort $2mn$ values without changing the cell circuitry.

Because of the two-way to one-way transformation of Section 2, the existence of a linear time local sort on a cellular mesh would imply the existence of a local sort on a one-way iterative mesh with a linear number of columns. Perhaps more useful is the converse observation that the nonexistence of a local sort for $m*n$ values on a one-way iterative array with $O(m+n)$ columns (of $m+1$ cells each) would imply the nonexistence of a linear time local sort on a cellular mesh. Since the one-way iterative array is a more restrictive model of computation, it may be easier to prove lower bounds for that model.

Arrays of Finite State Machines:

We have not required cells in the mesh to be finite state machines. However, if the cells of the two-way cellular mesh, P, *are* finite state machines, then so are the cells of the one-way iterative mesh Q which simulates P, as described in Section 2. Further, meshes P and Q can be interpreted as acceptors, where a mesh is said to accept its input if and only if the bottommost cell of the rightmost column ever enters an accepting state.

Linear array acceptors can be classified as either cellular (CA) or iterative (IA) according to whether the input is in the initial states of the cells or enters the first cell of the array, one value per time unit (we use the notation of [5]). In addition, linear array acceptors may have either two-way data flow (CA, IA) or one way data flow (OCA, OIA) between adjacent cells [5]. Linear arrays of length n work on input strings of length n and accept a string if and only if the rightmost cell ever enters an accepting state.

It is straightforward to show that CA's and IA's are both equivalent to linear space-bounded Turing Machines and thus are equivalent in computing power. Not so obvious is the equivalence between the OCA and OIA models, which was recently shown in [5]. The interesting open question is whether the OCA and CA are equivalent. It has been shown that resolving this question in the negative will be hard [1].

In two dimensions, the cellular and iterative models have been studied both for two-way data flow (CM) and for one-way data flow (OCM) [3, 4, 10, 11, 15]. It can be shown using techniques from [15] that the two-way mesh model is strictly more powerful than a one-way mesh *using the same number of cells.* (This is in contrast to the result of this paper which allows the number of columns of the one-way mesh to vary with t.) In the iterative case, the natural extension of IA and OIA to 2-dimensions would be the IM and OIM models in Figures 8a and b. In the case of the OIM, it might appear that, for the same number of cells, the iterative mesh model of Figure 9 would be more powerful, or at least more convenient, than the OIM of Figure 8b, but we conjecture that they are equivalent in computing power.

References

[1] J.H. Chang, O.H. Ibarra, and A. Vergis, "On the power of one-way communication", *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* Toronto, October 1986, 455-464.

[2] K. Culik, and S. Yu, "Translation of systolic algorithms between systems of different topology", *Proc. Int. Conf. on Parallel Processing,* August 1985, 756-763.

[3] C. Dyer, "One-way bounded cellular automata", *Information and Control 44,* 1980, 261-281.

[4] F.C. Hennie, *Iterative Arrays of Logical Circuits,* MIT Press, Cambridge, Mass. (1961).

[5] O.H. Ibarra, and T. Jiang, "On one-way cellular arrays", *SIAM J. Computing,* Vol. 16, No. 6, December 1987, 1135-1154.

[6] A.Z. Kotob, "Transforming computations with bi-directional data flow into ones with uni-directional data flow on linear systolic arrays", Master's thesis, North Carolina State University, 1987.

[7] S.R. Kosaraju, "On some open problems in the theory of cellular automata", *IEEE Trans. Computers, c-23,* 6, June 1974, 561-565.

[8] R.J. Lipton, R.E. Miller, and L. Snyder, "On an array sorting problem of Kosaraju", *Proc., Conf. on Information Sciences and Systems,* The Johns Hopkins University, 1977, 99-103.

[9] Y. Ma, S. Sen, and I.D. Scherson, "The distance bound for sorting on mesh connected processor arrays is tight", *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* Toronto, October 1986, 255-263.

[10] A. Rosenfeld, and D.L. Milgram, "Parallel/sequential array automata", *Information Processing Letters 2,* 1973, 43-46.

[11] C.D. Savage, "Computing majority on a mesh with one-way dataflow", *Information Processing Letters,* to appear.

[12] C.P. Schnorr, and A. Shamir, "An optimal sorting algorithm for mesh connected computers", *Proc. 8th Annual ACM Symp. on Theory of Computing,* California, May 1986, 255-263.

[13] I.D. Scherson, S. Sen, and A. Shamir, "Shear sort: a true two-dimensional sorting technique for VLSI networks", *Proc. Int. Conf. on Parallel Processing,* August 1986, 903-908.

[14] C. Thompson, and H.T. Kung, "Sorting on a mesh-connected parallel computer", *CACM,* 1977, 263-271.

[15] J. Chang, O. Ibarra, and M. Palis, "Efficient simulations of simple models" of parallel computation by space-bounded TM's and time-bounded alternating TM's, Revision of Tech. Rep. TR 85-47, Department of Computer Science, University of Minnesota, 1985.

Figure Captions

Figure 1. Mesh models of computation.
(a) Two-way cellular mesh.
(b) One-way iterative mesh

Figure 2. Main result.

Figure 3. Cell functions and data flow.
(a) Two-way data flow: cell function computes next state of center cell as $f(C,U,D,L)$.
(b) One-way data flow: cell function computes next state of center cell as $f(C,U,L)$.

Figure 4. Cell program, g, of one-way iterative array to simulate two-way cellular array with cell program f.

Figure 5. Flow of data in one-way simulation.

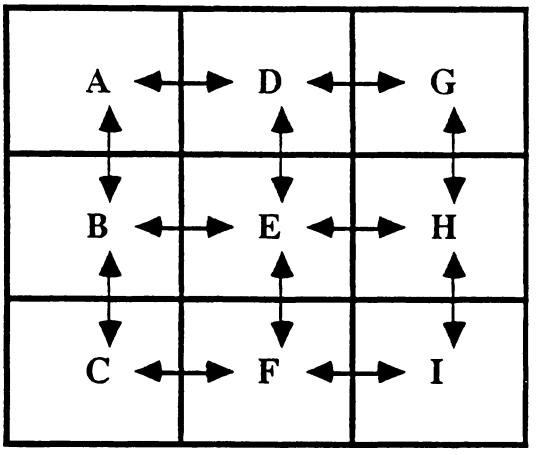
Figure 6. Simulation of two-way cellular array, P, by one-way iterative array, Q.
(a) Array P at time t.
(b) Array Q at time $3(t+1)+j-1$.

Figure 7. Simulation of two-way cellular P by one-way iterative Q.
(a) Array P at time $t=0$.
(b) Array Q at time 0.
(c) Array Q at time 1.
(d) Array Q at time 2.
(e) Array Q at time 3.
(f) Array Q at time 4.
(g) Array Q at time 5.
(h) Array Q at time 6.
(i) Array Q at time 7.
(j) Array Q at time 8.
(k) Array Q at time 9.
(l) Array Q at time 10.
(m) Array Q at time 11.
(n) Array Q at time 12.
(o) Array Q at time 13.

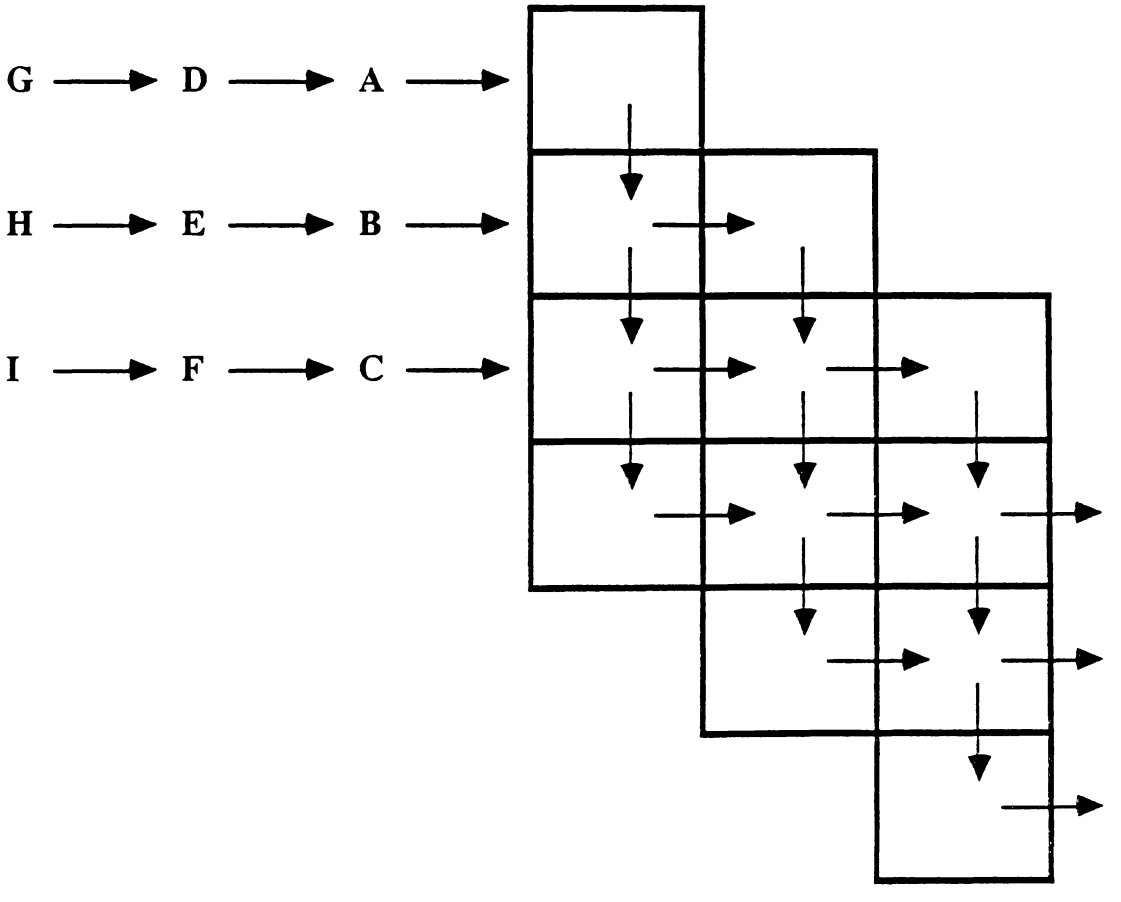
Figure 8. Iterative mesh models.
(a) Two-way.

(b) One-way.

Figure 9. Alternate one-way iterative mesh model.



(a) Two-way Cellular Mesh.



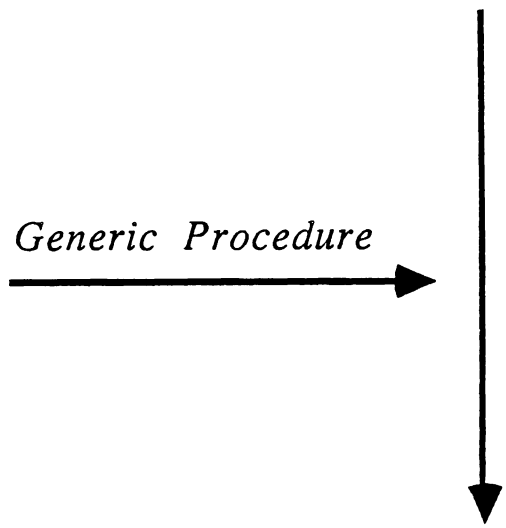
(b) One-way Iterative Mesh.

Figure 1. Mesh Models of Computation.

Two-way (Cellular) Mesh Algorithm

mesh size: m by n

time: t

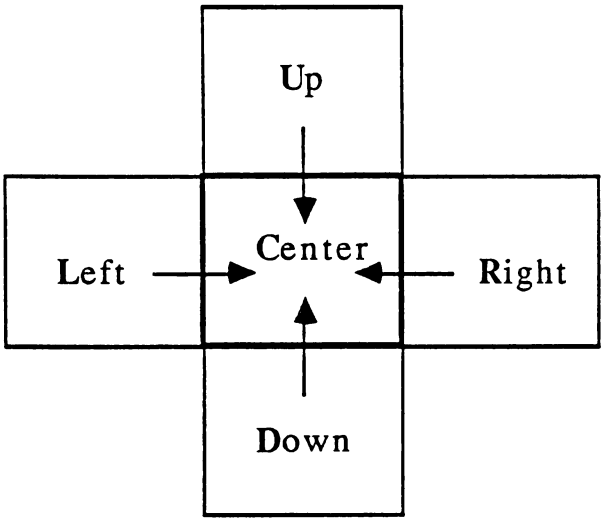


One-way (Iterative) Mesh Algorithm.

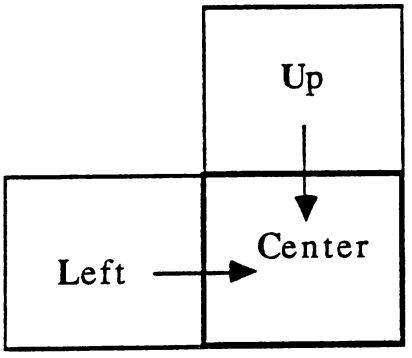
mesh size: t columns, (m+1) rows each

delay: 3t

Figure 2. Main Result.



(a) Two-way dataflow: Cell function computes next state of center cell as f(C, U, R, D, L).



(b) One-way dataflow: cell function computes next state of center cell as f(C, U, L).

Figure 3. Cell functions and Data Flow.

```
procedure g (Center, Up, Left);

with Center do
        if ON then
                begin
                if (C=XR) or (D=XR) then
                        ON := false
                        Output := C
                else if (C=XL) or (C=Null) or (C=XU) or (C=XD) then
                        Output := C
                else    Output := f (C, Up.C, Up.R, D, L)

                L := C
                D := R
                C := Up.R
                R := Left.Output
                end

        else
                if (Left.Output=XL) or (Up.R=XL) then
                        begin
                        ON := true
                        L := Null
                        D := Null
                        Output := Null
                        R := Left.Output
                        C := Up.R
                        end
```

Figure 4. Cell program, **g**, of One-way Iterative Array to Simulate Two-way Cellular Array with cell program f.

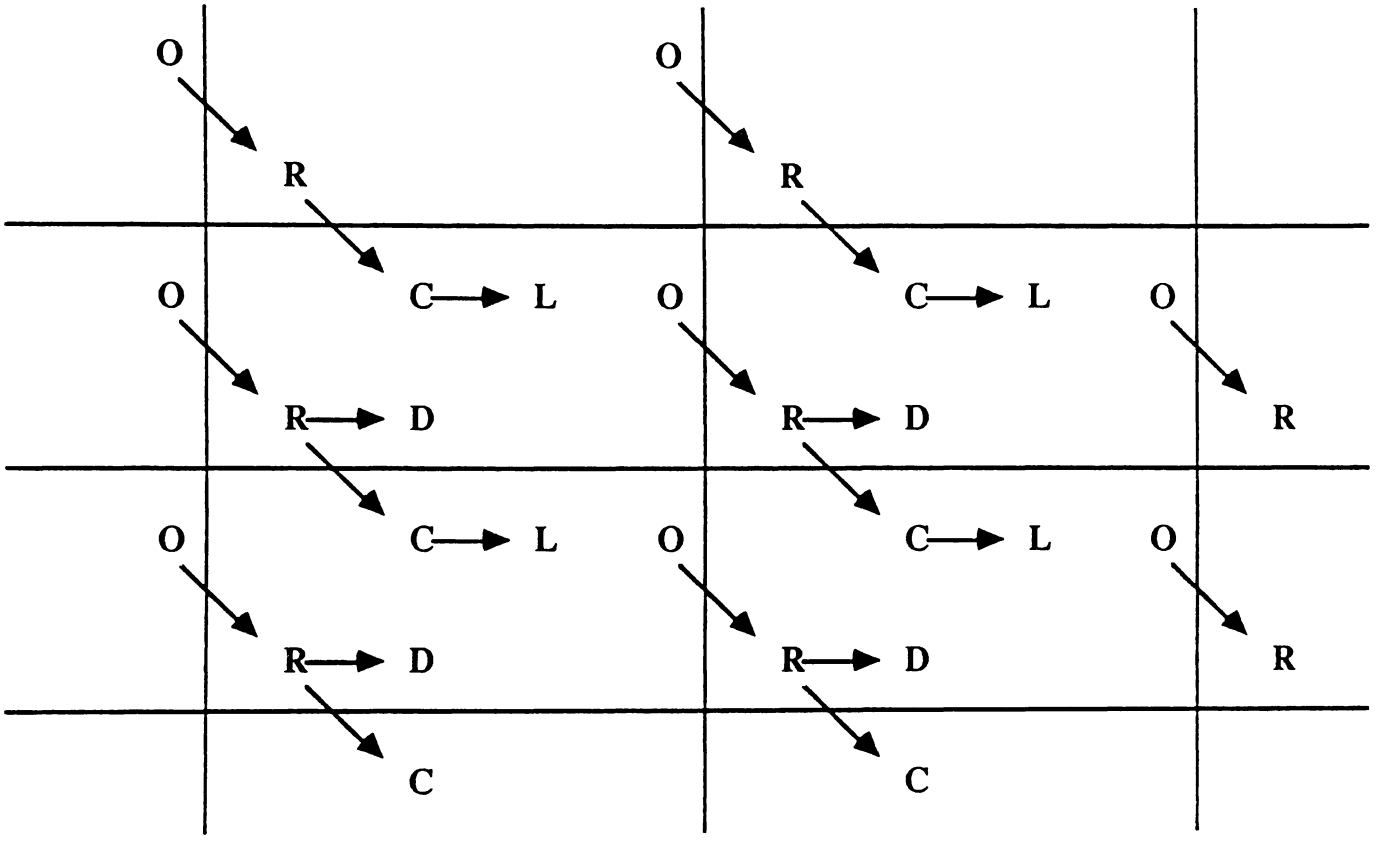
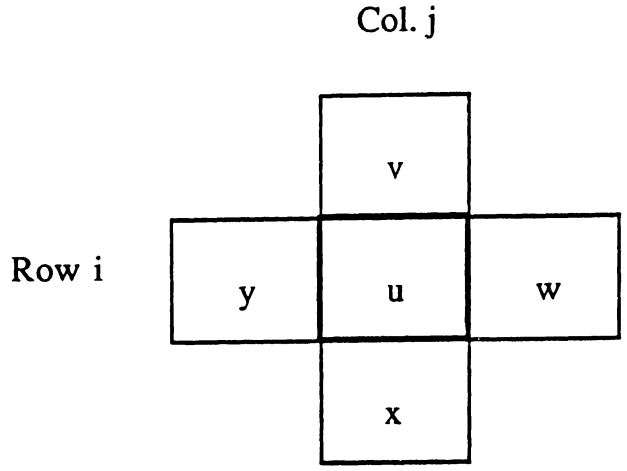
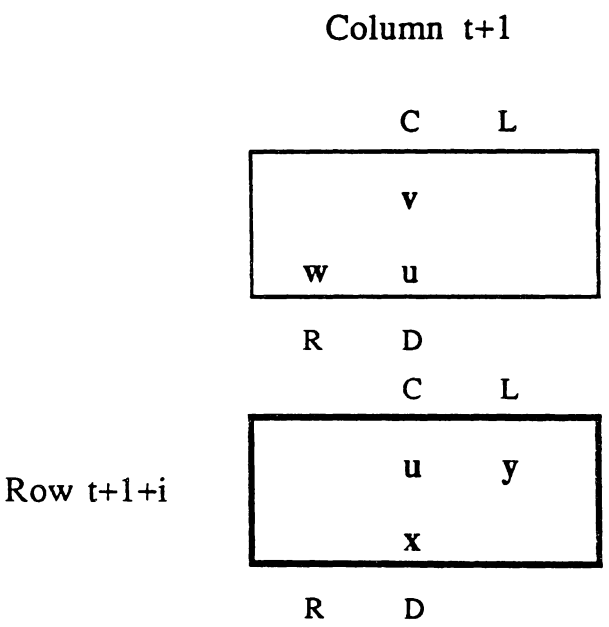


Figure 5. Flow of Data in One-way Simulation.

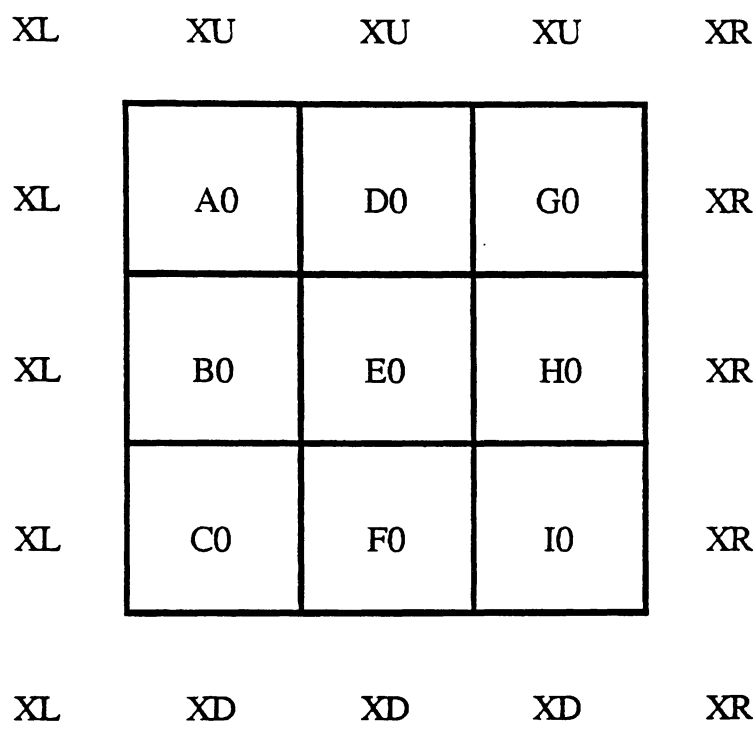


(a) Array P at time t



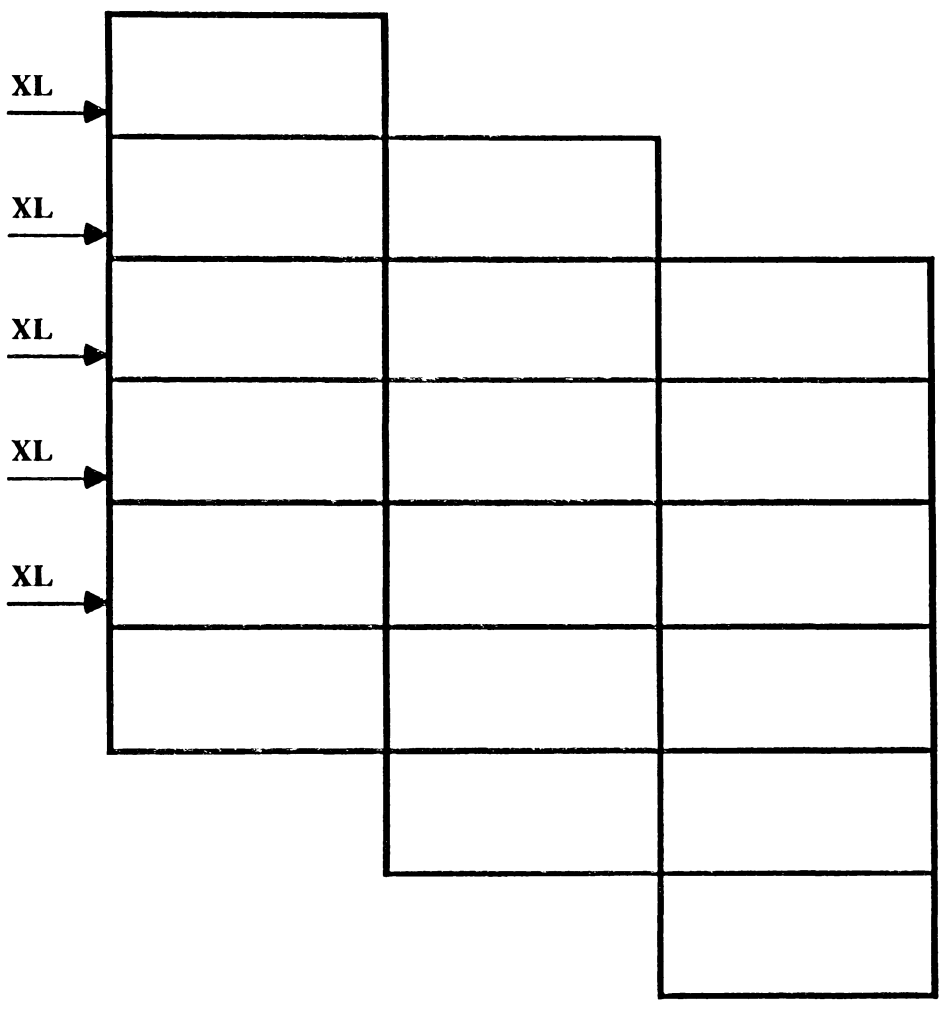
(b) Array Q at time 3(t+1)+j-1

Figure 6. Simulation of Two-way Cellular Array P by One-way Iterative Array Q

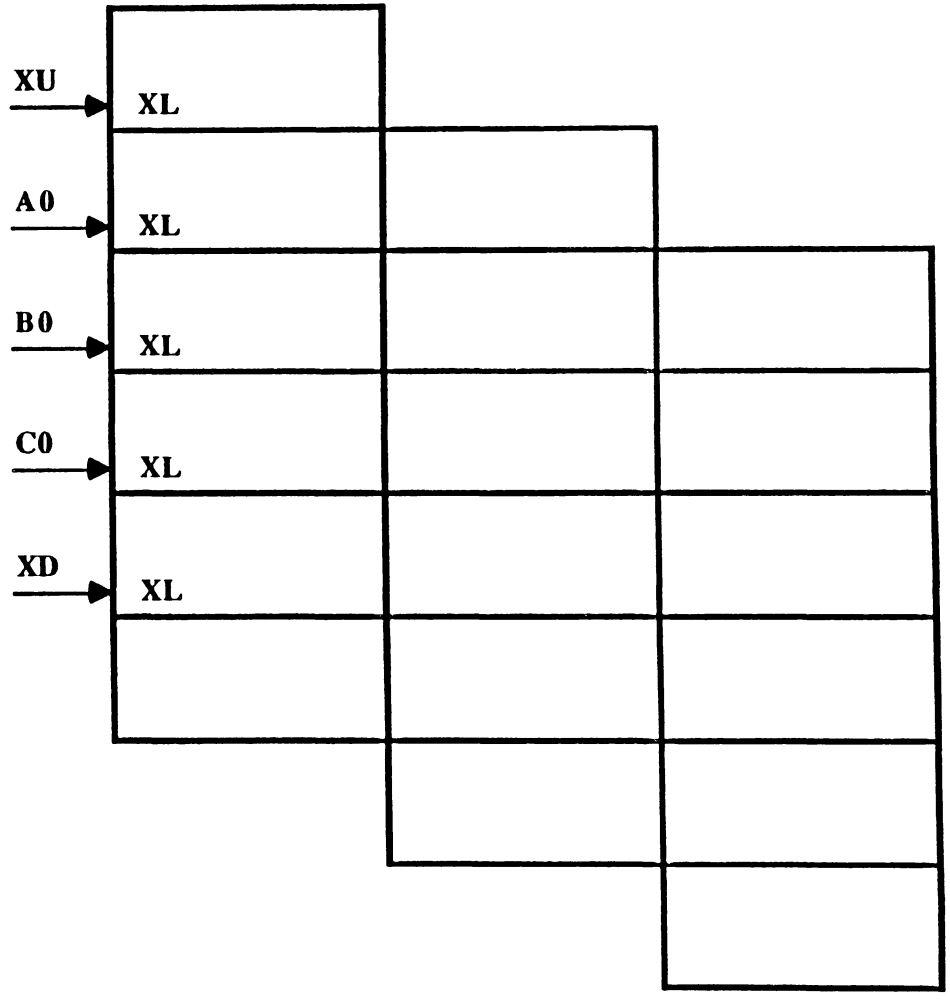


(a) Array P at time t=0

Figure 7. Simulation of Two-way Cellular P by One-way Iterative Q

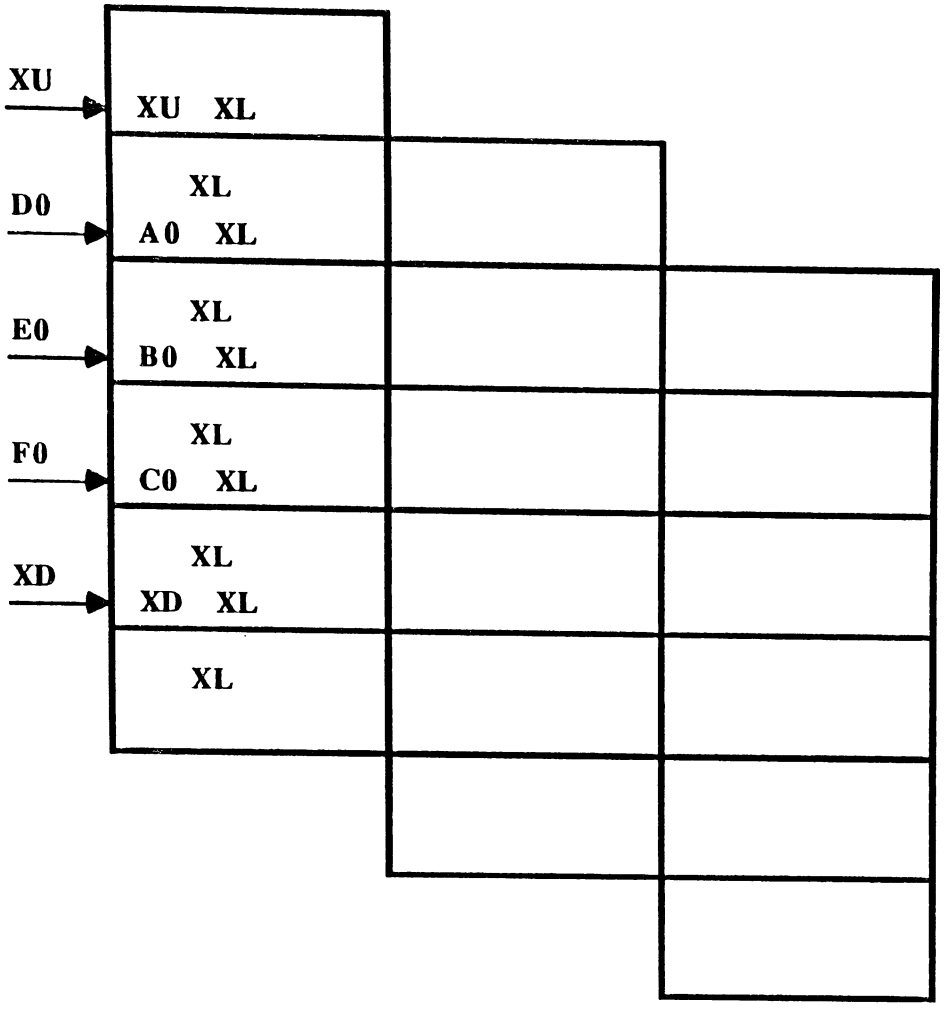


(b) Array Q at time 0

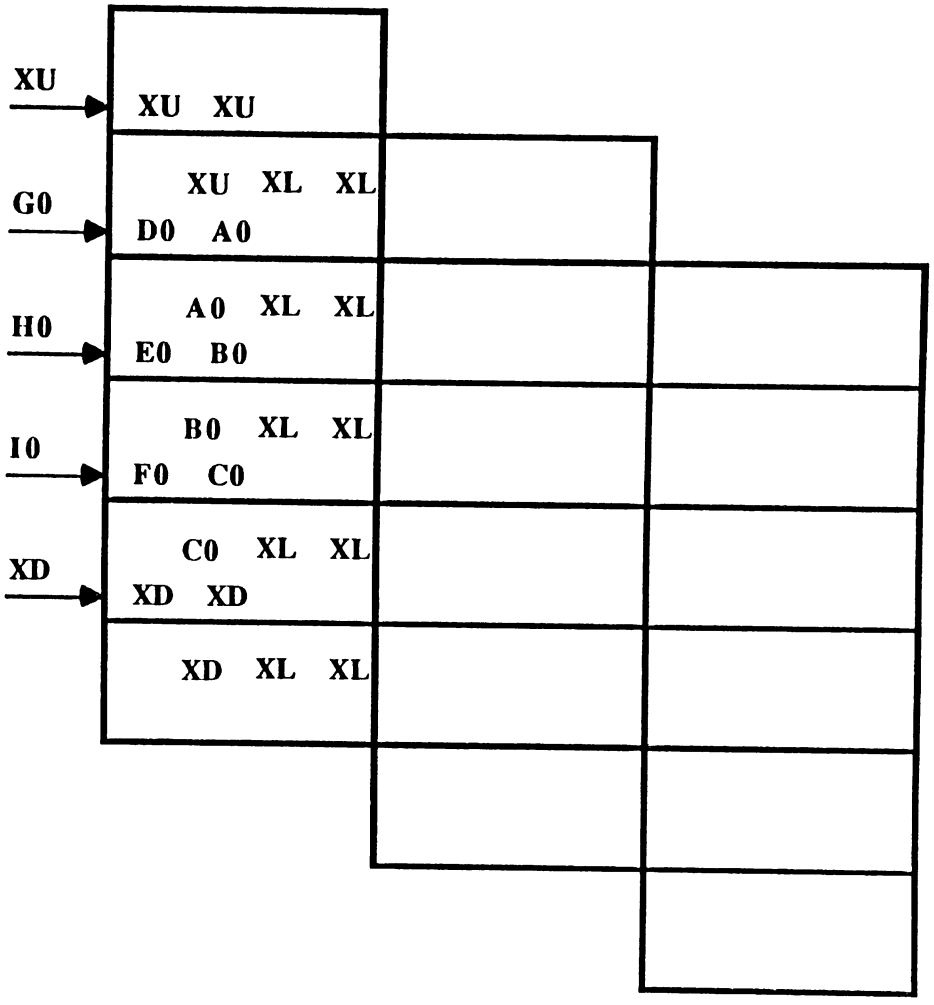


(c) Array Q at time 1

Figure 7. Simulation of Two-way Cellular P
by One-way Iterative Q

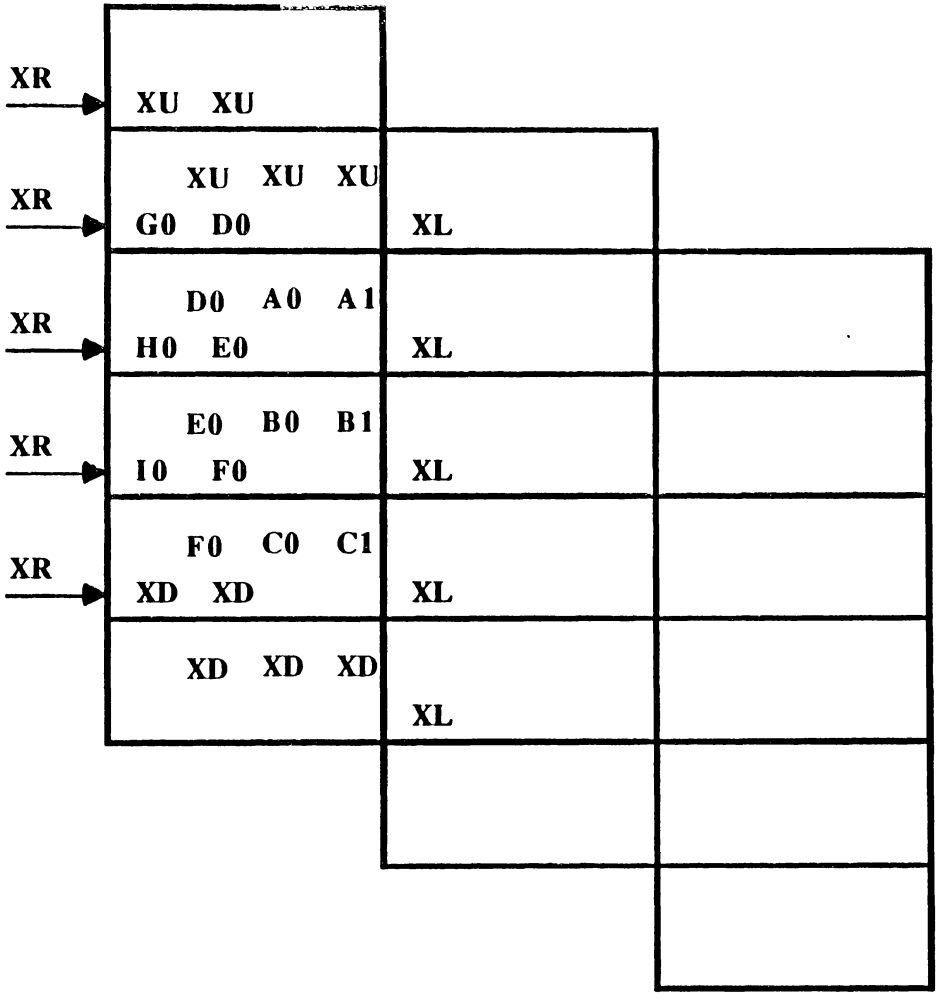


(d) Array Q at time 2

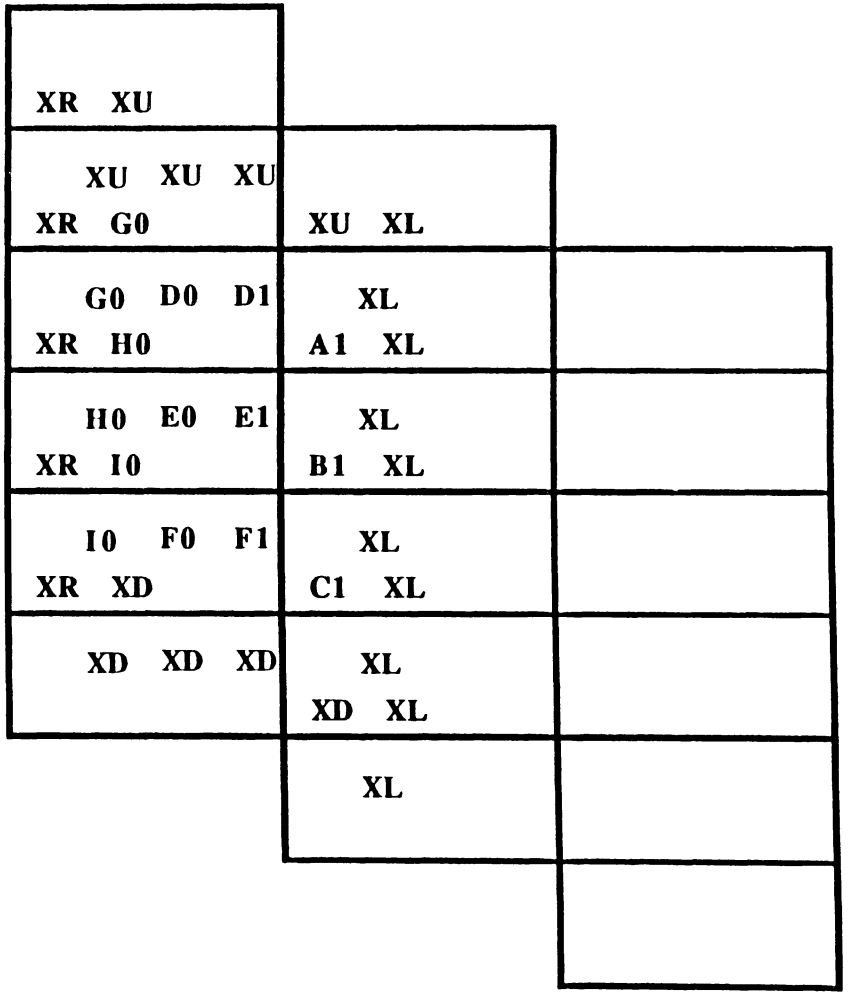


(e) Array Q at time 3

Figure 7. Simulation of Two-way Cellular *P*
by One-way Iterative *Q*

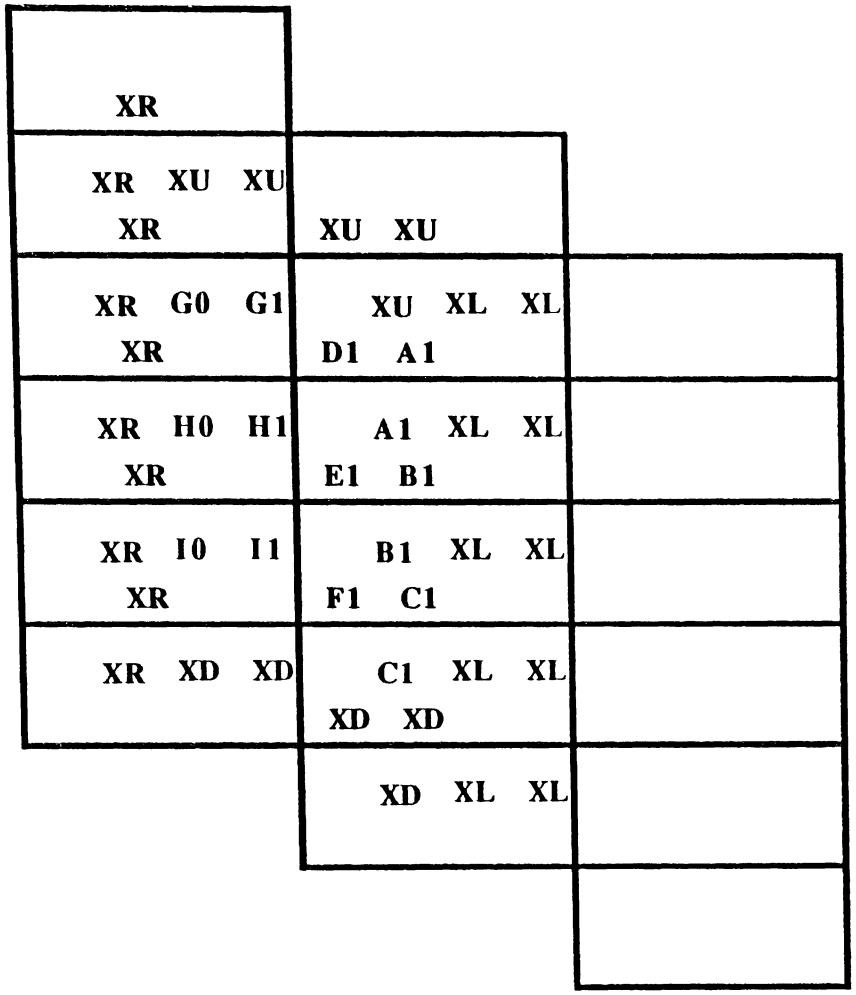


(f) Array Q at time 4

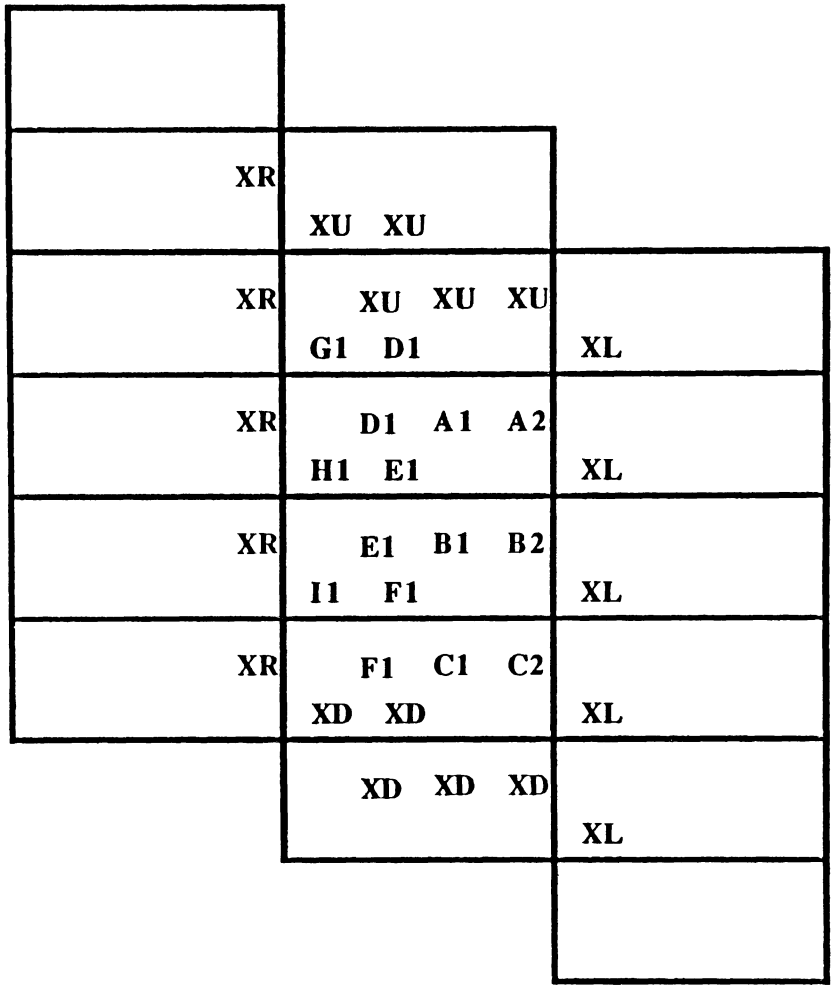


(g) Array Q at time 5

Figure 7. Simulation of Two-way Cellular *P*
by One-way Iterative *Q*

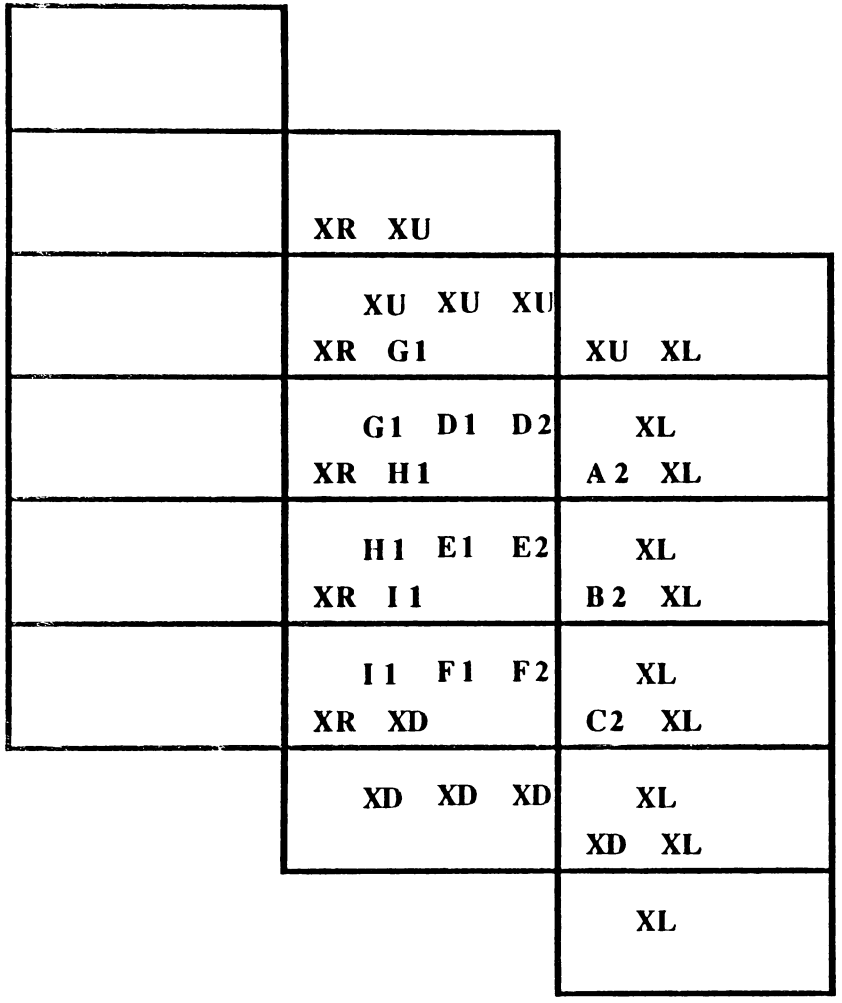


(h) Array Q at time 6

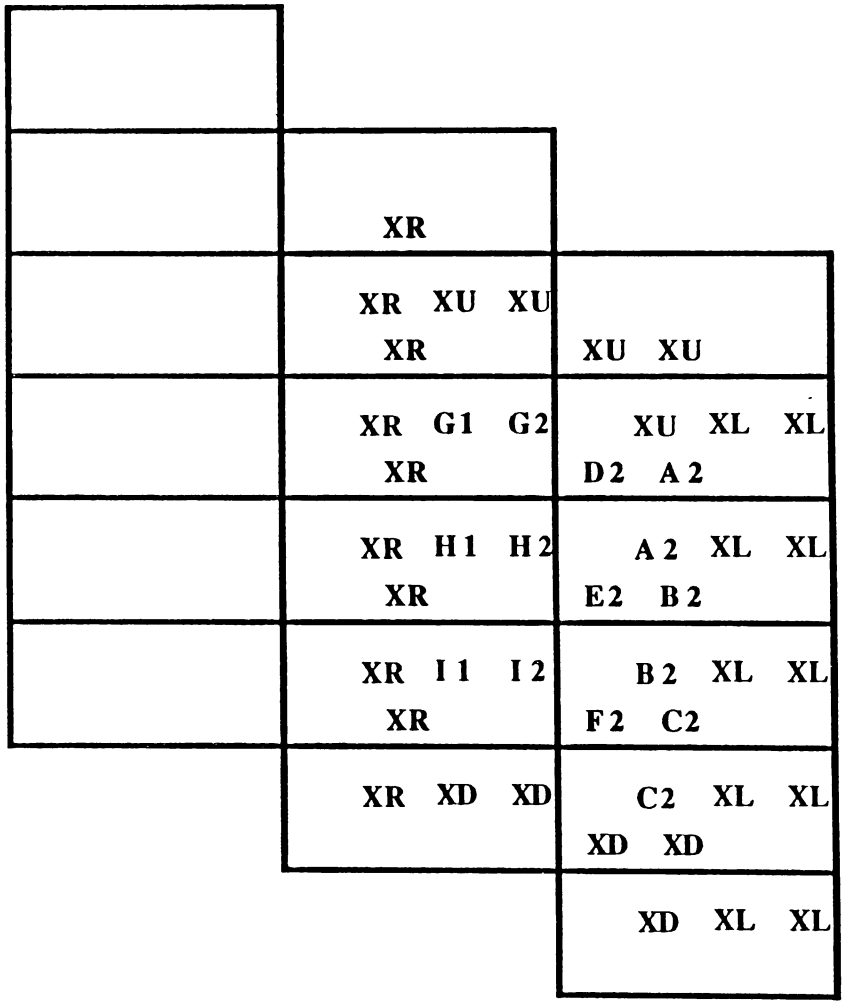


(i) Array Q at time 7

Figure 7. Simulation of Two-way Cellular P
by One-way Iterative Q

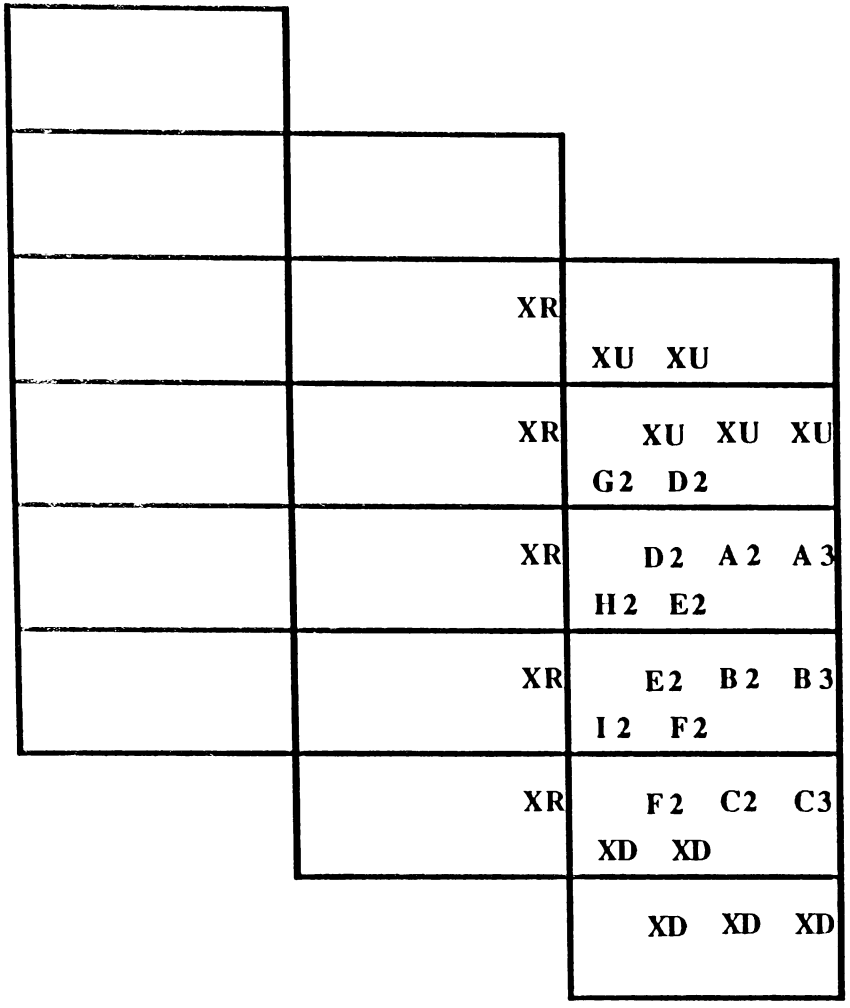


(j) Array Q at time 8

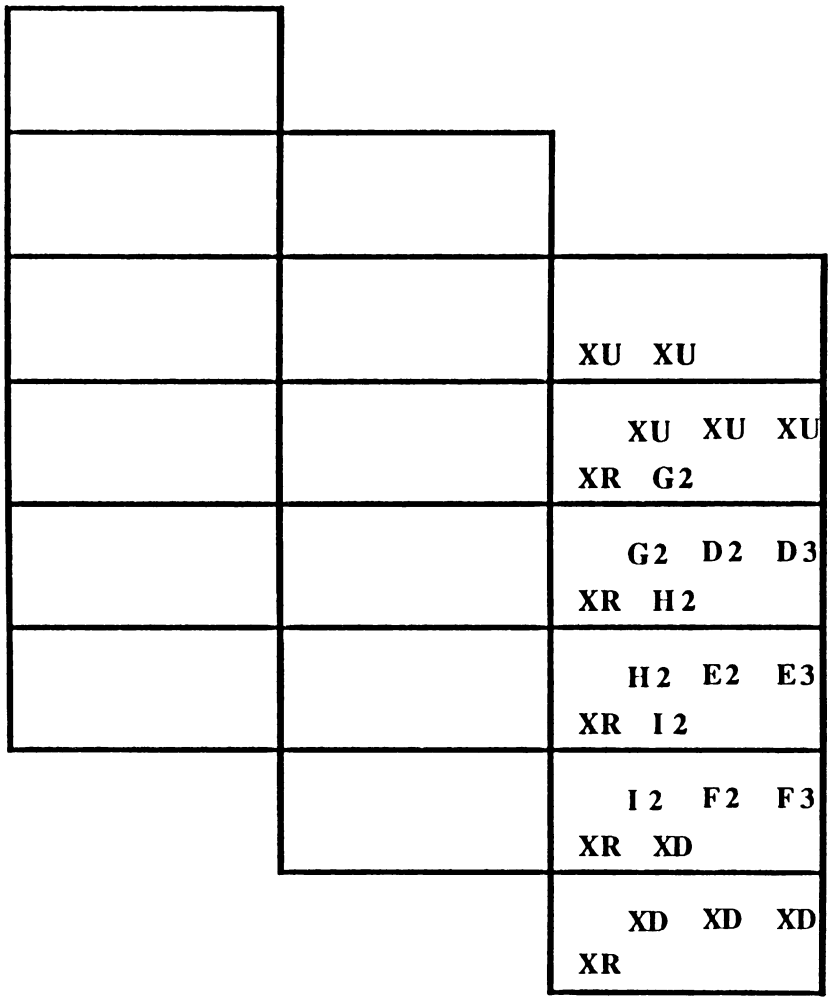


(k) Array Q at time 9

Figure 7. Simulation of Two-way Cellular *P*
by One-way Iterative *Q*

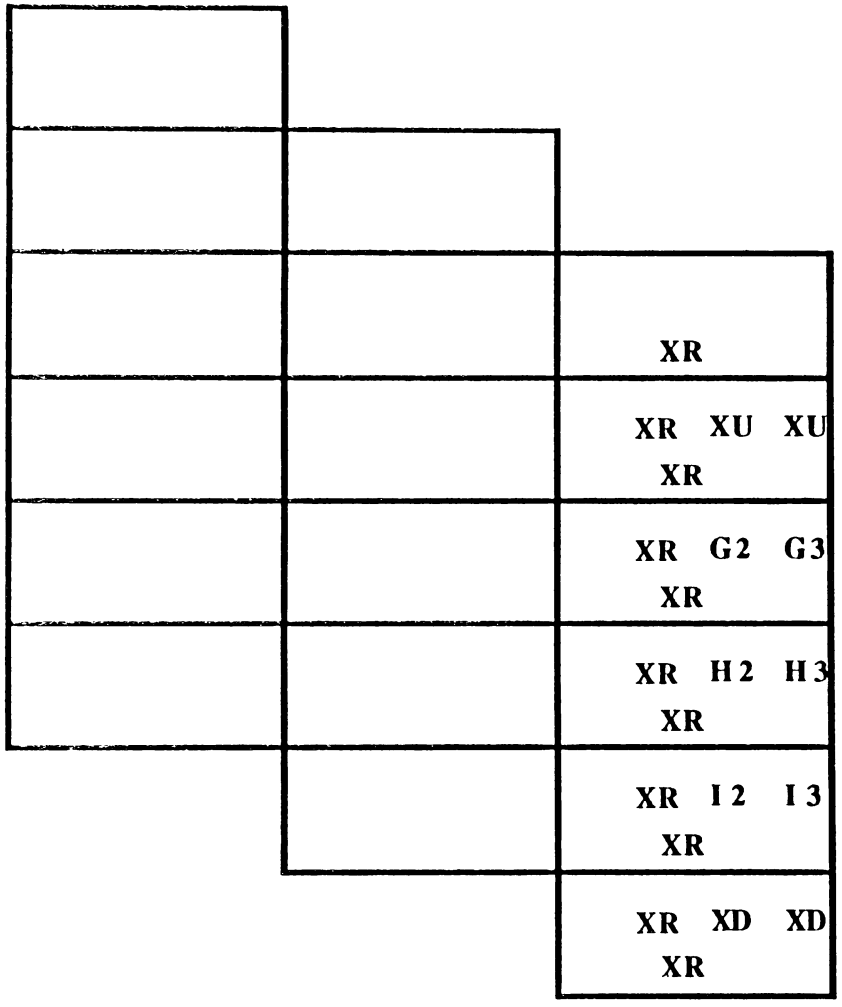


(l) Array Q at time 10

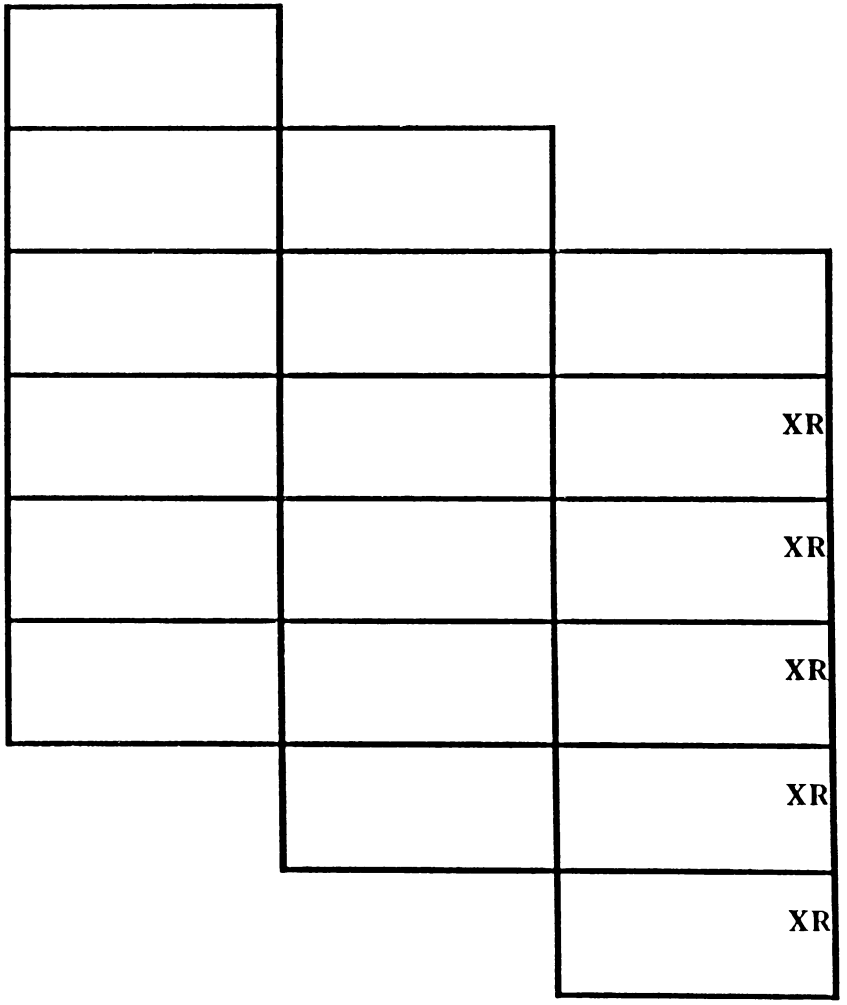


(m) Array Q at time 11

Figure 7. Simulation of Two-way Cellular P
by One-way Iterative Q

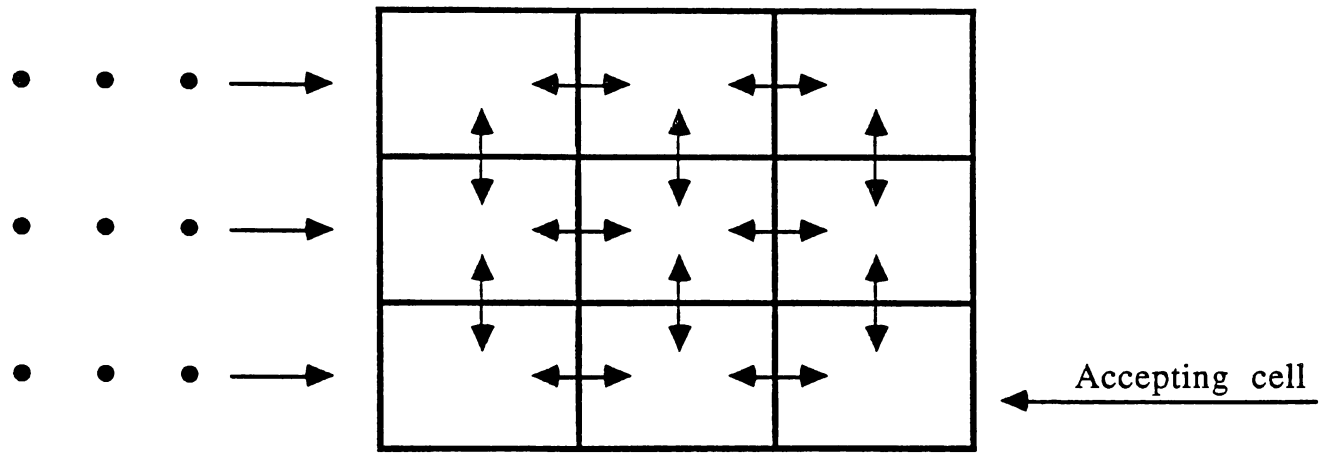


(n) Array Q at time 12

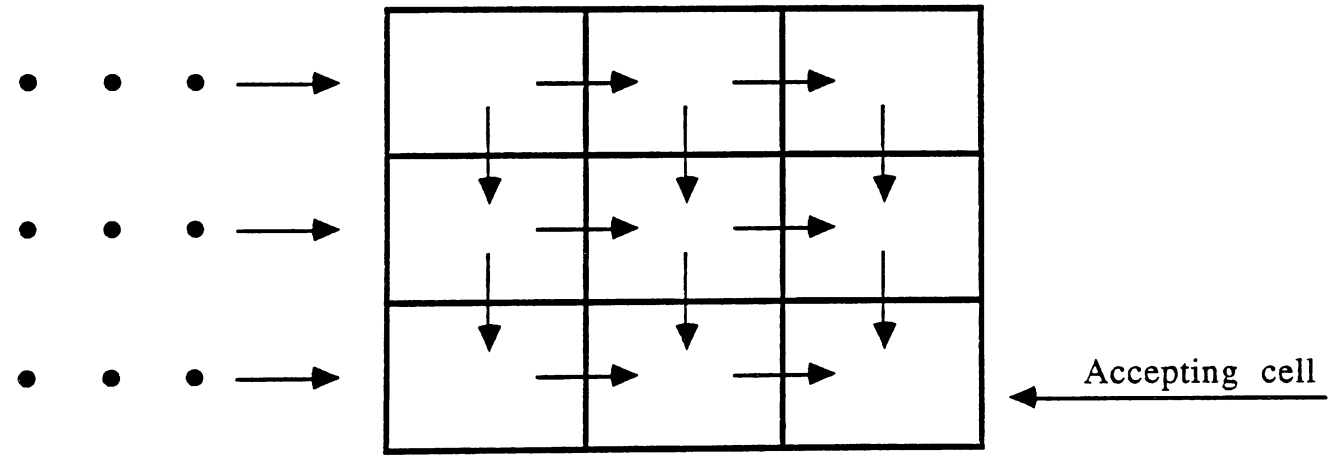


(o) Array Q at time 13

Figure 7. Simulation of Two-way Cellular *P*
by One-way Iterative *Q*



(a) Two-way



(a) One-way

Figure 8. Iterative Mesh Models

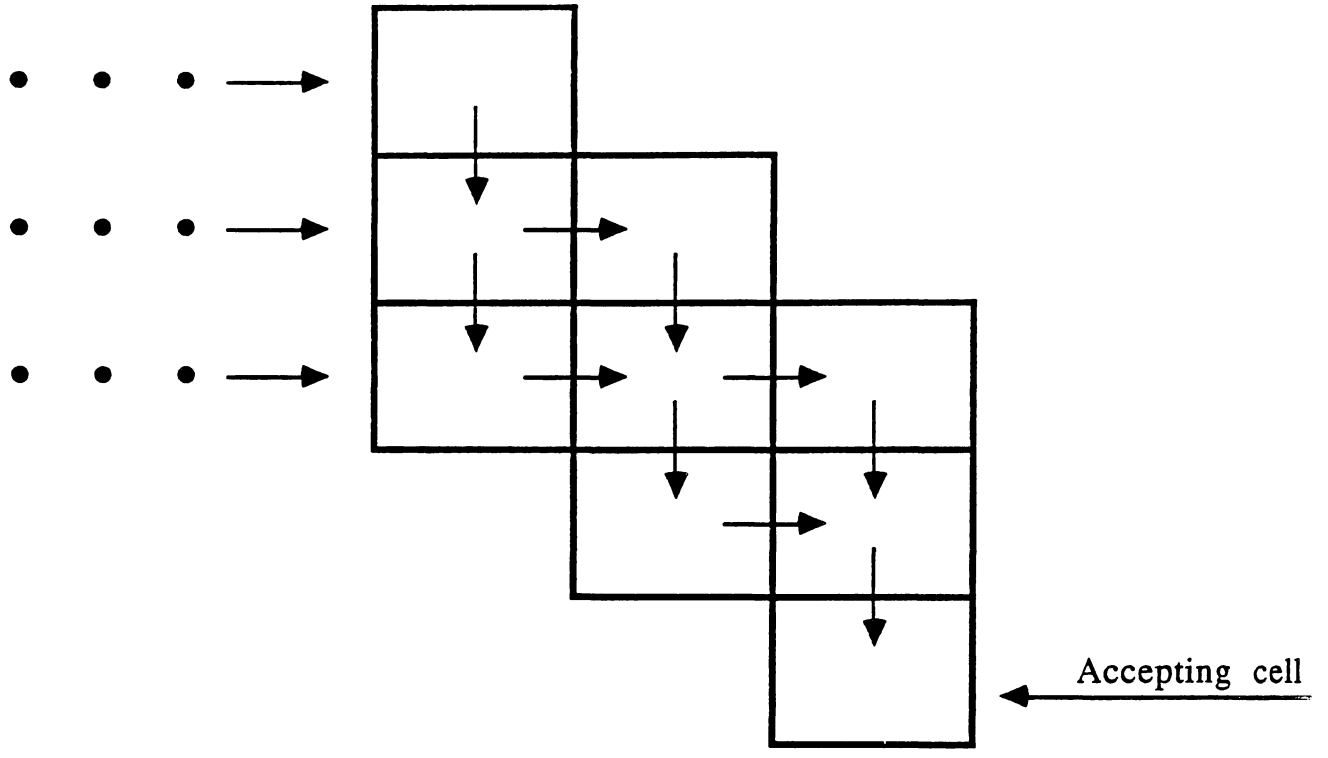


Figure 9. Alternate One-way Iterative Mesh Model