Simulation of Two-way Computations on Arrays with One-way Data Flow

Carla D. Savage
Matthias F.M. Stallmann
Anwer Z. Kotob

Center for Communications and Signal Processing
Department of Computer Science
North Carolina State University

CCSP-TR-87/20

October, 1987

Abstract

In this paper we describe a one-dimensional array model of computation which is powerful enough to solve a wide variety of signal processing problems, as well as combinatorial problems. In this model, input and output is done only at the first and last cells of the array; computation is synchronous and two-way data flow is allowed between adjacent processors. We show how any computation on an array with two-way dataflow can be transformed into a computation on an array in which data flow between adjacent processors is restricted to be one way. Specifically, a two-way array with n cells computing for t time units can be simulated by a one-way array with $O(n+t)$ cells computing for $O(n+t)$ time units. If we allow the one-way array to be circular, we can achieve this simulation with only $O(n)$ cells, even if we are required to keep the locations of the input/output cells fixed throughout the computation.

1. Introduction

In recent years systolic arrays have received much attention and many algorithms were designed to solve various signal processing and combinatorial problems on linear, as well as mesh-connected, systolic arrays. A significant portion of those algorithms require a bi-directional flow of data between adjacent cells in the array [Lei79, GL82, Kun81, Gue86, Rog82, and Kun85].

In this paper we describe the results of our study of ways for transforming *all* bi-directional linear array algorithms into uni-directional ones. This study was prompted by the following two issues.

Fault Tolerance:

Systolic arrays are a natural candidate for wafer scale integration (WSI). The fact that systolic arrays are made up of a large number of relatively small and simple cells, and that only local interconnections (to the nearest neighbor) are required, make them particularly suited for a WSI implementation.

However, with a WSI implementation, fabrication flaws are inevitable. For such an implementation to be practical, the array should "tolerate" some defective cells on a chip. A number of schemes have been devised to grant such arrays the capacity to tolerate faulty cells [LL85, SS87, KL84, GNE84, Smi81]. (In fact some of those schemes, [LL85, GNE84, Smi81], can be applied to almost anything implemented in WSI, not just systolic arrays).

A common approach is to provide redundant circuitry and then program the interconnections to avoid the defective cells [GNE84, Smi81], (laser-programming technology has been applied successfully for such purposes [LL85]). This is a very effective scheme for combating faulty cells, however it comes at a great cost in terms of wasted circuitry.

Another approach would be to program the interconnections to "skip over" the faulty cells [LL85]. This has the advantage that it does not require any redundant circuitry. However, it does mean increased wire length between electrically adjacent cells. Moreover, in such a scheme there is usually a tradeoff between the utilization of live cells and two cost functions: channel width and maximum wire length.

A fault tolerance scheme of particular interest [SS87, KL84] bypasses a faulty cell by a set of "bypass" registers (note that the input and output registers in a cell could be used as bypass registers in case the cell fails, and therefore no extra hardware is required to implement this scheme). This keeps the wire lengths and the clock speed at their original value. Moreover, all the live cells in the linear array are utilized. Kung and Lam [KL84] demonstrated that, with slight modifications, this approach can be applied to two dimensional mesh connected arrays while yielding a high utilization of the live cells.

When the bypass scheme is applied to uni-directional linear arrays it maintains the throughput of the flawless array while suffering only slight degradation in performance. A similar scheme could be applied to bi-directional linear arrays,

but the array's performance and throughput degrade rapidly with respect to the number of consecutive failed cells that need to be tolerated [KL84].

Recyclability:

The issue of decomposability arises whenever the available number of cells is less than the ideal number required for solving the problem at hand (whether due to failed cells or that the array is originally too small for the problem).

Usually the host computer is burdened with the task of decomposing the problem into pieces small enough to fit on the available hardware and then reconstructing the final result.

We have shown that an array with too few cells can still produce the correct result if the output is "recycled" $\left\lceil \frac{n}{k} \right\rceil$ times, where n is the number of cells required by the algorithm and k is the number of cells available [SS87]. This result holds as long as the array algorithm (i) is uni-directional, (ii) has all cells initialized to the same state, (iii) can tolerate an arbitrary delay between adjacent cells, and (iv) still works if the array has more than the ideal number of cells.

When applicable, our result relieves the host computer and the user from decomposition issues, and offers a uniform method for treating all decomposition/recycling tasks.

Note that a requirement for this recycling scheme to work is that the array be uni-directional. The other three requirements are easier to satisfy than uni-directionality.

We note that some of these results, obtained independently, were reported in [CY85].

The remainder of this paper is organized as follows. Section 2 presents the model of computation. In Section 3 we show how to simulate two-way (bi-directional) computation by one-way (uni-directional) computation for a particular subclass of two-way arrays. In Section 4 the general case is broken down into subcases which are considered in sections 5 and 6.

2. Model of Computation

In this section we describe the models of computation for the one-dimensional array and define what we mean by one-way dataflow and two-way dataflow. The array consists of n cells, labeled $1,2,\ldots,n$. Each cell has a finite number of registers, independent of the array size. Each register can hold a word and the length of the word may depend on the size of the array (n) or the size of the problem to be solved. (For this reason, cells cannot be regarded as finite state machines. However, some of our results will hold as well for an array of finite state machines and conversely.)

At any time during the operation of the array, each cell will be in a *state* completely defined by the values in its registers. Input and output to the array may occur only at cells 1 and n. Operation of the array is defined by a single program of which each cell has a copy. This program, called the *cell program* defines the state of a cell i at a given time unit t *in terms of the states of cell* i *and its neighbors at time* $t-1$.

More precisely, let $S(i,T)$ denote the state of cell i at time t and let Q be the set of all possible states. For convenience, we imagine that all input and output values are elements of Q and that Q contains a special state, S^*. We further imagine the existence of a cell 0 and a cell $n+1$ such that

$$S(0,t) = \begin{cases} \textit{the input to cell 1 at time } t+1, \text{ if } \textit{any} \\ S^*, \textit{ otherwise} \end{cases}$$

and

$$S(n+1,t) = \begin{cases} \textit{the (right) input to cell } n \textit{ at time } t+1, \text{ if } \textit{any} \\ S^*, \textit{ otherwise} \end{cases}$$

The cell program defines the operation of the array. For an array with *two-way data flow* between adjacent processors, the cell program, f, can be viewed as a mathematical function

$$f : Q \times Q \times Q \rightarrow Q$$

or, for example, as a PASCAL-like function

$$f(A,\ B,\ C : \textit{state}) : \textit{state}\ ;$$

where, for $t>0$ and $1 \le i \le n$,

$$S(i,t) = f(S(i-1,\ t-1),\ S(i,\ t-1),\ S(i+1,\ t-1))\ .$$

When $t=0$, $S(i,t) = S(i,0)$ represents the initial state of cell i. The output of cells 1 and n at time t, if any, is $S(1,t-1)$ and $S(n,t-1)$, respectively.

An array with *one-way data flow* between adjacent processors is one in which the program f in a cell i depends only on the states of cells i and $i-1$ at the previous time unit. In this case, f is a function of only two parameters,

$$f : Q \times Q \rightarrow Q$$

where for $t>0$ and $1 \le i \le n$,

$$S(i,t) = f(S(i-1,\ t-1),\ S(i,t-1)) \ .$$

In a practical sense, it is unnecessary and, in fact, wasteful to pass entire cell states through the array as data. However, this formalism makes it easier to generalize the notion of array computation and to simplify the generic transformations of arrays. For a particular application, it is easy to prune out the extra information so that only the essential data passes between cells.

3. Two-way Cellular Array to One-way Iterative Array

We first consider the case of cellular arrays with two-way data flow. This is a restriction of the model of computation described in Section 2 in which the n input values are initially stored in the n cells of the array. At the end of the computation, the n cells hold the n output values.

An example of this case is the odd-even transposition sort [Ull84]. Initially, n numbers to be sorted are stored in the array, one per cell. On odd time steps, cells $2i-1$ and $2i$ swap values, if they are out of order. On even time steps, cells $2i$ and $2i+1$ compare and swap, if necessary. After n time units, the values in the array are sorted.

Our goal in this section is to transform a two-way cellular array computation into a one-way iterative computation. A one-way iterative computation is a computation on a one-way array in which input values enter the first cell of the array and leave the last cell of the array as output. Further, all cells are initialized to the same null state.

We assume we are given the cell program, f, of a two-way cellular array, TC, and the initial states $s_1, s_2, \cdots, s_n$ of each cell of TC. The input value stored in cell i of TC is part of the state s_i. Let $f_1, f_2, \ldots, f_n$ be the states of the cells of TC at the end of the computation. Our goal is to determine the cell program, g, of a one-way iterative array, OI, which accepts as input $s_1, s_2, \cdots, s_n$ and produces as output $f_1, f_2, \cdots, f_n$.

The idea of the two-way to one-way transformation is illustrated in Figures 1 and 2. Figure 1 represents a two-way cellular computation in which cells are initialized to A_0, B_0, C_0, D_0, and E_0. After 5 time units, the cells are in states A_5, B_5, C_5, D_5, and E_5. As described in Section 2, we assume the existence of imaginary cells 0 and 6 which contain a special state, say S^*. Figures 2a and 2b show a simulation of the two-way cellular array of Figure 1 by a one-way iterative array, OI. Input to OI is the sequence $S^*, A_0, B_0, C_0, D_0, E_0, S^*$. Note that for $i=1,\ldots,5$, cell i computes and outputs A_i, B_i, C_i, D_i, E_i at time units $2i+1$, $2i+2$, $2i+3$, $2i+4$, $2i+5$, respectively. Thus, the final states of the two-way cellular array after t time units would be computed and output by cell t of the one-way array during time units $2t+1$ through $2t+5$.

To describe the operation of OI in the general case, each cell of OI stores two values, *old* and *new*. (Each of these values is a cell state of TC, the two-way cellular array being simulated by OI.) During a time unit, if x is the input value to a cell of

OI, the cell does the following:

(1) computes the value $f(old, new, x)$ to be output. (f is the cell program of *TC*)

(2) $old \leftarrow new$

(3) $new \leftarrow x$

Additional minor details are required in the cell program for *OI* to handle the boundary cases and these are described fully in [Kot87]. The result can be summarized as follows.

Theorem: A two-way cellular array (with n cells) computing for t time units can be simulated by a one-way iterative array with t cells computing for $2t+n$ time units.

Discussion: What is the penalty?

The penalty to be paid for achieving one-way versus two-way communication can be very small. The one-way time is $2t+n$, whereas the minimum time achievable by an n-cell iterative array computing an n-input, n-output function would be $2n$. However, this time of $2t+n$ is latency and *does not affect array throughput*. Successive problem instances can be pipelined. In addition, the time t for the two-way cellular array did not take into consideration the time to load and unload the array with input and output.

As for the complexity of each cell in the one-way array, storage is increased by about a factor of two and the complexity of the cell program is increased by a small additive constant. This should be regarded as an upper bound on the added complexity since for any specific application it may be possible to prune substantially the generic result of the two-way to one-way transformation.

A more serious concern is the number of cells required by the one-way iterative array if t is large. If $t = n$, which is frequently the case (as in odd-even transportation sort) then there is no increase in the number of cells for one-way computation. On the other hand, independent of t, note that at most $\lfloor n/2 \rfloor + 1$ consecutive cells are used during a given time unit and that the rightmost cell moves one cell to the right every two time units. Thus, by using techniques as described in Section 6, the one-way iterative computation can be done on a circular array of roughly $n/2$ cells, compensating for the increased cell complexity. (Actually, to maintain two fixed *I/O* cells for the circular array, the cell complexity may increase somewhat. See Section 6.) Since data is recirculated in this case, problem instances cannot be pipelined, but neither could they be for the original two-way celluar array.

In summary, then, not only is one-way data flow always achievable, but the penalty for one-way versus two-way data flow is small, whereas the advantages, in terms of fault tolerance, problem decomposition, and input delay tolerance, appear to be very high.

4. Two-way to One-way in the General Case

In the most general case of computation on an array, each cell could be intitalized to a different state. An example of this is the array in [Rog82, KRY81] which performs FIR filtering by storing a coeffcient of the convolution kernel in each cell. In addition, input and output could occur in both the first and last cell as in [Lip85].

A two-way array in which I/O is done at both the first and last cell can be "folded" to obtain a two-way array in which I/O is done only at the first cell. This array can be further modified to force output values arriving at cell 1 to bounce back to the last cell of the array and leave in proper sequence. Details of this are described in [Kot87].

It suffices, then, to consider two-way arrays in which input occurs only at the first cell and output only at the last cell. In the next section it is shown that any such array in which cells are required to be initialized to different states can be simulated by a two-way array (input first cell, output last) in which cells need not be initialized— or in which all cells can be initialized to the same state.

The remaining case to consider is that of a two-way iterative array, that is, a two-way array with input first cell, output last cell and all cells initialized to the same state. We can show the following.

Theorem: A two-way iterative array with n cells computing for t time units can be simulated by a one-way iterative array with $n+t-1$ cells in time $2t+1$.

Proof: The simulation is illustrated in figures 3 and 4. Basically, the computations are "shifted right" every other time unit, with details included to pass input right to the "first" cell and shift output right to the output cell. See [Kot87] for details.

In contrast to two-way cellular arrays, in many applications on two-way iterative arrays, especially to signal processing, t may be very large compared to n so that using $n+t-1$ cells to achieve one-way data flow would be unreasonable. Because of this, it is more appropriate to consider simulation on a one-way circular array. In Section 6 we show how to simulate a two-way iterative array with a one-way circular array with no increase in the number of cells.

5. Initialization of a Two-Way Computation

The following simulation converts a two-way computation in which cells are initialized to different states into a two-way computation in which cells are all initialized to the same state. In the original two-way computation, input comes from the left and has the form $S, x_1, \cdots, x_m, E$, where S and E are two symbols that do not correspond to any data (in fact, S and E may be the same symbol; we distinguish between them only for clarity). Cell i is initially in state q_i for $i = 1, \cdots, n$. Details that are specific to the computation are encapsulated by the function $f(A,B,C)$, where A, B, and C are the states of three adjacent cells. $f(A,B,C)$ computes a new state for the cell whose current state is B, and whose left (resp. right) neighbor is in state A (C). In a typical computation, f only needs to access part of the internal states A and C. Two special cases of f are the computation that takes place in the first and last physical cells. In the case of the first cell, A is taken

to be the input data (padded appropriately to make it of the same "type" as a state). For the last cell, C can either be the same as B or can have some special value such as *NULL* (this is dependent on the computation and does not affect our simulations).

Aside from f we assume that each cell has some mechanism for turning itself on at the beginning and off at the end of the computation. We view such a mechanism as being essentially equivalent to a control flag with values *OFF* and *ON* (this flag is part of the state of each cell). When a cell is *ON*, it computes f using the states of its two neighbors (which are transmitted at the end of the previous step). When a cell is *OFF*, it either retains its internal state or turns itself on. The symbol S in the input is used to turn cells on, while E turns them off. We assume that f is defined when the first parameter is E (to allow the computation to "know" when the end of input has occurred). We also assume that cells are reinitialized if necessary as they are turned off. A generic program for cell i during a single time unit is described below.

```
/* program for cell i; A, B, and C are "registers" holding the internal states
   of cells i-1, i, and i+1, respectively */
if cell i is OFF then
    if cell i-1 is ON then
        /* (or i = 1 and input is S) */
        change to ON end if
else /* cell i is ON */
    B := f(A,B,C)
    if cell i-1 is OFF then
        /* (or i = 1 and input is E) */
        change to OFF end if
end if
```

When the above computation is simulated by one in which cells need not be intialized, the initial state of each cell becomes part of the input. Input to the transformed computation takes the form $S,q_1,\cdots,q_n,M,x_1,\cdots,x_m,E$, where q_i is data representing the intial state of cell i and M is an additional special symbol (again, S, E, and M can all be the same symbol; we distinguish only for clarity). There are now three control states: *OFF*, *ON*, and *ACTIVE*. When a cell is *ON*, it merely transfers the data representing initial states. The first data item to reach a cell becomes its intial state. Subsequent data items are moved to the right. This ensures that cell 1 receives q_1 as its initial state, cell 2 receives q_2, and so on. In order for the transfer of initial state data to be accomplished, each cell must have an additional register *TRANS* capable of holding state data. When a cell is *ACTIVE*, however, it performs the computation associated with f. The simulation ensures that when a cell becomes *ACTIVE* both of its neighbors have received their initial state. The program for a cell during each time unit is described below.

```
/* program for cell i */
if cell i is OFF then
    if cell i−1 is ON then
        /* (or i = 1 and input is S) */
        B := S
        change to ON end if
else if cell i is ON then
    if B = S then
        B := TRANS(i−1) (or input data if i = 1)
        TRANS(i) := S
    else /* B contains valid state info */
        TRANS(i) := TRANS(i−1) end if
    if cell i−1 is ACTIVE then
        /* (or i = 1 and input is M) */
        change to ACTIVE end if
else /* cell i is ACTIVE */
    B := f(A,B,C)
    if cell i−1 is OFF then
        /* (or i = 1 and input is E) */
        change to OFF end if
end if
```

6. Simulating a Two-Way Iterative Computation on a One-Way Circular Array

A circular array (with one-way dataflow) has n cells arranged in a circular configuration, each cell having the ability to "read" the state of the previous cell. Cells are numbered 0 through $n-1$ so that cell i has the ability to read the state of cell $(i-1) \bmod n$. Cell 0 is the only cell that has the ability to read input and write output. A typical cell program is a function $h: Q \times Q \to Q$. Cell i (for $i = 1, \cdots, n-1$) computes $h(A,B)$, where A is the state of cell $i-1$ and B is the state of cell i. Cell 0 computes a slightly different function $h': Q \times Q \times \Sigma \to Q \times \Sigma$, i.e. h' also takes input into account and produces output as part of the result.

In our simulation there are three distinct "tracks" along which the computation takes place. The *input track* ensures that input gets from cell 0 to the cell that actually "reads" the input in the simulated computation. The *output track* ensures that output produced by the simulated computation reaches cell 0 in the correct sequence. The *computation track* accomplishes the actual simulation.

The input track of each cell has two data registers. *Cur_Input* is the current input data for the cell. *Cur_Input* remains unmodified until the computation track of the cell "reads" input. Then it is transferred to the computation and replaced by input from the cell to the left. *New_Input* is input data that is still moving to the right (has not reached its destination). Because later input are consumed farther to the right (within the circular order), the input track allows more recent input to overtake less recent input and move farther to the right. The part of h dealing with

the input track, h_I, can be described as follows.

```
function h_I(A,B) returns INPUT_STATE is
    /* If the computation requires data from the input track during this time
       step, Cur_Input(B) = empty by the time this function is executed.
       Note also that cell 0, instead of reading New_Input(A), always reads
       actual input data. */
    if Cur_Input(B) = empty then
        Cur_Input(B) := New_Input(A)
        New_Input(B) := empty
    else
        New_Input(B) := New_Input(A)
        (Cur_Input(B) stays the same)
    end if
end h_I
```

The output track is handled similarly by a function h_O. The output state of a cell includes two data registers. *New_Out* is output produced by the most recent computation at the cell. *New_Out* stays in place until all output produced at cells immediately to the left has overtaken it (such output was produced earlier and must therefore reach cell 0 sooner). *Pass* contains output which is moving right toward cell 0.

```
function h_O(A,B) returns OUTPUT_STATE is
    /* If the computation in this cell produced output during the current
       time step, New_Out now holds the value of that output. The Pass
       register of cell 0 is always empty when seen by cell 1 (its contents
       become actual output from the array). */
    if Pass(A) = empty then
        Pass(B) := New_Out(B)
        New_Out(B) := empty
    else
        Pass(B) := Pass(A)
    end if
end h_O
```

We are now ready to describe how the computation track of the cells functions. Aside from the details having to do with startup and boundary conditions (described below), each step of the two-way computation is simulated by two steps of the one-way computation (this part of the simulation is identical for linear and circular arrays). Each cell has two registers, each capable of holding state information for the original two-way computation (we assume all the cells in the two-way computation are initialized to the same state -- see Section 5). The registers are called *Middle* and *Right*, and their contents correspond to the state of this cell and the one to its right, respectively. Each cell also has a register *C_State* holding its control state. The different control states are introduced below. In every case, the

new state of a cell is a function of its current state and the *Middle* and *C_State* registers of the cell to its left.

The main idea behind the simulation is that a computation that took place in cell i at time t in the two-way array takes place in cell $(i+t+\alpha)$ mod n at time $2t+\beta$ in the one-way array, where α and β are constants related to how the startup sequence is accomplished. If we ignore boundary cases, the computation track works as follows. Note that g is the function computed by each cell in the original two-way computation.

```
function h_C(A,B) returns COMPUTATION_STATE is
    if C_State(B) = MAIN then
        Middle(B) := g(Middle(A),Middle(B),Right(B))
        C_State(B) := INTERMEDIATE
    else /* C_State(B) = INTERMEDIATE */
        Right(B) := Middle(B)
        Middle(B) := Middle(A)
        C_State(B) := MAIN
    end if
end h_C
```

The actual computation is performed during the *MAIN* control state. The *INTERMEDIATE* state is needed to route data to the right.

Figure 5 shows one possible implementation of the full computation state calculations, including startup sequence. This implementation, while not being ideal (some problems and their solutions are noted below), illustrates all of the issues that need to be dealt with.

As in the two-way computation, a cell is initially in the *OFF* control state. A special *START* state moves around the circle once to turn all cells on (control state *ON*). In the *ON* state (or the *START* state), only the input track of a cell is active.

After the start state has travelled all the way around the circle, it prompts cell 0 to begin its computation track in a state called *FIRST*. *FIRST* is exactly like *MAIN* except that the cell uses the value in its own *Cur_Input* register instead of the previous cell's *Middle* register. When a cell is in state *FIRST* it is simulating cell 1 of the original two-way computation.

Every time step after cell 0 begins the computation, a new cell activates its computation track in either the *MAIN* or *INTERMEDIATE* state. Meanwhile, the control state *FIRST* passes on to a new cell every other time step. When all of the cells on the circle have become activated, the cell in state *FIRST* sees a cell in state *MAIN* to its left. In the next time step, the same cell takes on control state *LAST* -- it is now behaving as the last cell of the original two-way computation and producing output. We assume for simplicity that the circle has exactly the right number of cells. If not, we either have to assume that the exact number of cells was not critical to the two way computation (as long as there are enough) or a moving counter has to count the number of active cells to determine when to enter state *LAST*. The

computation track in state *LAST* behaves like *INTERMEDIATE*, except that the cell moves data to the output track.

Because the circular array performs a computation of the original two-way array every other time unit, the rate of input and output for the array must be one item every other time unit. Use of the algorithm in Figure 5 along with the algorithms for input and output tracks actually forces input and output to be staggered: cell 0 reads n items during the first n time units, then no items during the next n time units, then n items again, and so on. Output behavior is similar.

If input needs to be read at a steady rate of one item every other time unit, the actual computation should not begin until exactly $2n$ time units have elapsed. This is to ensure that the first n items get beyond cell 0 before cell 0 consumes the first input so that the $n+1$st input takes the place of the first in cell 0. Allowing the start marker to move around the array twice before initiating the computation is one way to accomplish this goal (this requires a new control state *START'* for the second iteration of the start marker).

Staggered output can be remedied by having each cell produce a dummy output after each real output it produces. The array then generates a dummy output between every two real output values.

Another problem related to output is that the first real output arrives in the output track of cell $n-1$. This means that the first and second output values exit through cell 0 almost immediately after they are produced while the remaining output must traverse around the circle. The result is a large gap in time between when the second output value leaves cell 0 and when the third value leaves cell 0. Several remedies are possible. One is to redesign the circular array so that cell $n-2$ instead of cell 0 has the output port. Another is to shift the startup sequence so that computation begins in cell 2 instead of cell 0 (and adding two dummy inputs to the beginning of the input sequence). A third possibility is to mark the first two outputs in some special way so that they are passed from cell 0 to cell 1 instead of leaving the array. All solutions but the first require additional control states and a more complicated algorithm.

References

[CY85] K .Culik II, and S. Yu, "Translation of Systolic Algorithms Between Systems of Different Topology", *Proceedings of the 1985 International Conference on Parallel Processing*, D. Degroot editor, Computer Science Press, (1985, pp. 756-763).

[Fis81] Allan L. Fisher, "Systolic Algorithms for Running Order Statistics in Signal and Image Processing", *Proceedings of CMU Conference on VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele editors, Computer Science Press, (1981, pp. 265-272).

[GL82] Leo J. Guibas, and Frank M. Liang, "Systolic Stacks, Queues, and Counters", *Proceedings of 1982 Conference on Advanced Research in VLSI, MIT.*

[GNE84] Jan Grinberg, Graham R. Nudd, and R. David Etchells, "A Cellular VLSI Architecture", *Computer, Vol. 17, No. 1,* (Jan. 84, pp. 69-81).

[Gue86] Concettina Guerra, "Systolic Algorithms for Local Operations on Images", *IEEE Transactions on Computers, Vol. c-35, No. 1,* (Jan. 86, pp. 73-77).

[KL84] H. T. Kung, and Monica S. Lam, "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays", *Journal of Parallel and Distributed Computing, Vol. 1, No. 1,* (Aug. 84, pp. 32-63).

[Knu73] D. E. Knuth, "The Art of Computer Programming Vol. III: Sorting and Searching", Addison-Wesley, Reading, Mass. (1973).

[Kot87] Anwer Z. Kotob "Transforming bi-directional computation to unidirectional computation on linear systolic arrays", Master's Thesis, Computer Studies, NCSU, 1987.

[KRY81] H. T. Kung, Lawrence M. Ruane, and David W. L. Yen, "A Two-Level Pipelined Systolic Array for Convolutions", *Proceedings of CMU Conference on VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele editors, Computer Science Press, (1981, pp. 255-264).

[Kun81] H. T. Kung, "Why Systolic Architectures?", Technical report CMU-CS-81-148, Carnegie-Mellon University, Computer Science Department, (Nov. 81).

[Kun85] S. Y. Kung, "VLSI Array Processor for Signal Processing", In *Modern Signal Processing*, Thomas Kailath editor, Hemisphere Publishing Corp., (1985, pp. 393-440).

[Lei79] Charles E. Leiserson, "Systolic Priority Queues", *Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, (Jan. 1979, pp. 199-214).

[Lip85] Richard J. Lipton, and Daniel Lopresti, "A Systolic Array for Rapid String Comparison", *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, H. Fuchs editor, Computer Science Press, (1985, pp. 363-

376).

[LL85] Tom Leighton, and Charles E. Leiserson, "Wafer Scale Integration of Systolic Arrays", *IEEE Transactions on Computers, Vol. c-34, No. 5,* (May 85, pp. 448-461).

[Rog82] M. H. Rogers, "Specification of Algorithms for Systolic Array Elemenets", In *VLSI Architecture,* B. Randell, and P. C. Treleaven editors, Prentice-Hall International, (1983, pp. 212-224).

[Smi81] Robert T. Smith, James D. Chlipala, John F. M. Bindels, Roy G. Nelson, Frederick H. Fischer, and Thomas F. Mantz, "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM", *IEEE Journal of Solid State Circuits, Vol. sc-16, No. 5,* (Oct. 1981, pp. 506-514).

[SS87] Carla D. Savage, and Matthias F. M. Stallmann, "Fault Tolerance & Decomposability Issues in One-dimensional Array Architectures", in preparation.

[Ull84] J. D. Ullman, "Computational Aspects of VLSI", Computer Science Press, (1984).

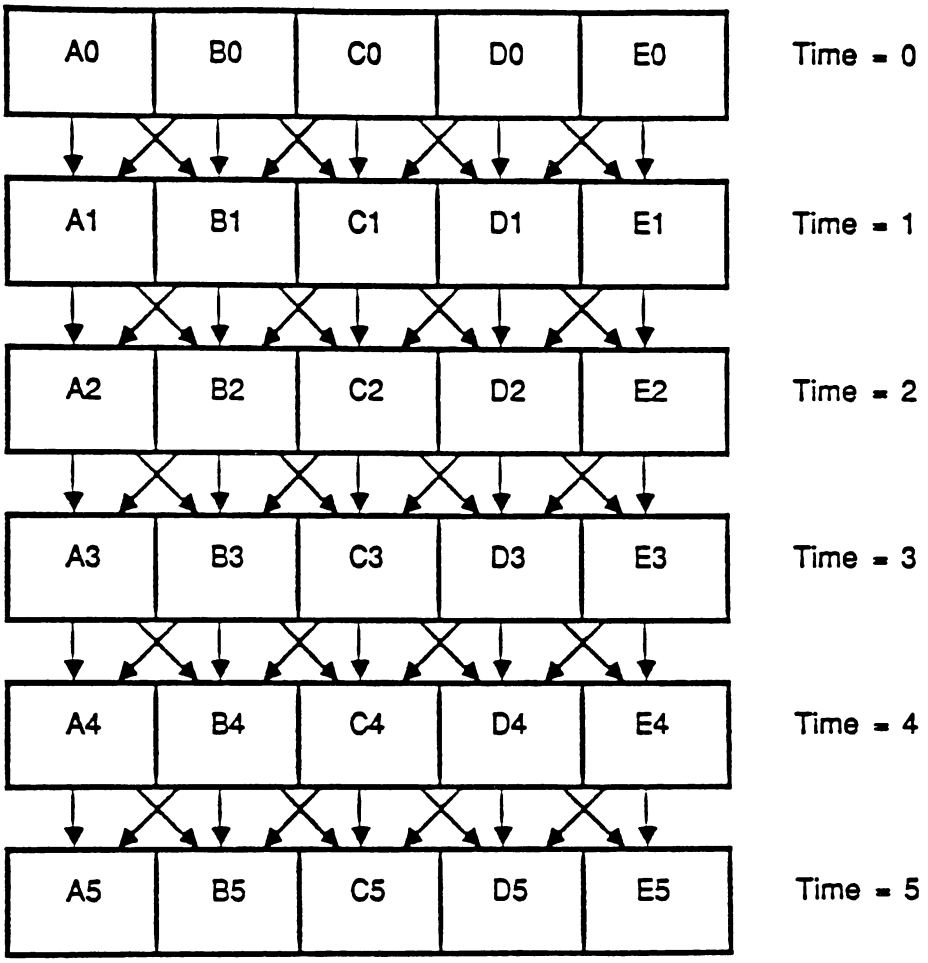

Figure 1: A Two-Way Cellular Array computing for 5 time units. Arrows indicate functional dependencies.

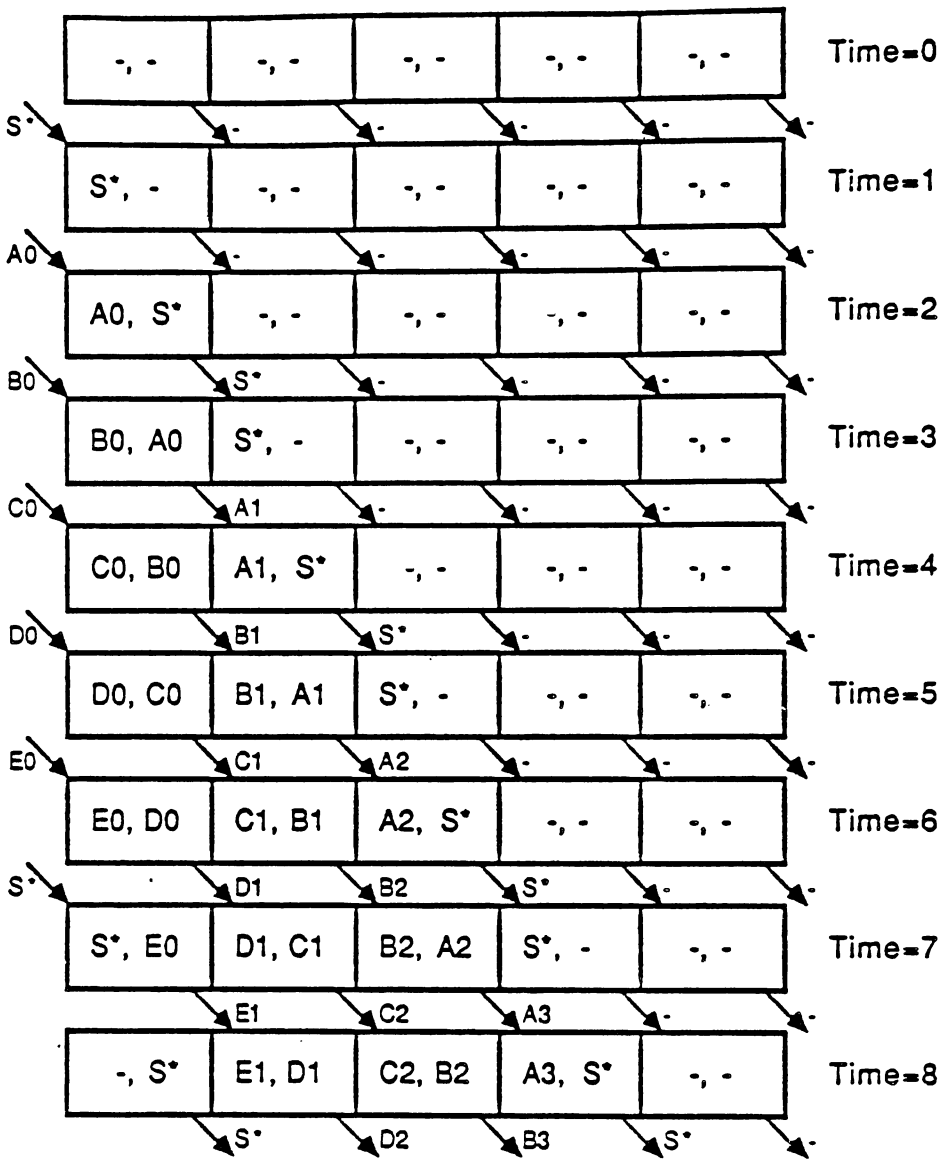

Figure 2a: A One-Way Iterative Array simulating the Two-Way Cellular Array of Figure 1. Time units 0 through 8.

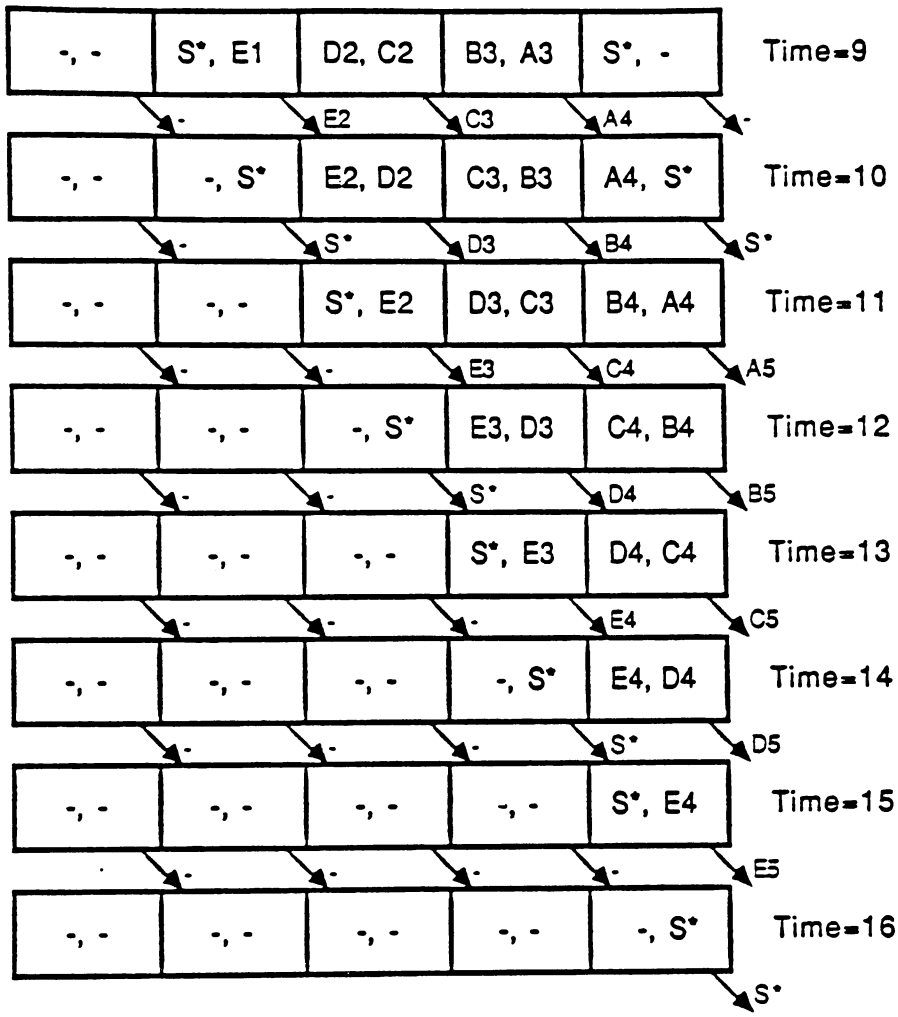

Figure 2b: A One-Way Iterative Array simulating the Two-Way Cellular Array of Figure 1. Time units 9 through 16.

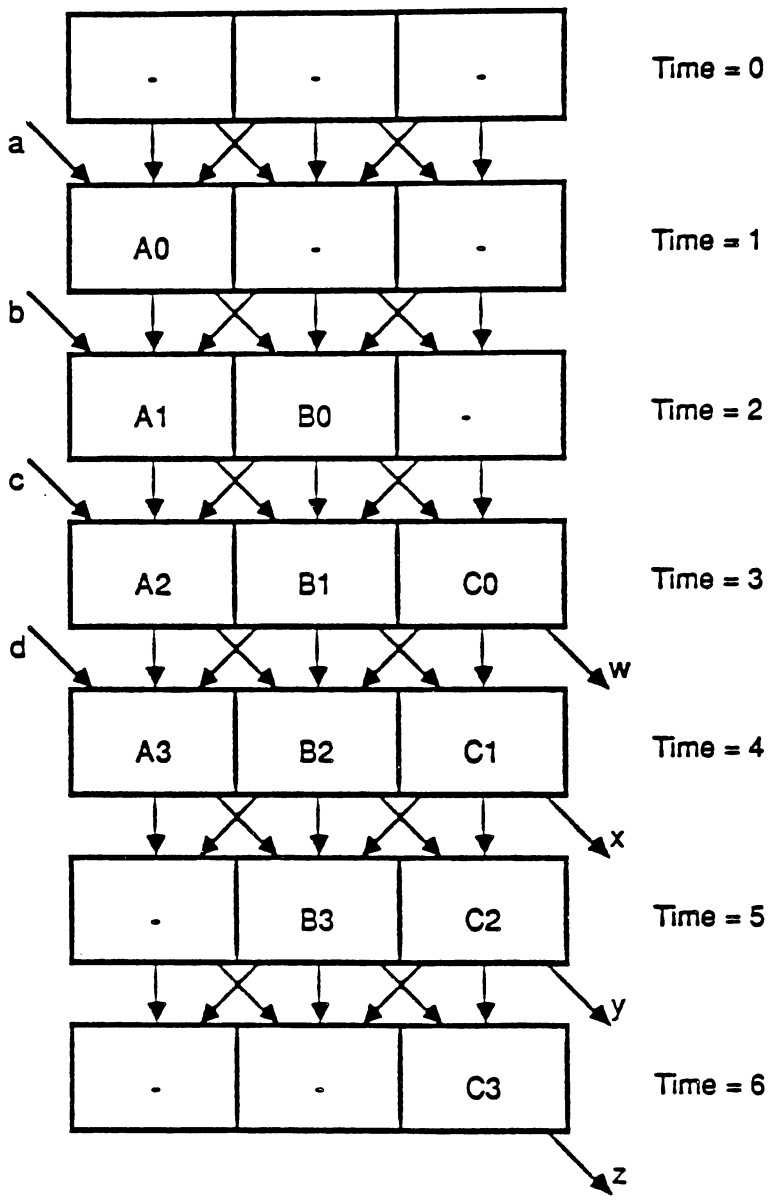

Figure 3: A Two-Way Iterative Array computing for 6 time units. Arrows indicate functional dependencies.

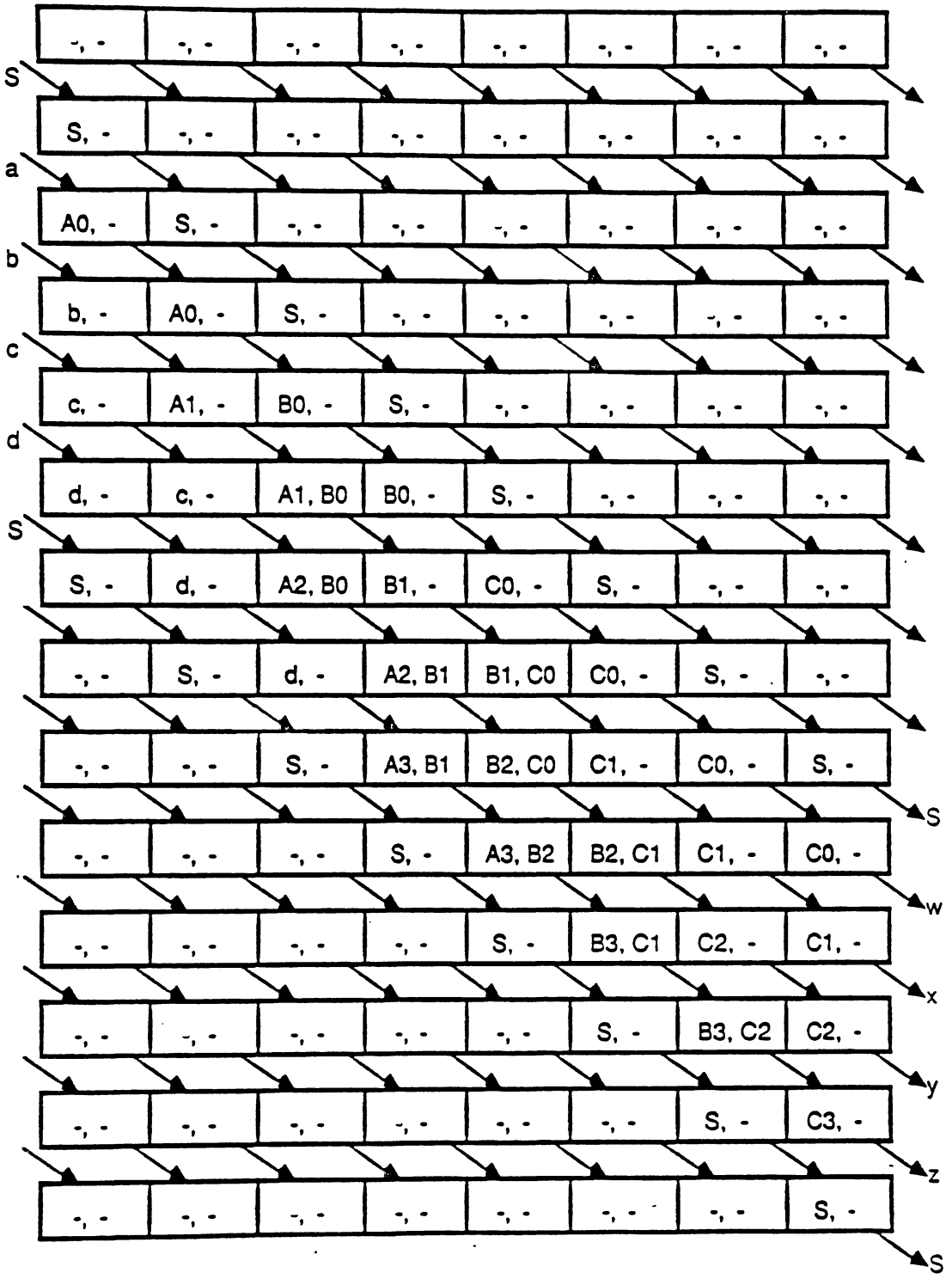

Figure 4: A One-Way Iterative Array simulating the Two-Way Iterative Array of Figure 3.

```
function h_C(A,B) returns COMPUTATION_STATE is
    if C_State(B) = OFF then
        if C_State(A) = START then
            start input track
            C_State(B) := START end if
    else if C_State(B) = START then
        C_State(B) := ON
    else if C_State(B) = ON then
        /* Stay in control state ON until the cell to the left is active (or in the special
          START state) */
        if C_State(A) = OFF or ON then
            no change
        else if C_State(A) = START then
            C_State(B) := FIRST
        else if C_State(A) = INTERMEDIATE then
            Right(B) := ∅
            Middle(B) := Middle(A)
            C_State(B) := MAIN
        else if C_State(A) = MAIN or FIRST then
            Middle(B) := ∅
            C_State(B) := INTERMEDIATE end if
    else if C_State(B) = FIRST then
        /* Perform computation as cell 0 of the two-way array. Then either become
          inactive or turn into the last cell of the computation. */
        Middle(B) := g(Cur_Input(B),Middle(B),Right(B))
        if C_State(A) = ON then
            C_State(B) := ON
        else if C_State(A) = MAIN then
            C_State(B) := LAST end if
    else if C_State(B) = LAST then
        /* Output data from previous computation (in the cell to the left). */
        initiate output track if not already active
        New_Out(B) := Middle(A)
        Middle(B) := Middle(A)
        Right(B) := ∅
        C_State(B) := MAIN
    else if C_State(B) = MAIN then
        Middle(B) := g(Middle(A),Middle(B),Right(B))
        C_State(B) := INTERMEDIATE
    else if C_State(B) = INTERMEDIATE then
        Right(B) := Middle(B)
        Middle(B) := Middle(A)
        if C_State(A) = INTERMEDIATE then
            C_State(B) := MAIN
        else if C_State(A) = ON or LAST then
            C_State(B) := FIRST end if
    end if
end h_C
```

Figure 5: Circular array algorithm.