Transforming Computations with Bi-directional Data Flow
Into ones with
Uni-directional Data Flow on Linear Systolic Arrays

Anwer Z. Kotob

Center for Communications and Signal Processing
Computer Studies Program
North Carolina State University

CCSP-TR-87/21

December 1987
Abstract

KOTOB, ANWER Z. Transforming Computations with Bi-directional Data Flow into ones with Uni-directional Data Flow on Linear Systolic Arrays. (Under the direction of Dr. Carla D. Savage.)

Uni-directional arrays appear to enjoy an advantage over their counterparts with bi-directional data flow in the areas of fault tolerance and of recyclability as an alternative to problem decomposition. This thesis presents viable methods for transforming all bi-directional linear array algorithms into uni-directional ones. The uni-directional functions produced have a space (and time) complexity that is a linear function of the size and/or the time units of operation of the bi-directional function. The complexity of an individual cell in the array is invariably increased, but only by a constant factor.

All bi-directional array algorithms are subdivided into four cases, and algorithms are presented that will transform a computation conforming to any one of the four cases into a uni-directional computation. In each case, the cell function of the original bi-directional array appears as a subroutine of the uni-directional cell function produced by the transformation.

Finally three sample applications which use a bi-directional data flow are presented, with figures showing snapshots first of their execution on bi-directional arrays and second, of their execution on the uni-directional arrays resulting from our transformation.
Chapter 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 bi-directional flow of data between adjacent cells in the array [GL, Gue86, Kun81, Kun85, Lei79, and Rog82].

The following two issues served as the primary incentives for studying ways of transforming bi-directional linear arrays into uni-directional ones.

Fault Tolerance:

Systolic arrays are a natural candidate for wafer scale integration (WSI). The facts 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" having 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: "in general, the longest interconnection between processors will be a communication bottleneck in the system." Moreover, in such a scheme there is usually a tradeoff between the utilization of live cells and the 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. This scheme also enjoys the advantage of utilizing all the live cells in a linear array, and can be generalized to two dimensional arrays. It has also been demonstrated that with slight modifications this approach could be applied to two dimensional mesh connected arrays while yielding a high utilization of the live cells [KL84].
When this 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, however, 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).

Traditionally the host computer was burdened with the task of decomposing the problem into pieces each of a size that will fit on the available hardware and then reconstructing the final result.

It has been shown however, that this is not necessary [SS87]. If the array algorithm is uni-directional, all the cells are initialized to the same state, the algorithm can tolerate an arbitrary delay between adjacent cells, and the algorithm still works if the array has more than the ideal number of cells then all that is needed is to "recycle" the output of the array the required number of times (equal to the ceiling of the ideal size desired divided by the number of available cells,) and that would produce the correct final result.
This relieves the host computer and the user from decomposition issues, and offers a uniform method for treating all decomposition/recycling tasks.

However as stated above, a requirement for this recycling scheme to work is that the array be uni-directional.

In this thesis I will describe our implementation of a system that transforms arrays with bi-directional data flow into ones that require just uni-directional data flow.

The rest of this thesis is organized as follows: In chapter 2 I will present the notion of a cell function, describe what I mean by uni-directional data flow and bi-directional data flow, and then will show the differences between functions that require uni-directional data flow and ones that require bi-directional data flow. Chapter 3 will divide all linear systolic arrays into 4 cases and will describe each case briefly. Chapters 4 through 7 will discuss in details the 4 cases of arrays and present a viable transformation for each. Chapter 8 will present an algorithm for transforming arrays of Case 2 into uni-directional systolic rings. Chapter 9 presents some sample applications, while the conclusions and remarks are presented in chapter 10.

Throughout the rest of this thesis figures will be included showing snapshots of arrays at different time units. Unless otherwise noted, in those figures, one or more hyphens are used to signify an empty register.
Chapter 2

Model of computation, cell functions, and data flow

In this thesis we will be concerned with linear systolic arrays. In such arrays the cells are arranged in a linear fashion with every cell having connections to its two neighbors (figures 2.1 through 2.4). Each cell is comprised of a processor with its own registers and memory such that a program can be downloaded into the cell's memory, and this program will control the operation of the cell. The contents of a processor's registers define the state of the cell at that time. The number of registers available to each processor is finite and is independent of the size of the array, however the size of each of the registers may depend on the size of the array or the size of the problem to be solved. Each cell in the array has the capability to read the contents of the registers of its neighboring cells, and only the "boundary" cells have the capability to do input or output. Although we describe the function of the array in terms of a program in a cell, the results apply equally well if the program is implemented in hardware.

In the previous paragraph the term "register" referred to a single physical register as defined by the hardware. However, in the rest of this thesis the term "register" will be used to refer to one or more physical registers that are logically grouped together. For example in the next few chapters the name Current _State register does not necessarily refer to a single register, but may refer to a whole set of physical registers.
The program that is stored in the processors’ memory is referred to as the “cell function”, and will describe the computation carried out by a cell in the array. It is assumed that every cell in the array has the same cell function. However, this does not preclude the possibility that different cells may actually be executing different parts of the function. If, for example, two cells were in different states then they could be executing essentially two different sub-programs. Every time unit the cell function will compute a new state for the cell. It will do so based on information stored in the state of the cell and the states of one or both of its neighboring cells. This dependency of the cell function on the state(s) of the neighboring cell(s) determines whether the array has a uni-directional data flow or a bi-directional one. If the function computes the new state depending only on the state of the cell and that of the cell to its left then the array has a uni-directional data flow (data is flowing from left to right). Whereas, if the function needs to know the states of both neighboring cells in order to compute the new state of the cell then the array has a bi-directional flow of data (data is flowing from left to right and from right to left).

Consider for example the odd-even transposition sorting algorithm [Knu73]. Here the numbers to be sorted are stored each in a different cell, and cells alternate between executing an odd step and an even step (that is, a cell executing an odd step this time unit will execute an even step in the next time unit, and vice versa). Also, a cell executing an odd step will have both its neighbors executing even steps (and vice versa). In odd steps the cell swaps values with its left neighbor if neces-
sary (that is, if the values stored in the two cells are out of order), while in even steps the cell swaps values with its right neighbor if necessary. The first cell does not have a cell to its left so it does nothing in odd steps, similarly the last cell does not have a cell to its right so it does nothing in even steps. In this example the cells that are in an odd step will be executing a different part of the program than those that are in an even step, and the first and last cells will be executing yet other parts of the program. In this array a cell needs two registers: One to store the value in, and one to tell it which step it is supposed to execute (the value in this register is changed from odd to even and vice versa every time unit). A cell will need those two registers (and only those two) regardless of the size of the array, however the size of the first register will depend on the values to be sorted. The algorithm also depends on every cell having access to the value stored in either of its neighbors, and hence requires a bi-directional flow of data in the array.

In the course of this work a software system [KSS87] was built that accepts a cell function employing bi-directional data flow and produces a new cell function that requires only uni-directional data flow, and optionally a complete Pascal program to simulate the operation of the new array. In this system the cell function was implemented as a Pascal function that accepts a number of arguments (two or three) representing the states of the cells needed for computing the new state of the cell, and will return a value representing the new state of the cell. The cell state is seldom a single value, usually it is described by a collection of values such as a Pascal record, and Pascal rules do not allow for a function to return such a value and
thus it was necessary to make the function return a pointer to the location in memory where the new state is stored. In this thesis I will not be concerned with the actual Pascal implementation (the details of which are described in [KSS87]) but will present an algorithmic representation using Pascal-like syntax.

As mentioned earlier the cell function will accept two or three arguments representing the states of the cells and will return a value that represents the new state of the current cell. In arrays with uni-directional data flow the function will accept two arguments: The second argument represents the state of the current cell, the cell whose computation is to be simulated, and the first argument represents the state of the cell to the left of the current cell. In arrays that have bi-directional data flow the function accepts three arguments: The second argument represents the state of the current cell, the first represents the state of the cell to the left of the current cell, and the third represents the state of the cell to the right of the current cell. Special cases arise with the “boundary” cells and are discussed separately for each “case” or class of arrays later.

Next I will present two examples of a cell function: The first will be an implementation of an FIR (Finite Impulse Response) filter and will demonstrate cell functions with uni-directional communication. The other will carry out the IIR (Infinite Impulse Response) filter and will demonstrate cell functions with bi-directional data flow.
A $K$-tap FIR filter that reads in a signal $x$ and produces the filtered signal $Y$

\[ Y_i = \sum_{k=0}^{K-1} (a_k x_{i-k}), \]

where $Y_i$ denotes the $i^{th}$ value of the $Y$ signal, similarly for $x$. The FIR filter described here is based on the convolution chip described in [KRY81]. In this implementation of the FIR filter every filter "tap" will constitute a cell, and thus every cell will need a register ($\text{Coeff}$) to store one of the coefficients $a_k$. Also it is necessary that the partial results travel through the array twice as fast as the values of the $x$ signal, thus the $x$ signal will have a delay register to pass through in every cell. This necessitates that every cell has a register to store the value of the signal to be passed to the right ($x\_valu$) and a delay register to store a value of the $x$ signal to be sent to the right one time unit later ($x\_delay$). The partial sum as calculated by the cell will be stored in the register $P\_Sum$. The partial sum calculated by the last cell in the array represents a value of the $Y$ signal and is sent out of the array as output. Every time unit, every cell in the array calculates a new partial sum and stores it in its $P\_Sum$ register. It does this by multiplying the value in the $x\_valu$ register of the cell to the left by the value in the cell's own $\text{Coeff}$ register and adding the result to the value in the $P\_Sum$ register of the cell to the left. Then the cell moves the value in its $x\_delay$ register into its $x\_valu$ register so that it can be read by the cell to the right, and then copies the value in the $x\_value$ register of the cell to the left into its $x\_delay$ register so that it can be sent to the cell to the right one time unit later. The first cell of the array does not have a cell to its left and thus will see a value of zero as the $x\_valu$ to its left, and
Acknowledgement

I would like to express my deepest appreciation to Dr. Carla D. Savage for her guidance, advice, extreme patience, and friendship. This work would not be finished without her candid help and encouragement. I am fortunate to have had the chance to work for Dr. Savage, and will always cherish my association with her.

Additional guidance from Dr. Matthias F. M. Stallmann is greatly appreciated as well. Special thanks to Dr. Shuaib H. Ahmad for his understanding and willingness to help.

Anwer Z. Kotob
October 1987.
# Table of Contents

1 Introduction 1

2 Model of computation, cell functions and data flow 5

3 Subdivision of bi-directional array computations into four cases 19

4 Case 1
   (Bi-directional cellular arrays → Uni-directional iterative arrays) 24

5 Case 2
   (Bi-directional iterative arrays → Uni-directional iterative arrays) 35

6 Case 3
   (Bi-directional arrays, cells initialized to different states, input at left end, output at right end → Bi-directional iterative arrays) 48

7 Case 4
   (Bi-directional arrays, input and output at left end, input and output at right end → Bi-directional arrays, input at left end, output at right end) 61

8 Circular arrays
   (Bi-directional iterative arrays → Uni-directional systolic rings) 77

9 Sample applications
   Example 1: Odd-Even Transposition Sort (Case 1) 96
   Example 2: String comparison (Case 4.2) 107
   Example 3: IIR filter (Case 3) 121

10 Concluding remarks 131

References 134
every time unit will see a new value of the \( x \) signal as the \( P_{Sum} \) to the left.

The following is how the cell function would look:

```plaintext
function FIR(A,B) returns State is
  /* This is the cell function for the FIR filter. \( B \) represents the state of the current cell, the cell whose computation is to be simulated, say Cell \( i \). \( A \) represents the state of Cell \( i-1 \). */
  result := B
  P_Sum(result) := P_Sum(A) + x_valu(A) * Coeff(B)
  x_valu(result) := x_delay(B)
  x_delay(result) := x_value(A)
  return result
end FIR
```

Figure 2.6 traces, for 6 time units, an array that implements a 3-tap FIR filter. The state of each cell is organized as shown in figure 2.5.
Figure 2.5: The cell state in the FIR program.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td>$A_1$</td>
<td>$A_2$</td>
</tr>
<tr>
<td>$x_0$</td>
<td>$x_0$</td>
<td>$x_0$</td>
</tr>
<tr>
<td>$A_0x_0$</td>
<td>$A_0x_0$</td>
<td>$A_0x_0$</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$A_0x_2$</td>
<td>$A_0x_2$</td>
<td>$A_0x_2$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>$A_0x_3$</td>
<td>$A_0x_3$</td>
<td>$A_0x_3$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$x_3$</td>
<td>$x_3$</td>
</tr>
<tr>
<td>$A_0x_4$</td>
<td>$A_0x_4$</td>
<td>$A_0x_4$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$x_4$</td>
<td>$x_4$</td>
</tr>
<tr>
<td>$A_0x_5$</td>
<td>$A_0x_5$</td>
<td>$A_0x_5$</td>
</tr>
<tr>
<td>$x_5$</td>
<td>$x_5$</td>
<td>$x_5$</td>
</tr>
</tbody>
</table>

Figure 2.6: FIR array, time units 0 - 6.
Now we will consider the IIR filter. A $K$-tap IIR filter can be represented by the following mathematical equation [CS81]:

$$
Z_n = \sum_{k=0}^{K-1} (a_k * x_{n-k}) + \sum_{k=1}^{K-1} (b_k * Z_{n-k})
$$

or by:

$$
Z_n = Y_n + \sum_{k=1}^{K-1} (b_k * Z_{n-k}),
$$

where

$$
Y_n = \sum_{k=0}^{K-1} (a_k * x_{n-k}).
$$

The calculation of the different $Y_n$'s is identical to an FIR filter (see above) and will not be further discussed. This way we can assume that the input to the filter is the signal $Y_i$ (the output of the FIR filter) and not the $x_i$. Partial sums will flow to the right, and the $Z_i$ values will flow to the left, with the last cell sending as output every $Z$ value it calculates. Every cell reads a $Z_i$ value from the cell to its right, reads a partial sum from the cell to its left, multiplies the $Z_i$ value by the coefficient stored in it and adds the result to the partial sum read, and this constitutes its new partial sum. The partial sum calculated by the last cell in the array is the $Z$ signal, this is sent as output the same time unit it is calculated and will be sent to the left in the time unit after its calculation (a delay of one time unit).

Thus every cell will need four registers: The first register, $Coeff$, will store the value of the coefficient of the cell, the second will store the partial sum, $Y_{signal}$, the third will store the results going to the left, $Z_{signal}$, and the fourth, $Position$, will be used to tell whether the cell is the last in the array or not. The cells will also have
another register that is not used except in the last cell, $Z_{\text{sig}_{\text{intr}}}$ (for intermediate Z signal), this will be used to store the Z value for an additional time unit until it is time for it to be sent to the left. The following is how the cell function would look:

```plaintext
function IIR(A, B, C) returns State is
/* This is the cell function for the IIR filter. B represents the state of the current cell, the cell whose computation is to be simulated, say Cell i. A represents the state of Cell i-1, and C represents the state of Cell i+1. */

result := B

if Position(B) = LAST then
    /* The cell is the last in the array. */
    \[ Y_{\text{signal}}(\text{result}) := Y_{\text{signal}}(A) + \text{Coeff}(B) \cdot Z_{\text{signal}}(B) \]
    \[ Z_{\text{signal}}(\text{result}) := Z_{\text{sig}_{\text{intr}}}(B) \]
    \[ Z_{\text{sig}_{\text{intr}}}(\text{result}) := Y_{\text{signal}}(\text{result}) \]
else /* The cell is not the last in the array. */
    \[ Y_{\text{signal}}(\text{result}) := Y_{\text{signal}}(A) + \text{Coeff}(B) \cdot Z_{\text{signal}}(B) \]
    \[ Z_{\text{signal}}(\text{result}) := Z_{\text{signal}}(C) \]
end if

return result
end IIR
```

Figure 2.8 shows 9 time units of an array that implements the above function. In other words, the array shown in figure 2.8 implements the recursive part of an IIR filter with 4 taps. Figure 2.7 presents the organization of the state of a cell as used in figure 2.8.
Figure 2.7: The cell state in the IIR program.
<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Y₀</th>
<th>Y₁</th>
<th>Y₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B₃</td>
<td>B₂</td>
<td>B₁</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.8a: IIR array, time units 0 - 4.
<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>Y_2</th>
<th>-----</th>
<th>Y_1+B_1Z_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>B_3</td>
<td>B_2</td>
<td>B_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Z_0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-----</td>
<td>-----</td>
<td>Z_1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y_3</th>
<th>-----</th>
<th>Y_2+B_2Z_0</th>
<th>-----</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_3</td>
<td>B_2</td>
<td>B_1</td>
<td></td>
</tr>
<tr>
<td>Z_0</td>
<td>-----</td>
<td>Z_1</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y_3+B_3Z_0</th>
<th>-----</th>
<th>Y_2+B_2Z_0+B_1Z_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_3</td>
<td>B_2</td>
<td>B_1</td>
</tr>
<tr>
<td>-----</td>
<td>Z_1</td>
<td>-----</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y_4</th>
<th>-----</th>
<th>Y_3+B_3Z_0+B_2Z_1</th>
<th>-----</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_3</td>
<td>B_2</td>
<td>B_1</td>
<td></td>
</tr>
<tr>
<td>Z_1</td>
<td>-----</td>
<td>Z_2</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y_4+B_3Z_1</th>
<th>-----</th>
<th>Y_3+B_3Z_0+B_2Z_1+B_1Z_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_3</td>
<td>B_2</td>
<td>B_1</td>
</tr>
<tr>
<td>-----</td>
<td>Z_2</td>
<td>-----</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>Z_3</td>
</tr>
</tbody>
</table>

Figure 2.8b: IIR array, time units 5 - 9.
Chapter 3

Subdivision of bi-directional array computations into four cases

In this thesis an array referred to as *cellular* has its cells initialized to different states and that constitutes all the input to the array, the result of the computation is in the final states of the cells after the array finishes computing, with no explicit output leaving the array. Also, an array referred to as *iterative* has its cells initialized to the same state, accepts input at its left end and produces output that leaves its right end.

For the purpose of transforming bi-directional linear arrays to uni-directional ones, *all* bi-directional linear systolic array algorithms were subdivided into four cases with the fourth case being further subdivided into two subcases. The following is an overview of the different cases with the details of the transformations being discussed in the next chapters.

Case 1: (Bi-directional cellular arrays.) An array conforming to Case 1 has its cells initialized to possibly different states (which constitutes all the input to the array), and the result of the computation is in the final states of the cells after the array finishes computing (no explicit output leaves the array). An example of this case would be odd-even transposition sort (discussed in chapter 2 and chapter 9), where the items to be sorted would be stored each in a different cell, and after the array finishes computing the items would be in sorted order and still each in a different cell.
Such an array will be transformed into a uni-directional iterative array where the cells would be initialized to the same state, the leftmost cell does the input, the rightmost cell does the output, and of course the flow of data is uni-directional from left to right. The result of the computation is in the output leaving the rightmost cell. The input to the array is nothing but the initial states in the original (bi-directional) program plus a start and an end marker. (In the odd-even transposition sort example the leftmost cell reads in the numbers or values to be sorted, and those values will leave as output from the right end of the array in sorted order).

Case 2: (Bi-directional iterative arrays.) In this case (both before and after the transformation) the cells of the array are initially in the same state, the leftmost cell does the input to the array, and the rightmost cell does the output of the array. The result of the computation is in the output leaving the rightmost cell. Where the original computation required bi-directional flow of data through the array, after the transformation the computation requires only uni-directional flow of data.

An example of this case would be median filtering in which the signal to be filtered is fed one value at a time into the array through the array's leftmost cell, and the resulting (filtered) signal leaves as output from the rightmost cell again, one value at a time.
Case 3: (Bi-directional arrays, cells initialized to different states, input at left end, output at right end.) An example of this case would be the infinite impulse response (IIR) filter (discussed in chapter 2 and chapter 9). In this case the cells of the array are originally initialized to (possibly) different states, (in the IIR filter example the coefficients of the filter are placed each in a different cell), the leftmost cell does the input to the array, the rightmost cell does the output of the array, and the result is in the output leaving the rightmost cell. (In the IIR filter example the signal to be filtered is read one value at a time by the leftmost cell of the array. The filtered signal leaves as output one value at a time from the rightmost cell of the array).

After the transformation, the cells of the array are initialized to the same state, the leftmost cell does the input of the array, and the rightmost cell does the output with the result of the computation being in the output leaving the rightmost cell. The initial states of the cells in the original array are read as the first part of the input, (the details of the transformation will be discussed in the chapter 6). After this transformation the computation is still bi-directional, but is now reduced to what we call Case 2. Hence, from this point on, any further transformations are done as discussed for Case 2.
Case 4: (Bi-directional arrays, input and output at left end, input and output at right end.) The fourth case is one in which data is input at both the leftmost cell and the rightmost cell, the computation requires a bi-directional flow of data, and output leaves both the leftmost cell and the rightmost cell. The result is in the output leaving either or both ends.

The first transformation done in this case is to transform it into an array that reads input through its leftmost cell only and outputs results through its rightmost cell only. This results in either Case 2 or Case 3, depending on whether or not the cells are initialized to the same state. The first transformation is done by folding the array onto itself, thus making each cell in the resulting array do the job of two cells of the original array with the possible exception of the very last cell (of the new array). The last cell in the new array requires special attention: if the number of cells in the original array was even then the last cell does the job of two cells of the original array, however if the number of cells in the original array was odd, then the last cell in the new array will do the job of but one cell of the original array. What has been described so far makes the leftmost cell do the input and the output, and thus the output has to be routed on a special output track so it will go out of the rightmost cell.

As mentioned earlier, this case is subdivided into two sub-cases that we call Case 4.2 and Case 4.3.
Case 4.2  An example of this case would be the calculation of the edit length (in terms of delete/insert operations) between two strings. In this case the cells of the array are all initialized to the same state. This implies that the above transformation will transform (reduce) the problem to what we call Case 2 (and so the name 4.2) and hence any further transformations are done as is discussed for Case 2.

Case 4.3: This case is similar to Case 4.2 except that the cells of the original array are originally initialized to (possibly) different states. After the above transformation is applied this case is reduced to what we call Case 3 (which gives it the name 4.3) and any further transformations are done as is discussed for Case 3.
Chapter 4

Case 1

(Bi-directional cellular arrays – Uni-directional iterative arrays.)

As mentioned in the previous chapter, an array of Case 1 ($A_1$) has its cells initialized to different states, and that constitutes all the input to the array. After $A_1$ finishes computing, the result is in the final states of the cells with no explicit output leaving the array. The goal is to transform such a computation into one that requires only uni-directional data flow, accepts input at the left end of the array and produces output that leaves the right end.

We assume the existence of two imaginary cells in $A_1$, Cell 0 and Cell $n + 1$, with Cell 0 containing a special state called $x_{left}$ and Cell $n + 1$ containing a special state called $x_{right}$. (The states $x_{left}$ and $x_{right}$ could be the same state, we differentiate between them here for the sake of convenience and clarity). This way the first cell knows that it is the first cell of $A_1$ (Cell 1) by detecting a state of $x_{left}$ in the cell to its left (Cell 0), and similarly the last cell of $A_1$ (Cell $n$) knows that it is the last cell, at least in the bi-directional array, by detecting a state of $x_{right}$ in the cell to its right (Cell $n + 1$).

Assuming that we are given a cell function $f_1$ of the bi-directional array $A_1$, we can develop a new cell function $g_A$ that will implement the same computation, except on a uni-directional array ($A_A$). $A_A$ would accept as input the initial states of the cells in $A_1$ and produce as output the final states of the cells in $A_1$. The input is accepted at the left end of $A_A$ and the output leaves the right end of
it. Assuming that the initial states of the cells of $Ar_1$ were $S_1, S_2, \ldots, S_n$, then the input to $Ar_A$ would be: $x_{left}, S_1, S_2, \ldots, S_n, x_{right}$. Each cell in $Ar_A$ holds three values named $Current\_State$, $Left\_State$, and $Output$, each of which is a cell state of $Ar_1$.

Assume that at some time unit Cell $j$ of $Ar_A$ is simulating the computation of Cell $k$ of $Ar_1$, in this case the values stored in Cell $j$ of $Ar_A$ are as follows: $Current\_State$ represents the state of Cell $k$ of $Ar_1$. $Left\_State$ represents the state that would have been stored in Cell $k-1$ of $Ar_1$. The state of Cell $k+1$ of $Ar_1$ will be input to cell $j$ of $Ar_A$. And $Output$ will store the result of the computation, which represents the state of Cell $k$ of $Ar_1$ at the end of the time unit. This value will be read and used by Cell $j+1$ of $Ar_A$ in the following time unit.

Figure 4.1 shows an array of type $Ar_1$ in which the cells are initialized to $A_0, B_0, C_0, D_0, \text{and } E_0$, and after five time units the cells are left in states $A_5, B_5, C_5, D_5, \text{and } E_5$. Figures 4.3a, 4.3b, and 4.3c show the same computation as the one in Figure 4.1 being simulated on a uni-directional array ($Ar_A$). Figure 4.2 shows the organization of the cell state as used in figure 4.3. Note that in the uni-directional array Cell $i$, for $i = 1, 2, \ldots, 5$, computes and outputs $A_i, B_i, C_i, D_i,$ and $E_i$ at time units $2i+1, 2i+2, 2i+3, 2i+4, \text{and } 2i+5$, respectively. In other words, all the computations that were carried on during time unit $i$ in $Ar_1$ (Cells 1...$n$) will be done in Cell $i$ of $Ar_A$ during time units $2i+1$ through $2i+n$. From that it can be seen that if $Ar_1$ had $n$ cells and computed for $t$ time units then $Ar_A$ will need $t$ cells and will compute for $2t+n+1$ time units.
Figure 4.1: Ar with bi-directional data flow.

Arrows show functional dependencies.
Figure 4.2: The cell state of the transformed array as used in figure 4.3.
Figure 4.3a: Uni-directional array that carries out the same computation as the one in Figure 4.1. Arrows show functional dependencies.

XL stands for xleft.
Figure 4.3b: Time units 5 - 10.
XR stands for xright.
<table>
<thead>
<tr>
<th>Time</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>XR</td>
<td>C3</td>
<td>D3</td>
<td>A4</td>
<td>B4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>E3</td>
<td>ON</td>
<td>C4</td>
<td>ON</td>
<td>A5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>E3</td>
<td>B4</td>
<td>C4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>XR</td>
<td>ON</td>
<td>D4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E3</td>
<td>XR</td>
<td>C4</td>
<td>D4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>E4</td>
<td>ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>E4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>XR</td>
<td>ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td>XR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>E5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3c: Time units 11 - 16.
Assume that the cell state for the bi-directional array $A_{r_1}$ is of type $State_1$, and that the cell state for the uni-directional array $A_{r_A}$ is of type $StateA$. $StateA$ has three fields: $Current\_State$, $Left\_State$, and $Output$, each of which is of type $State_1$. The new cell function $g_A$ accepts as input two arguments of type $StateA$ the second of which represents the state of the current cell, and the first of which represents the state of the cell to the left of the current cell. It returns a value of type $StateA$ which represents the new state of the current cell. Basically, array $A_{r_A}$ functions as follows: ignoring the boundary cases (discussed later) a cell would be storing two values: In the $Current\_State$ register it would hold the state of the cell of $A_{r_1}$ whose computation is to be simulated, say Cell $i$. In the $Left\_State$ register it would hold the state of Cell $i-1$ of $A_{r_1}$. At the beginning of the time unit the cell would read from the $Output$ register of the cell to its left the state of Cell $i+1$ of $A_{r_1}$. Now the cell knows the states of Cells $i-1$, $i$ and $i+1$ of $A_{r_1}$ which allows it to call $f_1$ (the cell function of $A_{r_1}$) with those arguments. $f_1$ will return the new state of Cell $i$ and that is sent as output to the cell to the right. After that the cell moves the old state of Cell $i$ of $A_{r_1}$ from its $Current\_State$ register into its $Left\_State$ register, and copies the state of Cell $i+1$ from the $Output$ register of the cell to the left into its $Current\_State$ register, and is ready to simulate the computation of Cell $i+1$ of $A_{r_1}$.

The following is a more detailed description of the new cell function $g_A$ that includes the code to tend to the boundary cases.
(1) When a cell finds (reads) a value of $x_{\text{left}}$ in the Output register of the cell to its left, it will, except as detailed in (2) below, turn ON and is ready to prepare to simulate the computation which was performed in the first cell (Cell1) of $Ar_1$. It stores a value of $x_{\text{left}}$ in its Current_State register and will initialize its Left_State and Output registers.

(2) If the cell has a value of $x_{\text{right}}$ stored in its Current_State register, it outputs a value of $x_{\text{right}}$ to the cell to its right signaling the end of the current input stream. Then it checks to see if there is a value of $x_{\text{left}}$ in the Output register of the cell to its left, this allows for the pipelining of individual input streams through the array $Ar_A$. If a value of $x_{\text{left}}$ was indeed found in the cell to the left, then the cell would initialize its Left_State register and store a value of $x_{\text{left}}$ in its Current_State register, at which point it is ready to prepare to simulate a computation that was performed in the first cell (Cell1) of $Ar_1$. Otherwise, it just turns OFF.

(3) If the cell has a value of $x_{\text{left}}$ stored in its Current_State register then it will prepare to simulate a computation that was performed by the first cell of $Ar_1$. It copies a value of $x_{\text{left}}$ into its Left_State and its Output registers, and copies the contents of the Output register of the cell to its left into its Current_State register. By storing a value of $x_{\text{left}}$ in the Left_State register, when the function $f_1$ is called in the following time unit (step 4, below) it will be sent a value of $x_{\text{left}}$ as the state of the cell to the left and hence the computation of the first cell of $Ar_1$ will be simulated. Also, by storing $x_{\text{left}}$ in the Output
register, the cell on the right will read it and start itself in the following time unit (step 1 above).

(4) If none of the above conditions holds, then the cell is simulating the computation that was performed by a cell of $A_r_A$. Thus it calls the cell function $f_1$ (of $A_r_A$) as follows:

$$f_1 (\text{Left\_State}, \text{Current\_State}, \text{Output of cell on the left}).$$

It would store the result returned by $f_1$ in its $\text{Output}$ register so that it (the result) can be used by the cell to the right. Then it should move the contents of its $\text{Current\_State}$ register into its $\text{Left\_State}$ register, and the contents of the $\text{Output}$ register of the cell on its left to its $\text{Current\_State}$ register. This way a cell in time step $i$ of $A_r_A$ that simulated the computation of $\text{Cell} \ j$ in time unit $k$ of $A_r_A$ will simulate the computation of $\text{Cell} \ j + 1$ in time unit $k$ of $A_r_A$ during the following time step (i.e. time step $i + 1$ of $A_r_A$).

The following is an example of how the new cell function would look:
function $g_A(A,B)$ returns StateA is  
	result := B  
	if $Output(A) = x_{\text{left}}$ and $Current\_State(B) \neq x_{\text{right}}$ then  
		/* This is step (1) above. The cell should turn ON and get ready to  
		simulate the computation of the first cell of $Ar_1$. */  
	
turn ON  
	$Output(result) := \emptyset$  
	$Left\_State(result) := \emptyset$  
	$Current\_State(result) := x_{\text{left}}$  
	else  
		if $Current\_State(B) = x_{\text{right}}$ then  
			/* This is step (2) above. The cell has (in the previous time unit)  
		
simulated the computation of the last cell of $Ar_1$ and is ready  
		
to turn OFF. */  
			$Output(result) := x_{\text{right}}$  
		
turn OFF  
		else  
			/* If input streams are pipelined then the cell should get  
		
turn OFF ready to simulate the computation of the first cell again. */  
			$Left\_State(result) := \emptyset$  
			$Current\_State(result) := x_{\text{left}}$  
		end if  
	else  
		/* This is step (3) above. The cell will prepare to simulate the  
		function of the first cell of $Ar_1$. It will do so by storing a value  
	
t of $x_{\text{left}}$ in its $Left\_State$ register. */  
		$Left\_State(result) := x_{\text{left}}$  
		$Current\_State(result) := Output(A)$  
		$Output(result) := x_{\text{left}}$  
		else  
			/* This is step (4) above. The cell will call the cell function of  
		$Ar_1$ and will store the value returned by it in its $Output$ re-  
	
gister and then will perform the data movement operations  
	
ecessary for operation in the time unit to come. */  
			$Output(result) := f_1(Left\_State(B),Current\_State(B),Output(A))$  
			$Left\_State(result) := Current\_State(B)$  
			$Current\_State(result) := Output(A)$  
		end if  
	end if  
	return result  
end $g_A$
Chapter 5
Case 2

(Bi-directional iterative arrays → Uni-directional iterative arrays.)

An array conforming to Case 2 ($Ar_2$) has all its cells initialized to the same state ($S_0$), accepts input at its left end (Cell 1) and produces output that leaves its right end (Cell n), and the computation employed requires bi-directional flow of data in the array. The result of the computation is in the output leaving the right end. The goal is to transform such a computation into one that accepts input at its left end, produces output that leaves the right end, but the computation would require only uni-directional flow of data. Given a cell function $f_2$ for $Ar_2$, a cell function $g_A$ can be developed that will implement the same computation except on a uni-directional array ($Ar_A$).

Every time step a cell in $Ar_2$ would calculate a new state for itself based on its own state and the states of its two neighboring cells. This is done by calling the function $f_2$ which accepts three arguments: The second argument represents the state of the current cell, say Cell i of $Ar_2$. The first argument represents the state of Cell i-1, and the third argument represents the state of Cell i+1. This is true for all the cells except for the first cell (Cell 1) and the last cell (Cell n). Cell 1 does not have a cell to its left, and so the first argument of $f_2$ in this case represents the input to the array at that time unit. Cell n does not have a cell to its right and so for the third argument, $f_2$ is sent a cell state initialized to $S_0$ (the initial cell state for all the cells). For convenience it is assumed that the input to the array is of the
same type as a cell's state. A preprocessor could easily transform raw input into the form desired. In figures 5.2 and 5.3 hyphens are used to indicate registers set to $S_0$. Figure 5.2 shows an array of type $A_{r_2}$ that accepts $a$, $b$, $c$, $d$, $e$, and $f$ as input, and produces $C_0$, $C_1$, $C_2$, and $C_3$ as output.

Figure 5.3 shows the same computation as the one in Figure 5.2 except on a uni-directional array ($A_{r_A}$). Each cell in $A_{r_A}$ holds the following information: Current_State, Right_State, Control_State, and Position. The organization of the cell state as used in figure 5.3 is shown in figure 5.1. Current_State is of the type used to describe the state of a cell of $A_{r_2}$ and holds the state of the cell of $A_{r_2}$ whose computation will be simulated, say Cell $i$ of $A_{r_2}$. Right_State is also of the type used to describe the state of a cell of $A_{r_2}$ and it holds the state of Cell $i+1$ of $A_{r_2}$, the cell to the right of the cell whose computation will be simulated. Control_State holds the control information necessary for the operation of $A_{r_A}$ and the different control states will be introduced as needed below. Position is capable of holding an integer and will be used only with the startup sequence as described below. Every time step a cell would compute a new state for itself depending on its own state and the state of the cell to its left, except in the case of the first cell which would calculate its new state based on its own state and the input to the array at that time unit. Initially, the cells of $A_{r_A}$ have their Control_State registers set to OFF, and their Current_State, and Right_State registers set to $S_0$ (the initial state for the cells of $A_{r_2}$).
Figure 5.1: The cell state of the transformed array as used in figure 5.3.

Figure 5.2: Ar\textsubscript{2} (with bi-directional data flow).
Arrows show functional dependencies.
Figure 5.3a: Array with uni-directional data flow that simulates the computation of the array of figure 5.2. Time units 0 through 6. Arrows show functional dependencies.
Figure 5.3b: Array with uni-directional data flow that simulates the computation of the array of figure 5.2. Time units 7 through 12.
A time step in $Ar_2$ is simulated by two time steps in $Ar_A$, a MAIN step followed by an INTERMEDIATE step. The computation is done in the MAIN step with the INTERMEDIATE step being used for data routing purposes only. Input is collected every time unit (on MAIN as well as INTERMEDIATE steps), and is read into the Current_State register of an initialized cell state. The very first input item read is given a Control_State of MAIN, while subsequent items are given Control_State values of OLD. Assuming that the $Ar_2$ array computed for $t$ time units during which it received the sequence $S_1$, $S_2$, ..., $S_t$ as input, then the input to $Ar_A$ would be START, $S_1$, $S_2$, ..., $S_t$, OFF. The special marker START is implemented as a cell state with its Current_State and Right_State registers initialized to $S_0$ and its Control_State set to “START”. Similarly, the OFF marker is implemented as a cell state with its Current_State and Right_State registers initialized to $S_0$ and its Control_State register set to “OFF”. If we assume that Cell 1 sees the input as the contents of an imaginary cell to its left, then the different $S_i$ values will be placed in the Current_State register of the imaginary cell, while the Right_State register of that imaginary cell would always be set to $S_0$. In the above scheme $S_1$ will have a Control_State of MAIN while the others will have a Control_State of OLD. A computation that took place in Cell $i$ at time unit $t$ of $Ar_2$ will be done in Cell $t + i - 1$ at time unit $2t$ of $Ar_A$. From the above it can be seen that if $Ar_2$ had $n$ cells and computed for $t$ time units then $Ar_A$ would need $n + t - 1$ cells and will compute for $2t$ time units.
The new cell function $g_A$ accepts two arguments, the second of which represents the state of the current cell and the first argument represents the state of the cell immediately to the left of the current cell. (In the case of the first cell the first argument represents the input to the array at that time unit). The value returned by $g_A$ represents the new state of the current cell. Ignoring the boundary cases, a cell in $Ar_A$ holds two values: In $Current\_State$ it holds the state of the cell to be simulated, say Cell $i$ of $Ar_2$. In $Right\_State$ it would hold the state of Cell $i+1$. If the cell is in control state $MAIN$ then it reads the state of Cell $i-1$ of $Ar_2$ from the $Current\_State$ register of the cell to its left. Now the cell knows the states of Cell $i-1$, Cell $i$, and Cell $i+1$, which allows it to call $f_2$, the cell function of $Ar_2$, and that returns the new state of Cell $i$ which is copied into the $Current\_State$ register of the cell and the cell changes its $Control\_State$ to $INTERMEDIATE$. If the cell is in control state $INTERMEDIATE$ then it moves the contents of its $Current\_State$ register into its $Right\_State$ register, and copies the contents of the $Current\_State$ register of the cell to its left into its own $Current\_State$ register and changes its control state to $MAIN$, this way if the cell simulated the computation of Cell $i$ at time unit $t$ of $Ar_2$ in the last time unit it will simulate the operation of Cell $i-1$ at time unit $t+1$ of $Ar_2$ in the next time unit.

As mentioned earlier the first item of input to $Ar_A$ is a $START$ marker, which is implemented as an initialized cell state with its $Control\_State$ set to $START$, and its $Position$ register set to an integer equal to twice the number of cells in $Ar_2$. The computation is basically "shifted to the right by one cell" every
other time unit. Ignoring the boundary conditions, the array has \( n \) cells actively involved in the computation, with the cells to the left of the computation being termed as *old* and have the duty of passing input to the right until it reaches the computation. The cells to the right of the computation are termed as *passive* and their duty is to pass the output to the right until it reaches the last cell. As the computation is shifted to the right, the leftmost *Passive* cell is *shifted in* and becomes actively involved in the computation, and the leftmost cell that was involved in the computation is *shifted out* and becomes *old*. The following is a more detailed and fuller description of the new cell function \( g_A \) that accounts for the boundary cases. (In the description below, to *initialize* a register means to set it to \( S_0 \).) Every time step of \( Ar_A \) one of the following will be carried on:

1. If the cell on the left has a *Control State* of *START* then the current cell should initialize its *Current State* and *Right State*, and set its *Control State* to *START*. If the integer in the *Position* register of the cell to the left is greater than zero then the cell will set its *Position* to one less than the value stored in the *Position* register of the cell to the left, otherwise it should be set to zero.

2. The following will be done if the cell has a *Control State* of *START*. If the *Position* register has a zero stored in it then the cell copies the *Current State* of the cell to the left into its own *Current State* register, and if the cell to the left is in control state *INTERMEDIATE* then the cell sets its *Control State* register to *MAIN* otherwise it sets its *Control State* to *PASSIVE*. If the Posi-
tion register has a positive integer stored in it and the cell to the left is in control state *MAIN* then the cell will simulate the action of a cell in $A_{r_2}$ and will call $f_2$ sending to it the *Current_State* of the cell to the left as the first argument, its own *Current_State* as the second argument, and its *Right_State* as the third argument. The value returned by $f_2$ will be stored in the *Current_State* register and the cell will set its *Control_State* to *INTERMEDIATE*. If the *Position* register has a positive integer stored in it and the cell to the left has a control state of *INTERMEDIATE* then the cell will copy the contents of the *Current_State* register of the cell to the left into its own *Current_State* register and will set its *Control_State* to *MAIN*.

(3) If the cell is in control state *INTERMEDIATE* then (unless the cell to the left is in control state *OFF*) it copies the contents of its *Current_State* register into its *Right_State* register, and the contents of the *Current_State* register of the cell to the left into its own *Current_State* register. If the cell to the left is in control state *INTERMEDIATE* then the cell will go into control state *MAIN* otherwise, it will copy the *Control_State* of the cell to the left into its own *Control_State* (this is how cells change from being actively involved in the computation into ones that are *old*). However, if the cell to the left is in control state *OFF* then the cell would initialize its *Current_State* and *Right_State* registers and sets its *Control_State* to *OFF* (the cell turns itself off).
(4) If the cell is in control state MAIN then (unless the cell to the left is in control state OFF) it reads the Current_State register of the cell to the left. This way it knows the states of Cell i-1, Cell i, and Cell i+1 of Ar₂ and that allows it to call $f_2$ (the cell function of Ar₂). $f_2$ returns the new state of Cell i of Ar₂ which the cell will store in its Current_State register. Then the cell will set its Control_State register to INTERMEDIATE. However, if the cell to the left is in control state OFF then the cell would initialize its Current_State and Right_State registers and sets its Control_State to OFF (the cell turns itself off).

(5) If the cell is in control state OLD then it copies the contents of the Current_State register of the cell to the left into its own Current_State register, initializes its Right_State register, and then copies the value of the Control_State register of the cell to the left into its own Control_State register (this is one way that cells can turn off).

(6) If the cell has a Control_State of PASSIVE then it should copy the contents of the Current_State register of the cell to the left into its own Current_State register. If the cell to the left is in control state INTERMEDIATE then the cell should set its Control_State register to MAIN (the cell is being shifted in), otherwise if the cell to the left is in control state OFF then the cell sets its Control_State register to OFF, otherwise it stays in control state PASSIVE.
(7) If the cell is in control state \textit{OFF} and the cell to the left is \textit{not} in control state \textit{START} then the cell does nothing and its state stays unchanged. (If the cell to the left is in control state \textit{START} then step (1) would be executed.)

The following is an example of how the new cell function would look:

```
function g_A(A, B) returns StateA is
  result := B
  if Control_State(A) = START then
    /* If the cell to the left is in control state \textit{START} then the current cell should initialise itself and get ready to compute. (Step 1 above). */
    Current_State(result) := S_0
    Right_State(result) := S_0
    Control_State(result) := START
    if Position(A) > 0 then
      Position(result) := Position(A) - 1
    else
      Position(result) := 0
    end if
  end if
  if Control_State(B) = START then
    /* This is step 2 above. */
    if Position(B) > 0 then
      /* In this case the cell will be actively involved in the computation in the next time unit. In this time unit it either does a computation and goes into a control state of \textit{INTERMEDIATE} or it copies the contents of the \textit{Current_State} register of the cell to the left into its own \textit{Current_State} register and goes into a control state of \textit{MAIN}. */
      if Control_State(A) = MAIN then
        Current_State(result) := f_2(Current_State(A), Current_State(B), Right_State(B))
        Control_State(result) := INTERMEDIATE
      else
        Current_State(result) := Current_State(left)
        Control_State(result) := MAIN
      end if
    else
      /* In this case it copies the contents of the \textit{Current_State} register of the cell to the left into its own \textit{Current_State} register. Then it goes into \textit{MAIN} or \textit{PASSIVE} control state depending on the contents of the \textit{Control_State} register of the cell to the left. */
      Current_State(result) := Current_State(A)
      if Control_State(A) = INTERMEDIATE then
        Control_State(result) := MAIN
      else
        Control_State(result) := PASSIVE
      end if
    end if
  else
    /* In this case it copies the contents of the \textit{Current_State} register of the cell to the left into its own \textit{Current_State} register. Then it goes into \textit{MAIN} or \textit{PASSIVE} control state depending on the contents of the \textit{Control_State} register of the cell to the left. */
    Current_State(result) := Current_State(A)
    if Control_State(A) = INTERMEDIATE then
      Control_State(result) := MAIN
    else
      Control_State(result) := PASSIVE
    end if
  end if
```
end if
else if $Control\_State(B) = \text{INTERMEDIATE}$ then

\*/ This is step 3 above. The cell should move the contents of its $Current\_State$ register into its $Right\_State$ register, and should copy the contents of the $Current\_State$ register of the cell to the left into its $Current\_State$ register. The contents of the cells $Control\_State$ register depends on the contents of the $Control\_State$ register of the cell to the left. Unless the cell to the left ($A$) is in control state $\text{OFF}$, in which case the cell should initialize its $Current\_State$ and $Right\_State$ registers and set its $Control\_State$ to $\text{OFF}$. */

if $Control\_State(A) \neq \text{OFF}$ then

$Right\_State(result) := Current\_State(B)$
$Current\_State(result) := Current\_State(A)$

if $Control\_State(A) = \text{INTERMEDIATE}$ then

$Control\_State(result) := \text{MAIN}$
else

$Control\_State(result) := Control\_State(A)$
end if
else /* $Control\_State(A) = \text{OFF}$ */

$Right\_State(result) := S_0$
$Current\_State(result) := S_0$
$Control\_State(result) := \text{OFF}$
end if
else if $Control\_State(B) = \text{MAIN}$ then

\*/ This is case 4 above. The cell should call $f_2$ to calculate a new value for its $Current\_State$ register and then should go into control state $\text{INTERMEDIATE}$. Unless the cell to the left ($A$) is in control state $\text{OFF}$, in which case the cell should initialize its $Current\_State$ and $Right\_State$ registers and set its $Control\_State$ to $\text{OFF}$. */

if $Control\_State(A) \neq \text{OFF}$ then

$Current\_State(result) := f_2(Current\_State(A),Current\_State(B),$
$Right\_State(B))$
$Control\_State(result) := \text{INTERMEDIATE}$
else /* $Control\_State(A) = \text{OFF}$ */

$Current\_State(result) := S_0$
$Right\_State(result) := S_0$
$Control\_State(result) := \text{OFF}$
end if
else if $Control\_State(B) = \text{OLD}$ then

\*/ This is case 5 above. The cell should copy the contents of the $Current\_State$ and the $Control\_State$ registers of the cell to the left into its own respective registers. */

$Current\_State(result) := Current\_State(A)$
$Right\_State(result) := S_0$
$Control\_State(result) := Control\_State(A)$
else if $Control\_State(B) = \text{PASSIVE}$ then

\*/ This is case 6 above. The cell should copy the contents of the $Current\_State$ register of the cell to the left into its own $Current\_State$ register. The contents of the cell's $Control\_State$ register depends on the contents of the $Control\_State$ register of the cell to the left. */

$Current\_State(result) := Current\_State(A)$
if $Control\_State(A) = \text{INTERMEDIATE}$ then

$Control\_State(result) := \text{MAIN}$
else if $Control\_State(A) = \text{OFF}$ then
\begin{verbatim}
Control_State(result) := OFF
else
   Control_State(result) := PASSIVE
end if
else if Control_State(B) = OFF then
   /* This is case 7 above. The cell is assumed \textit{off} and hence should do nothing. In other words its contents (state) should not be changed. */
   do nothing
end if
return result
end g_A
\end{verbatim}
Chapter 6

Case 3

(Bi-directional arrays, cells initialized to different states, input at left end, output at right end → Bi-directional iterative arrays.)

In this chapter we will discuss arrays that conform to Case 3, and present a scheme for transforming all such arrays into ones that conform to Case 2. An array of Case 3 ($Ar_3$) has its cells initialized to different states, accepts input at its left end (Cell 1), produces output that leaves its right end (Cell n), and carries out a computation that requires bi-directional data flow. The goal is to transform it into an array, $Ar_2$, that has all its cells initialized to the same state, accepts input at the left end, and produces output that leaves the right end. The computation in the resulting array still requires bi-directional data flow, but, since the array now conforms to Case 2, we can apply the transformations presented in chapters 5 and 8 to transform it into one that requires only uni-directional data flow.

Assuming we are given $f_3$, the cell function of $Ar_3$, we can develop $f_2$, a cell function that will carry out the same computation as $f_3$, except will operate on an array of type $Ar_2$. The function $f_3$ accepts three parameters: The second parameter represents the state of the cell whose computation is to be simulated, the current cell. The first parameter represents the state of the cell immediately to the left of the current cell, and the third parameter represents the state of the cell immediately to the right of the current cell. A cell in $Ar_3$ would call $f_3$ every time unit with the correct arguments and $f_3$ would return the new state of that cell;
this is true for all the cells except the two boundary cells. Cell 1 does not have a cell to its left, and so for the first argument it sends $f_3$ the input to the array at that time unit. The last cell in the array (Cell $n$) does not have a cell to its right, and so for the third argument it sends $f_3$ an empty cell state. The same applies to the cells of the resulting $Ar_2$ array which call the function $f_2$.

A cell in $Ar_2$ would need five registers to hold its state (see figure 6.1). Four of those registers, namely Current, Right, Out, and Pass, are each of the type used to describe the state of a cell in $Ar_3$. The register Current holds the state of the current cell: the cell whose computation is or will be simulated. The registers Right, and Pass are used only during the startup sequence. The register Right holds the state of the cell that was immediately to the right of the current cell in $Ar_3$. The register Pass is used to store cell states (of the type defined for $Ar_3$) being passed to the right to be used by the other cells. The register Out holds the output of the array generated during the startup sequence as it is being passed to the right until it reaches the output port in the last cell of the array. Out is used only during the startup sequence except the Out register in the last cell of the array (Cell $n$) which at any time unit $t$ will hold the output of the array at that time unit. The last register is called $C\_State$ and is used to hold the control information needed for the operation of $Ar_2$. The control states (values possible for $C\_State$) will be presented below. Initially, all the cells have their $C\_State$ registers set to OFF, and the rest of their registers initialized to NULL.
The basic idea is to implement a transformation similar to the one presented in chapter 4, but only until the computation can be distributed over \( n \) cells after which a cell in \( Ar_2 \) carries out its computation exactly as a cell in \( Ar_3 \) would. In other words, Cell \( i \) of \( Ar_2 \) will carry out the computation done during time unit \( i \) in Cell \( n \) through Cell \( i+1 \) of \( Ar_3 \) as described in chapter 4 (It computes each new state based on the Out register of the cell to the left (the input in case of Cell 1), its Current register, and its Right register, and stores the result in its Out register, then it copies the contents of its Current register into its Right register, and the contents of the Out register of the cell to the left (the input in the case of Cell 1) into its Current register). Then it carries out the computation done in Cell \( i \) of \( Ar_3 \) but does not pass the result to the right, instead it keeps it in its Current register. From then on it computes the new states based on the Current register of the cell to the left (the input in the case of Cell 1), its own Current register, and the Current register of the cell to the right.

If \( Ar_3 \) had its cells initialized to \( S_1, S_2, \ldots, S_n \), and computed for \( t \) time units during which it received the sequence \( I_1, I_2, \ldots, I_t \) as input, then, \( Ar_2 \) will have the following sequence as input: \( START, S_n, S_{n-1}, \ldots, S_2, S_1, I_1, I_2, \ldots, I_t \). \( START \) is implemented by an initialized cell state (all the fields are set to NULL) with the C_State field set to \( START \). If we assume that Cell 1 sees the input as the contents of an imaginary cell to its left, then the initial states will be placed in the Out register of that imaginary cell (with all the other registers set to NULL), while the "real" input to the array is put in the Current register of the imaginary cell.
(again, with all the other registers set to NULL). Cell 1 differentiates between an $S$ value and an $I$ value by examining the $Out$ register of the imaginary cell to its left: If it has a value other than NULL then it ($Out$) has an $S$ value, however if $Out$ had a value of NULL then the $Current$ register contains an $I$ value. Note that the initial states ($S_i$'s) are read in reverse order, from right to left, so that after the startup sequence, Cell $i$ of $Ar_2$ will simulate the operation of Cell $i$ of $Ar_3$. Figure 6.2 shows the array $Ar_3$ computing for 6 time units, and figure 6.3 shows an array $Ar_2$ simulating the operation of $Ar_3$. It can be seen that if $Ar_3$ had $n$ cells and computed for $t$ time units, then $Ar_2$ will have $n$ cells as well but will compute for $t + n + 1$ time units, such that a computation that took place in Cell $i_3$ of $Ar_3$ during time unit $t_3$ will be done in Cell $i_2$ of $Ar_2$ during time unit $t_2$ such that:

if $t_3 < i_3$ then

\[
i_2 = t_3 \\
\text{and } t_2 = 2*t_3 + n - i_3 + 1
\]

otherwise

\[
i_2 = i_3 \\
\text{and } t_2 = n + t_3 + 1.
\]

During the startup sequence, a cell in $Ar_2$ would call $f_3$ with the $Pass$ register of the cell to the left as the first argument (state of the cell to the left of the current cell), its $Current$ register as the second argument (the state of the current cell), and its $Right$ register as the third argument (the state of the cell to the right of the current cell). Whereas afterwards, the cell would call $f_3$ sending to it the $Current$ register of the cell to the left as the first argument, its own $Current$ register as the
second argument, and the *Current* register of the cell to the right as the third argument (analogous to what happens in $A_r^3$).
Figure 6.1: The cell state of the transformed array as used in figure 6.3.

Figure 6.2: Ar$_3$ (with bi-directional data flow). Arrows show functional dependencies.
Figure 6.3a: Array conforming to Case 2 that simulates the computation of the array of figure 6.2. Time units 0 through 5.

Arrows show functional dependencies.
Figure 6.3b: Array conforming to Case 2 that simulates the computation of the array of figure 6.2. Time units 6 through 11.

Arrows show functional dependencies.
The following is a complete and detailed description of the new cell function $f_2$.

(1) If the state is in control state (i.e. its $C_{\text{State}}$ is set to) $OFF$ then it does nothing (it keeps its state unchanged) unless the cell to the left is in control state $START$ in which case the cell would change its control state to $START$ (the cell stays off until it sees the start marker in the cell to its left).

(2) If the cell is in control state $START$ then it copies the contents of the $Out$ register of the cell to its left into its own $Out$ register (pass the output to the right). Then if the $Pass$ register of the cell to the left is not empty ($NULL$) then the cell would copy the contents of that register into its own $Current$ register and goes into control state $LAST$ (the cell is now ready to simulate the computation that took place in the last cell of the array $Ar_3$).

(3) If the cell is in control state $LAST$ (the cell is simulating the last cell in $Ar_3$) then it does one of the following:

If the $Pass$ register in the cell to the left is not set to $NULL$ (still in the startup sequence) the cell would call $f_3$ sending to it the contents of the $Pass$ register of the cell to the left as the first argument, the contents of its own $Current$ register as the second argument, and the contents of its $Right$ register as the third argument. The cell’s $Out$ and $Pass$ registers are set to the value returned by $f_3$ (the value is put in $Out$ so that it can leave the array as output, and is put in $Pass$ so that it can be read by the cell to the right in the following time unit). Then the cell moves the contents of its $Current$ register
into its Right register, copies the contents of the Pass register of the cell to the left into its Current register, and goes into control state INTERMEDIATE. This way the cell is ready to simulate the operation of a cell of Ar₃ (other than the last cell).

If the Pass register of the cell to the left is set to NULL then the startup sequence is over and the cell will simulate the operation of the last cell of Ar₃ until the array finishes computing. The cell will call f₃ sending to it the contents of the Current register of the cell to the left as the first argument, the contents of its own Current register as the second argument, and the contents of its Right register as the third argument (Right here will always be set to NULL). The value returned by f₃ represents the new state of the cell and is stored in the cell’s Current register and is put in the cell’s Out register (the output port of the array).

(4) If the cell is in control state INTERMEDIATE then it is still in the startup sequence and should simulate the operation of a cell of Ar₃ other than the last cell. It would do one of the following:

If the Pass register of the cell to the left is not set to NULL then the cell would call f₃ sending to it the contents of the Pass register of the cell to the left as the first argument, the contents of its Current register as the second argument, and the contents of its Right register as the third argument. The value returned by f₃ is stored in the cell’s Pass register to be read by the cell to the right in the following time unit. Then the cell would move the contents
of its Current register into its Right register, and copies the contents of the Pass register of the cell to the left into its own Current register.

If the Pass register of the cell to the left is set to NULL then the startup sequence is over for the cell: It calls \( f_3 \) sending to it the contents of the Current register of the cell to the left as the first argument, the contents of its Current register as the second argument, and the content of its Right register as the third argument. The value returned by \( f_3 \) represents the new state of the cell and is stored in the Current register of the cell. Then the cell sets its Out, Pass and Right registers to NULL, and goes into control state MIDDLE.

(5) If the cell is in control state MIDDLE then the startup sequence is over and the cell is simulating the operation of a cell of \( Ar_3 \) other than the last cell of the array. It would call \( f_3 \) sending to it the contents of the Current register of the cell to the left as the first argument, the contents of its Current register as the second argument, and the contents of the Current register of the cell to the right as the third argument. The value returned by \( f_3 \) represent the new state of the cell and is stored in the Current register of the cell.
The following is an example of how the new cell function $f_2$ would look:

function $f_2(A, B, C)$ returns State2 is

$Current(result) := \emptyset$
$Right(result) := \emptyset$
$Pass(result) := \emptyset$
$Out(result) := \emptyset$

if $C_{\text{State}}(B) = \text{OFF}$ then
    /* The cell stays OFF until it sees the start marker in the cell to the left. */
    if $C_{\text{State}}(A) = \text{START}$ then
        $C_{\text{State}}(result) := \text{START}$
    end if
else if $C_{\text{State}}(B) = \text{START}$ then
    $Out(result) := \text{Out}(A)$
    if $Pass(A) \neq \emptyset$ then
        $Current(result) := Pass(A)$
        $C_{\text{State}}(result) := \text{LAST}$
    else /* $Pass(A) = \emptyset$ */
        $C_{\text{State}}(result) := \text{START}$
    end if
else if $C_{\text{State}}(B) = \text{LAST}$ then
    if $Pass(A) \neq \emptyset$ then
        $Out(result) := f_3(Pass(A), Current(B), Right(B))$
        $Pass(result) := \text{Out}(result)$
        $Right(result) := Current(B)$
        $Current(result) := Pass(A)$
        $C_{\text{State}}(result) := \text{INTERMEDIATE}$
    else /* $Pass(A) = \emptyset$ */
        $Out(result) := f_3(Current(A), Current(B), Right(B))$
        $Current(result) := \text{Out}(result)$
        $C_{\text{State}}(result) := \text{LAST}$
    end if
else if $C_{\text{State}}(B) = \text{INTERMEDIATE}$ then
    if $Pass(A) \neq \emptyset$ then
        $Pass(result) := f_3(Pass(A), Current(B), Right(B))$
        $Right(result) := Current(B)$
        $Current(result) := Pass(A)$
        $C_{\text{State}}(result) := \text{INTERMEDIATE}$
    else /* $Pass(A) = \emptyset$ */
        $Current(result) := f_3(Current(A), Current(B), Right(B))$
        $C_{\text{State}}(result) := \text{MIDDLE}$
    end if
else /* $C_{\text{State}}(B) = \text{MIDDLE} */
    $Current(result) := f_3(Current(A), Current(B), Current(C))$
    $C_{\text{State}}(result) := \text{MIDDLE}$
end if

return (result)

end $f_2$

A simpler transformation based on the one presented in chapter 4 can be applied to an array conforming to Case 3 and that would change it directly into an array that requires only uni-directional data flow. However, we chose the transformation presented above because the resulting array can be further transformed into a linear uni-directional array requiring $2n + t$ cells (where $n$ and $t$ are respectively the number of cells and the number of time units of operation of the original $A_{r3}$ array). It can also be transformed into a circular uni-directional array with $n$ cells. On the other hand, using the simpler transformation produces a linear uni-directional array of $t$ cells and transforming the resulting array into a circular uni-directional array would require $t$ cells as well. Since most signal processing and combinatorial problems compute for a number of time units larger (in certain cases even much larger) than the number of cells they have, the transformation presented in this chapter is more interesting especially when transforming the array into a circular one.
Chapter 7

Case 4

(Bi-directional arrays, input and output at left end, input and output at right end
→ Bi-directional arrays, input at left end, output at right end.)

This chapter addresses arrays that accept input at both ends, employ a bi-directional data flow, and produce output that leaves both ends. The result of such a computation would be in the output leaving either or both ends. Arrays in this category ($Ar_4$) are those conforming to Case 4.2 or Case 4.3. Such arrays will be transformed into ones that accept input at the left end, produce output that leaves the right end, and employ bi-directional data flow as well ($Ar_{23}$). When the transformation presented in this chapter is applied to an array of Case 4.2 the result of the transformation is an array of Case 2, but if the transformation is applied to an array of Case 4.3 then the result would be an array of Case 3. The result of the transformation can be further transformed as applicable to Case 2 or Case 3.

Assuming $f_4$ is the cell function of $Ar_4$, $f_4$ would accept three parameters: The second parameter represents the state of the current cell, the first parameter represents the state of the cell immediately to the left of the current cell, and the third parameter represents the state of the cell immediately to the right of the current cell. A cell in $Ar_4$ would call $f_4$ every time unit with the correct parameters and $f_4$ would return the new state of that cell. This holds true for all the cells of $Ar_4$ except the boundary cells (Cell 1 and Cell $n$). Cell 1 does not have a cell to
its left, so instead of the state of the cell to its left it will send \( f_4 \) the input coming from the left at that time unit. Similarly, Cell \( n \) does not have a cell to its right, so instead of the state of the cell to its right it will send \( f_4 \) the input to the array coming from the right at that time unit.

Figure 7.1 shows such an array computing for 7 time units. It accepts the sequence \( L_1, L_2, \ldots, L_7 \) as input from the left, and the sequence \( R_1, R_2, \ldots, R_7 \) as input from the right. It produces the output sequence \( A_0, \ldots, A_6 \) at its left end, and the output sequence \( F_0, \ldots, F_6 \) at its right end.

The transformation can be achieved by folding the array onto itself such that Cell 1 of the new array (\( Ar_{23} \)) would simulate Cell 1 and Cell \( n \) of \( Ar_4 \). Similarly Cell 2 of \( Ar_{23} \) would simulate Cell 2 and Cell \( N - 1 \) of \( Ar_4 \), and so on for the rest of the cells. The last cell of \( Ar_{23} \) could be a special case: If the number of cells in \( Ar_4 \) (\( n \)) is even then the last cell will simulate Cell \( n/2 \) and Cell \( n/2 + 1 \) and will operate exactly like the other cells in \( Ar_{23} \). However, if \( n \) is odd then the last cell in \( Ar_{23} \) will simulate only Cell \( n \ div \ 2 + 1 \), (where the operator \( \div \) represents integer division). Just doing the aforementioned transformation would result in an array that accepts input at its left end and produces output that leaves its left end as well. However, the requirement is that the array accept input at its left end and produce output that leaves its right end. Thus two tracks should be added to the above array to route the output to the right. Two tracks are needed such that one will route the output that would have left the left end of \( Ar_4 \) and the other to route the output that would have left the right end of \( Ar_4 \).
Figure 7.1: Ar (with bi-directional data flow).

Arrows show functional dependencies.
As mentioned earlier each cell of $Ar_{23}$ will simulate the operation of two cells of $Ar_4$, and so will have two registers ($Up$, and $Down$) to hold the part of its state that is concerned with the actual computation (see figure 7.2). Each of those registers is capable of holding a value of the type used to define the state of a cell in $Ar_4$. Cell $i$ of $Ar_{23}$ will simulate the operation of Cell $i$ and Cell $n - i + 1$ of $Ar_4$, for $i = 1, 2, ..., n/2$ (except for the last cell of $Ar_{23}$ if $n$ is odd, where Cell $n \text{ div } 2 + 1$ of $Ar_{23}$ will simulate the operation of only Cell $n \text{ div } 2 + 1$ of $Ar_4$). This way in Cell $i$ of $Ar_{23}$ the register $Up$ would hold the state of Cell $i$ of $Ar_4$, while the register $Down$ would hold the state of Cell $n - i + 1$ of $Ar_4$. Also mentioned earlier are two output tracks, and so each cell in $Ar_{23}$ will have two registers associated with the output tracks, $UpOut$ and $DownOut$. $UpOut$ and $DownOut$ hold the output of the array as it is being passed to the right until it eaches the output port at the last cell in the array. $UpOut$ holds the output that would have left the left end of $Ar_4$, while $DownOut$ holds the output that would have left the right end of the array. Incidentally, $UpOut$ transfers the values generated by the $Up$ register of the first cell of $Ar_{23}$ and $DownOut$ transfers the values generated by the $Down$ register of the first cell. Output is transferred to the right by making each cell copy the contents of the $UpOut$ and $DownOut$ registers of the cell to the left into its own $UpOut$ and $DownOut$ registers, except for the first cell which will copy the contents of its $Up$ and $Down$ registers into its $UpOut$ and $DownOut$ registers, respectively.
Each cell in the array also has a register labeled *Position* which can take any one of the following values: *FIRST*, *MIDDLE*, *ODDLAST*, and *EVENLAST*. A value of *FIRST* in the *Position* register identifies the cell as the first cell in \( A_{r23} \) and hence it will process the output as described earlier. A value of *ODDLAST* identifies the cell as the last cell in \( A_{r23} \) and that the original \((A_r)\) array had an odd number of cells, while a value of *EVENLAST* identifies the cell as the last cell of \( A_{r23} \) and that the array \( A_r \) had an even number of cells. A value of *MIDDLE* in the *Position* register means that the cell is neither the first cell nor the last cell of \( A_{r23} \). A typical \( A_{r23} \) array would have a *FIRST* cell, either an *ODDLAST* or an *EVENLAST* cell, and zero or more *MIDDLE* cells.

Each cell also has a register called *TempUp* which could be used for temporary storage. This register is used only by the last cell in \( A_{r23} \) and then only if that cell has its *Position* register set to *EVENLAST*, as will be described later.

Each cell in \( A_{r23} \) also has two registers \((Num_1 \) and \( Num_2)\) each capable of holding an integer. Those registers are used only with the startup sequence as follows: Each cell reads the *Num1* and *Num2* registers of the cell to its left. Cell 1 will see \( Num_1 = Num_2 = n \) where \( n \) is the number of cells in \( A_r \). First each cell will copy the contents of the *Num1* register of the cell to the left into its own *Num1* register. By comparing the values stored in the *Num1* and *Num2* registers of the cell to the left, a cell can tell its position in the array and will accordingly set its *Position* and *Num2* registers. If the values are equal then the cell is the first cell in the array, and it sets its *Num2* register to be 2 less than the value on the left. If
Num2 of the cell to the left equals 1 then the cell is the last cell and the original number of cells is odd. If Num2 of the cell to the left is two then the cell is the last cell in the array and the original number of cells is even. If none of the previous condition holds then the state is an internal cell and will set its Num2 register to be 2 less than the Num2 register of the cell to the left.

Each cell also has a Control_State register that carries the control information necessary for the operation of the new array. The different control states will be presented below.

If Ar4 computed for t time units during which it received the sequences \(L_1, L_2, \ldots, L_i\) as input from the left, and \(R_1, R_2, \ldots, R_i\) as input from the right then Ar23 will have the following sequence as input: START, Null, \(I_1, I_2, \ldots, I_i\), Null1, Null2, \(\ldots, Null_{n_{23}}\), where \(n_{23}\) is the number of cells in Ar23 and is computed as described below. START is implemented as a cell state with its Control_State register set to SETUP and its Num1 and Num2 registers set to \(n\), the number of cells in Ar4. \(I_z\) is implemented as a cell state with its Control_State register set to CALC, its Up register set to \(L_z\), and its Down register set to \(R_z\). Nullz is implemented as an empty cell state with its Control_State set to NULL, those are used after the array has finished processing the actual input but is still waiting for all the output values to reach the output port. From the above it can be seen that if Ar4 had \(n\) cells and computed for \(t\) time units then Ar23 will have \(n_{23} = \text{ceiling}(n/2)\) cells and will compute for \(t + n_{23} + 1\) time units. It follows that a computation that was done in Cell \(i\) at time unit \(j\) in Ar4 will be done in Cell \(i\)
at time unit $j+1$ of $Ar_{23}$ if $i \leq (n \ div 2 + 1)$ and in Cell $n - i + 1$ at time unit $j+1$ otherwise.

Figure 7.3 shows an array $Ar_{23}$ simulating the operation of the array $Ar_4$ shown in figure 7.1. (Figure 7.2 shows the organization of the cell state as used in figure 7.3.)

![Table]

<table>
<thead>
<tr>
<th>Up</th>
<th>Up Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down</td>
<td>Down Out</td>
</tr>
<tr>
<td>num1</td>
<td>num2</td>
</tr>
<tr>
<td>Position</td>
<td>Control State</td>
</tr>
</tbody>
</table>

Figure 7.2: The cell state of the transformed array as used in figure 7.3
Figure 7.3a: Array conforming to Case 2 that carries out the same computation as the one in figure 7.1. Arrows show functional dependencies.
Figure 7.3b: The details about num1 & num2 are omitted.
Every time unit each cell in $A_{r_{23}}$ will read the state of the cell to its left and that of the cell to its right and then will call the new cell function $f_{23}$. The result returned by $f_{23}$ represents the new state of the cell. The following is a complete and detailed description of the new cell function $f_{23}$.

Depending on the Control State of the cell to the left one of the following will be carried out.

(1) If the cell to the left is in control state SETUP then the cell should get ready to compute. It initializes its Up, Down, UpOut, and DownOut registers, sets its Control State to SETUP (so that the cell to the right can see it in the next time unit), copies the contents of the Num1 register of the cell to the left into its own Num1 register, and then it should determine its position in the array by examining the contents of the Num2 register of the cell to the left and possibly comparing it to the value stored in Num1. Let $X$ stand for the integer stored in the Num2 register of the cell to the left. If $X$ is equal to the integer stored in the Num1 register then the cell is the first cell of the array, it will set its Position register to FIRST and its Num2 register to $X - 2$. If $X = 1$ then the cell is the last cell in the array and the number of cells in $A_{r_{4}}$ was odd, thus the cell will set its Position register to ODDLAST and its Num2 register to zero. If $X = 2$ then the cell is the last cell in the array and the number of cells in $A_{r_{4}}$ was even, so the cell should set its Position register to EVENLAST and its Num2 register to zero. If none of the preceding conditions is true then the cell is an internal cell and hence it should set its Position
register to \textit{MIDDLE} and its \textit{Num2} register to $X - 2$. (Note that setting the \textit{Num2} register to zero in the last cell of the array is not really necessary, however it doesn’t hurt to give it a value, and if the array had more cells than is needed then a cell can detect that it is not concerned with the computation by detecting a zero in the \textit{Num2} register of the cell to the left). The contents of \textit{Num2} are decremented by two by every cell in the array to account for the two cells of $Ar_4$ that are represented or simulated by the cell of $Ar_{23}$.

(2) If the cell to the left is in control state \textit{CALC} then the cell should set its \textit{Control\_State} register to \textit{CALC} and then do one of the following steps depending on its position in the array:

If the cell is the first cell then it should call $f_4$ sending to it the \textit{Up} register of the cell to the left as the first argument, its own \textit{Up} register as the second parameter, and the \textit{Up} register of the cell to the right as the third parameter. The result returned by $f_4$ represents the new state of Cell 1 of $Ar_4$ and should be stored in the \textit{Up} register. Then the cell will call $f_4$ sending to it the \textit{Down} register of the cell to the \textit{right} as the first argument, its own \textit{Down} register as the second argument, and the \textit{Down} register of the cell to the left as the third argument. The result returned by $f_4$ here represents the new state of Cell $n$ of $Ar_4$ and should be stored in the \textit{Down} register. Then in order to let the output reach the output port of the array the cell should copy the contents of its \textit{Up} and \textit{Down} registers to its \textit{UpOut} and \textit{DownOut} registers respectively.
If the Position register has a value of MIDDLE then the cell is internal to the array and is simulating the cells $i$ and $n-i+1$ of $Ar_4$. The cell should call $f_4$ sending to it the Up register of the cell to the left as the first argument, its own Up register as the second argument, and the Up register of the cell to the right as the third argument. The value returned by $f_4$ represents the new state of Cell $i$ of $Ar_4$ and should be stored in the Up register. Then the cell should call $f_4$ sending to it the Down register of the cell to the right as the first argument, its own Down register as the second argument, and the Down register of the cell to the left as the third argument. The result returned by $f_4$ represents the new state of Cell $n-i+1$ of $Ar_4$ and should be stored in the Down register. Then the cell should copy the contents of the UpOut and DownOut registers of the cell to the left into its own UpOut and DownOut registers respectively.

If the Position register has an EVENLAST stored in it then the cell is the last cell in the array, and is simulating the operation of Cell $n/2$ and Cell $n/2+1$ of $Ar_4$. Note that the two cells of $Ar_4$ that are simulated by this cell used to be adjacent to each other in $Ar_4$ and the cell simulated by the Up register used to be the left of the cell simulated by the Down register. The cell should call $f_4$ sending to it the Up register of the cell to the left as the first argument, its own Up register as the second argument, and its own Down register as the third argument. The result returned by $f_4$ represents the new state of Cell $n/2$ and should be stored in the Up register, however the current value of the
$Up$ register is needed for the simulation of Cell $n/2+1$ in the $Down$ register. Thus the value returned is stored in the temporary storage register $TempUp$ until the current value of the $Up$ register is no longer needed. Then the cell should call $f_4$ sending to it its own $Up$ register as the first argument, its $Down$ register as the second argument, and the $Down$ register of the cell to the left as the third argument. The result returned by $f_4$ here represents the new state of Cell $n/2+1$ and should be stored in the $Down$ register. At this time the value in the $Up$ register is no longer needed and thus the cell should copy the contents of the $TempUp$ register into the $Up$ register. Then the cell should copy the contents of the $UpOut$ and $DownOut$ registers of the cell to the left into its own $UpOut$ and $DownOut$ registers respectively.

If the $Position$ register has a value of $ODDLAST$ stored in it then the cell is the last cell in the array and is simulating only one cell of $Ar_4$. Note that in $Ar_4$ the cell being simulated used to be immediately to the right (left) of the cell in the $Up$ ($Down$) register of the cell to the left (right). Also note that the cell in the $Up$ register of the cell to the left expects to find the state of Cell $n \text{ div } 2+1$ in the $Up$ register, while the cell in the $Down$ register of the cell to the left expects to find it in the $Down$ register, and hence it is necessary to keep a copy of the state in the $Up$ register and another in the $Down$ register. Thus, the cell should call $f_4$ sending to it the $Up$ register as the first argument, its own $Up$ register as the second argument, and the $Down$ register of the cell to the left as the third argument. The result returned by $f_4$ will be
stored in the *Up* register and a copy of it will be stored in the *Down* register. Then the cell should copy the contents of the *UpOut* and *DownOut* registers of the cell to the left into its own *UpOut* and *DownOut* registers respectively.

(3) If the cell to the left is in control state *NULL* then there’s not any more *real* input to process and the array is functioning just to allow all the output items to reach the output port. The cell should initialize itself and set its *Control_State* to *NULL*.

The following is an example of how the new cell function (*f*₂₃) would look:

```plaintext
function *f*₂₃(*A*, *B*, *C*) returns State₂₃ is
  result := *B*
  if *Control_State*(*A*) = SETUP then
    *Up*(result) := ∅
    *Down*(result) := ∅
    *UpOut*(result) := ∅
    *DownOut*(result) := ∅
    *Control_State*(result) := SETUP
    *Num₁*(result) := *Num₁*(A)
  
  else if *Num₂*(A) = *Num₁*(A) then
    *Position*(result) := FIRST
    *Num₂*(result) := *Num₂*(A) − 2
  else if *Num₂*(A) = 1 then
    *Position*(result) := ODDLAST
    *Num₂*(result) := 0
  else if *Num₂*(A) = 2 then
    *Position*(result) := EVENLAST
    *Num₂*(result) := 0
  else
    *Position*(result) := MIDDLE
    *Num₂*(result) := *Num₂*(A) − 2
  end if

  else if *Control_State*(*A*) = CALC then
    *Control_State*(result) := CALC
    if *Position*(*B*) = FIRST then
      /* Simulate the computation of Cell 1. */
      *Up*(result) := *f*₄(*Up*(A), *Up*(B), *Up*(C))
    
    /* Simulate the operation of Cell n. */
  
end if
```
Down(result) := f 4(Down(C),Down(B),Down(A))

/* Put the result of the computation on the output track so that it can be passed to
the right until it reaches the output port at the last cell of the array. */
UpOut(result) := Up(result)
DownOut(result) := Down(result)

else if Position(B) = MIDDLE then
/* Simulate the operation of Cell i. */
Up(result) := f 4(Up(A),Up(B),Up(C))

/* Simulate the operation of Cell n - i. */
Down(result) := f 4(Down(C),Down(B),Down(A))

/* Pass the contents of the UpOut and DownOut registers of the cell to the left,
to the right. */
UpOut(result) := UpOut(A)
DownOut(result) := DownOut(A)

else if Position(B) = EVENLAST then
/* Simulate the operation of Cell n/2. */
Up(result) := f 4(Up(A),Up(B),Down(B))

/* Simulate the operation of Cell n/2 + 1. */
Down(result) := f 4(Up(B),Down(B),Down(A))

/* Copy the contents of the UpOut and DownOut registers of the cell to the left
into own UpOut and DownOut registers (the output ports). */
UpOut(result) := UpOut(A)
DownOut(result) := DownOut(A)

else /* Position(B) = ODDLAST */
/* Simulate the operation of Cell Ndiv2 + 1 */
Up(result) := f 4(Up(A),Up(B),Down(A))

/* Make a copy of the result available in the Down register so that it can used by
the cell in Down register of the cell to the left. */
Down(result) := Up(result)

/* Copy the contents of the UpOut and DownOut registers of the cell to the left
into own UpOut and DownOut registers (the output ports). */
UpOut(result) := UpOut(A)
DownOut(result) := DownOut(A)

end if

else /* Control_State(A) = NULL */
Up(result) := ∅
Down(result) := ∅
UpOut(result) := ∅
DownOut(result) := ∅
ControlState(result) := NULL

end if
return result
end $f_{23}$

The cell function in the fashion presented above is geared towards transforming arrays conforming to Case 4.2 into ones conforming to Case 2. To be used to transform arrays conforming to Case 4.3 into arrays conforming to Case 3, the cell should not initialize its registers to $NULL$ upon encountering a $SETUP$ or $NULL$ signal because the registers would already be initialized to their counterparts in $Ar_{43}$. 
Chapter 8

Circular Arrays

(Bi-directional iterative arrays – Uni-directional systolic rings.)

In the previous chapters it was shown how arrays conforming to Case 3, Case 4.2, and Case 4.3 can be transformed into ones that conform to Case 2, and how an array conforming to Case 2 can be transformed into one that requires only uni-directional flow of data. However, when transforming an array of Case 2, that had \( n \) cells and computed for \( t \) time units, into one that requires uni-directional flow of data, the resulting array requires \( n + t - 1 \) cells and will compute for \( 2t + 1 \) time units. In many applications such as signal processing algorithms, \( t \) can be much larger than \( n \) and hence using \( n + t - 1 \) cells could be impractical or even not possible.

Note that after the transformation a maximum of \( n \) cells are actively involved in the computation while the rest only pass input or output to the right. As the computation migrates to the right every other time unit, a cell on the right is \textit{shifted in} and becomes actively involved with the computation while a cell on the left is \textit{shifted out} and is no longer involved with the computation. This permits us to use only \( n \) cells if we configure the array in a circular or ring fashion such that Cell 1 is connected to Cell \( n \). This way when the computation migrates out of Cell \( n \) instead of going into Cell \( n + 1 \) it goes back into Cell 1, which would have been left behind in the original transformation.
Original array with bi-directional data flow.

Circular array with uni-directional data flow.

Figure 8.1
This chapter presents a scheme for transforming an array of Case 2 into a uni-directional circular array with the same number of cells (see figure 8.1). Given a cell function \( f_2 \) of \( Ar_2 \), a cell function \( g_B \) can be developed that will carry out the same computation except on a uni-directional circular array \( (Ar_B) \).

A straightforward transformation is not feasible: With time the computation circles around the array a number of times, so that if we allow the input to follow we would need very large input buffers at every cell and that would undermine any savings in hardware gained by using a circular array.

In this implementation of the transformation, input and output are handled in a slightly peculiar way. The array reads input every time unit, but for only \( n \) time units (where \( n \) is the number of cells in the array). Then, the array stops accepting input for \( n \) time units. This cycle (accepting input for \( n \) time units, then not accepting input for \( n \) time units) is repeated over and over until the array finishes computing. In [SSK87] a modification to this implementation is described that would allow the array to accept input at a constant rate (every other time unit), however that modification requires an additional delay of \( n \) time units. Handling input is discussed further in the following paragraph. Output is sent out of the array in similar batches as well. The first two output items leave the array during time units \( 3n+2 \) and \( 3n+3 \). Following that there will be \( n \) time units with no output leaving the array, followed by \( n \) time units with an item of output leaving the array every time unit. This cycle (\( n \) time units with no output followed by \( n \) time units with an item leaving the array every time unit) is repeated until the
array finishes computing. In [SSK87] a modification is described that will allow the output to leave the array at a constant rate (every other time unit).

Figure 8.2 shows a two-way array ($Ar_2$) computing for 11 time units. It accepts $a, b, ..., k$ as input and produces $D_0, D_1, ..., D_7$ as output. Figure 8.4 shows a uni-directional circular array carrying out the same computation as the array in figure 8.2. The organization of the cell state as used in figure 8.4 is shown in figure 8.3.
Figure 8.2: $A_{r_2}$ (with bi-directional data flow).

Arrows show functional dependencies.
<table>
<thead>
<tr>
<th>Control State</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Current State</td>
<td>Right State</td>
</tr>
<tr>
<td>Current Input</td>
<td>New Output</td>
</tr>
<tr>
<td>New Input</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Figure 8.3: The cell state of the transformed array as used in figure 8.4.
Figure 8.4a: Circular array with uni-directional data flow that simulates the array of figure 8.2. Time units 0 through 5.

Arrows show functional dependencies.
Figure 8.4b: Circular array with uni-directional data flow. Time units 6 through 13.
Figure 8.4c: Circular array with uni-directional data flow. Time units 14 through 21.
Figure 8.4d: Circular array with unidirectional data flow. Time units 22 through 29.
Every cell in \( A_rB \) has two registers that are used to store input, \( Current\_Input \) and \( New\_Input \) (see figure 8.3). \( New\_Input \) represents input that is being passed to the right. Whenever Cell \( i \) has a value stored in its \( New\_Input \) register, that value will be read in the following time unit by Cell \( i+1 \). The value stored in the \( Current\_Input \) register of Cell \( i \) will stay in Cell \( i \) to be used by the computation taking place in Cell \( i \) the next time Cell \( i \) is the first cell of the computation. In the time unit after it is used this value will be replaced by input from the cell to the left (found in the \( New\_Input \) register of Cell \( i-1 \)). Whereas the array \( A\_r2 \) sampled input every time unit, the array \( A\_rB \) will sample input every time unit for \( n \) time units and then will stop sampling input for another \( n \) time units and so on. This allows the input to be folded. Input item 1 will go into the \( Current\_Input \) register of Cell 1. When the first cell reads input item 2 it will see that its \( Current\_Input \) register is full and so will put that in its \( New\_Input \) register to be read by Cell 2 in the following time unit. In the following time unit Cell 2 will read the value in the \( New\_Input \) register of Cell 1 (input item 2) into its \( Current\_Input \) register, and Cell 1 will read input item 3 into its \( New\_Input \) register. This is maintained for \( n \) time units after which the array does not accept input for another \( n \) time units after which the array starts accepting input again. By this time the value in the \( Current\_Input \) register of Cell 1 would have been used up by the computation, and hence input item \( n+1 \) will go into the \( Current\_State \) register of Cell 1, and the same cycle as before is repeated.
The output is treated somewhat similarly. Every cell in $Ar_B$ has two registers that are used with the output, $Pass$ and $New\_Output$. $Pass$ holds the output that is moving right to reach Cell 1 where it will leave the array. Whenever Cell $i-1$ has a value stored in its $Pass$ register this value will be copied by Cell $i$ into its own $Pass$ register until it reaches Cell 1. $New\_Output$ holds the output produced by the computation at that cell. The value will be stored in $New\_Output$ until all output produced by the cells to the left has passed it by (output produced by the cells to the left is older and hence should get to Cell 1 first). If the $Pass$ register of the cell to the left is empty then there isn’t any more output coming from the left that is older than the value stored in $New\_Output$ and thus the cell should move the contents of its $New\_Output$ register into its $Pass$ register. The $Pass$ register of Cell 1 will be always empty when seen by Cell 2 (Cell 1 does the output of the array and hence does not need to pass output to the right).

Each cell in $Ar_B$ also has three other registers: $Control\_State$, $Current\_State$, and $Right\_State$. $Control\_State$ holds the control information necessary for the operation of the new cells. The different control states will be introduced as needed later. $Current\_State$ holds the state of the cell of $Ar_2$ whose computation is to be simulated. $Right\_State$ holds the state of the cell of $Ar_2$ immediately to the right of the cell whose computation will be simulated. Note that each of $Current\_Input$, $New\_Input$, $Pass$, $New\_Output$, $Current\_State$ and $Right\_State$ is capable of holding a value of the type used to describe the state of a cell in $Ar_2$. 
Here again each time step of $Ar_2$ is simulated with two time steps, a *MAIN* step followed by an *INTERMEDIATE* step, with the computation being simulated during the *MAIN* step while the *INTERMEDIATE* step is used solely for data routing purposes. The new cell function accepts two arguments: The *second* argument represents the state of the current cell, and the *first* argument represents the state of the cell to the left of the current cell. Due to the fact that the array is connected in a circular (ring-like) fashion, Cell 1 will see Cell $n$ to its left. Since Cell 1 does the input and the output of the array its operation will be somewhat peculiar. The *Pass* register of Cell 1 will always be empty when viewed by Cell 2. Also Cell 1 does not see the *New_Input* register of Cell $n$ (which will always be empty), instead it will see the input to the array at that time unit.

Assuming that $Ar_2$ computed for $t$ time units during which it received $S_1, S_2, \ldots, S_t$ as input, then the input to the $Ar_B$ array would be: *START*, $S_1, S_2, \ldots, S_n$, *Null_1*, *Null_2*, ..., *Null_n*, $S_{n+1}, S_{n+2}, \ldots, S_{2n}$, *Null_1*, *Null_2*, ..., *Null_n*, $S_{2n+1}$, ... For *START* Cell 1 would see to its left the state of Cell $n$ (which would be initialized at that time) except with a *Control_State* of *START*. For the rest of the input Cell 1 would see to its left the exact state of Cell $n$ except that the *New_Input* would be set to the actual input value. For *Null_* the *New_Input* register would be empty (initialized). From the above it can be seen that if $Ar_2$ had $n$ cells and computed for $t$ time units then $Ar_B$ will have $n$ cells as well and will compute for $t_B$ time units such that:

$$t_B := 2n(t \text{ div } n) + n + (t \text{ mod } n) + 2 \quad \text{if } (t \text{ mod } n) \leq 1$$
and

\[ t_B := 2n((t \div n)+1)+(t \mod n)+2 \quad \text{otherwise}. \]

It follows that a computation that took place in Cell \( i \) at time unit \( t \) in \( Ar_2 \) will be simulated in time unit \( 2t+n \) of \( Ar_B \) in Cell \( i_B \) such that:

\[ i_B := n \quad \text{if } (i+t-1) \mod n = 0 \]

and

\[ i_B := (i+t-1) \mod n \quad \text{otherwise}. \]

There are more similarities than dissimilarities between the transformation presented below and the one presented in chapter 5. The notable exceptions are:

Except during the startup sequence all the cells are actively involved with the computation, and hence there are no \( OLD \) or \( PASSIVE \) cells in this array. A direct consequence of this is that the register \( Position \) is not needed. The \( first \) and the \( last \) cell of the \textit{computation} (as opposed to the first cell and the last cell of the array) function in a special way: The first cell will use the value stored in the \textit{Current Input} register instead of the \textit{Current State} register of the cell to the left, and the last cell should put the result of the computation in the \textit{New Output} register. Thus the operation of the first cell is implemented as a special case of the \textit{MAIN} step while that of the last cell is implemented as a special case of the \textit{INTERMEDIATE} step. The following is a complete and detailed description of the new cell function \( g_B \).

(1) If the cell is in control state \( OFF \) and the cell to the left is in control state \( START \) then the cell will initialize itself and will go into control state \( START \).
(2) If the cell is in control state START then it will go into control state ON.

(3) The following will be done if the cell is in control state ON.

If the cell to the left is in control state OFF or ON then the cell will do nothing, it will keep its present state.

If the cell to the left is in control state START then the cell will go into control state FIRST. This happens with the first cell of the array, after the control state START has gone around the array (it has reached Cell n) it is time for the first cell to simulate the operation of the first cell of the computation.

If the cell to the left is in control state INTERMEDIATE then the cell will initialize its Right_State register, will copy the contents of the Current_State register of the cell to the left into its own, and will go into control state MAIN.

If the cell to the left has a control state of MAIN or FIRST then the cell will initialize its Current_State register and will go into control state INTERMEDIATE.

(4) If the cell is in control state FIRST then it will perform the computation as if it were Cell 1 in Ar2.

If the Current_Input register is empty then there is no more input to process and the array is still operating in order to get the various output values generated to the output port, and so the cell will initialize its Current_State, Right_State, Current_Input, and New_Input registers.

However if the Current_Input register is not empty then the cell will call \( f_2 \)
(the cell function of $Ar_2$) and will send it its $Current\_Input$ register as the first argument, its $Current\_State$ register as the second, and its $Right\_State$ register as the third argument. Then the cell will initialize its $Current\_Input$ register, store the value returned by $f_2$ in its $Current\_State$ register.

After doing one of the above two steps the cell will do the following: If the cell to the left is in control state $ON$ then the cell will change into control state $ON$, however if the cell to the left is in control state $MAIN$ then the cell will change into control state $LAST$.

(5) If the cell is in control state $LAST$ then it should output the result from the previous computation and do an $INTERMEDIATE$ step. The result from the previous computation is in the $Current\_State$ register of the cell to the left, and hence the contents of that register will be copied into the $New\_Output$ register of the cell, then the cell will copy the contents of the $Current\_State$ register of the cell to its left into its own, and will initialize its $Right\_State$ register.

(6) A cell that is in control state $MAIN$ will simulate the actual computation of a cell in $Ar_2$ (other than the very first cell, which will be simulated by a cell in the control state $FIRST$). Hence, unless its $Current\_State$ register is empty, the cell should call $f_2$ (the cell function of $Ar_2$) sending to it the $Current\_State$ register of the cell to the left as the first argument, its own $Current\_State$ register as the second argument, and its $Right\_State$ register as the third argument. The value returned by $f_2$ represents the new state of the
cell and will be placed in the \textit{Current\_State} register. However, if the \textit{Current\_State} register is empty then there is no more input to process and the array is functioning just to let the result of the last few computations travel around the array and to the output port (Cell 1). Thus the cell should initialize its \textit{Current\_State} and \textit{Right\_State} registers.

After doing one of the above two steps the cell should go into control state \textit{INTERMEDIATE}.

\(7\) If the cell is in control state \textit{INTERMEDIATE} then it should perform the following data routing steps. It should copy the contents of its \textit{Current\_State} register into its \textit{Right\_State} register, and the contents of the \textit{Current\_State} register of the cell to the left into its own \textit{Current\_State} register. If the cell to the left is in control state \textit{INTERMEDIATE} then the cell would go into control state \textit{MAIN}, otherwise if the cell to the left is in control state \textit{ON} or \textit{LAST} then the cell would go into control state \textit{FIRST}.

After doing one of the seven previous steps, the cell would do the following: If its \textit{Current\_Input} register is empty then it will copy the contents of the \textit{New\_Input} register of the cell to the left into its \textit{Current\_Input} register, otherwise it will copy the contents of the \textit{New\_Input} register of the cell to the left into its own \textit{New\_Input} register. If the \textit{Pass} register of the cell to the left is empty then the cell would move the contents of its \textit{New\_Output} register into its \textit{Pass} register, otherwise the cell would copy the contents of the \textit{Pass} register of the cell to the left into its own \textit{Pass} register. (Note that if a cell copies the contents of register \(Q\) into
Chapter 9

Sample applications

The following examples will demonstrate the application of the transformations discussed in earlier chapters. The first example will be an implementation of the odd-even transposition sort, and will demonstrate transforming an array of Case 1 (bi-directional cellular array) \((Ar_1)\) into a uni-directional iterative array \((Ar_2)\). The second example will implement the string comparison algorithm presented in [Lip85] and will demonstrate transforming an array of Case 4.2 (bi-directional array, cells initialized to the same state, input and output at left end, input and output at right end) \((Ar_{42})\) into an array of Case 2 (bi-directional iterative array) \((Ar_2)\). The last (third) example is an implementation of an IIR filter and will demonstrate transforming an array of Case 3 (bi-directional array, cells initialized to different states, input at left end, output at right end) \((Ar_3)\) into an array of Case 2 (bi-directional iterative array) \((Ar_2)\).

The code segments presented in this chapter are closest to standard Pascal, with some modifications for the sake of simplicity. The most notable violation of Pascal rules is that a function will be allowed to return a result that is not a simple type, i.e. a result that is a user defined record.

Example 1: Odd-Even Transposition Sort:
(Bi-directional cellular array → Uni-directional iterative array.)

This example will carry out the odd-even transposition sort [Knu73]. Here, the elements to be sorted are each stored in a different cell. This initialization con-
stitutes all the input to the array. Furthermore, there is no explicit output leaving the array, the result of the computation is in the final states of the cells. That is, after the array finishes computing, the items will still be each in a different cell, but in order. The computation requires a bi-directional flow of data.

The function requires that each cell store a parity for itself that can be set to either 0 or 1 (even, and odd parity respectively). The parity of a cell is flipped every time unit, that is a cell that has even parity during time unit $t$ will have odd parity during time unit $t + 1$. Also, an odd parity cell will have both its neighbors with even parity, and vice versa. An odd parity cell will get the minimum of the values stored in itself and the cell to its right, while an even parity cell will get the maximum of the values stored in itself and the cell to its left. Special cases arise with the first and last cells of the array. The first cell does not have a cell to its left, and hence does nothing when it has an even parity (except for flipping it parity). Similarly, the last cell does not have a cell to its right, and thus does nothing but flip its parity when it has an odd parity.
The state of a cell is described by the following:

\[
\text{State} = \text{record}
\begin{align*}
\text{valu} & : \text{integer}; \quad \{ \text{This is the value or item stored in the cell.} \} \\
\text{parity} & : (0, 1); \quad \{ \text{This holds the parity of the cell. A cell} \\
& \quad \text{with even parity gets the max of values in} \\
& \quad \text{itself and the cell to the right. An odd} \\
& \quad \text{parity cell gets the min of values in itself} \\
& \quad \text{and the cell to the left.} \\
\text{inst} & : (\text{xl}, \text{xr}, \text{calc}); \quad \{ \text{We assume the existence of two imaginary} \\
& \quad \text{cells Cell 0 and Cell } n+1. \text{ The 'inst' field} \\
& \quad \text{of Cell 0 will be set to 'xl', and that of} \\
& \quad \text{Cell } n+1 \text{ will be set to 'xr'. The other cells} \\
& \quad \text{will have their 'inst' fields set to 'calc'.} \\
& \quad \text{The first cell will know that it is the first} \\
& \quad \text{by detecting an 'xl' in the cell to the left.} \\
& \quad \text{Similarly, the last cell will know it by} \\
& \quad \text{detecting an 'xr' in the cell to the right.} \\
\end{align*}
\]
The following is a cell function for the odd-even transposition sort:

```plaintext
function f1 (Left, Current, Right : State1 ) : State1;

(*
* This function defines the operation of a cell in the array. The parameter
* Current represents the state of the cell whose computation is to be
* simulated, say Cell i. Left represents the state of Cell i−1, and
* Right represents the state of Cell i+1.
* *
* Procedure and function definitions.
* *
* min : Accepts two integers and returns the smaller of the two.
* max : Accepts two integers and returns the larger of the two.
* fmiddle : Will simulate the operation of a cell that is not a boundary
* cell, that is neither the first nor the last cell. Will be
* described in detail later.
* flleft : Will simulate the operation of the first cell of the array. Will
* be described in detail later.
* fright : Will simulate the operation of the last cell of the array. Will
* be described in detail later.
*)

begin {function f1}

if Left.inst = xl { current cell is the leftmost cell. }  
  return (flleft (Current, Right))

else if Right.inst = xr { current cell is the rightmost cell. }  
  return (fright (Left, Current))

else { current cell is a 'middle' cell. }  
  return (fmiddle (Left, Current, Right));

end; {function f1}
```
function fniddle(Left, Current, Right : State1): State1;

(*
* This function describes the operation of a cell with neighboring
* cells on both sides, that is a cell that is not at either end of
* the array. On odd parity steps (parity=1) the value in the cell
* is replaced with the smaller of itself and the value in the cell
* to the right. On even parity steps (parity=0) the value is replaced
* by the larger of itself and the value in the cell to the left.
*)

var
  result : State1;

begin

with result do
  begin
    parity := not (Current.parity); { Flip the parity. }
    inst := calc; { should always have calc. }

    if Current.parity then { odd parity, i.e. parity=1 } 
      valu := min (Current.valu, Right.valu) 
    else { even parity, i.e. parity=0} 
      valu := max (Current.valu, Left.valu) 

    end; {with result do}

return (result)

end; {function fniddle}
function fleft (Current, Right : State1) : State1;

(*
* This function defines the operation of the first (leftmost) cell
* of the array, i.e. Cell 1. On odd parity steps (parity=1) the
* value in the cell is replaced with the smaller of itself and the
* value in the cell to the right. On even parity steps (parity=0)
* the cell keeps its value (of course it still has to change its parity).
*)

var
  result : State1;

begin

  with result do
  begin
    parity := not (Current.parity); { Flip the parity. }  
    inst := calc; { should always have calc. } 
    if Current.parity then
      valu := min (Current.valu, Right.valu) 
    else
      valu := Current.valu { the cell keeps its own value.}
  end; {with result do}

  return (result)

end; {function fleft}
function fright (Left, Current : State1 ) : State1;

(*
* This function defines the operation of the last (rightmost) cell
* of the array, i.e. Cell n. On odd parity steps (parity=1) the
* value in the cell is not changed (only the parity is flipped).
* On even parity steps (parity=0) the value in the cell is replaced
* with the larger of itself and the value in the cell to the left.
*)

var
  result : State1;

begin
  with result do
    begin
      parity := not (Current.parity); { Flip the parity. }
      inst := calc; { should always have calc. }
      if not (Current.parity) then { even parity, i.e. parity=0 }
        valu := max (Current.valu, Left.valu)
      else { odd parity, i.e. parity=1 }
        valu := Current.valu { the cell keeps its own value. }
    end; {with result do}
  return (result)
end; {function fright}

Figure 9.1 shows an array of 5 cells, each executing the above cell function for
5 time units. The cells are initialized to 60, 21, 81, 55, and 17. Odd numbered
cells are initialized to have a parity of ODD while, even numbered cells are
initialized to have a parity of EVEN, (in figure 9.1 O stands for odd parity
and $E$ stands for even parity). In figure 9.1 the $inst$ field of the cell state is not shown, all cells are assumed to have their $inst$ field set to $calc$ with the exception of the two imaginary cells (not shown in the figure): The imaginary cell to the left is assumed to have its $inst$ field set to $xl$, and the one to the right is assumed to have its $inst$ field set to $xr$.

The above cell function which requires a bi-directional array can be transformed into a cell function for a uni-directional array as described in chapter 4. An array whose cells execute the resulting cell program will carry out the same computation as the one in figure 9.1 except with uni-directional data flow. Figure 9.2 shows such an array computing for 16 time units. The cell state in figure 9.2 is as discussed in Chapter 4, with the $Left$ and $Current$ states being organized as follows: $value, parity$. All contents of $Left$ and $Current$ are assumed to have an $inst$ field set to $calc$ with the exception of those marked $XL$ which will have an $inst$ field of $xl$, and those marked $XR$ which will have an $inst$ field of $xr$. 
Figure 9.1
Figure 9.2a: Time units 0 through 7.
Figure 9.2b: Time units 8 through 16.
Example 2: String comparison:
(Bi-directional array, cells initialized to the same state, input and output at left end, input and output at right end – Bi-directional iterative array.)

The cell function presented here implements the algorithm presented in [Lip85]. It accepts as input two strings of characters and the result of the computation is the number of delete/insert operations necessary to change one string into the other. All the cells of the array are initialized to the same state, one of the strings is accepted at the left end of the array, while the other is accepted at the right end, hence the array conforms to Case 4.2.

The array needs $2n - 1$ cells, $n$ being the number of characters in either string (the two strings are assumed to be of the same length). The input at either end of the array is assumed to be the characters of the string (one character at a time), with each character being followed by an integer such that the integer following a certain character is larger than the integer that followed the previous character by one, e.g. A 1 B 2 C 3 ... This sequence is then followed by $2n - 1$ “pass output” instructions (to allow the last two input items to be processed and the result to reach the output port). The result of the computation (the number of delete/insert steps required) is the last integer that the array sends out as output.

The cells have 3 sets of registers: one set will hold the information about the cell (the internal registers), another set will hold information to be read by the cell to the left (the left output registers), and the last set will hold the
information to be read by the cell to the right (the right output registers). The
following is a quick overview of the algorithm, for a more detailed description
see [Lip85]:

If the cell sees a pass instruction to either side of it then it copies the contents
of the right output registers of the cell to the left to its own right output regis-
ters, and the contents of the left output registers of the cell to the right into
its own left output registers.

If the cell sees that a character is being passed to it from either end then it
does the following: It copies the contents of the left output registers of the cell
to the right into its own left output registers and the contents of the right out-
put registers of the cell to the left into its own right output registers. If it is
being passed a character from only one end then it resets the CompMade flag
(a comparison was not made). If it is being passed a character from both ends
then it sets the CompMade flag (a comparison was made), and then compares
the two characters and sets the EQ flag if the two characters are equal, and
resets it if they are not equal. (Note that the afore mentioned flags together
with a register that can hold an integer make up the internal registers of the
cell.)

Otherwise the cell does the following: If the cell has its CompMade flag reset
(a comparison was not made during the previous time unit) then the cell
copies the larger of the two numbers being passed to it into its internal regis-
ters. Otherwise if the CompMade flag is set and the EQ flag is also set (the
two characters were the same) and the two numbers being passed to the cell are equal and are equal to one larger than the number stored in the internal register then it increments the value in its internal registers by two. After doing one of the previous two possible steps the cell copies the value stored in its internal registers to its left output and its right output registers.
The state of a cell is described by the following:

State42 = record
  int = record { internal registers. } [106x580] [125x580]
    value : integer;
    CompMade : boolean { True if a comparison was made. }
    EQ : boolean { True if the two characters compared } [144x551] [143x536] [143x505] [143x480]
      { were the same. } [143x465]
  end;

lo = record { Left output registers. } [120x476]
  value : integer { values being passed to the left. } [144x461] [144x446]
  ch : char { characters being passed to the left. } [144x431]
  inst : (value, charac, null, pass) { this defines whether the cell is } [254x417] [255x402] [255x387] [255x372] [255x357]
    { passing a value, a character, } [254x327] [254x312] [254x298]
    { a null instruction, or a pass } [254x283] [254x268]
    { instruction. The cells are initialized } [254x254] [254x239] [254x224] [254x209]
    { as passing a null instruction. } [254x195] [254x180]
  end;

ro = record { Right output registers. } [122x313]
  value : integer { values being passed to the right. } [145x298]
  ch : char { characters being passed to the right. } [145x283]
  inst : (value, charac, null, pass) { this defines whether the cell is } [255x254]
    { passing a value, a character, } [255x239]
    { a null instruction, or a pass } [255x224]
    { instruction. The cells are initialized } [255x209]
    { as passing a null instruction. } [255x195]
  end;

end;
The following is the cell function described above:

```
function f42 (Left, Current, Right : State42) : State42;

(*
 * This function describes the operation of a cell in the array. The
 * parameter Current represents the state of the cell whose computation
 * is to be simulated, say Cell i. Left represents the state of
 * Cell i - 1, and Right represents the state of Cell i + 1.
 *
 * Procedures and functions used:
 *
 * passer: defines the operation of the cell if it reads a pass instruction
 * from either of its neighbors.
 * charaction: defines the operation of the cell if it receives a character
 * from either of its neighbors.
 * valaction: defines the operation of the cell if it receives only values
 * both its neighbors.
 *)

begin {function f42}

if (right.lo.inst = pass) or (left.ro.inst = pass) then
   f42 := passer (left, current, right)

else if (right.lo.inst = charac) or (left.ro.inst = charac) then
   f42 := charaction (left, current, right)

else
   f42 := valaction (left, current, right)

end; {function f42}
```
function passer (left, current, right : State42) : State42;

(*
 * After all the characters of the string have been read, the input should
 * have enough 'pass' instruction to allow the result (computed in the
 * middle cell of the array) to reach the ends.
 * Upon reading a pass instruction the cell copies the content of the
 * left output register of the cell to the right into its own left output
 * register, and the contents of the right output register of the cell to
 * the left into its own right output register.
 *)

var
    result : State42;

begin  {function passer}

with result do
    begin
        begin
            lo := right.lo;
            ro := left.ro
        end;

    passer := result

end;   {function passer}
function charaction (left, current, right : State42) : State42;

(*
* This function is called whenever the cell reads a character from either
* of its neighbors.
* If one of its neighbors does not pass it a character than its sets its
* registers to reflect that, otherwise it sets its registers to show that
* a comparison was made and the result of comparing the two characters
* (equal or not equal).
* Then it passes what it has read from the left to the right, and what it
* has read from the right to the left.
*)

var
  result : State42;

begin  {function charaction}

  with result do
    begin
      int := center.int;
      lo := right.lo;
      ro := left.ro;

      if (right.lo.inst = charac) and (left.ro.inst = charac) then
        begin
          int.CompMade := true;
          int.EQ := (left.ro.ch = right.lo.ch)
        end
      else
        int.CompMade := false
      end;  {with statement}

  charaction := result

end;  {charaction}
function valaction (left, current, right : State42) : State42;

(*
* This function calculates the effect of the comparison, that is supposed
* to have happened in the previous time unit, on the edit length.
* If no comparison was made then the cell accepts the larger of the
* two values being passed to it. Otherwise it keeps the value stored
* in its internal registers unless the characters that were compared
* in the previous time unit were not equal, the two numbers passed
* to it are equal, and the numbers passed to it are equal to the
* number stored in the cell’s internal registers added to one, in which
* case the cell increments the value in its internal registers by two.
*
* Then the cell copies the value in its internal registers to its
* left output and its right output registers.
*)

var
    result : State42;

begin

    result := current;

    with result do
        begin
            if not (current.int.CompMade) then
                int.value := max (left.ro.value, right.lo.value)
            else if not (current.int.EQ) then
                if (left.ro.value=right.lo.value) and (left.ro.valu=current.int.valu+1)
                    then int.valu := current.int.value + 2;

            lo.inst := value;
            lo.value := int.value;
            ro := lo
        end; {with statement}

    valaction := result

end; {valaction}
Figure 9.3 shows an array of 5 cells executing the above cell function for 10 time units. The array reads in the strings $ABC$ and $BAC$ and computes the number of editing steps required to transform one into the other (an edit step is a delete or an insert operation). The result of the computation is the last integer that is output from the array, 2 in this example. In figure 9.3 the internal register are represented by three adjacent values: an integer representing the value stored, followed by a Y/N (Yes-True/No-False) field representing the flag $CompMade$, followed by another Y/N field representing the $EQ$ flag, for example $2,Y,N$ means that the cell stores the value 2, has done a comparison, and the two characters compared were not the same. For the left output and right output registers the figure shows only the character if the $inst$ field is $charac$, only the value if the $inst$ field is $value$, only $pass$ if the $inst$ field is $pass$, and only $null$ if the $inst$ field is $null$.

Figure 9.4 shows the array resulting from applying the transformation presented in Chapter 7 to the cell function above. The resulting array simulates the operation of the array in figure 9.3 while conforming to Case 2. The contents of the $Up$, $Down$, $UpOut$, and $downOut$ registers of a cell in the array of figure 9.4 are organized as described above for the array of figure 9.3.
Figure 9.3a: Time units 0 through 4.
Figure 9.3b: Time units 5 through 10.
Figure 9.4a: Time units 0 through 4.
Figure 9.4b: Time units 5 through 9.
Figure 9.4c: Time units 10 through 14
Example 3: IIR filter:
(Bi-directional array, cells initialized to different states, input at left end, output at right end – Bi-directional iterative array.)

This example illustrates transforming an array conforming to Case 3 into an array conforming to Case 2. The cell function is an implementation of the recursive part of an infinite impulse response (IIR) filter and is basically the same as the one presented in Chapter 2.

As mentioned in chapter 2 (and in [CS81]), a K-tap IIR filter that accepts a signal \( x \) and produces the signal \( Z \) can be represented by:

\[
Z_n = \sum_{k=0}^{K-1} (a_k \cdot x_{n-k}) + \sum_{k=1}^{K-1} (b_k \cdot Z_{n-k})
\]  

(1)

or by:

\[
Z_n = Y_n + \sum_{k=1}^{K-1} (b_k \cdot Z_{n-k})
\]  

(2)

where

\[
Y_n = \sum_{k=0}^{K-1} (a_k \cdot x_{n-k}).
\]

The cell function presented here implements equation 2: It accepts the signal \( Y \) as input (the signal \( Y \) is computed from the signal \( x \) using a K-tap FIR filter similar to the one discussed in chapter 2), and produces the signal \( Z \) as output. The function is a two-slow system meaning that it accepts an input item and produces an output item only every other time unit, and at any time unit only half the cells are doing useful computation.

Every cell will have a register to store a partial sum, a register to store the coefficient, and a register to store a value of the \( Z \) signal. Every time unit a
cell multiplies the contents of its coefficient register by the contents of its Z-
signal register, adds the result to the contents of the partial sum register of
the cell to the left, and stores the sum in its own partial sum register. Then it
copies the value stored in the Z-signal register of the cell to the right into its
own Z-signal register. It can be seen that the partial sums flow from left to
right while the Z-signal values flow from right to left, and hence the array has
a bi-directional data flow. Every cell is given its coefficient before the array
starts computing and hence the cells are initialized to possibly different
values.

The operation of the last cell differs somewhat from that of the other cells.
The partial sum calculated by the last cell constitutes a value of the Z-signal
and is sent out of the array as output, and is sent to the left after a delay of
one time unit.
The state of a cell is described by the following:

\[
\text{State3} = \text{record}
\]

\[
\begin{align*}
Y &: \text{integer}; \quad \{\text{Holds the partial sum calculated by the cell.}\} \\
Z &: \text{integer}; \quad \{\text{Holds a value of the } Z\text{-signal as it is being passed to the left.}\} \\
\text{Zint} &: \text{integer}; \quad \{\text{Used only in the last cell of the array to introduce a delay of one time unit in the path of the } Z\text{-signal.}\} \\
\text{Coeff} &: \text{integer}; \quad \{\text{Holds the coefficient assigned to the cell.}\} \\
\text{Position} &: (\text{mdle, last}); \quad \{\text{Tells the position of the cell in the array:}\} \\
\end{align*}
\]

\[
\begin{align*}
\quad &: \quad \{\text{The last cell has this field set to "last",}\} \\
\quad &: \quad \{\text{the others have it set to "mdle".}\} \\
\end{align*}
\]

end;
The following is the cell function for the IIR filter.

function f3 (Left, Current, Right : State3) : State3;

(*
* This cell function defines the operation of a cell in the array.
* Current represents the state of the current cell,
* Left represents the state of the cell to the left of the
* current cell, and
* Right represents the state of the cell to the right of
* the current cell.
*)

var
    result : State3;

begin {function f3}
    result := Current;

    if Current.Position = Last then { The current cell is the last in the array. }
        begin
            result.Z := Current.Zint;
            result.Zint := result.Y
        end
    else begin { The current cell is NOT the last cell. }
        result.Z := Right.Z
    end;

    return (result)
end; {function f3}
In this example the value zero is used to indicate an empty register, however it is recommended that a flag be used (rather than a special value) to indicate such special states of the register (e.g. empty, full, etc.). That would make the functions easier to debug, read, and explain.

Figure 9.5 shows an array of 4 cells, each executing the above function for 12 time units. The result of the computation are the contents of the $Y$ register of the last cell. The coefficients of the cells are set to 1, 1, 0, and -1 from left to right respectively. The array accepts the sequence 17, 15, 10, 3, 17, 12 as input and produces 7, 8, 2, 8, and 24 as output.

The above cell function can be transformed, as described in chapter 6, into a cell function for a bi-directional iterative array (Case 2). Figure 9.6 shows an array executing the resulting cell function. The contents of the $Current$, $Right$, $Pass$, and $Out$ registers is the same as the one used to define the cell state in figure 9.5.
Cell state as used in the figure below.

Figure 9.5a: Organization of the cell state, and Time units 0 through 4.
Figure 9.5b: Time units 5 through 12.
Figure 9.6a: Time units 0 through 5.
Figure 9.6b: Time units 6 through 11.
Figure 9.6c: Time units 12 through 17.
Chapter 10

Concluding Remark

This work shows that there are no algorithms that are inherently bi-directional, only algorithms that are at best more convenient if implemented using bi-directional data flow. Any algorithm can be solved on a uni-directional array, and if systolic rings are used then the ring will have the same number of processing elements (cells) as the original bi-directional array (though the complexity of a single cell will increase by a constant factor) even if only one cell does all the input and all the output of the array. With this in mind, researchers might lean towards solving problems using only uni-directional data flow (even if the cell function has to be somewhat more complex) because that will put the benefits of recyclability and improved fault tolerance at their disposal [SS87].

As mentioned earlier, during the course of this study a software system that implements the above transformations was built [KSS87]. It is hoped that this system will provide a unified approach for solving problems on linear systolic arrays. The system is made up of a compiler and a simulator. The simulator accepts a cell function and simulates an array executing it, while the compiler implements the transformations discussed in this thesis.

However, the real value of the system lies elsewhere. A user would design a bi-directional algorithm, and use the system to transform it into a uni-directional one. He can then use the resulting function as is or can prune it to relieve it of the extra work that is not necessary in that particular case. The real value of the
system can be felt when by examining the function produced by the system (pruned or not) the user might deduce some relationships or hints that will enable him/her to design a uni-directional function that will carry out the same computation as the original function. The new function should exceed or at least match the one produced by the system in terms of space efficiency, performance in time, and readability.

The transformations presented in this thesis produce cell functions that achieve the required transformations with acceptable space efficiency and performance in time. All the resulting functions are linear in both time and space, however the constant factors might be improved upon. It was mentioned in [CY85] that they could transform a bi-directional real time Cellular Automaton (array conforming to Case 1 having \( n \) cells and computing for \( n \) time units) into a uni-directional iterative array in \( 2n - 1 \) time units, whereas the transformation presented here will require \( 3n \) time units. Thus, one avenue for possible future work is to try to improve upon the transformations presented here for savings in time or space or both.

The transformations presented in this thesis apply to linear systolic arrays, and a possible area for future work is to use the knowledge gained by examining those transformations to build upon them, extend them, or devise ones that apply to different kinds of arrays. Work is underway to devise ways that will transform two-dimensional (mesh-connected) arrays with bi-directional data flow (data flows from top to bottom, bottom to top, left to right, and right to left) into ones that
have only uni-directional data flow (data flows only from top to bottom and from left to right) [Sav87].

The two main reasons that prompted this study are that linear arrays with uni-directional data flow have better fault tolerance capabilities, and algorithms that are designed as described in chapter 1 are “recyclable” [SS87]. Hence, as far as the software system is concerned, the next logical step is to build into it facilities that will allow the user to “fail” certain cells, and facilities that will allow for recycling of the output as described in [SS87].
References


[GL] Leo J. Guibas, and Frank M. Liang, “Systolic Stacks, Queues, and Counters”, manuscript.


[Sav87] Carla D. Savage, personal communication.


register $R$ and $Q$ is empty, then $R$ would be empty after the operation. Also, if a cell moves the contents of $Q$ into $R$ then $Q$ would be empty after the transaction, also if $Q$ was empty to start with then $R$ would be empty as well after the transaction.

The following is an example of how the new cell function ($g_B$) would look:

```plaintext
function $g_B(A,B)$ returns StateB is
result := B
if Control_State(B) = OFF then
   /* The cell will stay OFF until it can see a START to its left. */
   if Control_State(A) = START then
      result := Ø
      Control_State(result) := START
   end if
else if Control_State(B) = START then
   Control_State(result) := ON
else if Control_State(B) = ON then
   /* Stay in the control state ON until the cell to the left is active or in the special state START. */
   if Control_State(A) = OFF or ON then
      do nothing
   else if Control_State(A) = START then
      Control_State(result) := FIRST
   else if Control_State(A) = INTERMEDIATE then
      Right_State(result) := Ø
      Current_State(result) := Current_State(A)
      Control_State(result) := MAIN
   else if Control_State(A) = MAIN then
      Current_State(result) := Ø
      Control_State(result) := INTERMEDIATE
   else if Control_State(A) = FIRST then
      Current_State(result) := Ø
      Control_State(result) := INTERMEDIATE
   end if
else if Control_State(B) = FIRST then
   /* This is the first cell of the computation, and will simulate the operation of the first cell of $A r_2$. After carrying out the computation it will either become inactive or will turn into the last cell of the computation. */
   if Current_Input(B) = Ø then
      Current_State(result) := Ø
      Right_State(result) := Ø
      Current_Input(result) := Ø
      New_Input(result) := Ø
   else
      Current_State(result) := f2(Current_Input(B), Current_State(B),
```
Current_Input(result) := ∅
end if
if Control_State(A) = ON then
    Control_State(result) := ON
else if Control_State(A) = MAIN then
    Control_State(result) := LAST
else
    erroneous combination.
end if
else if Control_State(B) = LAST then /* Output the result of the previous computation (in the cell to the left). */
    New_Output(result) := Current_State(A)
    Current_State(result) := Current_State(A)
    Right_State(result) := ∅
    Control_State(result) := MAIN
else if Control_State(result) = MAIN then
    if Current_State(A) = ∅ then
        Current_State(result) = ∅
        Right_State(result) := ∅
    else
        Current_State(result) := \( f_2(\text{Current}_\text{State}(A), \text{Current}_\text{State}(B), \text{Right}_\text{State}(B)) \)
    end if
    Control_State(result) := INTERMEDIATE
else if Control_State(B) = INTERMEDIATE then
    Right_State(result) := Current_State(B)
    Current_State(result) := Current_State(A)
    if Control_State(A) = INTERMEDIATE then
        Control_State(result) := MAIN
    else if Control_State(A) = ON or LAST then
        Control_State(result) := FIRST
    else
        erroneous combination.
    end if
end if
end if
if Current_Input(result) = ∅ then
    Current_Input(result) := New_Input(A)
    New_Input(result) := ∅
else
    New_Input(result) := New_Input(A)
end if
if Pass(A) = ∅ then
    Pass(result) := New_Output(result)
    New_Output(result) := ∅
else
    Pass(result) := Pass(A)
end if
return result
end \( g_B \)