8
Data Flow and Systolic Array Architectures

8.1 INTRODUCTION

This chapter describes the structure of two parallel architectures, *data flow* and *systolic array*. In the data flow architecture an instruction is ready for execution when data for its operands have been made available. Data availability is achieved by channeling results from previously executed instructions into the operands of waiting instructions. This channeling forms a flow of data, triggering instructions to be executed. An outcome of this is that many instructions are executed simultaneously, leading to the possibility of a highly concurrent computation. The next section details the theory behind the data flow concept and explains one of the well-known designs of a data flow machine, the MIT machine.

In a systolic array there are a large number of identical simple processors or processing elements (PEs). The PEs are arranged in a well-organized structure, such as a linear or two-dimensional array. Each PE has limited private storage and is connected to neighboring PEs. Section 8.3 discusses several proposed architectures for systolic arrays and also provides a general method for mapping an algorithm to a systolic array.

8.2 DATA FLOW ARCHITECTURE

To demonstrate the behavior of a data flow machine, a graph, called a *data flow graph*, is often used. The data flow graph represents the data dependencies between individual instructions. It represents the steps of a program and serves as an interface between system architecture and user programming language. The nodes in the data flow graph, also called *actors*, represent the operators and are connected by input and output arcs that carry tokens bearing values. Tokens are placed on and removed from the arcs according to certain firing rules. Each actor requires certain input arcs to have tokens before it can be fired (or executed). When tokens are present on all required input arcs of an actor, that actor is enabled and thus fires. Upon firing, it removes one token from the required input arcs, applies the specified function to the values associated with the tokens, and places the result tokens on the output arcs (see Figure 8.1a). Each node of a data flow graph can be represented (or stored) as an *activity template* (see Figure 8.1b). An activity template consists of fields for operation type, for storage of input tokens, and for destination addresses. Like a data flow graph, a collection of activity templates can be used for representing a program. Figure 8.2 represents a set of actors that is used in a data flow graph. There are two types of tokens: data tokens and Boolean tokens. To distinguish the type of inputs to an actor, solid arrows are used for carrying data tokens and outlined arrows for Boolean tokens. The switch actor places the input token on one of the output arcs based on the Boolean token arriving on its input. The merge actor places one of the input tokens on the output arc based on the Boolean token arriving on its input. The *T* gate places the input token on the output arc whenever the Boolean token arriving on its input is true, as the *F* gate does whenever the Boolean token arriving on its input is false. As an example, Figure 8.3 represents a data flow graph with its corresponding templates for the following *if* statement:

$$\text{if } x>3 \{ y = (x+2) * 4; \} \text{ else } \{ y = (x-1) * 4; \}.$$

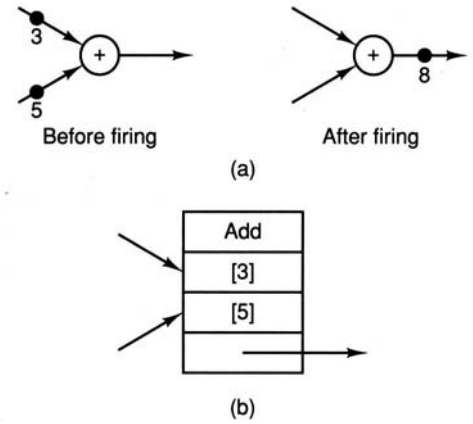

Figure 8.1 (a) Data flow graph and (b) its corresponding activity template.

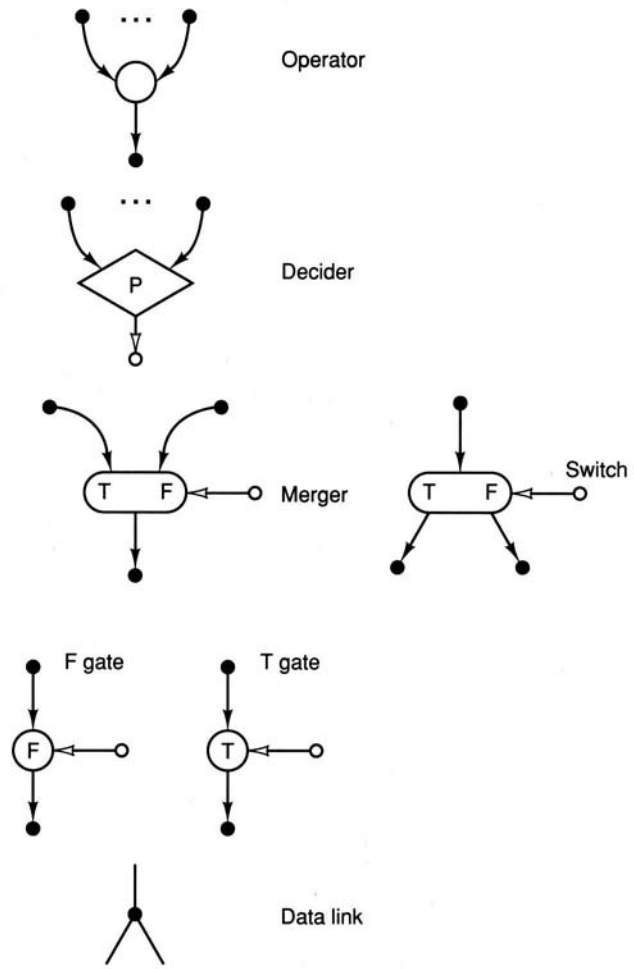

Figure 8.2 Set of actors for constructing a data flow graph.

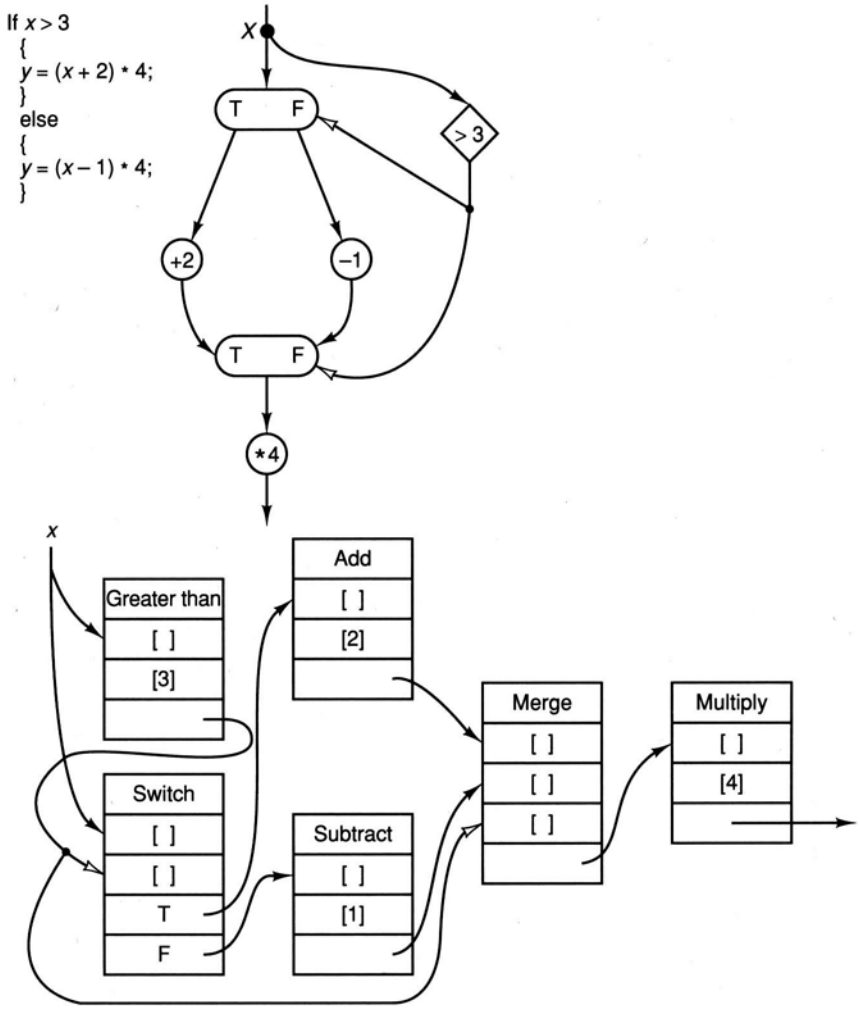

Figure 8.3 A data flow graph and its corresponding activity template for an *if* statement.

Another example is shown in Figure 8.4. For a given *N*, the data flow graph represents computation of *N*!. Note in this graph that, the Boolean input arcs to both mergers are initialized to false tokens. At the start this causes the data input token *N* to move to the output of both mergers.

```
Input N
    i = N;
    While  i > 1
        {
        N = N * (i – 1);
        i = i – 1;
        }
Output N
```

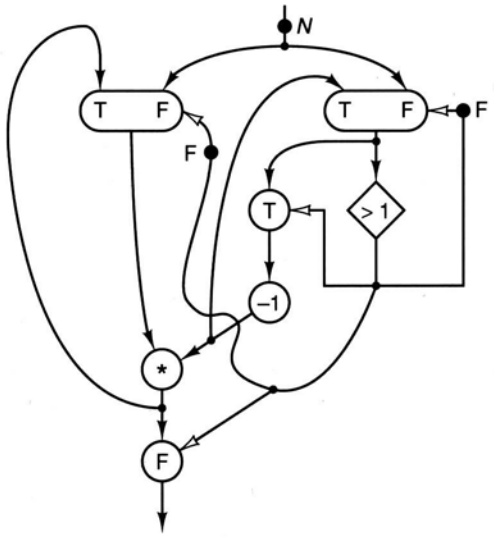

Figure 8.4 A data flow graph for calculating *N*!.

8.2.1. Basic Structure of a Data Flow Computer

To demonstrate the structure of a data flow machine, Figure 8.5 represents the main elements of a data flow machine called the MIT data flow computer [DEN 80]. The MIT computer consists of five major units: (1) the processing unit, consisting of specialized processing elements; (2) the memory unit, consisting of instruction cells for holding the instructions and their operands (i.e., each instruction cell holds one activity template); (3) the arbitration network, which delivers instructions to the processing elements for execution; (4) the distribution network for transferring the result data from processing elements to memory; and (5) the control unit, which manages all other units.

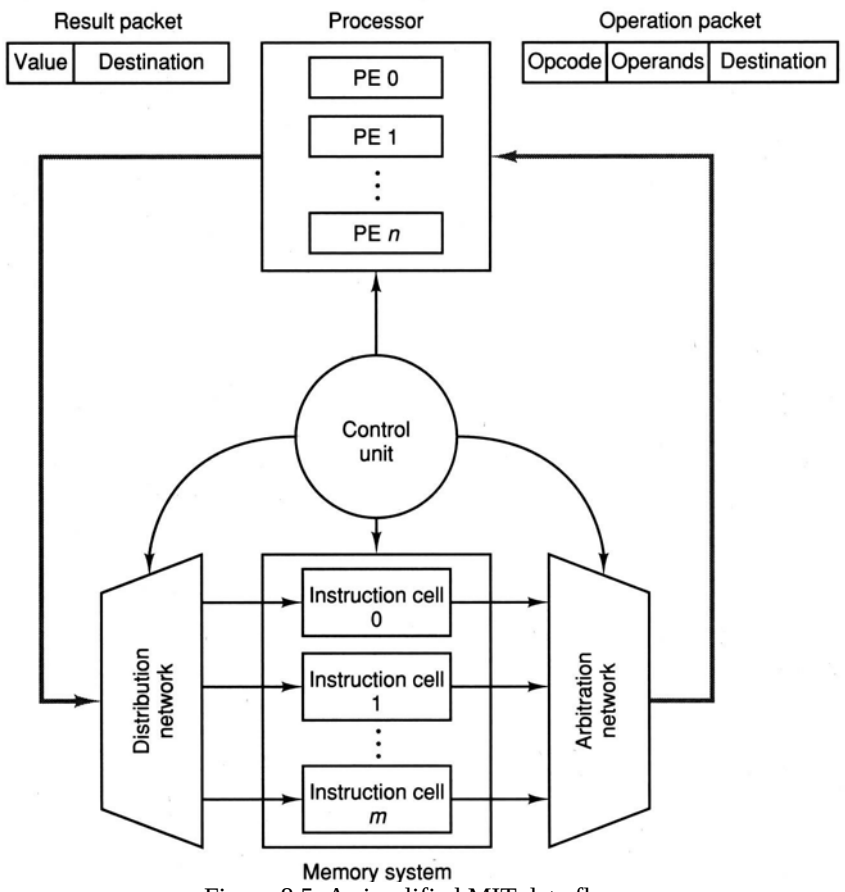

Figure 8.5 A simplified MIT data flow.

An instruction cell holds an instruction consisting of an operation code (opcode), operands, and a destination address. The instruction is enabled when all the required operands and control signals are received. The arbitration network sends the enabled instruction as an operation packet to the proper processing element. Once the instruction is executed, the result is sent back through the distribution network to the destination in memory. Each result is sent as a packet, which consists of a value and a destination address.

Proposed data flow machines can be classified into two groups: *static* and *dynamic*. In a static data flow machine, an instruction is enabled whenever all the required operands are received and another instruction is waiting for the result of this instruction; otherwise, the instruction remains disabled [DEN 80, COR 79]. In other words, each arc in the data flow graph can carry at most one token at any instance. An example of this type of data flow graph is shown in Figure 8.6. The multiply instruction must not be enabled until its previous result has been used by the add instruction. Often, this constraint is enforced through the use of acknowledgment signals.

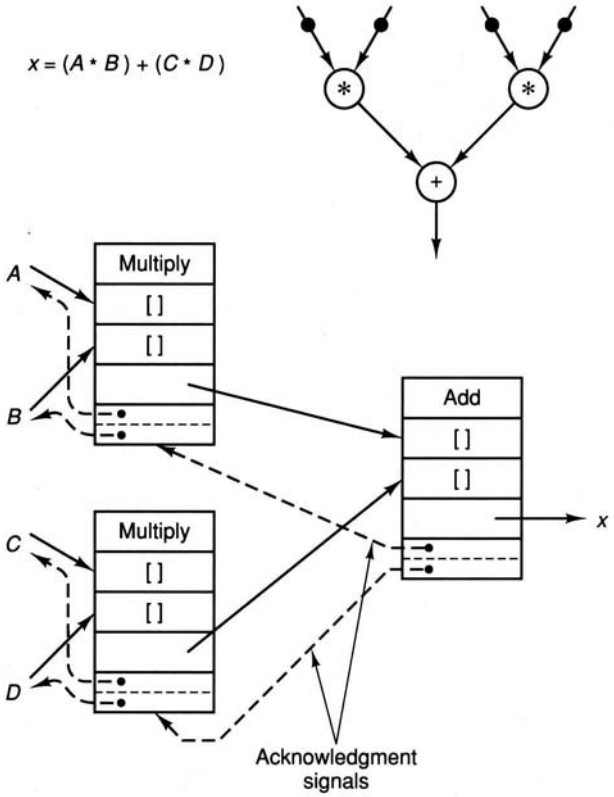

Figure 8.6 An example of a data flow graph for a static data flow machine. Each arc in the data flow graph can carry at most one token at any instance.

In a dynamic data flow machine an instruction is enabled whenever all the required operands are received. In this case, several sets of operands may become ready for an instruction at the same time. In other words, as shown in Figure 8.7, an arc may contain more than one token. Compared with static data flow, the dynamic approach allows more parallelism because an instruction need not wait for an empty location in another instruction to occur before placing its result. However, in the dynamic approach a mechanism must be established to distinguish instances of different sets of operands for an instruction. One way would be to queue up the instances of each operand in order of their arrival [DAV 78]. However, maintaining many large queues becomes very expensive. To avoid queues, the result packet format is often extended to include a label field; hence matching operands can be found by comparing the labels [ARV 80, GUR 80]. An associative memory, called *matching store*, can be used for matching these labels. As an example, Figure 8.8 represents a possible architecture for a memory unit of a dynamic data flow machine. Each time that a result packet arrives at the memory unit, its address and label fields are stored in the matching store. The result packet's value field is stored in the data store. The matching store uses the address and label fields as its search key for determining which instruction is enabled. In Figure 8.8, it is assumed that three packets have arrived at the memory unit at times t_0, t_1, and t_2. At time t_2, when both operands with the same label for the add instruction (stored at location 100 in instruction store) have arrived, the add instruction becomes enabled.

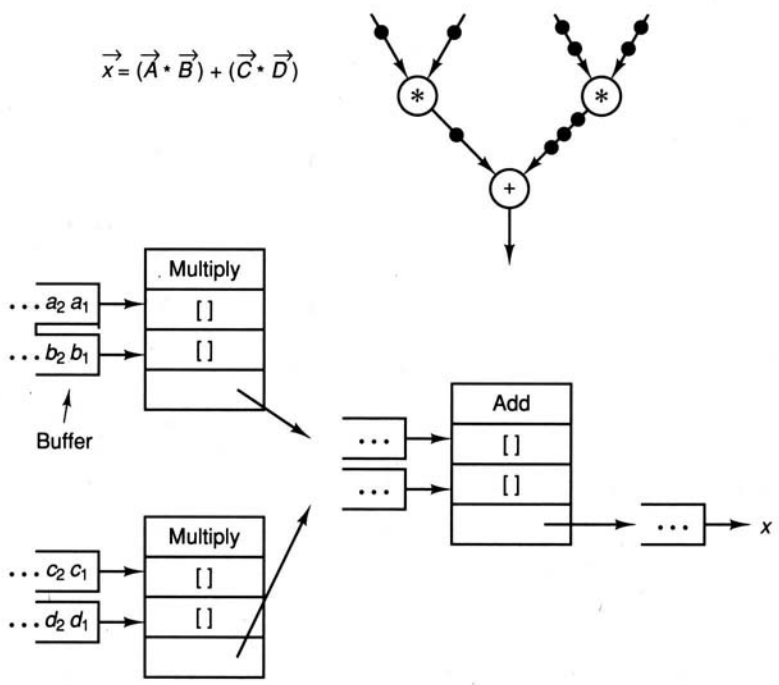

Figure 8.7 An example of a data flow graph for a dynamic data flow machine. An arc in the data flow graph may carry more than one token at any instance.

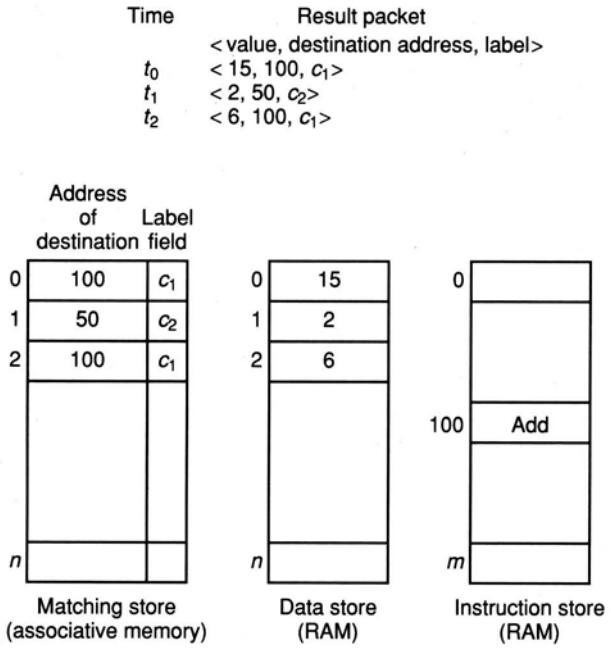

Figure 8.8 A possible memory unit for a dynamic data flow computer.

8.3 SYSTOLIC ARRAYS

In application fields of scientific computing, it is very often necessary to solve simultaneous linear equations, in particular, a large-scale linear system of equations. Finding fast, accurate, and cost-effective methods for solving a large-scale linear system of equations has been greatly needed for centuries, both by scientists and engineers. Nevertheless, we are always dealing with matrix algebra, like LU decomposition,

inversion, and multiplication. Due to the lengthy sequences of arithmetic computations, most large-scale matrix algebra is performed on high-speed digital computers using well-developed software packages. But a major drawback in processing matrix algebra on general-purpose computers by software programs is the need for long computation time. Also, in a general-purpose computer the main memory is not large enough to accommodate very large-scale matrices. Thus, many time-consuming I/O transfers are needed in addition to the CPU computation time.

To alleviate this problem, the use of parallel computers has been adopted and special-purpose machines have been introduced. One solution to the need for highly parallel computational power is the connection of a large number of identical simple processors or processing elements (PEs). Each PE has limited private storage, and is only allowed to be connected to neighboring PEs. Thus all PEs are arranged in a well-organized structure such as a linear or two-dimensional array. This type of structure, referred to as a *systolic array*, provides an ideal layout for VLSI implementation. Often, interleaved memories are used to feed data into such arrays.

Usually, a systolic array has a rectangular or hexagonal geometry, but it is possible for it to have any geometry. With VLSI technology, it is possible to provide extremely high but inexpensive computational capability with a system consisting of a large number of identical small processors organized in a well-structured fashion. In other words, through progress in VLSI technology, a low-cost array of processors with high-speed computations can be utilized.

Various designs of systolic arrays with different data stream schemes for matrix multiplication have been proposed. Some of the proposed designs are *hexagonal arrays*, *pipelined arrays*, *semibroadcast arrays*, *wavefront arrays*, and *broadcast arrays*. In this section, we will discuss these proposed designs and the drawbacks of each, thereby making a performance comparison among them. Finally, a general method is given for mapping an algorithm to a systolic array.

8.3.1 Basic Terminology and Proposed Arrays of Processors

Before describing various systolic arrays, some terminology common to all designs will be given. First, the processing element primarily used in each design is basically an inner-product step processor that consists of three registers: R_a, R_b, and R_c. These registers are used to perform the following multiplication and addition in one unit of computational time:

$$R_c = R_c + R_a * R_b.$$

The unit of computational time is defined as $t_a + t_m$, where t_a and t_m are the time to perform one addition and one multiplication, respectively.

To compare different proposed arrays of processors, two factors are considered: number of required PEs and turnaround time. Let P denote the number of required PEs. The turnaround time, T, is defined as the total time, in time unit $t_a + t_m$, needed to complete the entire computation.

In the following paragraphs, several proposed architectures for systolic arrays are discussed. The structure of each design is illustrated by an example that performs the computation $C=A*B$, where

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

Hexagonal array. The Hexagonal array, proposed by Kung and Leiserson [KUN 78], is a good example of the effective use of a large number of PEs arranged in a well-organized structure. In a hexagonal array, each PE has a simple function and communicates with neighbor PEs in a pipelined fashion. PEs on the boundary also communicate with the outside world. Figure 8.9 presents a hexagonal array for the multiplication of two 3-by-3 matrices *A* and *B*. Each circle represents a PE that has three inputs and three outputs. The elements of the *A* and *B* matrices enter the array from the west and east along two diagonal data streams. The entries of *C* matrix, with initial values of zeros, move from the south to north of the array. The input and output values move through the PEs at every clock pulse. For example, considering the current situation in Figure 8.9, the input values a_{11} and b_{11}, and the output value c_{11} arrive at the same processor element, PE_i, after two clock pulses. Once all these values have arrived, the PE computes a new value for c_{11} by performing the following operation:

$$c_{11} = c_{11} + a_{11} * b_{11}.$$

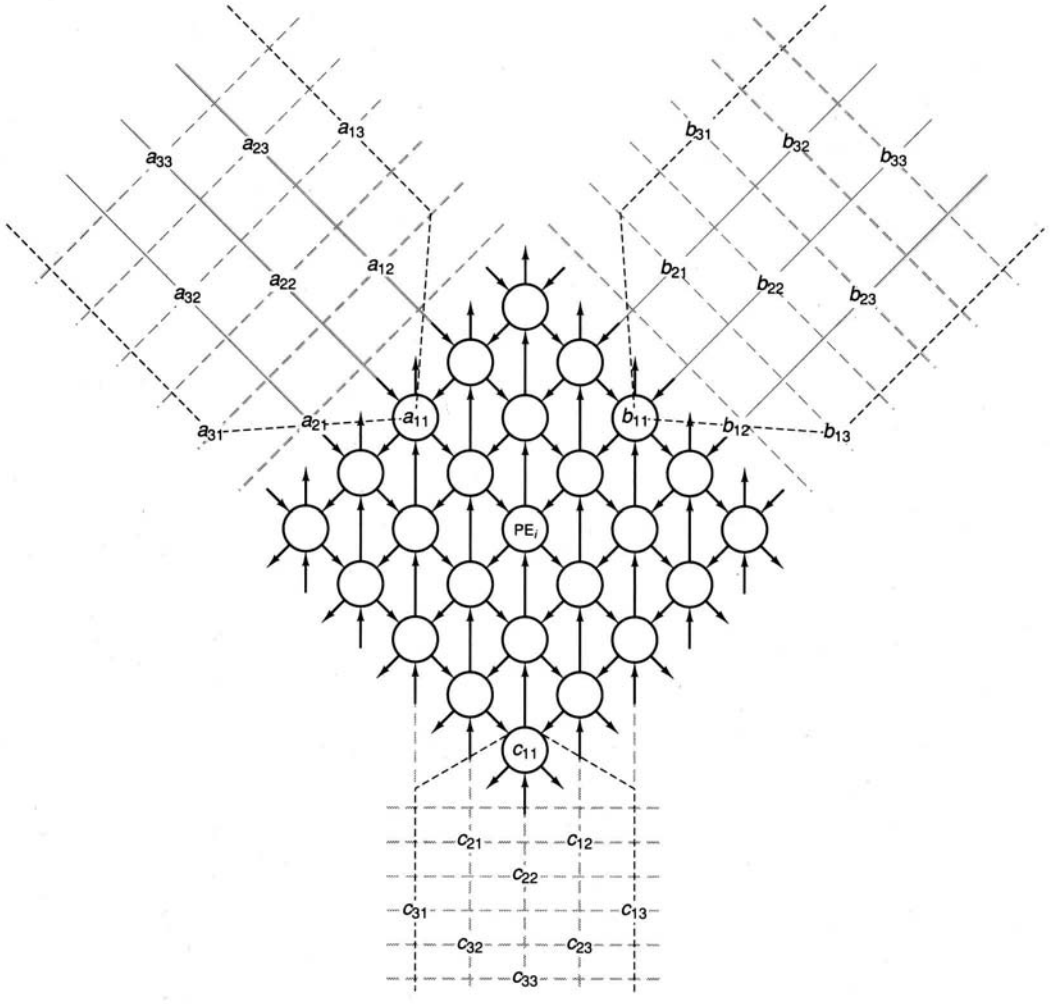

Figure 8.9 Block diagram of a hexagonal array.

There are 25 PEs in the hexagonal array of Figure 8.9. However, only 19 of them contribute to the computation; those are the PEs on which the elements of matrix *C* pass through. Assuming that the first

element of input data (i.e., a_{11} or b_{11}) enters the array after one unit of time, the turnaround time of such an array is 10 units. In general, for a hexagonal array with two *n*-by-*n* matrices, we have the following:

$P = (2n - 1)^2$,
$T = 4n - 2$ when part of the result stays in the array,
$T = 5n - 3$ when the result is transferred out of the array.

Note that only $3n^2 - 3n + 1$ out of $(2n-1)^2$ PEs contribute toward the computation. The major drawback of hexagonal arrays is that they do not pipe data elements into every PE in every time unit. This causes the utilization of PEs to be less than 50%. Also, the hexagonal arrays require a large number of PEs. Because of these drawbacks, several other designs, which were intended to make improvements in this area, have been proposed. Some of them are described next.

Pipelined array. The pipelined array was proposed by Hwang and Cheng [HWA 80]. As shown in Figure 8.10, this architecture has a rectangular array design. The elements of matrices *A* and *B* are fed from the lower and upper input lines in a pipelined fashion, one skewed row or one skewed column at a time. In general, for a pipelined array with two *n*-by-*n* matrices, we have the following:

$$P = n(2n - 1),$$
$$T = 4n - 2.$$

The pipelined array has a relatively simple design. The data flow is basically the same as for the hexagonal array, but uses less PEs. However, similar to hexagonal array, it does not pipe data elements into every PE in every time unit. This idles at least 50% of the PEs at any given time.

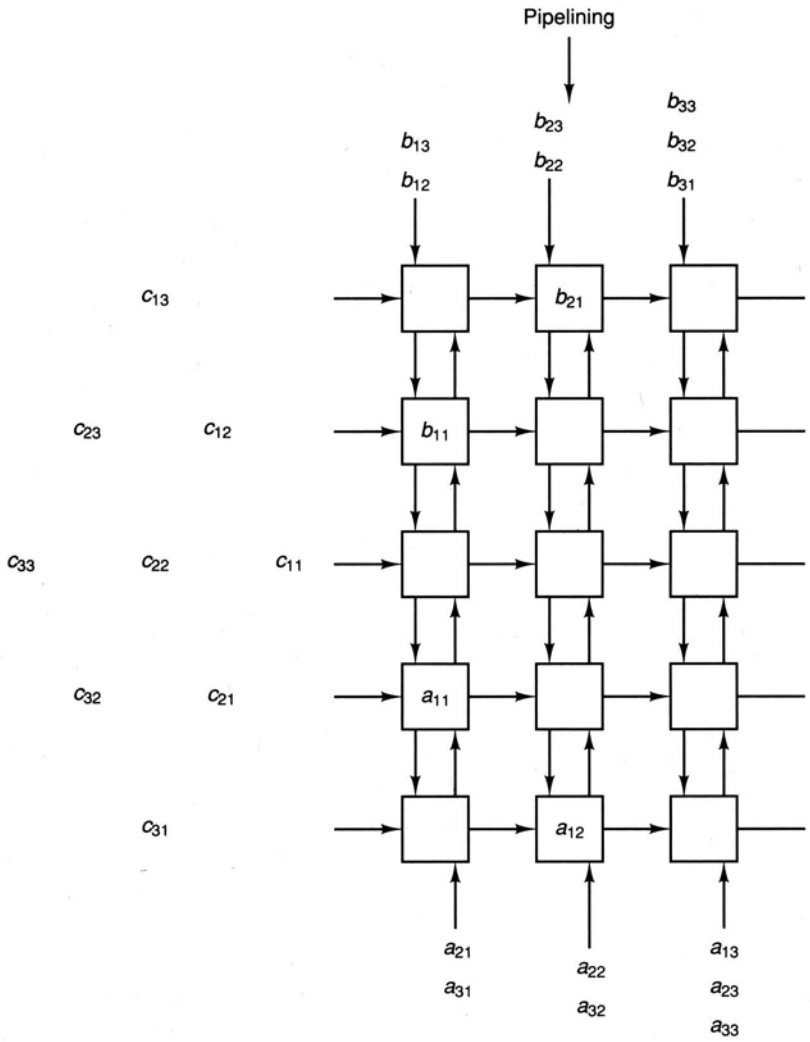

Figure 8.10 Block diagram of a pipelined array.

Semibroadcast array. The semibroadcast array was proposed by Huang and Abraham [HUA 82]. Figure 8.11 represents the structure of a semibroadcast array with data streams. Here semibroadcast means one-dimensional data broadcasting. For example, in Figure 8.11, only the matrix *A* is broadcast from the left side of the array, while matrix *B* is still pipelined into the array from the top edge. The result, matrix *C*, can be either left in the array or transferred out of the array. In general, for a semibroadcast array with two *n*-by-*n* matrices, we have the following:

$P = n^2$,
$T = 2n$ when the result stays in the array,
$T = 3n$ when the result is transferred out of the array.

Compared with the previous designs, the semibroadcast array uses less PEs and also requires less turnaround time. The most controversial point is that the propagation delay of the broadcast data transfer may be longer than that for a nonbroadcast array. Therefore, the degree of complexity controlling a semibroadcast array may be higher than that of a pipelined array. This exemplifies the trade-off between cost and time. Probably the simplest implementation of broadcast arrays is made by connecting the PEs using common bus lines. But the cost, compared to pipelined arrays, would be higher. For example, in a VLSI implementation the bus lines require extra layout area.

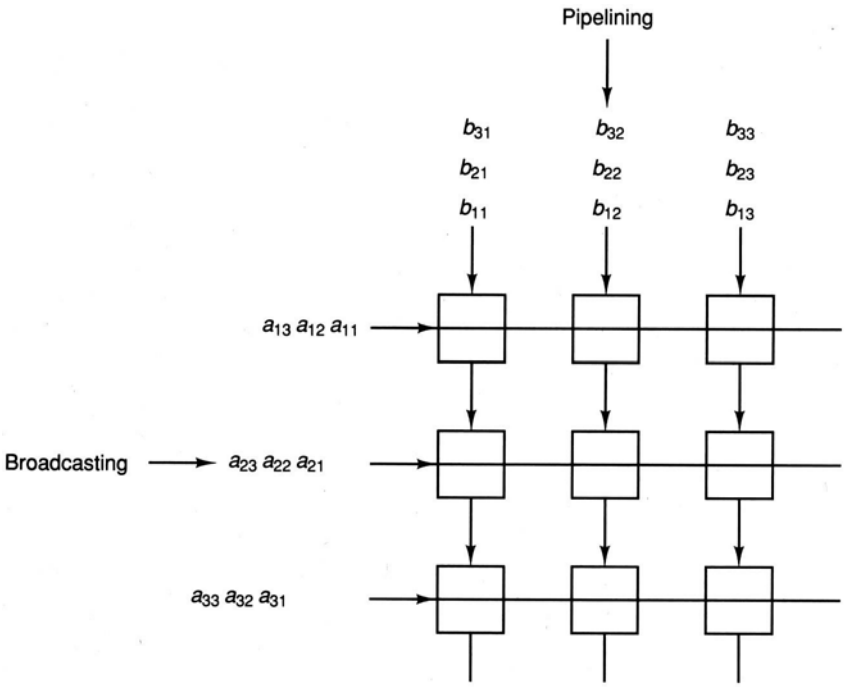

Figure 8.11 Block diagram of a semibroadcast array.

Wavefront array. The wavefront array was proposed by Kung and Arun [KUN 82]. The structure of a wavefront array with a data stream is shown in Figure 8.12. Given two *n*-by-*n* matrices *A* and *B*, the matrix *A* can be decomposed into columns A_i and matrix *B* into rows B_j. Thus

$$C = A_{1*}B_1 + A_{2*}B_2 + ... + A_{n*}B_n.$$

The matrix multiplication can then be carried out by the following *n* iterations:

$$C^{(k)} = C^{(k-1)} + A_{k*}B_k \qquad \text{recursively, for } k = 1, 2, ..., n.$$

These iterations can be performed by applying the concept of a wavefront. Successive pipelining of the wavefronts will accomplish the computation of all iterations. As the first wave propagates, we can execute the second iteration in parallel by pipelining a second wavefront immediately after the first. For example, in Figure 8.12, the process starts with $PE_{1,1}$ computing $C_{11}^{(1)}=C_{11}^{(0)}+a_{11*}b_{11}$. The computational activity then propagates to the neighboring processor elements, $PE_{1,2}$ and $PE_{2,1}$. While $PE_{1,2}$ and $PE_{2,1}$ are executing $C_{12}^{(1)}=C_{12}^{(0)}+a_{11}*b_{12}$ and $C_{21}^{(1)}=C_{21}^{(0)}+a_{21*}b_{11}$, respectively, the $PE_{1,1}$ can execute $C_{11}^{(2)}= C_{11}^{(1)}+a_{12*}b_{21}$. In general, for a wavefront array with two *n*-by-*n* matrices, we have the following:

$P = n^2$,
$T = 3n-2$, when the result stays in the array,
$T = 4n-2$, when the result is transferred out of the array.

The wavefront array represents a well-synchronized structure. The data flow is basically the same as for a hexagonal array except that it does not pipe elements of matrix *C* into the array. This implies that it pipes data elements into every PE during computation for some amount of time (at least half of the total turnaround time). However, its drawback is that the utilization of PEs is at most 50%. Compared to the previous designs, it uses many fewer PEs than does the pipelined array, but it needs more turnaround time than does semibroadcasting.

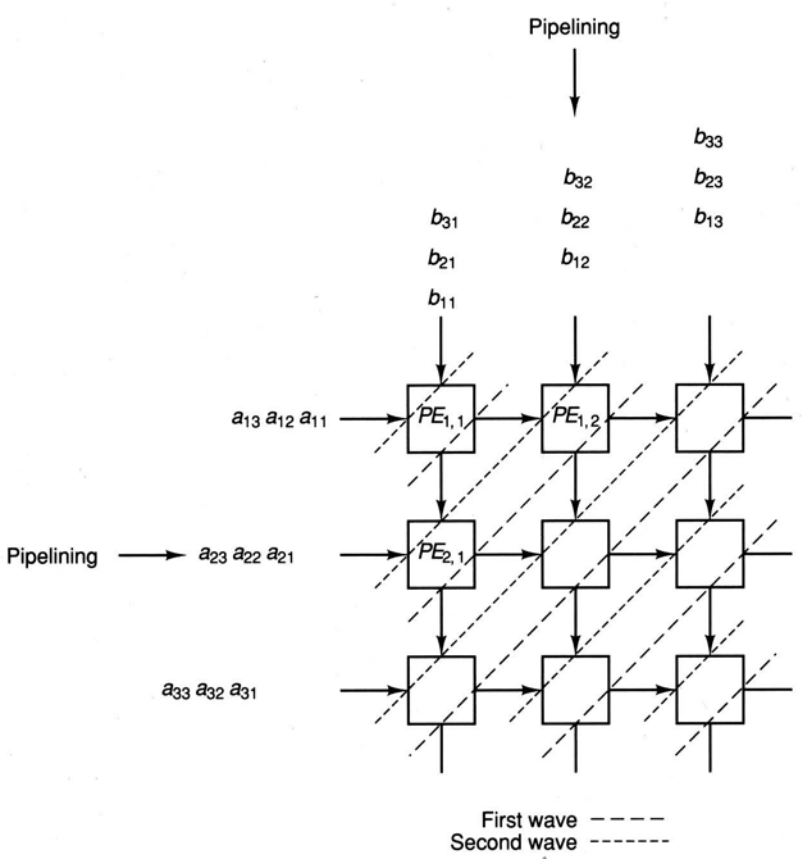

Figure 8.12 Block diagram of a wavefront array.

Broadcast array. The broadcast array was proposed by Chern and Murata [CHE 83]. The structure of a broadcast array with data stream is shown in Figure 8.13. In this design, a two-dimensional data broadcast scheme is introduced. That is, the data can be broadcast in two directions, by row and by column, across the array. In general, for two n-by-n matrices, we have

$P = n^2$,
$T = n$ when the result stays in the array,
$T = 2n$ when the result is transferred out of the array.

The broadcast array has less turnaround time than the previous designs. Also, it has higher utilization of PEs. Furthermore, for dense matrix multiplication, no matrix data rearrangement is required. However, the most controversial point about this design is, as with the semibroadcast array, the time delay due to data broadcasting. Again, the degree of complexity for controlling the broadcast array may be higher than that of the pipelined array. The data bus lines needed by broadcast arrays are twice as many as those needed by semibroadcast arrays. Hence they occupy more layout area.

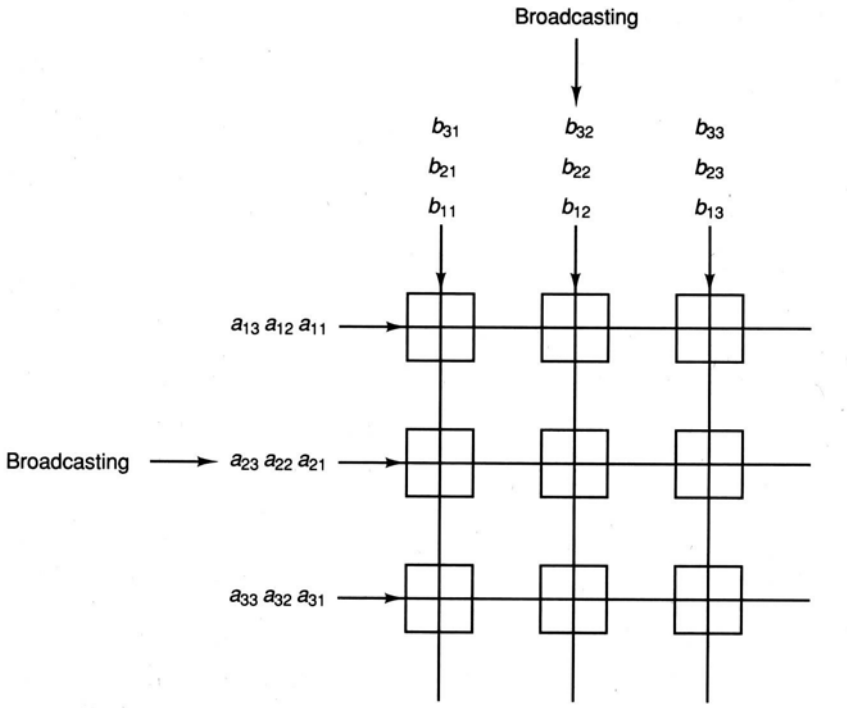

Figure 8.13 Block diagram of a broadcast array.

8.3.2 Mapping Algorithm to Systolic Architecture

Kuhn and Moldovan have proposed procedures for mapping algorithms with loops into systolic arrays [KUH 80, MOL 83]. The mapping procedures start by transforming the loop indexes of a given algorithm to new loop indexes, which allows parallelism and pipelining in the algorithm. Buffer variables are introduced in the transformed algorithm to implement the flow of a value of a set of loop indexes, (broadcast data) in a pipelined fashion. The data pipelining makes the algorithm suitable for VLSI design. Once the transformation is done, the transformed algorithm is mapped into a VLSI array. At this step, the function of each PE and the interconnections between them are determined.

A technique for mapping an algorithm into a systolic array is described next. The following two definitions are given to facilitate the explanation of such a technique.

Definition 1. An algorithm A can be defined as a five tuple $A=(I^n, C, D, X, Y)$, where

I^n	is a finite index set of A,
C	is the set of computations of A,
D	is the set of data dependencies,
X	is the set of input variables of A,
Y	is the set of output variables of A.

Definition 2. Two algorithms $A = (I^n, C, D, X, Y)$ and $A' = (I'^n, C', D', X', Y')$ are equivalent if and only if

1. Algorithm A is input/output equivalent to A'; that is, $X=X'$ and $Y=Y'$.
2. The index set of A' is the transformed index set of A; $I'^n = T(I^n)$, where T is a bijection and monotonic function.
3. Any operation of A corresponds to an identical operation in A', and vice-versa; thus $C=C'$.
4. Data dependencies of A' are the transformed data dependencies of A; $D' = T(D)$.

The matrix T is a bijection, and it is also a monotonic function. This is because we need to keep the data dependence of the original algorithm after transformation.

The matrix T is partitioned into two functions as follows:

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$$

where $\Pi : I^n \rightarrow I'^k \quad (n > k)$, and $S : I^n \rightarrow I'^{n-k}$.

The first k coordinates of elements I'^n can be related to time. The last n-k coordinates can be related to (space) the geometrical properties of the algorithm. In other words, time is associated with the new lexicographical ordering imposed on the elements I'^n, and this is given only by their first k coordinates. The last n-k coordinates can be chosen to satisfy expectations about the geometrical properties of the algorithm. In the remainder of this section, we consider transformed functions for which the ordering imposed by the first coordinate of the index set is an execution ordering. That is,

$$\prod : I^n \rightarrow I'^1$$
$$S : I^n \rightarrow I'^{n-1}.$$

The mapping $\prod$ is selected such that the transformed data dependence matrix D' has positive entries in the first row. This ensures a valid ordering, That is, $\prod d_i > 0$ for any $d_i \in D$. Thus a computation indexed by $I \in I^n$ will be processed at time $\prod * I$.

Steps of the mapping procedure. We use the following steps for the mapping procedure:

1. Buffer all the variables.
2. Determine the PEs functions by collecting the assignment statements in the loop bodies into m input and n output functions.
3. Apply a linear reindexing transformation T.
4. Find connections between processors and the direction of data flow.

As an example, consider the following algorithm which represents multiplication of two 2-by-2 matrices A and B

```
for (k=1; k<=2; k++)
    for (i=1; i<=2; i++)
        for (j=1; j<=2; j++)
            C(i,j)=C(i,j) + B(k,j) * A(i,k);.
```

Figure 8.14 represents the index set for this algorithm. In this figure, each index element is shown as a three tuple (k,i,j). Note that for both index elements (k,i,1) and (k,i,2), the same value of $A(i,k)$ is used; that is, the value $A(i,k)$ can be piped on the j direction. Similarly, values $B(k,j)$ and $C(i,j)$ can be piped on i and k directions, respectively. Based on these facts, the algorithm can be rewritten by introducing buffering variables A^{j+1}, B^{i+1}, and C^{k+1}, as follows:

```
for (k=1; k<=2; k++)
    for (i=1; i<=2; i++)
        for (j=1; j<=2; j++)
        {
            A^{j+1}(i,k) =  A^j(i,k);
            B^{i+1}(k,j) =  B^i(k,j);
            C^{k+1}(i,j) =  C^k(i,j) + B^i(k,j) * A^j(i,k);
        }.
```

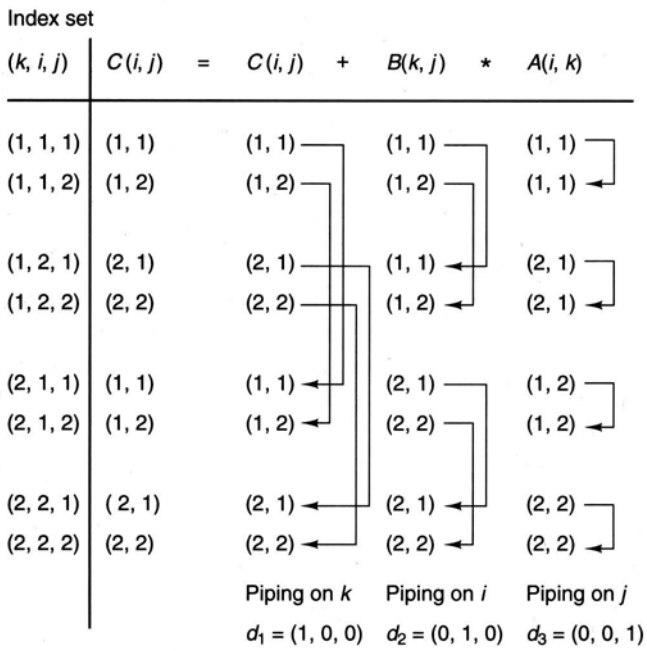

Figure 8.14 Index set

The set of data dependence vectors can be found by equating indexes of all possible pairs of generated and used variables. In the preceding code, the generated variable $C^{k+1}(i,j)$ and used variable $C^{k}(i,j)$ gives us $d_1 = (k+1-k, i-i, j-j) = (1,0,0)$. Similarly, $<B^{i+1}(k,j)$ and $B^{i}(k,j)>$ and $<A^{j+1}(i,k)$ and $A^{j}(i,k)>$ give us $d_2 = (0,1,0)$ and $d_3 = (0,0,1)$, respectively. Figure 8.14 should provide insight in to understanding the logic followed in finding the data dependence vectors. The dependence matrix $D = [d_1 \mid d_2 \mid d_3]$ can be expressed as

$$D = \begin{bmatrix} d_1 & d_2 & d_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We are looking for a transformation T that is a bijection and monotonic increasing of the form

$$T\begin{bmatrix} \Pi \\ S \end{bmatrix}$$

where $\prod d_i > 0.$ Let

$$T = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}$$

The condition $\prod d_i > 0$ (for i=1, 2, 3) implies that

$$(t_{11}\, t_{12}\, t_{13}) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} > 0 \rightarrow t_{11} > 0,$$

$$(t_{11}\, t_{12}\, t_{13}) \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} > 0 \rightarrow t_{12} > 0,$$

$$(t_{11}\, t_{12}\, t_{13}) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} > 0 \rightarrow t_{13} > 0.$$

To reduce the turnaround time, we try to choose the smallest values for t_{11}, t_{12}, and t_{13} such as

$$t_{11} = t_{12} = t_{13} = 1; \text{ that is, } \prod = (1,1,1).$$

For example, if we chose one index from the index set given in Figure 8.14, say (1,1,2), then

$$\prod \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = (1,1,1) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = 4.$$

Here, 4 indicates the reference time at which the computation corresponding to index (1,1,2) is performed.

Our choice of S will determine the interconnection of the processors. In the selection of mapping S, we are now restricted only by the fact that T must be a bijection and consist of integers. A large number of possibilities exists, each leading to different network geometries. Two of the options are

$$S_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } S_2 = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

Throughout this example, S_1 is selected. Thus,

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for the multiplication of two n-by-n matrices, 2^n PEs are needed. Thus, for our example, four PEs are needed. The interconnection between these processors is defined by

$$S_1 d_i = \begin{bmatrix} x \\ y \end{bmatrix},$$

where x and y refer to the movement of the variable along the directions i and j, respectively. Thus

$$S_1 d_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

means that variable c does not travel in any direction and is updated in time.

$$S_1 d_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

means that variable b moves along the direction i with a speed of one grid per time unit. And

$$S_1 d_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

means that variable a moves along the direction j with a speed of one grid per time unit.

Figure 8.15 represents the interconnections between the PEs. At time 1, b_{11} and a_{11} enter $PE_{1,1}$, which contains variable c_{11}. Then each PE performs a multiply and an add operation; that is, $c_{11}=c_{11}+a_{11*}b_{11}$. At time 2, a_{11} leaves $PE_{1,1}$ and enters $PE_{1,2}$. At the same time, a_{12} enters $PE_{1,1}$, b_{12} enters $PE_{1,2}$, b_{11} enters $PE_{2,1}$, and b_{21} enters $PE_{1,1}$. Again, each PE performs a multiply and an add operation. Thus

$$c_{11}=c_{11}+a_{12}*b_{21},$$
$$c_{12}=c_{12}+a_{11*}b_{12},$$
$$c_{21}=c_{21}+a_{21*}b_{11}.$$

This process continues until all the values are computed.

Figures 8.16 and 8.17 represent the mapping between original and transformed indexes.

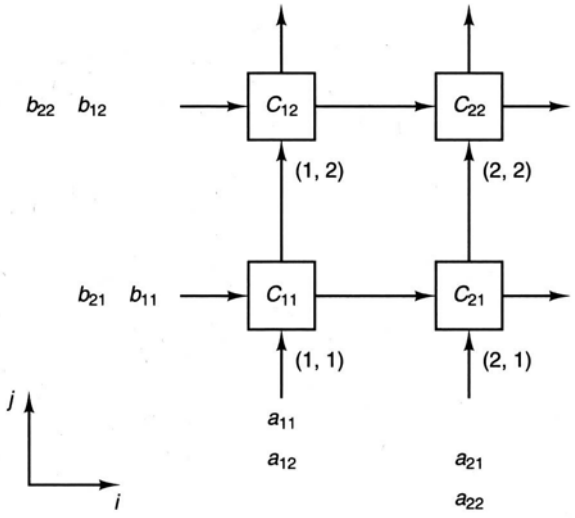

Figure 8.15 Required interconnection between the PEs.

Transfer matrix		Original index		Transformed index	C's	A's	B's	Time	PE element
T	*	$[1, 1, 1]^t$	=	$[3, 1, 1]^t$	c_{11}	a_{11}	b_{11}	3	(1, 1)
T	*	$[1, 1, 2]^t$	=	$[4, 1, 2]^t$	c_{12}	a_{11}	b_{12}	4	(1, 2)
T	*	$[1, 2, 1]^t$	=	$[4, 2, 1]^t$	c_{21}	a_{21}	b_{11}	4	(2, 1)
T	*	$[1, 2, 2]^t$	=	$[5, 2, 2]^t$	c_{22}	a_{21}	b_{12}	5	(2, 2)
T	*	$[2, 1, 1]^t$	=	$[4, 1, 1]^t$	c_{11}	a_{12}	b_{21}	4	(1, 1)
T	*	$[2, 1, 2]^t$	=	$[5, 1, 2]^t$	c_{12}	a_{12}	b_{22}	5	(1, 2)
T	*	$[2, 2, 1]^t$	=	$[5, 2, 1]^t$	c_{21}	a_{22}	b_{21}	5	(2, 1)
T	*	$[2, 2, 2]^t$	=	$[6, 2, 2]^t$	c_{22}	a_{22}	b_{22}	6	(2, 2)

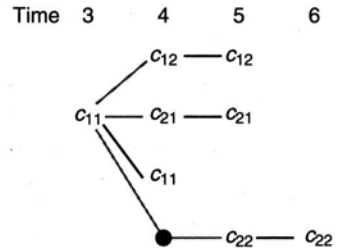

Figure 8.16 Mapping the original index set to the transformed index set.

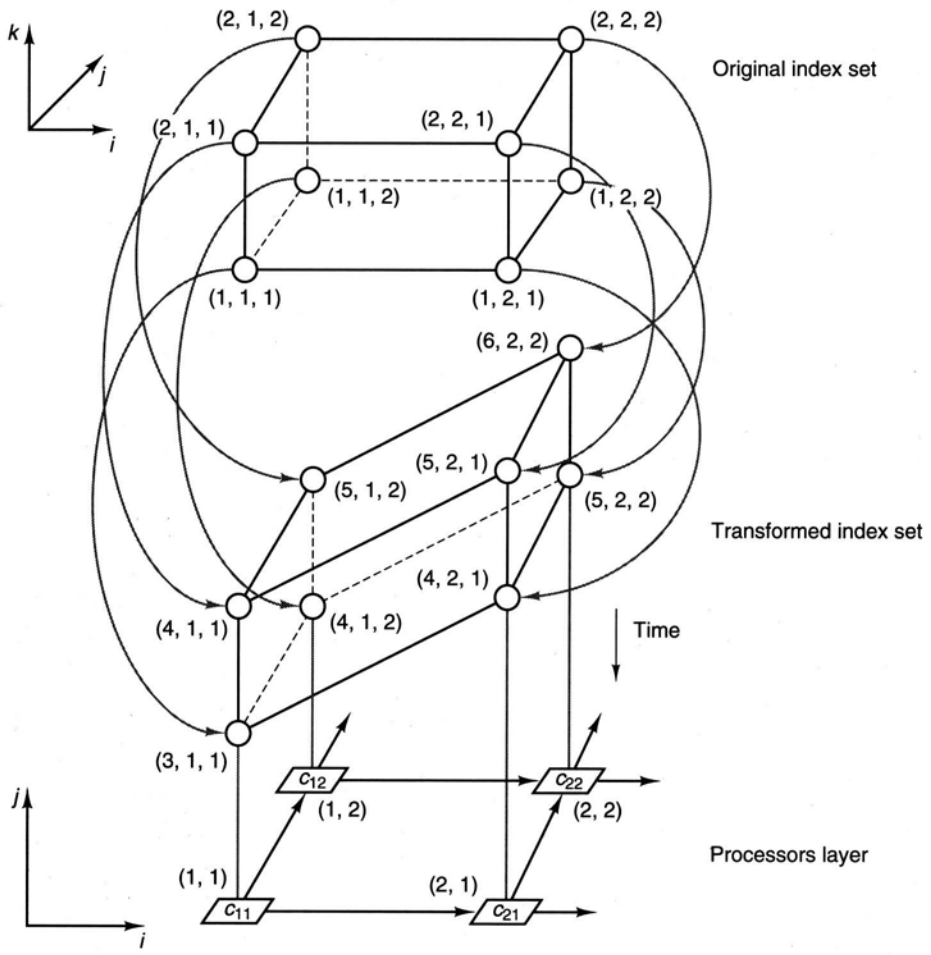

Figure 8.17 The order in which transformed indexes will be computed by the PEs.