CHAPTER

FIVE

STRUCTURES AND ALGORITHMS FOR ARRAY PROCESSORS

This chapter deals with the interconnection structures and parallel algorithms for SIMD array processors and associative processors. The various organizations and control mechanisms of array processors are presented first. Interconnection networks used in array processors will be characterized by their routing functions and implementation methods. We then study the structure of associative memory and parallel search in associative array processors. SIMD algorithms are presented for matrix manipulation, parallel sorting, fast Fourier transform, and associative search and retrieval operations.

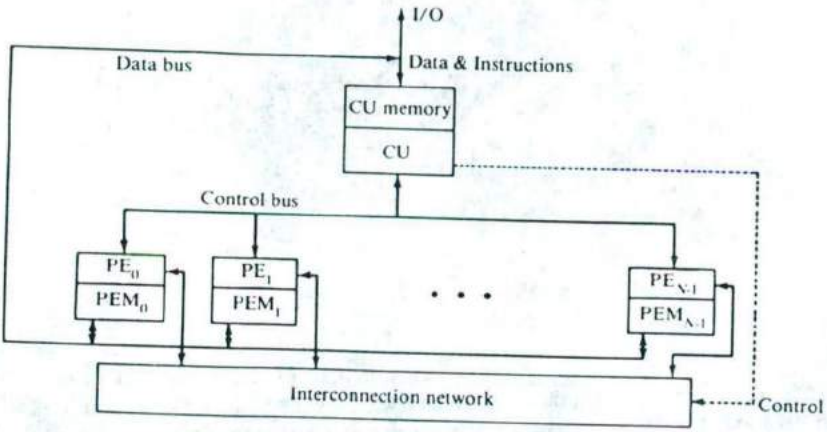
5.1 SIMD ARRAY PROCESSORS

A synchronous array of parallel processors is called an *array processor*, which consists of multiple processing elements (PEs) under the supervision of one control unit (CU). An array processor can handle *single instruction and multiple data* (SIMD) streams. In this sense, array processors are also known as *SIMD computers*. SIMD machines are especially designed to perform vector computations over matrices or arrays of data. In this book, the terms array processors, parallel processors, and SIMD computers are used interchangeably.

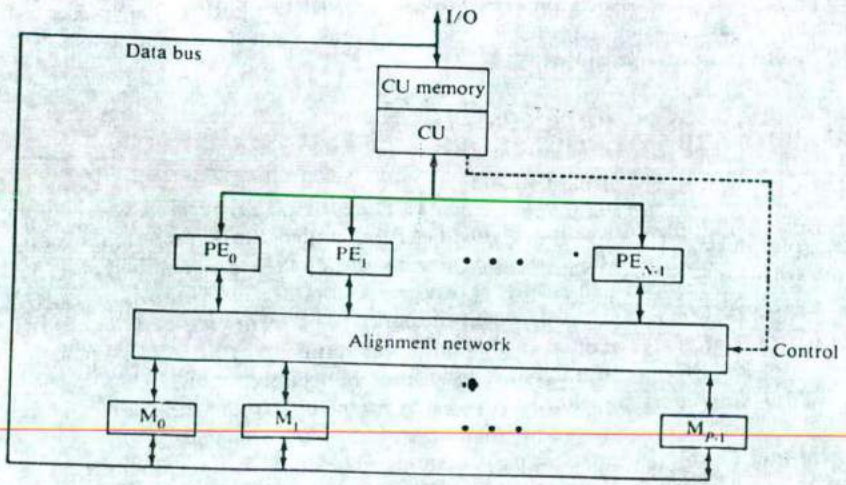
SIMD computers appear in two basic architectural organizations: *array processors*, using random-access memory; and *associative processors*, using content-addressable (or associative) memory. The first three sections of this chapter deal primarily with array processors. We will study associative processors in Section 5.4 as a special type of array processor whose PEs correspond to the words of an associative memory.

5.1.1 SIMD Computer Organizations

In general, an array processor may assume one of two slightly different configurations, as illustrated in Figure 5.1. Configuration I was introduced in Chapter 1. It has been implemented in the well-publicized Illiac-IV computer. This configuration is structured with N synchronized PEs, all of which are under the control of one CU. Each PE_i is essentially an arithmetic logic unit (ALU) with attached



(*a*) Configuration I (Illiac IV)



(*b*) Configuration II (BSP)

Figure 5.1 Architectural configurations of SIMD array processors.

working registers and local memory PEM_i for the storage of distributed data. The CU also has its own main memory for the storage of programs. The system and user programs are executed under the control of the CU. The user programs are loaded into the CU memory from an external source. The function of the CU is to decode all the instructions and determine where the decoded instructions should be executed. Scalar or control-type instructions are directly executed inside the CU. Vector instructions are broadcast to the PEs for distributed execution to achieve spatial parallelism through duplicate arithmetic units (PEs).

All the PEs perform the same function synchronously in a lock-step fashion under the command of the CU. Vector operands are distributed to the PEMs before parallel execution in the array of PEs. The distributed data can be loaded into the PEMs from an external source via the system data bus, or via the CU in a broadcast mode using the control bus. Masking schemes are used to control the status of each PE during the execution of a vector instruction. Each PE may be either *active* or *disabled* during an instruction cycle. A masking vector is used to control the status of all PEs. In other words, not all the PEs need to participate in the execution of a vector instruction. Only enabled PEs perform computation. Data exchanges among the PEs are done via an inter-PE communication network, which performs all necessary data-routing and manipulation functions. This interconnection network is under the control of the control unit.

An array processor is normally interfaced to a host computer through the control unit. The host computer is a general-purpose machine which serves as the "operating manager" of the entire system, consisting of the host and the processor array. The functions of the host computer include resource management and peripheral and I/O supervisions. The control unit of the processor array directly supervises the execution of programs, whereas the host machine performs the executive and I/O functions with the outside world. In this sense, an array processor can also be considered a back-end, attached computer, similar in function to those pipeline attached processors studied in Chapter 4.

Another possible way of constructing an array processor is illustrated in Figure 5.1*b*. This configuration II differs from the configuration I in two aspects. First, the local memories attached to the PEs are now replaced by parallel memory modules shared by all the PEs through an alignment network. Second, the inter-PE permutation network is replaced by the inter-PE memory-alignment network, which is again controlled by the CU. A good example of a configuration II SIMD machine is the Burroughs Scientific Processor (BSP). There are N PEs and P memory modules in configuration II. The two numbers are not necessarily equal. In fact, they have been chosen to be relatively prime. The alignment network is a path-switching network between the PEs and the parallel memories. Such an alignment network is desired to allow conflict-free accesses of the shared memories by as many PEs as possible.

Array processors became well publicized with the hardware-software development of the Illiac-IV system. Since then, many SIMD machines have been constructed to satisfy various parallel-processing applications. The Burroughs *Parallel Element Processing Ensemble* (PEPE) and the Goodyear Aerospace

Staran are two associative array processors. Extended from the Illiac-IV design are the *Burroughs Scientific Processor* (BSP) and the Goodyear Aerospace *Massively Parallel Processor* (MPP).

Formally, an SIMD computer C is characterized by the following set of parameters:

$$C = \langle N, F, I, M \rangle \tag{5.1}$$

where N = the number of PEs in the system. For example, the Illiac-IV has $N = 64$, the BSP has $N = 16$, and the MPP has $N = 16{,}384$.

F = a set of data-routing functions provided by the interconnection network (in Figure 5.1*a*) or by the alignment network (in Figure 5.1*b*).

I = the set of machine instructions for scalar-vector, data-routing, and network-manipulation operations.

M = the set of masking schemes, where each mask partitions the set of PEs into the two disjoint subsets of enabled PEs and disabled PEs.

This model provides a common basis for evaluating different SIMD machines. We will characterize various data-routing functions in the next section when we study interconnection networks for SIMD machines. The instruction sets of important array processors will be discussed with those example SIMD computers in Chapter 6.

In addition to regular SIMD machines, several algorithmic array processors have been developed as back-end attachments to host machines. Among them are the IBM 3838 and the Datawest MATP. These attached array processors are highly pipelined for array processing. They are not SIMD machines as discussed above. The reason that these pipeline attached processors are commercially known as "array" processors lies in the fact that they are used for processing arrays of data. Details of the Illiac-IV, BSP, MPP and multiple-SIMD computers using a shared pool will be treated in Chapter 6.

5.1.2 Masking and Data-Routing Mechanisms

In this chapter, we consider only configuration I of an SIMD computer. Each PE_i is a processor (Figure 5.2) with its own memory PEM_i; a set of working registers and flags, namely A_i, B_i, C_i, and S_i; an arithmetic logic unit; a local index register I_i; an address register D_i; and a data-routing register R_i. The R_i of each PE_i is connected to the R_j of other PEs via the interconnection network. When data transfer among PEs occurs, it is the contents of the R_i registers that are being transferred. We shall denote the N PEs as PE_i for $i = 0, 1, 2, \ldots, N-1$, where the index i is the address of PE_i. To facilitate future illustrations, we assume $N = 2^m$ or $m = \log_2 N$ binary digits are needed to encode the address of a PE. The address register D_i is used to hold the m-bit address of the PE_i. This PE structure is essentially based on the design in Illiac-IV.

Some array processors may use two routing registers, one for input and the other for output. We will simply consider the use of one R_i per PE_i in which the

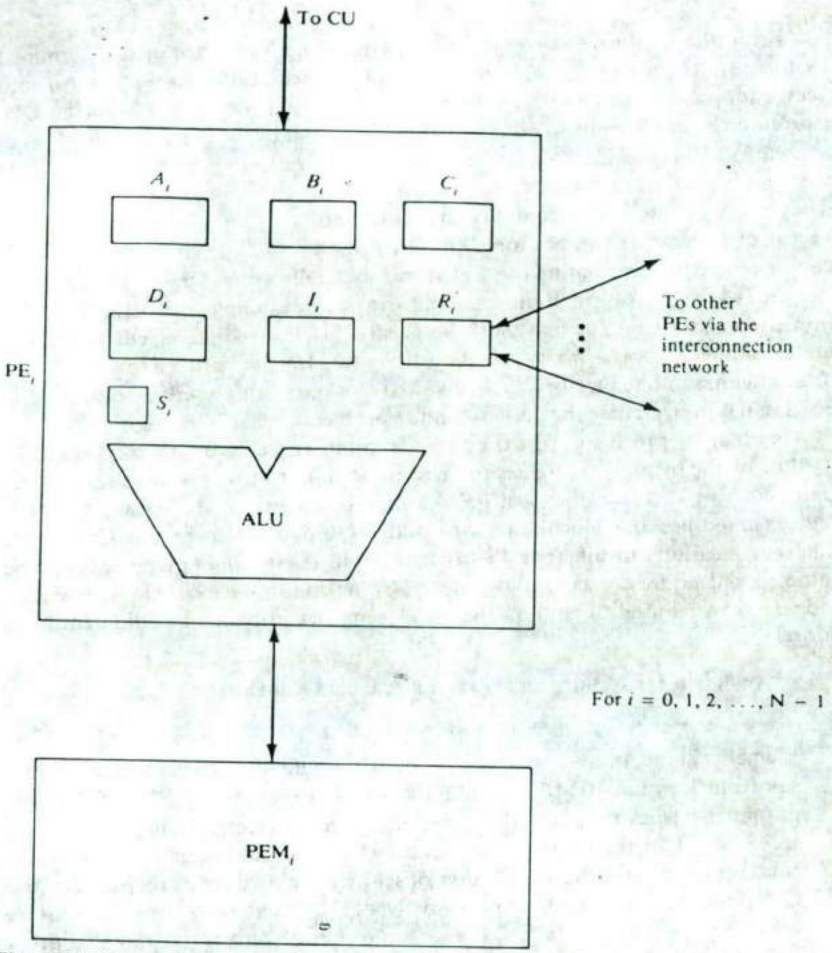


Figure 5.2 Components in a Processing Element (PE_i).

inputs and outputs of R_i are totally isolated by using master-slave flip-flops. Each PE_i is either in the active or in the inactive mode during each instruction cycle. If a PE_i is active, it executes the instruction broadcast to it by the CU. If a PE_i is inactive, it will not execute the instructions broadcast to it. The masking schemes are used to specify the status flag S_i of PE_i. The convention $S_i = 1$ is chosen for an active PE_i and $S_i = 0$ for an inactive PE_i. In the CU, there is a global index register I and a masking register M. The M register has N bits. The ith bit of M will be denoted as M_i. The collection of S_i flags for $i = 0, 1, 2, \ldots, N - 1$ forms a status register S for all the PEs. Note that the bit patterns in registers M and S are exchangeable upon the control of the CU when masking is to be set.

From the hardware viewpoint, the physical length of a vector is determined by the number of PEs. The CU performs the segmentation of a long vector into vector loops, the setting of a global base address, and the offset increment. Distributing vector elements to different PEMs is crucial to the efficient utilization of an array of PEs. Ideally, N elements of a vector are retrieved from different PEMs simultaneously. In the worst case, all the vector elements are all in a single PEM. They must be fetched sequentially one after another. A one-dimensional linear vector of n elements can be stored in all PEMs if $n \leq N$. Long vectors ($n > N$) can be stored by distributing the n elements cyclically among the N PEMs. Difficulty may arise in using high-dimensional arrays. For example, in matrix computations, rows and columns may both be needed in intermediate calculations. The matrix should be stored in a way to allow the parallel fetch of either a row, a column, or a diagonal in one memory cycle. Skewed storage methods will be discussed in Section 6.4 to overcome the access-conflict problem.

In an array processor, vector operands can be specified by the registers to be used or by the memory addresses to be referenced. For memory-reference instructions, each PE_i accesses the local PEM_i, offset by its own index register I_i. The I_i register modifies the global memory address broadcast from the CU. Thus, different locations in different PEMs can be accessed simultaneously with the same global address specified by the CU. The following example shows how indexing can be used to address the local memories in parallel at different local addresses.

Example 5.1 Consider an array of $n \times n$ data elements:

$$A = \{A(i, j), 0 \leq i, j \leq n - 1\} \tag{5.2}$$

Elements in the jth column of A are stored in n consecutive locations of PEM_j, say from location 100 to location $100 + n - 1$ (assume $n \leq N$). If the programmer wishes to access the principal diagonal elements $A(j, j)$ for $j = 0, 1, \ldots, n - 1$ of the array A, then the CU must generate and broadcast an effective memory address 100 (after offset by the global index register I in the CU, if there is a base address of A involved). The local index registers must be set to be $I_j = j$ for $j = 0, 1, \ldots, n - 1$ in order to convert the global address 100 to local addresses $100 + I_j = 100 + j$ for each PEM_j. Within each PE, there should be a separate memory address register for holding these local addresses. However, if one wishes to address a row of the array A, say the ith row $A(i, j)$ for $j = 0, 1, 2, \ldots, n - 1$, all the I_j registers will be reset to be for all $j = 0, 1, 2, \ldots, n - 1$ in order to ensure the parallel access of the entire row.

Example 5.2 To illustrate the necessity of data routing in an array processor, we show the execution details of the following vector instruction in an array of N PEs. The sum $S(k)$ of the first k components in a vector A is desired for each k from 0 to $n - 1$. Let $A = (A_0, A_1, \ldots, A_{n-1})$. We need to compute the following n summations:

$$S(k) = \sum_{i=0}^{k} A_i \qquad \text{for } k = 0, 1, \ldots, n - 1 \tag{5.3}$$

These n vector summations can be computed recursively by going through the following $n - 1$ iterations defined by:

$$S(0) = A_0$$

$$S(k) = S(k-1) + A_k \qquad \text{for } k = 1, 2, \ldots, n-1. \tag{5.4}$$

The above recursive summations for the case of $n = 8$ are implemented in an array processor with $N = 8$ PEs in $\lceil \log_2 n \rceil = 3$ steps, as shown in Figure 5.3. Both data routing and PE masking are used in the implementation. Initially, each A_i, residing in PEM_i, is moved to the R_i register in PE_i for $i = 0, 1, \ldots, n-1$ ($n = N = 8$ is assumed here). In the first step, A_i is routed from R_i to R_{i+1} and added to A_{i+1} with the resulting sum $A_i + A_{i+1}$ in R_{i+1}, for $i = 0, 1, \ldots, 6$. The arrows in Figure 5.3 show the routing operations and the shorthand notation $i - j$ is used to refer to the intermediate sum $A_i + A_{i+1} + \cdots + A_j$. In step 2, the intermediate sums in R_i are routed to

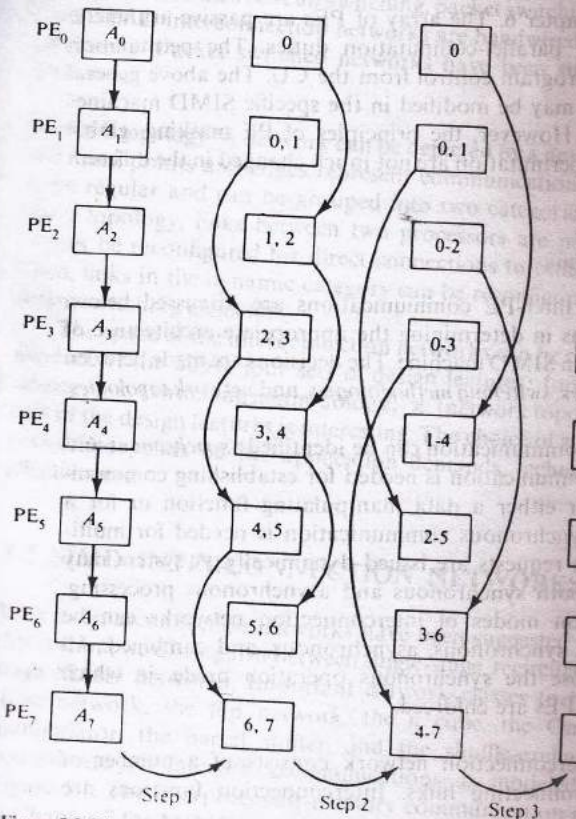


Figure 5.3 The calculation of the summation $S(k) = \sum_{i=0}^{k} A_i$, $k = 0, 1, \ldots, 7$ in an SIMD machine.

R_{i+2} for $i = 0$ to 5. In the final step, the intermediate sums in R_i are routed to R_{i+4} for $i = 0$ to 3. Consequently, PE_k has the final value of $S(k)$ for $k = 0, 1, 2, \ldots, 7$, as shown by the last column in Figure 5.3.

As far as the data-routing operations are concerned, PE_7 is not involved (receiving but not transmitting) in step 1. PE_7 and PE_6 are not involved in step 2. Also PE_7, PE_6, PE_5, and PE_4 are not involved in step 3. These unwanted PEs are masked off during the corresponding steps. During the *addition* operations, PE_0 is disabled in step 1; PE_0 and PE_1 are made inactive in step 2; and PE_0, PE_1, PE_2, and PE_3 are masked off in step 3. The PEs that are masked off in each step depend on the operation (data-routing or arithmetic-addition) to be performed. Therefore, the masking patterns keep changing in the different operation cycles, as demonstrated by the example. Note that the masking and routing operations will be much more complicated when the vector length $n > N$.

Array processors are special-purpose computers for limited scientific applications. We will describe the detailed structures of the CU and PEs with several SIMD computer designs in Chapter 6. The array of PEs are passive arithmetic units waiting to be called for parallel-computation duties. The permutation network among PEs is under program control from the CU. The above general structures of SIMD computers may be modified in the specific SIMD machines to be presented in Chapter 6. However, the principles of PE masking, global versus local indexing, and data permutation are not much changed in the different machines.

5.1.3 Inter-PE Communications

Network design decisions for inter-PE communications are discussed below. These are fundamental decisions in determining the appropriate architecture of an interconnection network for an SIMD machine. The decisions are made between *operation modes*, *control strategies*, *switching methodologies*, and *network topologies*.

Operation mode Two types of communication can be identified: *synchronous* and *asynchronous*. Synchronous communication is needed for establishing communication paths synchronously for either a data manipulating function or for a data instruction broadcast. Asynchronous communication is needed for multiprocessing in which connection requests are issued dynamically. A system may also be designed to facilitate both synchronous and asynchronous processing. Therefore, the typical operation modes of interconnection networks can be classified into three categories: synchronous, asynchronous, and combined. All existing SIMD machines choose the synchronous operation mode, in which lock-step operations among all PEs are enforced.

Control strategy A typical interconnection network consists of a number of switching elements and interconnecting links. Interconnection functions are

realized by properly setting control of the switching elements. The control-setting function can be managed by a centralized controller or by the individual switching element. The latter strategy is called *distributed control* and the first strategy corresponds to *centralized control*. Most existing SIMD interconnection networks choose the centralized control on all switch elements by the control unit.

Switching methodology The two major switching methodologies are *circuit switching* and *packet switching*. In circuit switching, a physical path is actually established between a source and a destination. In packet switching, data is put in a packet and routed through the interconnection network without establishing a physical connection path. In general, circuit switching is much more suitable for bulk data transmission, and packet switching is more efficient for many short data messages. Another option, integrated switching, includes the capabilities of both circuit switching and packet switching. Therefore, three switching methodologies can be identified: circuit switching, packet switching, and integrated switching. Most SIMD interconnection networks are handwired to assume circuit switching operations. Packet switched networks have been suggested mainly for MIMD machines.

Network topology A network can be depicted by a graph in which nodes represent switching points and edges represent communication links. The topologies tend to be regular and can be grouped into two categories: *static* and *dynamic*. In a static topology, links between two processors are passive and dedicated buses cannot be reconfigured for direct connections to other processors. On the other hand, links in the dynamic category can be reconfigured by setting the network's active switching elements.

The space of the interconnection networks can be represented by the cartesian product of the above four sets of design features: {operation mode} × {control strategy} × {switching methodology} × {network topology}. Not every combination of the design features is interesting. The choice of a particular interconnection network depends on the application demands, technology supports, and cost-effectiveness.

5.2 SIMD INTERCONNECTION NETWORKS

Various interconnection networks have been suggested for SIMD computers. In this section, we distinguish between single-stage, recirculating networks and multi-stage SIMD networks. Important network classes to be presented include the Illiac network, the flip network, the n cube, the Omega network, the data manipulator, the barrel shifter, and the shuffle-exchange network. We shall concentrate on inter-PE communications as modeled by configuration I in Figure 5.1. The interprocessor-memory communication networks will be studied in Chapter 7 for MIMD operations.

5.2.1 Static Versus Dynamic Networks

The topological structure of an SIMD array processor is mainly characterized by the data-routing network used in interconnecting the processing elements. Formally, such an inter-PE communication network can be specified by a set of data-routing functions. If we identify the addresses of all the PEs in an SIMD machine by the set $S = \{0, 1, 2, \ldots, N-1\}$, each routing function f is a *bijection* (a one-to-one and onto mapping) from S to S. When a routing function f is executed via the interconnection network, the PE_i copies the contents of its R_i register into the $R_{f(i)}$ register of $PE_{f(i)}$. This data-routing operation occurs in all active PEs simultaneously. An inactive PE may receive data from another PE if a routing function is executed, but it cannot transmit data. To pass data between PEs that are not directly connected in the network, the data must be passed through intermediate PEs by executing a sequence of routing functions through the interconnection network.

The SIMD interconnection networks are classified into the following two categories based on network topologies: *static networks* and *dynamic networks*.

Static networks Topologies in the static networks can be classified according to the dimensions required for layout. For illustration, one-dimensional, two-dimensional, three-dimensional, and hypercube are shown in Figure 5.4. Examples of one-dimensional topologies include the *linear array* used for some pipeline architectures (Figure 5.4*a*). Two-dimensional topologies include the *ring*, *star*, *tree*, *mesh*, and *systolic array*. Examples of these structures are shown in Figures 5.4*b* through 5.4*f*.

Three-dimensional topologies include the *completely connected chordal ring*, 3 *cube*, and 3-*cube-connected-cycle* networks depicted in Figures 5.4*g* through 5.4*j*. A D-dimensional, W-wide hypercube contains W nodes in each dimension, and there is a connection to a node in each dimension. The mesh and the 3 cube are actually two- and three-dimensional hypercubes, respectively. The cube-connected-cycle is a deviation of the hypercube. For example, the 3-cube-connected-cycle shown in Figure 5.4*j* is obtained from the 3 cube.

Dynamic networks We consider two classes of dynamic networks: *single-stage* versus *multistage*, as described below separately:

Single-stage networks A single-stage network is a switching network with N *input selectors* (IS) and N *output selectors* (OS), as demonstrated in Figure 5.5. Each IS is essentially a 1-to-D demultiplexer and each OS is an M-to-1 multiplexer where $1 \leq D \leq N$ and $1 \leq M \leq N$. Note that the crossbar-switching network is a single-stage network with $D = M = N$. To establish a desired connecting path, different path control signals will be applied to all IS and OS selectors.

The single-stage network is also called a *recirculating* network. Data items may have to recirculate through the single stage several times before reaching their final destinations. The number of recirculations needed depends on the

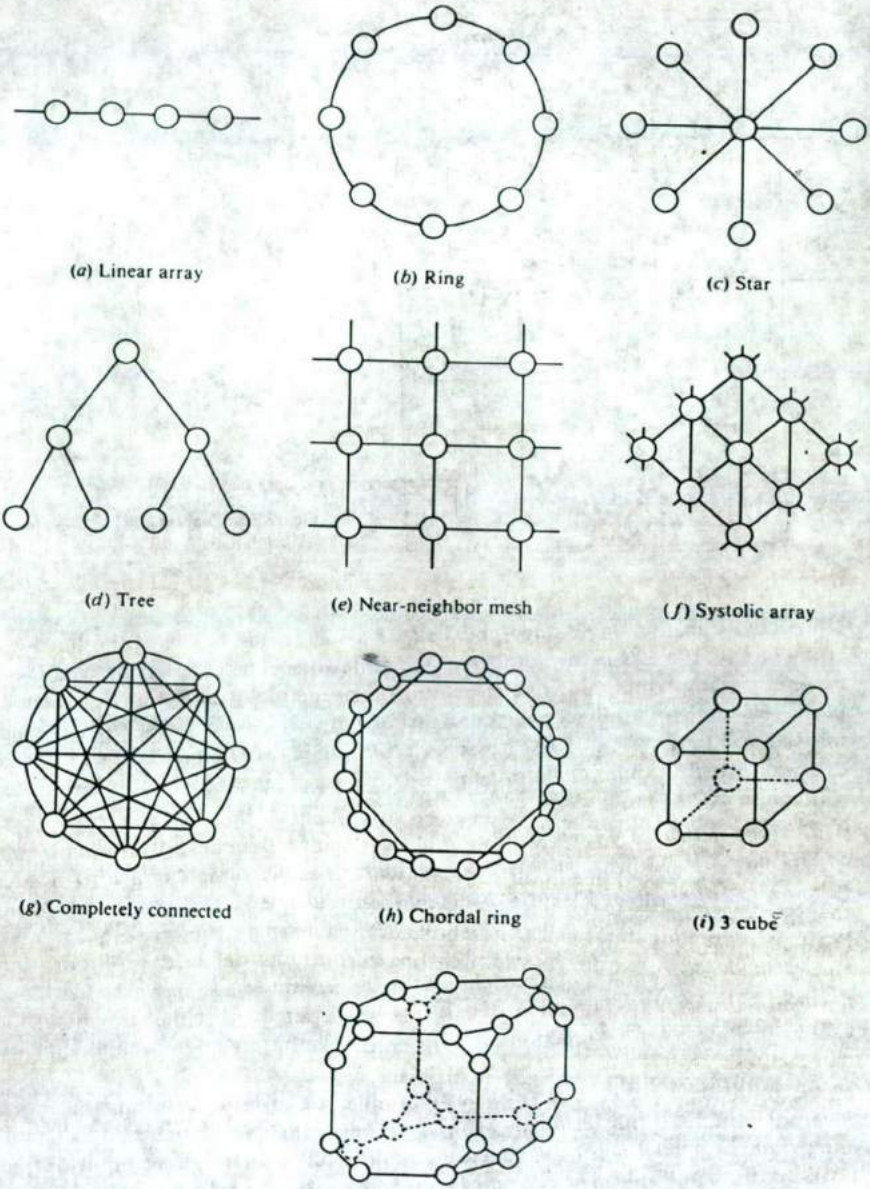


Figure 5.4 Static interconnection network topologies. (Courtesy of Feng, *IEEE Computer*, December 1981.)

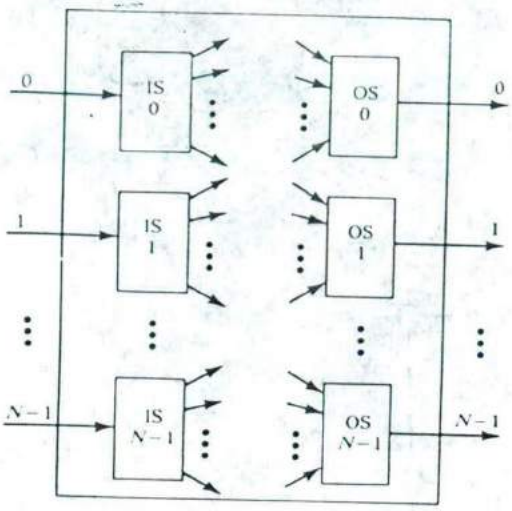


Figure 5.5 Conceptual view of a single-stage interconnection network.

connectivity in the single-stage network. In general, the higher is the hardware connectivity, the less is the number of recirculations. The crossbar network is an extreme case in which only one circulation is needed to establish any connection path. However, the fully connected crossbar networks have a cost $O(N^2)$, which may be prohibitive for large N. Most recirculating networks have cost $O(N \log N)$ or lower, which is definitely more cost-effective for large N.

Multistage networks Many stages of interconnected switches form a *multistage SIMD network*. Multistage networks are described by three characterizing features: the *switch box*, the *network topology*, and the *control structure*. Many switch boxes are used in a multistage network. Each box is essentially an interchange device with two inputs and two outputs, as depicted in Figure 5.6. Illustrated are four states of a switch box: *straight*, *exchange*, *upper broadcast*, and *lower broadcast*. A two-function switch box can assume either the straight or the exchange states. A four-function switch box can be in any one of the four legitimate states.

A multistage network is capable of connecting an arbitrary input terminal to an arbitrary output terminal. Multistage networks can be one-sided or two-sided. The *one-sided networks*, sometimes called full switches, have input-output ports on the same side. The *two-sided multistage networks*, which usually have an input side and an output side, can be divided into three classes: blocking, rearrangeable, and nonblocking.

In *blocking networks*, simultaneous connections of more than one terminal pair may result in conflicts in the use of network communication links. Examples of a blocking network are the *data manipulator*, *Omega*, *flip*, *n cube*, and *baseline*.

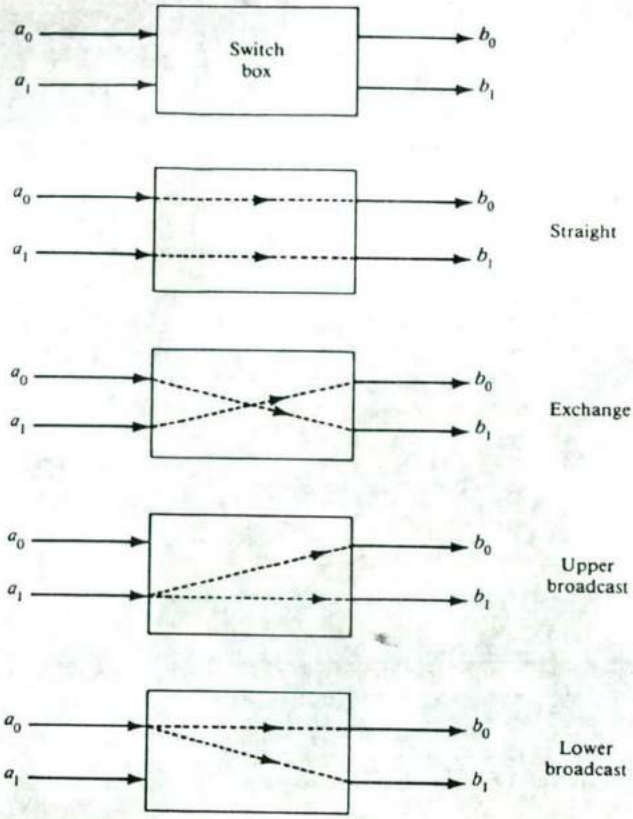
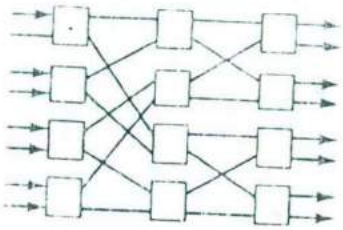


Figure 5.6 A two-by-two switching box and its four interconnection states.

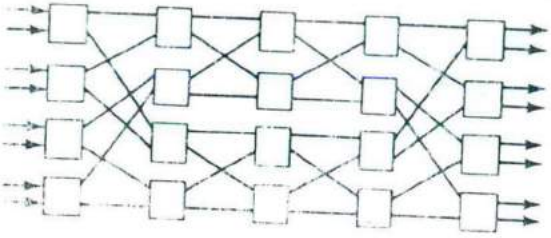
Some of these networks will be introduced in subsequent sections. Figure 5.7*a* shows the interconnection pattern in the baseline network.

A network is called a *rearrangeable network* if it can perform all possible connections between inputs and outputs by rearranging its existing connections so that a connection path for a new input-output pair can always be established. A well-defined network, the Benes network, shown in Figure 5.7*b*, belongs to this class. The Benes rearrangeable network topology has been extensively studied for use in synchronous data permutation and in asynchronous interprocessor communication.

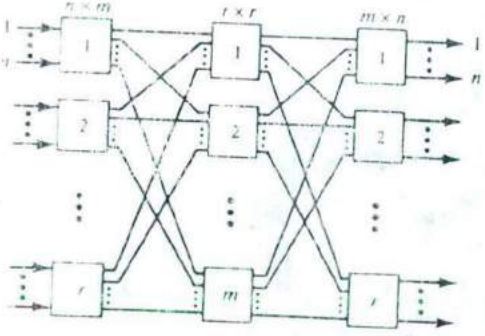
A network which can handle all possible connections without blocking is called a *nonblocking network*. Two cases have been considered in the literature. In the first case, the *Clos network*, shown in Figure 5.7*c*, a one-to-one connection is made between an input and an output. The other case considers one-to-many connections. Here, a generalized connection network topology is generated to



(a) 8 × 8 baseline network



(b) 8 × 8 Benes network



(c) Clos network

Figure 5.7 Several multistage interconnection networks.

pass any of multiple mappings of inputs onto outputs. The *crossbar switch network* can connect every input port to a free output port without blocking.

Generally, a multistage network consists of n stages where $N = 2^n$ is the number of input and output lines. Therefore, each stage may use $N/2$ switch boxes. The interconnection patterns from stage to stage determine the network topology. Each stage is connected to the next stage by at least N paths. The network delay is proportional to the number n of stages in a network. The cost of a size N multistage network is proportional to $N \log_2 N$. The control structure of a network determines how the states of the switch boxes will be set. Two types of control structures

are used in a network construction. The *individual stage control* uses the same control signal to set all switch boxes in the same stage. In other words, all boxes at the same stage must be set to assume that same state. Therefore, it requires n sets of control signals to set up the states of all n stages of switch boxes.

Another control philosophy is to apply *individual box control*. A separate control signal is used to set the state of each switch box. This offers higher flexibility in setting up the connecting paths, but requires $n^2/2$ control signals, which will significantly increase the complexity of the control circuitry. A compromise design is to use *partial stage control*, in which $i + 1$ control signals are used at stage i for $0 \leq i \leq n - 1$. Various network topologies and control structures of both recirculating and multistage inter-PE communication networks are described in subsequent sections.

5.2.2 Mesh-Connected Illiac Network

A single-stage recirculating network has been implemented in the Illiac-IV array processor with $N = 64$ PEs. Each PE_i is allowed to send data to any one of PE_{i+1}, PE_{i-1}, PE_{i+r}, and PE_{i-r}, where $r = \sqrt{N}$ (for the case of the Illiac-IV, $r = \sqrt{64} = 8$) in one circulation step through the network. Formally, the Illiac network is characterized by the following four routing functions:

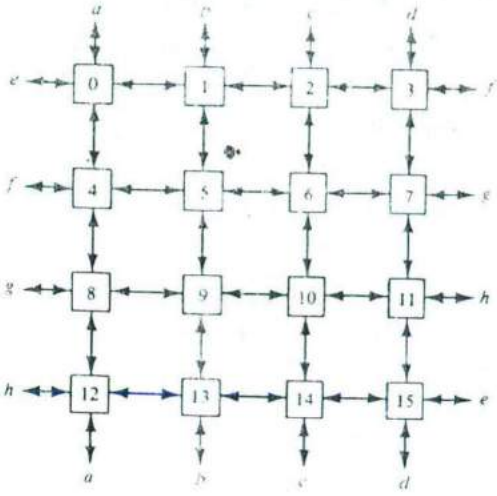
$$\begin{aligned} R_{+1}(i) &= (i + 1) \bmod N \\ R_{-1}(i) &= (i - 1) \bmod N \\ R_{+r}(i) &= (i + r) \bmod N \\ R_{-r}(i) &= (i - r) \bmod N \end{aligned} \tag{5.5}$$

where $0 \leq i \leq N - 1$. In practice, N is commonly a perfect square, such as $N = 64$ and $r = 8$ in the Illiac-IV network.

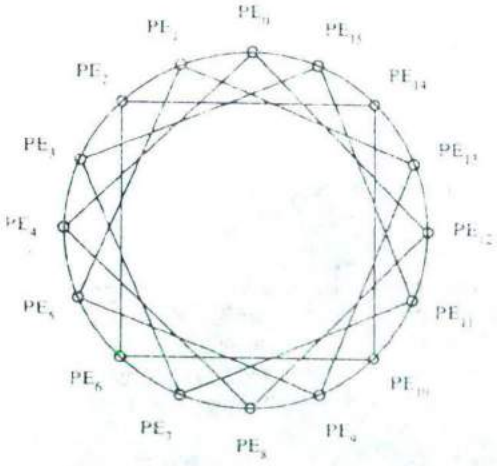
A reduced Illiac network is illustrated in Figure 5.8*a* for $N = 16$ and $r = 4$. The real Illiac network has a similar structure except larger in size. All the index arithmetic in Eq. 5.5 is modulo N. Comparing with the formal model shown in Figure 5.5, we observe that the outputs of IS_i are connected to the inputs of OS_j for $j = i + 1, i - 1, i + r$, and $i - r$. On the other hand, OS_j gets its inputs from IS_i for $i = j - 1, j + 1, j - r$, and $j + r$, respectively.

Each PE_i in Figure 5.8 is directly connected to its four nearest neighbors in the mesh network. In terms of permutation cycles, we can express the above routing functions as follows: Horizontally, all the PEs of all rows form a linear circular list as governed by the following two permutations, each with a single cycle of order N. The permutation cycles $(a\ b\ c)\ (d\ e)$ stand for the permutation $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow a$ and $d \rightarrow e$, $e \rightarrow d$ in a circular fashion within each pair of parentheses:

$$\begin{aligned} R_{+1} &= (0\ 1\ 2\ \cdots\ N-1) \\ R_{-1} &= (N-1\ \cdots\ 2\ 1\ 0) \end{aligned} \tag{5.6}$$



(a) The mesh connections



(b) The mesh redrawn

Figure 5.8 An Illiac network with $N = 16$ PEs.

Vertically, the distance r shifting operations are characterized by the following two permutations, each with r cycles of order r each:

$$R_{+r} = \prod_{i=0}^{r-1} (i \;\; i+r \;\; i+2r \;\; \cdots \;\; i+N-r)$$
$$R_{-r} = \prod_{i=0}^{r-1} (i+N-r \;\; \cdots \;\; i+2r \;\; i+r \;\; i) \tag{5.7}$$

For the example network of $N = 16$ and $r = \sqrt{16} = 4$, the shift by a distance of four is specified by the following two permutations, each with four cycles of order four each:

$$R_{+4} = (0\ 4\ 8\ 12)(1\ 5\ 9\ 13)(2\ 6\ 10\ 14)(3\ 7\ 11\ 15)$$
$$R_{-4} = (12\ 8\ 4\ 0)(13\ 9\ 5\ 1)(14\ 10\ 6\ 2)(15\ 11\ 7\ 3)$$

It should be noted that when either the R_{+1} or R_{-1} routing function is executed, data is routed as described in Eq. 5.6 only if all PEs in the cycle are active. When the routing function R_{+r} or R_{-r} is executed, data are permuted as described in Eq. 5.7 only if PE_{i+kr} where $0 \le k \le r-1$ are active for each i. The shifting operation in a cycle will be suspended if any PE required in the cycle is disabled. For an example, the cycle (1 5 9 13) in the above permutation R_4 will not be executed if one or more among PE_1, PE_5, PE_9, and PE_{13} is disabled by masking.

The Illiac network is only a partially connected network. Figure 5.8*b* shows the connectivity of the example Illiac network with $N = 16$. This graph shows that four PEs can be reached from any PE in one step, seven PEs in two steps, and eleven PEs in three steps. In general, it takes I steps (recirculations) to route data from PE_i to any other PE_j in an Illiac network of size N where I is upper-bounded by

$$I \le \sqrt{N} - 1 \tag{5.8}$$

Without a loss of generality, we illustrate the cases when PE_0 is a source node in Figure 5.8. PE_1, PE_4, PE_{12}, or PE_{15} is reachable in one step from PE_0. In two steps, the network can route data from PE_0 to PE_2, PE_3, PE_5, PE_8, PE_{11}, PE_{13}, or PE_{14}. In the worst case of three routing steps, the following eight routing sequences take place in the network:

$$0 \xrightarrow{R_{+1}} 1 \xrightarrow{R_{+1}} 2 \xrightarrow{R_{+4}} 6 \qquad 0 \xrightarrow{R_{+4}} 4 \xrightarrow{R_{+4}} 8 \xrightarrow{R_{-1}} 7$$

$$0 \xrightarrow{R_{-4}} 12 \xrightarrow{R_{-1}} 11 \xrightarrow{R_{+4}} 7 \qquad 0 \xrightarrow{R_{-4}} 12 \xrightarrow{R_{+4}} 8 \xrightarrow{R_{-1}} 7$$

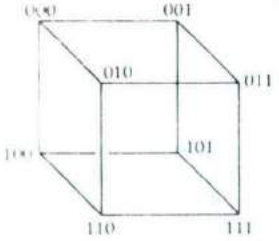
$$0 \xrightarrow{R_{-4}} 12 \xrightarrow{R_{-4}} 8 \xrightarrow{R_{+1}} 9 \qquad 0 \xrightarrow{R_{-1}} 15 \xrightarrow{R_{+4}} 11 \xrightarrow{R_{-1}} 10$$

$$0 \xrightarrow{R_{+1}} 1 \xrightarrow{R_{+4}} 13 \xrightarrow{R_{+4}} 9 \qquad 0 \xrightarrow{R_{-1}} 15 \xrightarrow{R_{-1}} 14 \xrightarrow{R_{+1}} 10$$

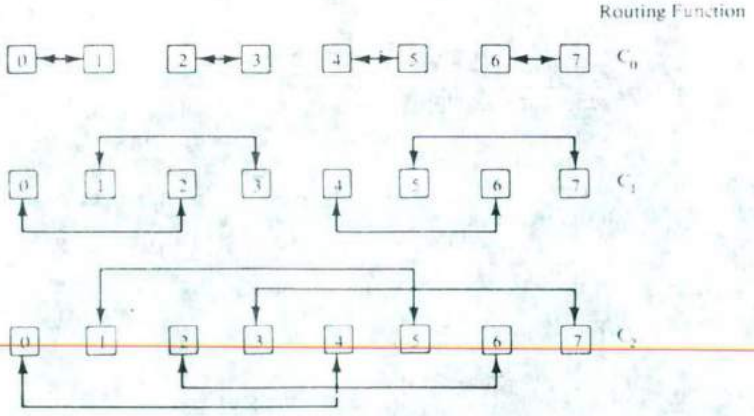
In the Illiac-IV computer, at most seven ($\sqrt{64} - 1$) steps are needed to route data from any one PE to another PE. Of course, if we increase the connectivity in Figure 5.8, the upper bound given in Eq. 5.8 can be lowered. We shall demonstrate this by other network types in subsequent sections. When the network is strongly connected (i.e., with 15 outgoing links per node in Figure 5.8), the upper bound on recirculation steps can be reduced to one at the expense of significantly increased hardware in the crossbar network.

5.2.3 Cube Interconnection Networks

The cube network can be implemented as either a recirculating network or as a multistage network for SIMD machines. A three-dimensional cube is illustrated in Figure 5.9*a*. Vertical lines connect vertices (PEs) whose addresses differ in the most significant bit position. Vertices at both ends of the diagonal lines differ in



(*a*) A 3-cube of 8 nodes



(*b*) The recirculating cube network

Figure 5.9 The cube interconnection network.

the middle bit position. Horizontal lines differ in the least significant bit position. This unit-cube concept can be extended to an n-dimensional unit space, called an *n cube*, with n bits per vertex. A cube network for an SIMD machine with N PEs corresponds to an n cube where $n = \log_2 N$. We shall use the binary sequence $A = (a_{n-1} \cdots a_2\ a_1 a_0)_2$ to represent the vertex (PE) address for $0 \le A \le N - 1$. The complement of bit a_i will be denoted as $\bar{a}_i$ for any $0 \le i \le n - 1$.

Formally, an n-dimensional cube network of N PEs is specified by the following n routing functions:

$$C_i(a_{n-1} \cdots a_1 a_0) = a_{n-1} \cdots a_{i+1} \bar{a}_i a_{i-1} \cdots a_0 \qquad \text{for } i = 0, 1, 2, \ldots, n-1 \qquad (5.9)$$

In the n cube, each PE located at a corner is directly connected to n neighbors. The neighboring PEs differ in exactly one bit position. Pease's binary n cube, the flip network used in STARAN, and the programmable switching network proposed for the Phoenix project are examples of cube networks.

In a recirculating cube network, each IS_A for $0 \le A \le N - 1$ is connected to n OSs whose addresses are $a_{n-1} \cdots a_{i+1} \bar{a}_i a_{i-1} \cdots a_0$ for $0 \le i \le n - 1$. On the other hand, each OS_T with $T = t_{n-1} \cdots t_1 t_0$ gets its inputs from ISs whose addresses are $t_{n-1} \cdots t_{i+1} \bar{t}_i t_{i-1} \cdots t_0$ for $0 \le i \le n - 1$. To execute the C_i routing function, IS_j selects the $C_i(j)$ output line and the OS_j selects the $C_i(j)$ input line for all j such that $0 \le j \le N - 1$.

The implementation of a single-stage cube network is illustrated in Figure 5.9*b* for $N = 8$. The interconnections of the PEs corresponding to the three routing functions C_0, C_1, and C_2 are shown separately. If one assembles all three connecting patterns together, the 3 cube shown in Figure 5.9*a* should be the result.

The same set of cube-routing functions, C_0, C_1, and C_2, can also be implemented by a three-stage cube network, as modeled in Figure 5.10 for $N = 8$.

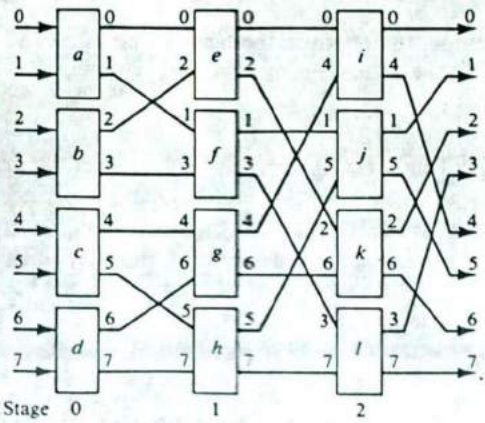


Figure 5.10 A Multistage cube network for $N = 8$.

Two-function (*straight* and *exchange*) switch boxes are used in constructing multistage cube networks. The stages are numbered as 0 at the input end and increased to $n-1$ at the output end. Stage i implements the C_i routing function (Eq. 5.9) for $i = 0, 1, 2, \ldots, n-1$. This means that switch boxes at stage i connect an input line to the output line that differs from it only at the ith bit position. Based on this interconnection requirement, individual box control is assumed in a multistage cube network.

The STARAN flip network and Pease's binary n-cube networks are both implemented with multiple stages. These two cube networks have the same topology, as shown in Figure 5.10. They differ from each other only in their control structures. The flip network has two control mechanisms, the flip control and the shift control. Under the individual-stage flip control, an n-bit vector $F = f_{n-1} \cdots f_1 f_0$ determines how stages will be set. Stage i switch boxes are set to exchange if $f_i = 1$, and set to straight if $f_i = 0$. For example, if $F = 001$ (for $N = 8$), the network connects input line $a_2 a_1 a_0$ to output line $a_1 a_2 \bar{a}_0$. This partial-stage shift control allows barrel shifts of data from input A to output $(A + 2^m) \pmod{2^p}$ where $0 \le m \le p \le n$, using $i + 1$ control lines at stage i for $0 \le i \le n - 1$. The individual box control of the Pease's n-cube network is much more flexible than the partial-stage control in the flip network. In other words, the n-cube network can perform not only all the connections that STARAN can, but also some connections it cannot.

The cube-routing function C_i for each i in Eq. 5.9 corresponds to performing the following permutation on N PEs:

$$P_i = \prod_{j=0}^{N-1} (j \;\; C_i(j)) \tag{5.10}$$

where the ith bit of j equals zero and PE_j and $PE_{C_i(j)}$ are both active. For an example, the routing function C_2 executed on a 3-cube network corresponds to the following permutation over eight PEs:

$$P_2 = (0\;\;4)(1\;\;5)(2\;\;6)(3\;\;7)$$

If all the switch boxes in stage i are set to exchange, the network performs the P_i permutation at stage i. In general, the following multistage permutation is conducted in an n-stage cube network:

$$P = \prod_{i=0}^{n-1} \left(\prod_{j=0}^{N-1} (j \;\; C_i(j)) \right) \tag{5.11}$$

where the ith bit j equals 0 and the stage i switch boxes whose inputs are labeled as j and $C_i(j)$ are set to exchange. For the example design in Figure 5.10, the permutation $(0\;1)(0\;2)(0\;4) = (0\;1\;2\;4)$ is performed only if the top row boxes are set to exchange and the rest are set to straight.

Masking may change the data-routing patterns in a cube network. The general practice is to disable all PEs belonging to the same cycle of a permutation. In the above example, if both PE_2 and PE_6 become inactive by masking, the cycles (2 6)

are removed and the cube-routing function C_2 performs only the partial permutation (0 4) (1 5) (3 7). However, if only PE_2 is disabled in the above example, the above partial permutation will still be performed, but data in both PE_2 and PE_6 will be transferred to PE_2, causing a two-to-one conflicting transfer. PE_6 will not receive any data, so the mapping will not be onto either. Masking should be carefully applied to cube networks because of the *send-active* and *receive-inactive* nature of data transfers among the PEs.

5.2.4 Barrel Shifter and Data Manipulator

Barrel shifters are also known as *plus-minus-2^i* (PM2I) networks. This type of network is based on the following routing functions:

$$\begin{aligned} B_{+i}(j) &= (j + 2^i) \pmod N \\ B_{-i}(j) &= (j - 2^i) \pmod N \end{aligned} \tag{5.12}$$

where $0 \le j \le N - 1$, $0 \le i \le n - 1$, and $n = \log_2 N$. Comparing Eq. 5.12 with Eq. 5.5, the following equivalence is revealed when $r = \sqrt{N} = 2^{n/2}$:

$$\begin{aligned} B_{+0} &= R_{+1} \\ B_{-0} &= R_{-1} \\ B_{+n/2} &= R_{+r} \\ B_{-n/2} &= R_{-r} \end{aligned} \tag{5.13}$$

This implies that the Illiac routing functions are a subset of the barrel-shifting functions. In addition to adjacent (± 1) and fixed-distance ($\pm r$) shiftings, the barrel-shifting functions allow either forward or backward shifting of distances which are the integer power of two, i.e., $\pm 1, \pm 2, \pm 4, \pm 8, \ldots, \pm 2^{n/2}, \ldots, \pm 2^{n-1}$. Instead of having just four nearest neighbors as in the Illiac mesh networks, each PE in a barrel shifter is directly connected to $2(n - 1)$ PEs. Therefore, the connectivity in a barrel shifter is increased from the Illiac network by having $(2n - 5) \cdot 2^{n-1}$ more direct links. As demonstrated in Figures 5.8*b* and 5.11 for $N = 16$ ($n = 4$, $r = 4$), the Illiac network has 32 direct links and the same size barrel shifter has 56 links. The two networks are identical only when the size is reduced to be no greater than $n = 2$ or $N = 5$.

The barrel shifter can be implemented as either a recirculating single-stage network or as a multistage network. Figure 5.12 shows the interconnection patterns in a recirculating barrel shifter for $N = 8$. The barrel shifting functions $B_{\pm 0}$, $B_{\pm 1}$, and $B_{\pm 2}$ are executed by the interconnection patterns shown. For a single-stage barrel shifter of size $N = 2^n$, the minimum number of recirculations B is upper bounded by

$$B \le \frac{\log_2 N}{2} \tag{5.14}$$

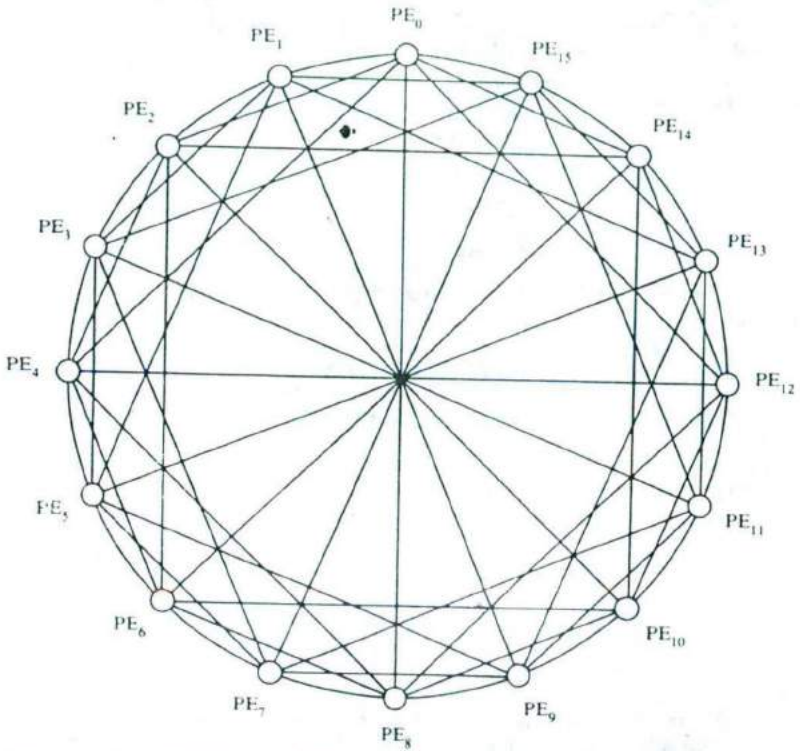


Figure 5.11 Connectivity of a barrel shifter for $N = 16$.

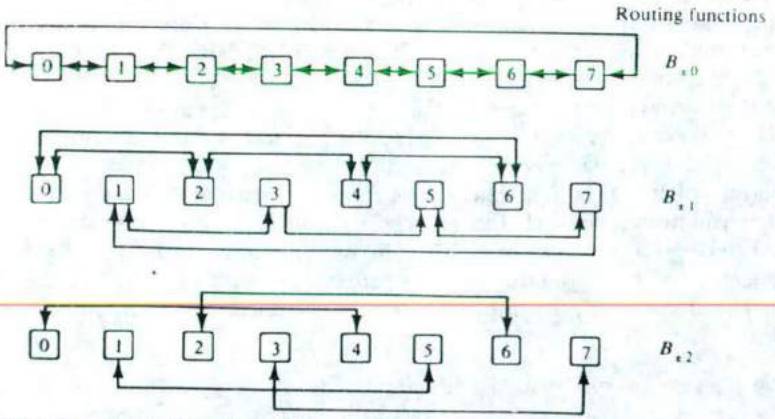


Figure 5.12 A recirculating barrel shifter for $N = 8$.

For the example barrel shifter with $N = 16$, it takes at most two steps to route data from a PE to any other PE. If we assume PE_0 as the source node, PE_0 can reach PE_1, PE_2, PE_4, PE_8, PE_{12}, PE_{14}, or PE_{15} in one step. In two steps, PE_0 can reach PE_3, PE_5, PE_6, PE_7, PE_9, PE_{10}, PE_{11}, or PE_{13}. Thus, one step is saved by using the same size Illiac network. If one replaces the 64-node Illiac-IV network by a 64-node barrel shifter, at most three routing steps are needed (instead of seven steps). The speedup of a barrel shifter over the Illiac network of the same size can be expressed by

$$S_N = \frac{\sqrt{N} - 1}{(\log_2 N)/2} = \frac{2^k - 1}{k} \tag{5.15}$$

where $N = 2^{2k}$. Therefore, the larger the network, the higher the speedup ratio. For very large networks with $N = 2^{2k}$, the speedup approaches $2^k/k$, as demonstrated in Table 5.1.

A barrel shifter has been implemented with multiple stages in the form of a *data manipulator*. As shown in Figure 5.13, the data manipulator consists of n stages of N cells. Each cell is essentially a controlled shifter. This network is designed for implementing data-manipulating functions such as *permuting*, *replicating*, *spacing*, *masking*, and *complementing*. To implement a data-manipulating function, proper control lines of the six groups $(u_1^i, u_2^i, h_1^i, h_2^i, d_1^i, d_2^i)$ in each column must be properly set through the use of the control register and the associated decoder.

The schematic logic circuit of a typical cell in a data manipulator is shown in Figure 5.14. For $0 \le k \le N - 1$ and $0 \le i \le n - 1$, the kth cell at stage i (column 2^i) has three inputs, three sets of outputs, and three control signals. Individual stage control is used with three sets of control signals per stage. The control lines u^i, h^i, and d^i are connected to the AND gates in each cell of stage i. The u^i line controls the backforward barrel shifting (-2^i) and the d^i line controls forward barrel shifting $(+2^i)$. The horizontal line corresponds to no shifting under the control of the h^i signal. Note that stage i performs the distance 2^i shiftings. By passing data through the n stages from left to right, the shifting distance decreases from 2^{n-1} to 2^{n-2} and eventually to 2^1 and 2^0 at the output end. Note that all the

Table 5.1 A comparison of bounds on the minimum routing steps in Illiac network and in barrel shifter of various sizes

k	Network size $N = 2^{2k}$	Illiac network (Eq. 5.5)	Barrel shifter (Eq. 5.14)	Speed up (Eq. 5.15)
2	16	3	2	1.50
3	64	7	3	2.33
4	256	15	4	3.75
5	1024	31	5	6.20

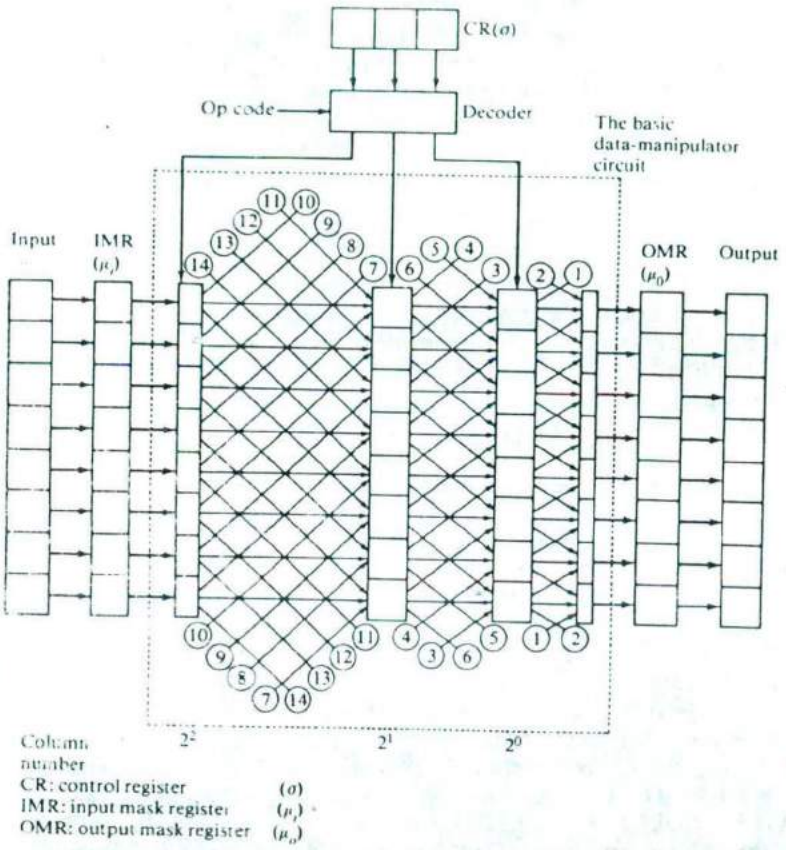


Figure 5.13 The data manipulator for $N = 14$. (Courtesy of *IEEE Trans. Computers*, Feng 1974.)

shifting operations at all stages are module N. This is reflected by the wraparound connections in the data manipulator.

In terms of permutations, the B_{+i} routing function can be expressed by the following product of 2^i cycles by size 2^{n-i} each:

$$\prod_{k=0}^{2^i-1} (k \;\; k+2^i \;\; k+2\cdot 2^i \;\; k+3\cdot 2^i \;\cdots\; k+N-2^i) \tag{5.16}$$

For the example network of $N = 8$, the B_{+1} function is represented by the following permutation (0 2 4 6) (1 3 5 7). Similarly, for B_{-1} we have the following permutation in cycle notation:

$$\prod_{k=0}^{2^i-1} (k+N-2^i \;\cdots\; k+3\cdot 2^i \;\; k+2\cdot 2^i \;\; k+2^i \;\; k) \tag{5.17}$$

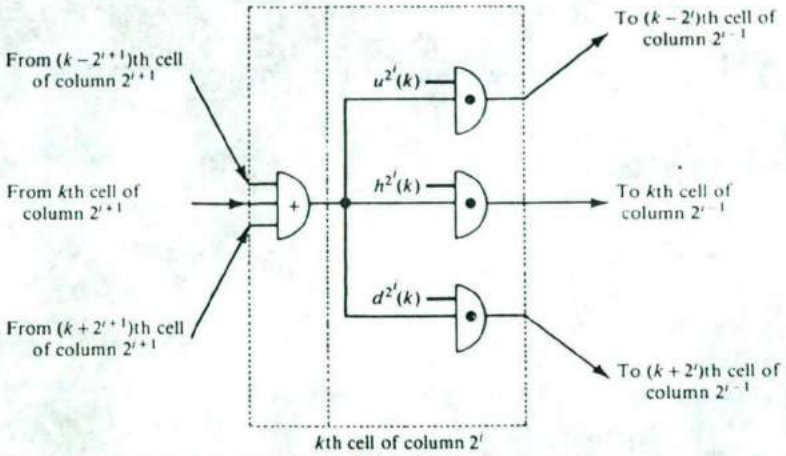


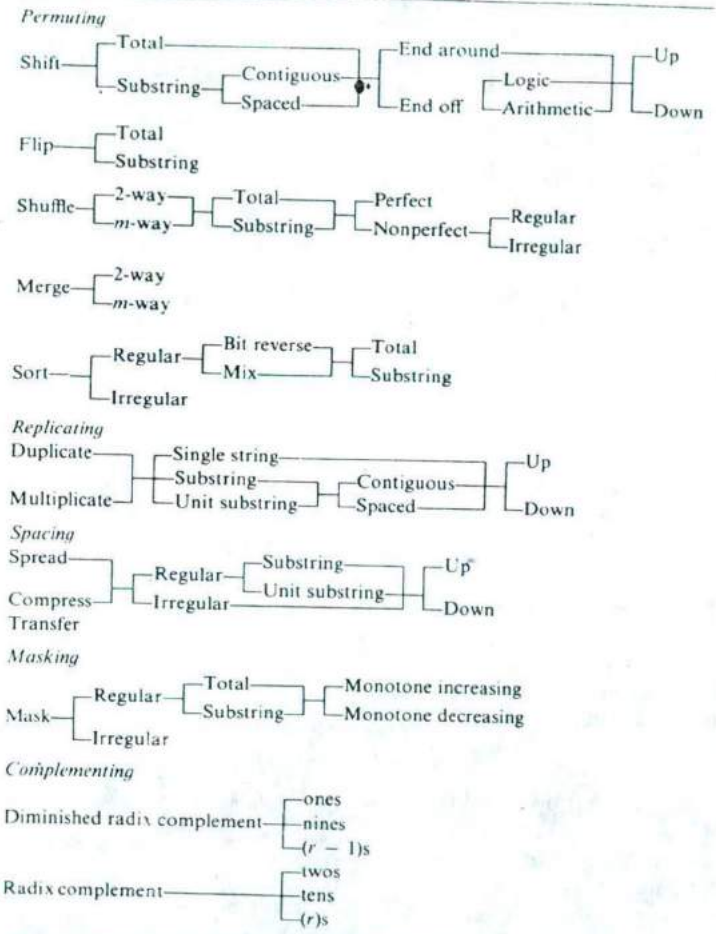
Figure 5.14 The logic design of an intermediate cell in the data manipulator.

Listed in Table 5.2 are data-manipulation functions that are implementable with the data manipulator. The network can perform various types of permutations such as *shift*, *flip*, *shuffle*, *merge*, and *sort*. It can be also used to *replicate* and *space* the data. The network does not provide the capability of masking or complementing. However, we include them in the table for the sake of completeness. Primitive operations among the aforementioned data-manipulation functions are listed below:

1. Total shift up, end around
2. Total shift up, end off
3. Spaced substrings shift up, end around
4. Contiguous substring shift up, end off
5. Spaced substrings shift up, end off
6. Substring flip
7. Multiplicate spaced substring up
8. Spread substrings with 2^s spacing up

Additional data-manipulating functions can be generated by using different sequences of primitive operations. One can augment the data manipulator by introducing individual cell control instead of individual stage control. This relaxation in control will increase the functional flexibility at the expense of significantly increased control cost. A prototype data manipulator has been implemented and attached as an interface device to the STARAN computer at the Rome Air Force Development Center in New York. Because of the uniform structure and low wire density buildup, the data manipulator is a good candidate for VLSI implementation.

Table 5.2 Data-manipulating function (Feng)



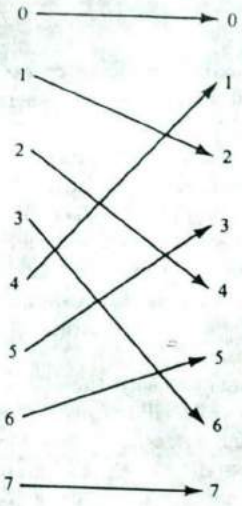
(Courtesy of *IEEE Trans. Computers*, March 1974.)

5.2.5 Shuffle-Exchange and Omega Networks

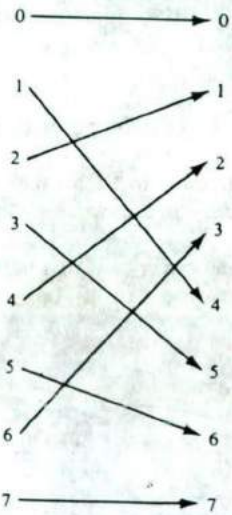
The class of shuffle-exchange networks is based on two routing functions *shuffle* (*S*) and *exchange* (*E*). Let $A = a_{n-1} \cdots a_1 a_0$ be a PE address:

$$S(a_{n-1} \cdots a_1 a_0) = a_{n-2} \cdots a_1 a_0 a_{n-1} \tag{5.18}$$

where $0 \leq A \leq N - 1$ and $n = \log_2 N$. The cyclic shifting of the bits in A to the left for one bit position is performed by the S function. This action corresponds to perfectly shuffling a deck of N cards, as demonstrated in Figure 5.15. The *perfect*



(*a*) The perfect shuffle



(*b*) The inverse perfect shuffle

Figure 5.15 The perfect shuffle and the inverse perfect shuffle for $N = 8$.

shuffle cuts the deck into two halves from the center and intermixes them evenly. The *inverse perfect shuffle* does the opposite to restore the original ordering. The exchange-routing function E is defined by;

$$E(a_{n-1} \cdots a_1 a_0) = a_{n-1} \cdots a_1 \bar{a}_0 \tag{5.19}$$

The complementing of the least significant digit means the exchange of data between two PEs with adjacent addresses. Note that $E(A) = C_0(A)$, where C_0 was the cube routing function defined in Eq. 5.9.

These shuffle-exchange functions can be implemented as either a recirculating network or a multistage network. For $N = 8$, a single-stage recirculating shuffle-exchange network is shown in Figure 5.16. The solid line indicates *exchange* and the dashed line indicates *shuffle*. The use of a recirculating shuffle-exchange network for parallel processing was proposed by Stone. There are a number of parallel algorithms that can be effectively implemented with the use of the shuffle and exchange functions. The examples include the fast Fourier transform (FFT), polynomial evaluation, sorting, and matrix transposition, etc.

The shuffle-exchange functions have been implemented with the multistage Omega network by Lawrie. The Omega network for $N = 8$ is illustrated in Figure 5.17. An $N \times N$ Omega network consists of n identical stages. Between two adjacent stages is a perfect-shuffle interconnection. Each stage has $N/2$ switch boxes under independent box control. Each box has four functions (straight, exchange, upper broadcast, lower broadcast), as illustrated in Figure 5.6. The switch boxes in the Omega network can be repositioned as shown in Figure 5.17*b* without violating the perfect-shuffle interconnections between stages.

The n-cube network shown in Figure 5.10 has the same interconnection topology as the repositioned Omega (Figure 5.17*b*). The two networks differ in two aspects:

1. The cube network uses two-function switch boxes, whereas the Omega network uses four-function switch boxes.
2. The data-flow directions in the two networks are opposite to each other. In other words, the roles of input-output lines are exchanged in the two networks.

Based on the above differences, the n-cube and Omega networks have different capabilities even with isomorphic topologies. Suppose we wish to establish the

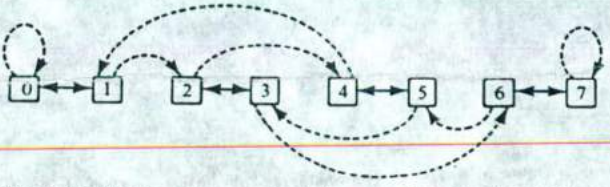
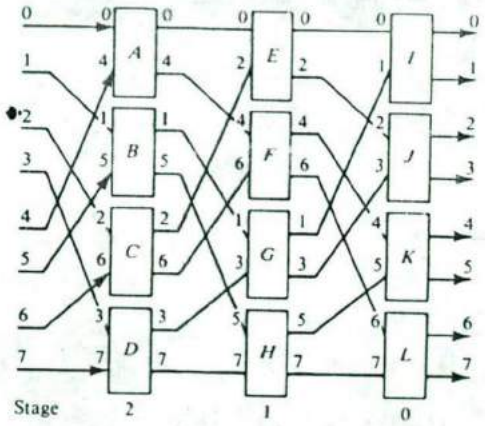
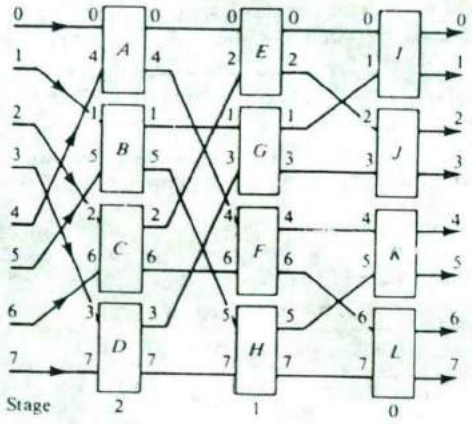


Figure 5.16 Shuffle-exchange recirculating network for $N = 8$. (Solid lines are *exchanges* and dashed lines are *shuffle*.)



(*a*) The shuffle-exchange network for $n = 8$ (Omega network)



(*b*) the Omega network with switch box repositioned

Figure 5.17 The multistage Omega network proposed by Lawrie (1975).

I/O connections zero to five and one to seven. The Omega network (Figure 5.17*a*) can perform this task, whereas the *n*-cube (Figure 5.10) network cannot. On the other hand, the *n*-cube network can connect five to zero and seven to one, but the Omega network cannot. In general, the Omega network can perform one to many connections, while the *n*-cube network cannot. However, if one considers only bijections (one-to-one connections), the *n*-cube and Omega networks are functionally equivalent by some relabeling techniques.

If one applies the shuffle function S i times, written as S^i, the binary sequence in Eq. 5.18 will be shifted cyclically to the left i bit positions. The shuffle function S (Eq. 5.18) corresponds to the following permutation cycles:

$$\prod_{j=0}^{N-1} (j \ S(j) \ S^2(j) \cdots) \tag{5.20}$$

where, for each cycle, j has not appeared in a previous cycle. The largest cycle in the above permutation has order n. For $N = 8$, the shuffle function corresponds to the permutation (0) (1 2 4) (3 5 6) (7).

The exchange function E (Eq. 5.19) can be expressed as a product of $N/2$ cycles of order two, provided that the index j is even:

$$\prod_{j=0}^{N-2} (j \ j+1) \tag{5.21}$$

For $N = 8$, the exchange function results in the following permutation function: (0 1) (2 3) (4 5) (6 7).

Comparing various multistage SIMD networks, we conclude that increasing flexibilities in interconnection capabilities are found in the following order: the flip network, the binary n cube, Omega, and data manipulator. The increased flexibility is obtained with increased cost. The flip network is the only network among the four that has been commercially constructed for a large network size of $N = 256$. The cost-effectiveness of a network for a particular application is the fundamental question to be answered before entering the design and construction phases. One can always use a single-stage recirculating network to simulate the multistage counterpart. Table 5.3 shows the lower and upper bounds for all such simulations. Entries in row i and column j are the lower and upper bounds on the number of recirculations needed for the single-stage network i to simulate the multistage network j, where $n = \log_2 N$. These simulation bounds can be used to analyze network suitability and capability for a particular class of applications.

Table 5.3 Lower and upper bounds on the number of transfers for the network in row i to simulate the network in column j, where $n = \log_2 N$

	Bound	Illiac	Barrel	Shuffle-exchange	Cube
Illiac	Lower	—	$\sqrt{N}/2$	$1 + \sqrt{N}/2$	$1 + \sqrt{N}/2$
	Upper	—	$\sqrt{N}/2$	$3\sqrt{N} - 4$	$1 + \sqrt{N}/2$
Barrel	Lower	1	—	n	2
	Upper	1	—	$2n - 2$	2
Shuffle-exchange	Lower	$2n - 1$	$2n - 1$	—	$n + 1$
	Upper	$2n$	$2n$	—	$n + 1$
Cube	Lower	n	n	n	—
	Upper	n	n	n	—

5.3 PARALLEL ALGORITHMS FOR ARRAY PROCESSORS

The original motivation for developing SIMD array processors was to perform parallel computations on vector or matrix types of data. Parallel processing algorithms have been developed by many computer scientists for SIMD computers. Important SIMD algorithms can be used to perform matrix multiplication, fast Fourier transform (FFT), matrix transposition, summation of vector elements, matrix inversion, parallel sorting, linear recurrence, boolean matrix operations, and to solve partial differential equations. We study below several representative SIMD algorithms for matrix multiplication, parallel sorting, and parallel FFT. We shall analyze the speedups of these parallel algorithms over the sequential algorithms on SISD computers. The implementation of these parallel algorithms on SIMD machines is described by *concurrent* ALGOL. The physical memory allocations and program implementation depend on the specific architecture of a given SIMD machine.

5.3.1 SIMD Matrix Multiplication

Many numerical problems suitable for parallel processing can be formulated as matrix computations. Matrix manipulation is frequently needed in solving linear systems of equations. Important matrix operations include matrix multiplication, L-U decomposition, and matrix inversion. We present below two parallel algorithms for matrix multiplication. The differences between SISD and SIMD matrix algorithms are pointed out in their program structures and speed performances. In general, the inner loop of a multilevel SISD program can be replaced by one or more SIMD vector instructions.

Let $A = [a_{ik}]$ and $B = [b_{kj}]$ be $n \times n$ matrices. The multiplication of A and B generates a product matrix $C = A \times B = [c_{ij}]$ of dimension $n \times n$. The elements of the product matrix C is related to the elements of A and B by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \qquad \text{for } 1 \le i \le n \text{ and } 1 \le j \le n \tag{5.22}$$

There are n^3 cumulative multiplications to be performed in Eq. 5.22. A cumulative multiplication refers to the linked multiply-add operation $c \leftarrow c + a \times b$. The addition is merged into the multiplication because the multiply is equivalent to multioperand addition. Therefore, we can consider the unit time as the time required to perform one cumulative multiplication, since add and multiply are performed simultaneously.

In a conventional SISD uniprocessor system, the n^3 cumulative multiplications are carried out by a serially coded program with three levels of DO loops corresponding to three indices to be used. The time complexity of this sequential program is proportional to n^3, as specified in the following SISD algorithm for matrix multiplication.

Example 5.3: An $O(n^3)$ algorithm for SISD matrix multiplication

```
For i = 1 to n Do
  For j = 1 to n Do
    c_ij = 0 (initialization)
    For k = 1 to n Do
      c_ij = c_ij + a_ik · b_kj (scalar additive multiply)
    End of k loop
  End of j loop
End of i loop
```
(5.23)

Now, we want to implement the matrix multiplication on an SIMD computer with n PEs. The algorithm construct depends heavily on the memory allocations of the A, B, and C matrices in the PEMs. Suppose we store each row vector of the matrix across the PEMs, as illustrated in Figure 5.18. Column vectors are then stored within the same PEM. This memory allocation scheme allows parallel access of all the elements in each row vector of the matrices. Based in this data distribution, we obtain the following parallel algorithm. The two **parallel do** operations correspond to *vector load* for initialization and *vector multiply* for the inner loop of additive multiplications. The time complexity has been reduced to $O(n^2)$. Therefore, the SIMD algorithm is n times faster than the SISD algorithm for matrix multiplication.

Example 5.4: An $O(n^2)$ algorithm for SIMD matrix multiplication

```
For j = 1 to n Do
Par for k = 1 to n Do
  c_ik = 0 (vector load)
For j = 1 to n Do
Par for k = 1 to n Do
  c_ik = c_ik + a_ij · b_jk (vector multiply)
End of j loop
End of i loop
```
(5.24)

It should be noted that the *vector load* operation is performed to initialize the row vectors of matrix C one row at a time. In the *vector multiply* operation, the same multiplier a_{ij} is broadcast from the CU to all PEs to multiply all n elements $\{b_{ik}$ for $k = 1, 2, \ldots, n\}$ of the ith row vector of B. In total, n^2 *vector multiply*

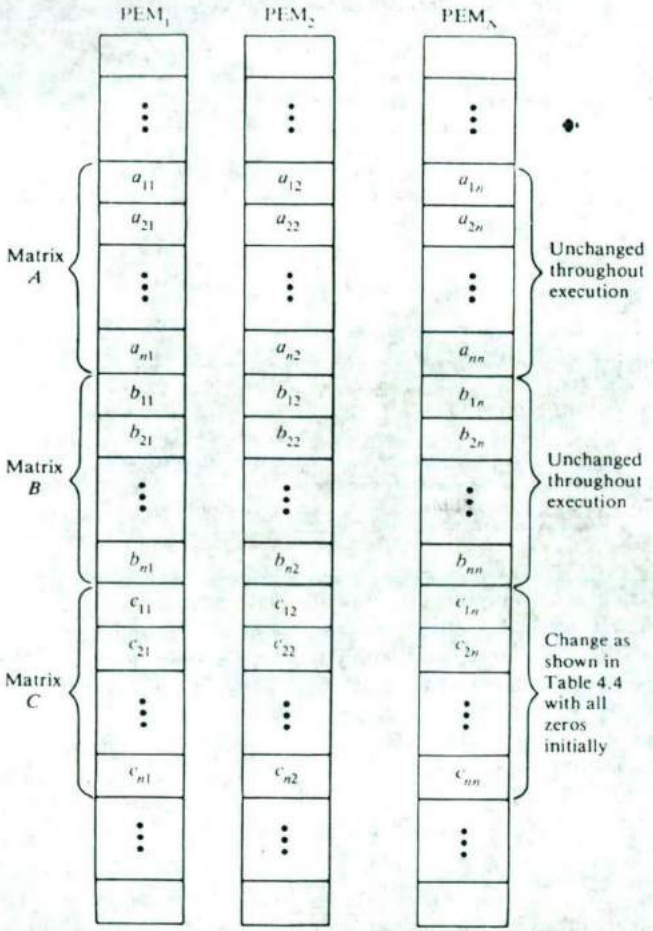


Figure 5.18 Memory allocation for SIMD matrix multiplication.

operations are needed in the double loops. The successive memory contents in the execution of the above SIMD matrix multiplication program are illustrated in Table 5.4. Each *vector multiply* instruction implies n parallel scalar multiplications in each of the n^2 iterations. This algorithm in Example 5.4 is implementable on an array of n PEs.

If we increase the number of PEs used in an array processor to n^2, an $O(n \log_2 n)$ algorithm can be devised to multiply the two $n \times n$ matrices A and B. Let $n = 2^m$ and recall the binary cube network described in Section 5.2.3. Consider an array processor whose $n^2 = 2^{2m}$ PEs are located at the 2^{2m} vertices of a $2m$-cube network. A $2m$-cube network can be considered as two $(2m - 1)$-cube networks linked

Table 5.4 Successive contents of the C array in memory

Outer loop i	Inner loop j	Parallel SIMD operations on $k = 1, 2, \ldots, n$			
		$c_{i1} \leftarrow c_{i1} + a_{ij} \times b_{j1}$	$c_{i2} \leftarrow c_{i2} + a_{ij} \times b_{j2}$	$\cdots$	$c_{in} \leftarrow c_{in} + a_{ij} \times b_{jn}$
1	1	$c_{11} \leftarrow c_{11} + a_{11} \times b_{11}$	$c_{12} \leftarrow c_{12} + a_{11} \times b_{12}$		$c_{1n} \leftarrow c_{1n} + a_{11} \times b_{1n}$
	2	$c_{11} \leftarrow c_{11} + a_{12} \times b_{21}$	$c_{12} \leftarrow c_{12} + a_{12} \times b_{22}$		$c_{1n} \leftarrow c_{2n} + a_{12} \times b_{2n}$
	n	$c_{11} \leftarrow c_{11} + a_{1n} \times b_{n1}$	$c_{12} \leftarrow c_{12} + a_{1n} \times b_{n2}$		$c_{1n} \leftarrow c_{1n} + a_{1n} \times b_{nn}$
2	1	$c_{21} \leftarrow c_{21} + a_{21} \times b_{11}$	$c_{22} \leftarrow c_{22} + a_{21} \times b_{12}$		$c_{2n} \leftarrow c_{2n} + a_{21} \times b_{1n}$
	2	$c_{21} \leftarrow c_{21} + a_{22} \times b_{21}$	$c_{22} \leftarrow c_{22} + a_{22} \times b_{22}$		$c_{2n} \leftarrow c_{2n} + a_{22} \times b_{2n}$
	n	$c_{21} \leftarrow c_{21} + a_{2n} \times b_{n1}$	$c_{22} \leftarrow c_{22} + a_{2n} \times b_{n2}$		$c_{2n} \leftarrow c_{2n} + a_{2n} \times b_{2n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	1	$c_{n1} \leftarrow c_{n1} + a_{n1} \times b_{11}$	$c_{n2} \leftarrow c_{n2} + a_{n1} \times b_{12}$		$c_{nn} \leftarrow c_{nn} + a_{n1} \times b_{1n}$
	2	$c_{n1} \leftarrow c_{n1} + a_{n2} \times b_{21}$	$c_{n2} \leftarrow c_{n2} + a_{n2} \times b_{22}$		$c_{nn} \leftarrow c_{nn} + a_{n2} \times b_{2n}$
	n	$c_{nn} \leftarrow c_{nn} + a_{nn} \times b_{n1}$	$c_{n2} \leftarrow c_{n2} + a_{nn} \times b_{n2}$		$c_{nn} \leftarrow c_{nn} + a_{nn} \times b_{nn}$
Local memory		PEM_1	PEM_2	$\cdots$	PEM_n

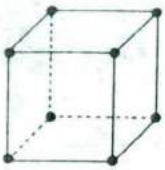
together by $2m$ extra edges. In Figure 5.19, a 4-cube network is constructed from two 3-cube networks by using $8 = 2^3$ extra edges between corresponding vertices at the corner positions. For clarity, we simplify the 4-cube drawing by showing only one of the eight fourth dimension connections. The remaining connections are implied.

Let $(p_{2m-1}p_{2m-2}\cdots p_m p_{m-1}\cdots p_1 p_0)_2$ be the PE address in the $2m$ cube. We can achieve the $O(n \log_2 n)$ compute time only if initially the matrix elements are favorably distributed in the PE vertices. The n rows of matrix A are distributed over n distinct PEs whose addresses satisfy the condition

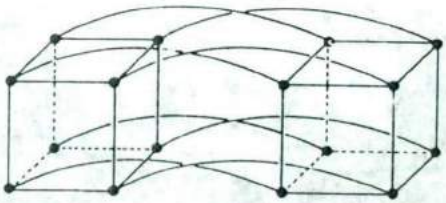
$$p_{2m-1}p_{2m-1}\cdots p_m = p_{m-1}p_{m-2}\cdots p_0 \tag{5.25}$$

as demonstrated in Figure 5.20*a* for the initial distribution of four rows of the matrix A in a 4×4 matrix multiplication ($n = 4$, $m = 2$). The four rows of A are then broadcast over the fourth dimension and front to back edges, as marked by row numbers in Figure 5.20*b*.

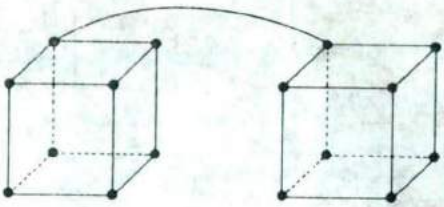
The n columns of matrix B (or the n rows of matrix B^t) are evenly distributed over the PEs of the $2m$ cubes, as illustrated in Figure 5.20*c*. The four rows of B^t are then broadcast over the front and back faces, as shown in Figure 5.20*d*. Figure 5.21 shows the combined results of A and B^t broadcasts with the inner product ready to be computed. The n-way broadcast depicted in Figure 5.20*b* and 5.20*d* takes $\log n$ steps, as illustrated in Figure 5.21 in $m = \log_2 n = \log_2 4 = 2$ steps. The matrix multiplication on a $2m$-cube network is formally specified below.



(*a*) A 3 cube



(*b*) A 4 cube formed from two 3 cubes



(*c*) The 4 cube showing only one of eight fourth-dimension connections

Figure 5.19 The construction of $(m + 1)$-cube from two m-cubes. (Courtesy of Thomas, 1981.)

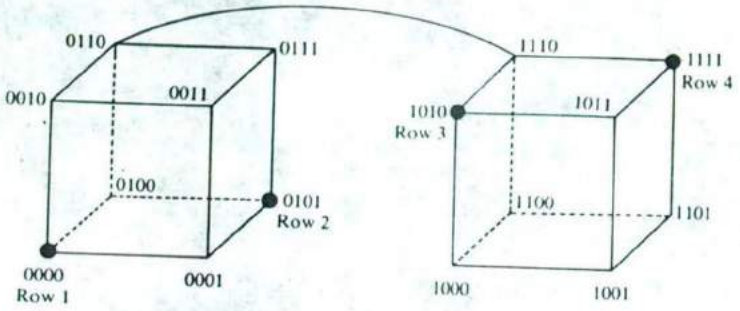
Example 5.5: An $O(n \log_2 n)$ algorithm for matrix multiplication

1. Transpose B to form B^t over the m cubes $x_{2m-1} \cdots x_m 0 \cdots 0$ in $n \log_2 n$ steps (Figure 5.20*c*).
2. N-way broadcast each row of B^t to all PEs in the m cube

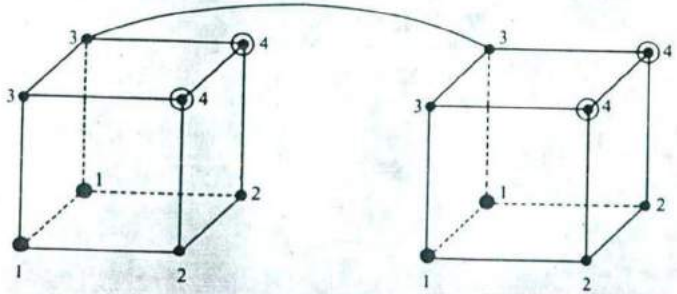
$$p_{2m-1} \cdots p_m x_{m-1} \cdots x_0$$

in $n \log_2 n$ steps (Figure 5.20*d*).

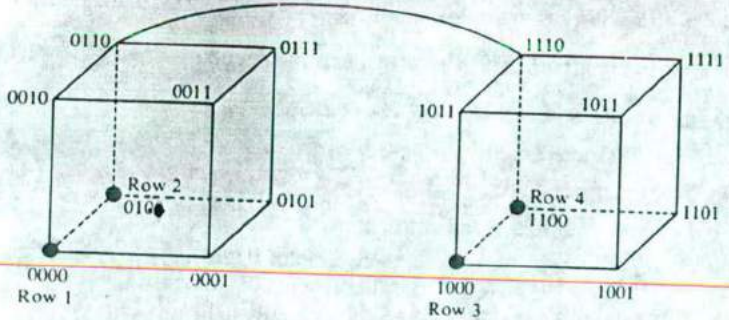
3. N-way broadcast each row of A residing in PE $p_{2m-1} \cdots p_m p_{m-1} \cdots p_0$ to all PEs in the m cube $x_{2m-1} \cdots x_m p_{n-1} \cdots p_0$ in $n \log_2 n$ steps (Figure 5.20*b*). All the n rows can be broadcast in parallel.
4. Each PE now contains a row of A and a column of B and can form the inner product in $O(n)$ steps (Figure 5.21). The n elements of each result row can be brought together within the same PEs which initially held a row of A in $O(n)$ steps.



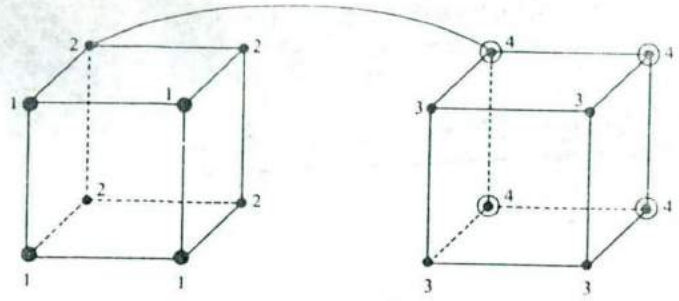
(*a*) Initial distribution of rows of A



(*b*) 4-way broadcase of rows of A



(*c*) Initial distribution of rows of B^t



(*d*) 4-way broadcase of rows of B^t

Figure 5.20 Allocation of the elements of two 4 × 4 matrices in a 4-cube of 16 PEs.

The above algorithm takes a total of $3n \log_2 n + O(n)$ time steps to complete, which equals $O(n \log_2 n)$. This demonstrates a gain in speed over the $O(n^2)$ algorithm in Example 5.4 at the expense of using n^2 PEs over the use of only n PEs in the slow algorithm. In Chapter 10, we shall further show a VLSI hardware approach to complete the n-by-n matrix multiplication in $O(n)$ time using $O(n^2/m^2)$ VLSI processor arrays, each consisting of an array of $O(m^2)$ PEs for pipelined inner-product computations.

5.3.2 Parallel Sorting on Array Processors

An SIMD algorithm is to be presented for sorting n^2 elements on a mesh-connected (Illiac-IV-like) processor array in $O(n)$ routing and comparison steps. This shows a speedup of $O(\log_2 n)$ over the best sorting algorithm, which takes $O(n \log_2 n)$ steps on a uniprocessor system. We assume an array processor with $N = n^2$ identical PEs interconnected by a mesh network (Figure 5.22) similar to Illiac-IV except that the PEs at the perimeter have two or three rather than four neighbors. In other words, there are no *wraparound* connections in this simplified mesh network.

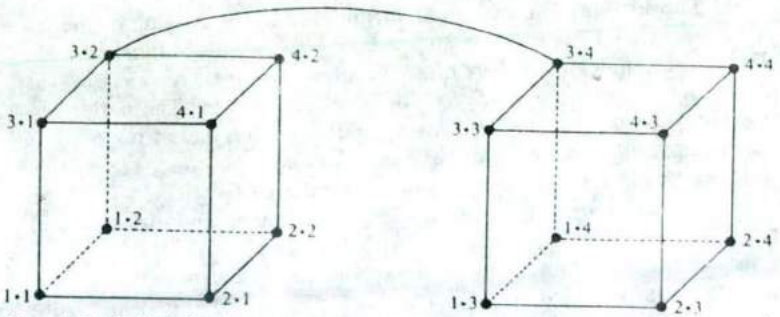


Figure 5.21 The final distributions of the rows of matrix A and the columns of matrix B ready for inner product in the 16 PEs of a 4-cube array processor.

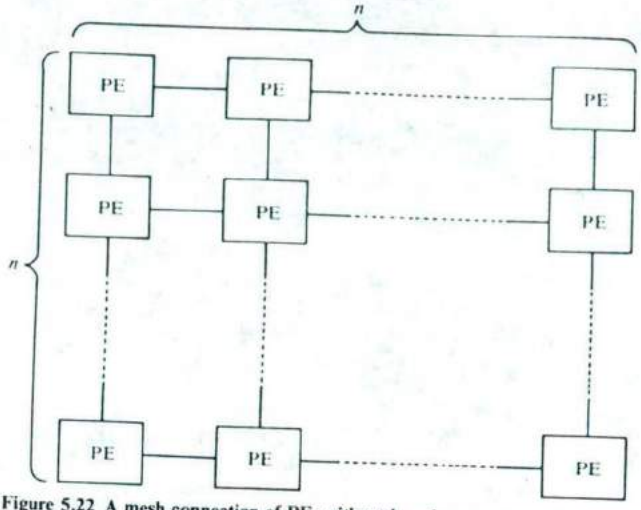


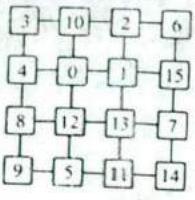
Figure 5.22 A mesh connection of PEs without boundary wraparound connections for SIMD sorting.

Eliminating the wraparound connections simplifies the array-sorting algorithm. The time complexity of the array-sorting algorithm would be affected by, at most, a factor of two if the wraparound connections were included.

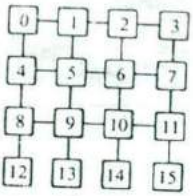
Two time measures are needed to estimate the time complexity of the parallel-sorting algorithm. Let t_R be the *routing time* required to move one item from a PE to one of its neighbors, and t_C be the *comparison time* required for one comparison step. Concurrent data routing is allowed. Up to N comparisons may be performed simultaneously. This means that a comparison-interchange step between two items in adjacent PEs can be done in $2t_R + t_C$ time units (route left, compare, and route right). A mixture of horizontal and vertical comparison interchanges requires at least $4t_R + t_C$ time units.

The sorting problem depends on the indexing schemes on the PEs. The PEs may be indexed by a bijection from $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ to $\{0, 1, \ldots, N-1\}$, where $N = n^2$. The sorting problem can be formulated as the moving of the jth smallest element in the PE array for all $j = 0, 1, 2, \ldots, N-1$. Illustrated in Figure 5.23 are three indexing patterns formed after sorting the given array in part *a* with respect to three different ways for indexing the PEs. The pattern in part *b* corresponds to a *row-majored indexing*, part *c* corresponds to a *shuffled row-major* indexing, and is based on a *snake-like row-major indexing*. The choice of a particular indexing scheme depends upon how the sorted elements will be used. We are interested in designing sorting algorithms which minimize the total routing and comparison steps.

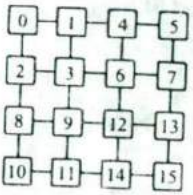
The longest routing path on the mesh in a sorting process is the transposition of two elements initially loaded at opposite corner PEs, as illustrated in Figure 5.24. This transposition needs at least $4(n-1)$ routing steps. This means that no



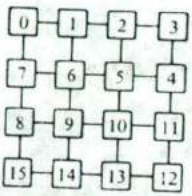
(*a*) Initial loading pattern before sorting



(*b*) Sorted pattern with row-major indexing



(*c*) Sorted pattern with shuffled row major indexing



(*d*) Sorted pattern with snakelike row-major indexing

Figure 5.23 Sorting patterns with respect to three ways of indexing the PEs.

algorithm can sort n^2 elements in a time of less than $O(n)$. In other words, an $O(n)$ sorting algorithm is considered optimal on a mesh of n^2 PEs. Before we show one such optimal sorting algorithm on the mesh-connected PEs, let us review Batcher's *odd-even merge sort* of two sorted sequences on a set of linearly connected PEs shown in Figure 5.25. The *shuffle* and *unshuffle* operations can each be implemented with a sequence of interchange operations (marked by the double-arrows in Figure 5.26). Both the perfect shuffle and its inverse (unshuffle) can be done in $k - 1$ interchanges or $2(k - 1)$ routing steps on a linear array of $2k$ PEs.

Batcher's odd-even merge sort on a linear array has been generalized by Thompson and Kung to a square array of PEs. Let $M(j, k)$ be a sorting algorithm for merging two sorted j-by-$k/2$ subarrays to form a sorted j-by-k array, where j and k are powers of 2 and $k > 1$. The snakelike row-major ordering is assumed in all the arrays. In the degenerate case of $M(1, 2)$, a single comparison-interchange step is sufficient to sort two unit subarrays. Given two sorted columns of length $j \geq 2$, the $M(j, 2)$ algorithm consists of the following steps:

Example 5.6: The $M(j, 2)$ sorting algorithm

J1: Move all odds to the left column and all evens to the right in $2t_k$ time.
J2: Use the *odd-even transposition sort* to sort each column in $2jt_k + jt_c$ time.
J3: Interchange on each row in $2t_k$ time.
J4: Perform one comparison-interchange in $2t_k + t_c$ time.

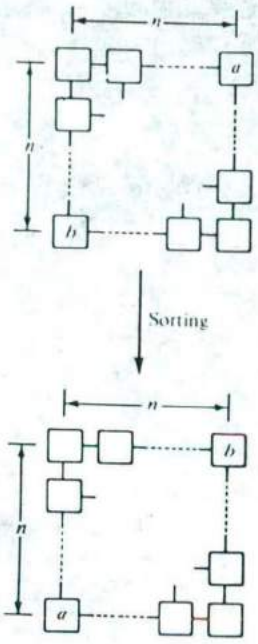


Figure 5.24 The transposition of two elements at opposite corner PEs.

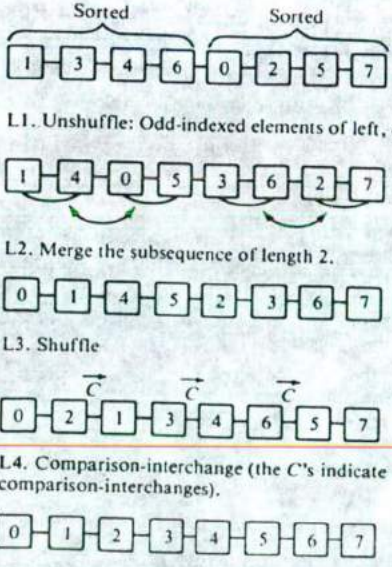


Figure 5.25 Batcher's odd-even merge of two sorted sequences on a linear array of PEs.

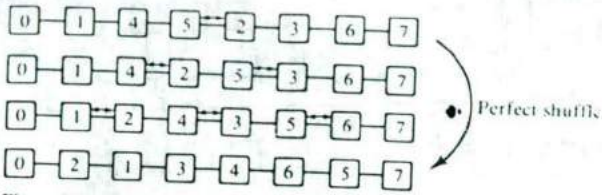


Figure 5.26 The implementation of a perfect shuffle by a sequence of interchange operations.

The above $M(j, 2)$ algorithm is illustrated in Figure 5.27 for an $M(4, 2)$ sorting. When $j > 2$ and $k > 2$, the $M(j, k)$ sorting algorithm for a meshed-connected array processor is recursively specified as follows:

Example 5.7: The $M(j, k)$ sorting algorithm

M1: If $j > 2$, perform a single interchange step on even rows so that columns contain either all evens or all odds. If $j = 2$, the columns are already segregated, so nothing else needs to be done (time: $2t_k$).
M2: Unshuffle each row [time: $(k - 2) \cdot t_k$].
M3: Merge by calling algorithm $M(j, k/2)$ on each half of the array [time: $T(j, k/2)$].
M4: Shuffle each row [time: $(k - 2) \cdot t_k$].
M5: Interchange on even rows (time: $2t_k$).
M6: Comparison-interchange adjacent elements (every *even* with the next *odd*) (time: $4t_k + t_c$).

For $j > 2$ and $k > 2$, the $M(j, k)$ sorting algorithm is illustrated in Figure 5.28 for the case of $M(4, 4)$. Steps M1 and M2 unshuffle the elements. Step M3 recursively merges the *odd subsequences* and the *even subsequences*. Steps 4 and 5 shuffle the *odd* and *even* together. M6 performs the final comparison interchange. Two sorted 4-by-2 subarrays are being merged to form a 4-by-4 sorted array in snakelike row-major ordering. Let $T(j, k)$ be the time required to perform all the steps in the $M(j, k)$ sorting algorithm. In the degenerated case of $k = 2$, we have

$$T(j, 2) = (2j + 6)t_R + (j + 1)t_C \tag{5.26}$$

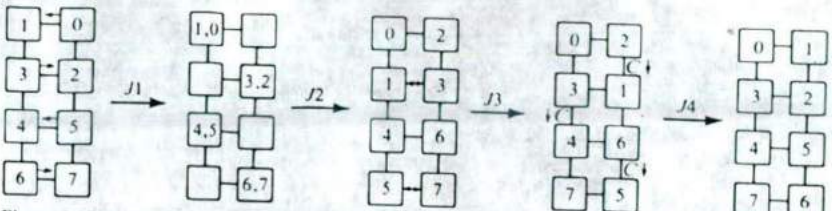


Figure 5.27 Data routing, comparison, and interchange operations performed in the $M(4, 2)$ sorting algorithm. (Courtesy of *IEEE Trans. Computers*, Thompson and Kung 1977.)

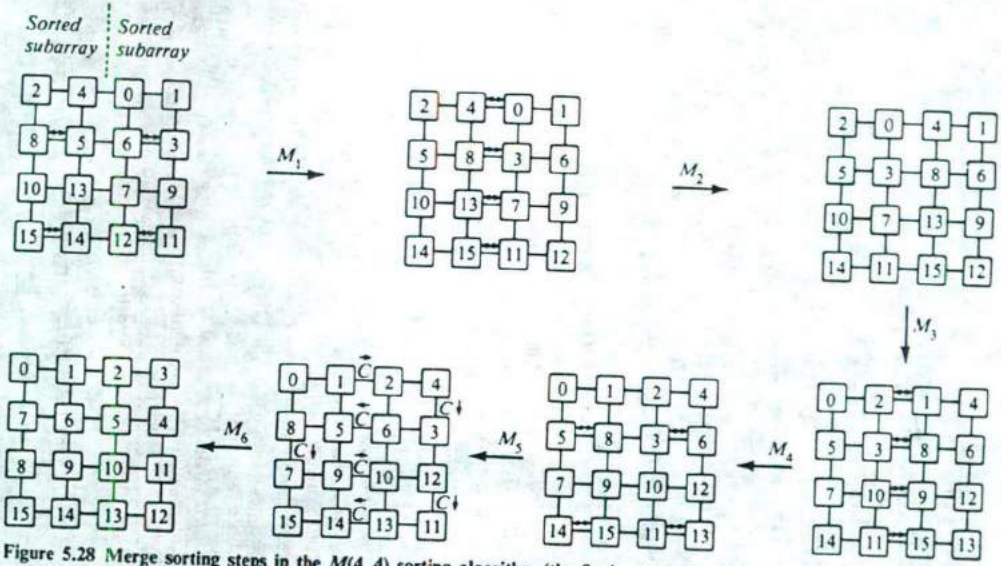


Figure 5.28 Merge sorting steps in the $M(4, 4)$ sorting algorithm (the final sorted 4×4 array has snake-like row major ordering). (Courtesy of *IEEE Trans. Computers*, Thompson and Kung 1977.)

For $k > 2$, the time function is recursively computed by:

$$T(j, k) = (2k + 4)t_R + t_C + T(j, k/2) \tag{5.27}$$

By repeated substitution, we have the following time bound:

$$T(j, k) \leq (2j + 4k + 4\log_2 k)t_R + (j + \log_2 k)t_C \tag{5.28}$$

For an $n \times n$ array of PEs, the $M(n, n)$ sort algorithm can be done in $T(n, n)$ time, which is proportional to $O(n)$:

$$T(n, n) = (6n + 4\log_2 n)t_R + (n + \log_2 n)t_C = O(n) \tag{5.29}$$

Combining an *upwards* merge with the *sideways* merge (just described), one can further tighten the above bound to within a factor of $6n$ under the assumption that $t_C \leq t_R$. This parallel sorting algorithm has a speedup of $\log_2 n$ over the best $O(n \log_2 n)$ algorithm for serial sorting.

5.3.3 SIMD Fast Fourier Transform

SIMD algorithms for performing one-dimensional and two-dimensional *fast Fourier transform* (FFT) are presented in this section. Let $s(k), k = 0, 1, \ldots, M - 1$ be M samples of a time function. The discrete Fourier transform of $s(k)$ is defined to be the discrete function $x(j), j = 0, 1, \ldots, M - 1$ where

$$x(j) = \sum_{k=0}^{M-1} s(k) \cdot W^{jk} \qquad j = 0, 1, \ldots, M - 1 \tag{5.30}$$

and $W = e^{2\pi i/M}$ and $i = \sqrt{-1}$.

Consider the use of an SIMD machine with $N = M/2$ PEs to perform an M-point FFT of the discrete signal sequence $\{s(m),\ 0 \leq m \leq M - 1\}$. The algorithm is a parallel implementation of the *decimation-in-frequency* (DIF) technique illustrated in Figure 5.29 for the case of $M = 16$ sampling points. The DIF algorithm divides the input sequence $\{s(m)\}$ into two half subsequences $\{f(m)\}$ and $\{g(m)\}$ such that $f(m) = s(m)$ and

$$g(m) = s\left(m + \frac{M}{2}\right) \qquad \text{for } m = 0, 1, \ldots, \frac{M}{2} - 1$$

The FFT of the M-point sequence can be computed in terms of the two $M/2$-point FFTs of the two sequences $\{f(m) + g(m)\}$ and $\{[f(m) - g(m)] \cdot W^m\}$, where $0 \leq m < M/2$. For M being a power of 2, repeated applications of this dividing process require $O(M \log_2 M)$ array operations.

For the parallel FFT algorithm, PE_i initially contains $s(k)$ and $s(k + M/2)$, where $0 \leq k < N$. As in the serial method, $\log_2 M$ stages of computations are needed. At each stage, $M/2$ *butterfly* operations, shown in Figure 5.30, are executed. The items being paired in a butterfly at stage k where $0 \leq k < \log_2 M$ are those whose indices differ in the $(\log_2 M - k - 1)$th bit position of the binary representation. Because of this difference in a given bit position, the cube interconnection

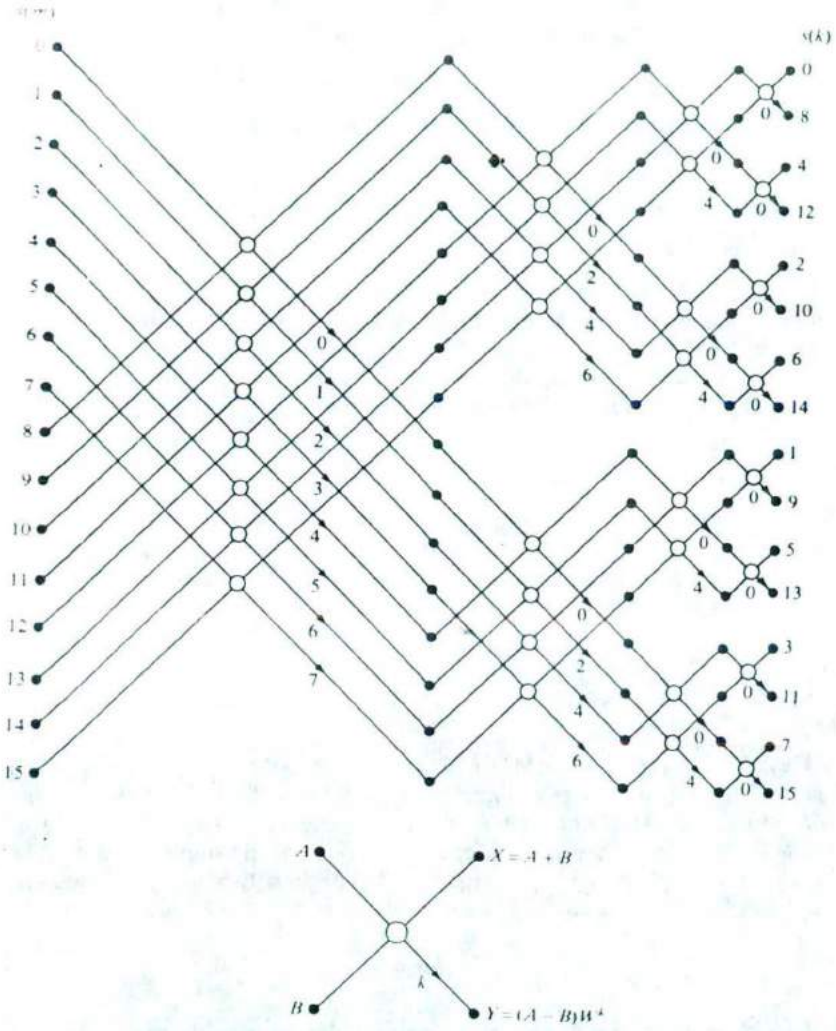


Figure 5.29 Operations in a 16-point FFT based on the decimation-in-frequency algorithm.

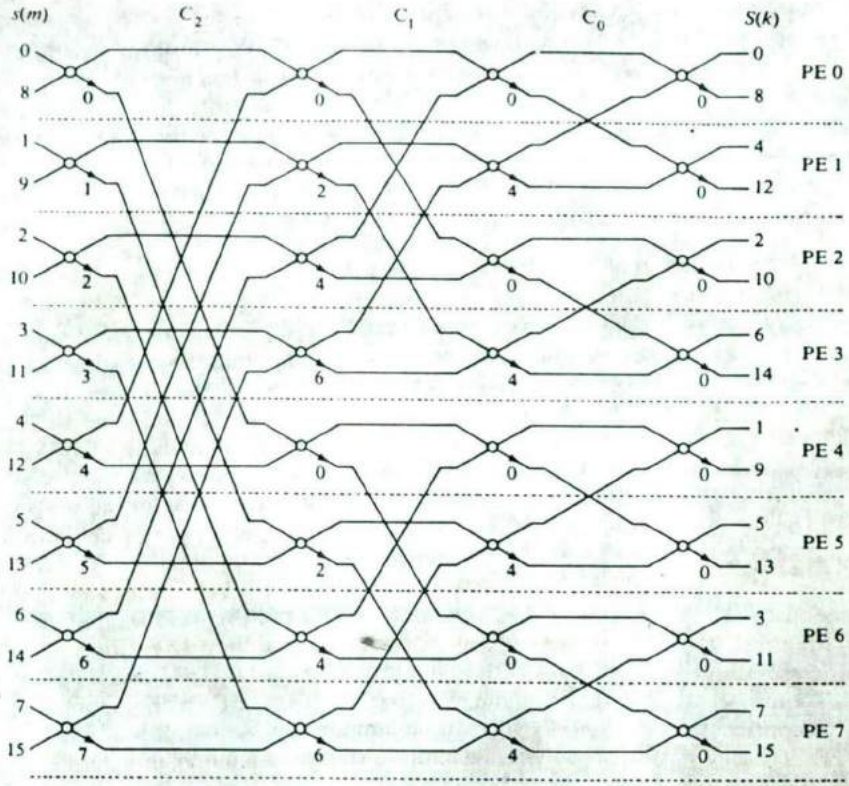


Figure 5.30 Computation of a 16-point FFT in an SIMD machine with 8 PEs. (Courtesy of IEEE *Proc. 5th Int'l Conf. on PRIP*, Mueller, et al. 1980.)

network provides a natural means for specifying the interprocessor data transfers required for the FFT algorithm. This network consists of n routing functions C_i for $0 \leq i < \log_2 N$ as in Eq. 5.9. Figure 5.30 illustrates data transfers and computations performed in the one-dimensional FFT algorithm for an SIMD machine with $N = M/2$ PEs.

The above algorithm performs the M-point FFT calculations using $\log_2(M/2)$ parallel data transfers. This is a lower bound on the number of data transfers required to perform an M-point FFT when the M points are initially distributed over $M/2$ PEs. The number of parallel butterfly operations performed is $\log_2 M$, where each butterfly involves two complex additions and one complex multiplication in each PE. The number of butterfly steps is reduced from $(M/2)\log_2 M$ in a serial FFT algorithm to $\log_2 M$ for a parallel FFT algorithm, using $M/2$ PEs.

Because $M/2$ PEs may not be available for the computation of an M-point FFT, it is of interest to consider FFT algorithms which use fewer PEs. A simple

solution for using fewer PEs is to replicate the steps in the $M/2$ PE algorithm. For example, if $N = M/4$, two computations that were performed in parallel in different PEs in the $M/2$ PE algorithm are now performed sequentially in the same PE, as shown for $N = 16$ in Figure 5.31. The number of butterfly steps performed is $2\log_2 M$. $2(\log_2 M - 2) = 2[\log_2(M/4)]$ parallel data transfers are required. This approach can be generalized to perform an M-point FFT in $M/2^k$ PEs for $2 \leq k \leq \log_2 M$. For $N = M/2^k$ PEs, each PE will initially contain 2^k elements. The number of parallel butterfly steps performed will be $2^{k-1}\log_2 M$. The data transfers will be performed the C_i functions for $\log_2 M - k - 1 \geq i \geq 0$; each C_i function will be replicated 2^{k-1} times. The total number of parallel data transfers will therefore be $2^{k-1}(\log_2 M - k)$.

We consider next the two-dimensional FFT algorithm for processing an M-by-M signal array. A standard approach to computing the two-dimensional FFT of a signal array S is to perform the one-dimensional FFT on the rows of S, giving an intermediate matrix G, then performing the one-dimensional FFT on the columns of G. The resulting matrix F is the two-dimensional FFT of S. An SIMD algorithm which uses $N = M^2/2$ PEs is presented below.

The implementation of a two-dimensional FFT makes use of the previous work done for one-dimensional FFTs. The PEs are logically partitioned into M rows of $M/2$ PEs. Each row of PEs is given a row of the input matrix S, with two matrix elements in each PE. The two-dimensional FFT is implemented by simultaneously having each row of PEs compute the FFT of its row of the input matrix to obtain G. The PEs are then logically reconfigured to form M columns of $M/2$ processors, with each column of PEs having a column of G. Then each column of PEs computes the FFT of its column of G to obtain F, the FFT of the input matrix. This approach can be considered a row-column method in that it transforms the rows of the matrix S to produce G, then transforms the columns of the intermediate matrix G to produce F.

Initially, the PEs are logically configured as M rows of $M/2$ PEs, logically numbered (i, j), where $0 \leq i < M$ and $0 \leq j < M/2$. The physical address of PE (i, j) is $i(M/2) + j$. The physical address can be represented in binary as

$$p_{2\mu-2}p_{2\mu-3}\cdots p_{\mu-1}p_{\mu-2}\cdots p_1 p_0$$

where $\mu = \log_2 M$. Bits $p_{\mu-2}\cdots p_0$ are the binary representation of j, and bits $p_{2\mu-2}\cdots p_{\mu-1}$ are the binary representation of i. The input matrix S is distributed such that PE (i, j) has $S(i, j)$ and $S(i, j + M/2)$. Thus, each row of PEs can perform the one-dimensional FFT on its row of S with $N = M/2$. In this case, the cube functions required for data transfers will exchange data based on the lower order $\mu - 1$ bits of the physical address; i.e., the functions will act on j independently of i. Thus, the one-dimensional FFT can be performed on each row independently and simultaneously. The result G is distributed to each column of PEs which holds two columns of G, with each PE holding one element from each of the two columns of G.

The PEs are now logically reconfigured to form M columns of $M/2$ PEs, with each column of PEs having one column of G. Two matrix elements are in each PE.

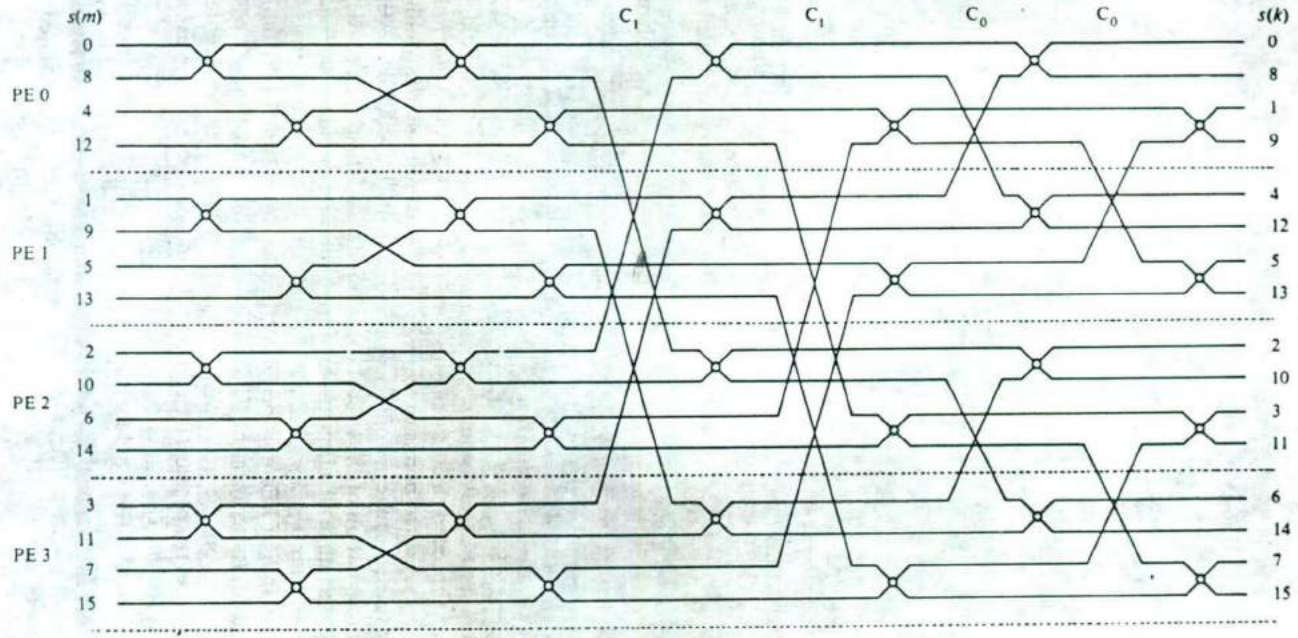


Figure 5.31 Computation of a 16-point FFT in an SIMD machine with 4 PEs. (Courtesy of IEEE *Proc. 5th Int'l Conf. on PRIP*, Mueller, et al. 1980.)

Table 5.5 SIMD machine reconfiguration for two-dimensional FFT computations [notation (i, j) denotes the PE_{ij}]

Configuration: M-by-$M/2$ array			
$(0, 0)$	$(0, 1)$	$\cdots$	$(0, M/2 - 1)$
$(1, 0)$	$(1, 1)$	$\cdots$	$(1, M/2 - 1)$
$\vdots$			$\vdots$
$(M - 1, 0)$	$(M - 1, 1)$	$\cdots$	$(M - 1, M/2 - 1)$

Configuration: $M/2$-by-M array						
$(0, 0)$	$(M/2, 0)$	$(0, 1)$	$(M/2, 1)$	$\cdots$	$(0, M/2 - 1)$	$(M/2, M/2 - 1)$
$(1, 0)$	$(M/2 + 1, 0)$	$(1, 1)$	$(M/2 + 1, 1)$	$\cdots$		
$\vdots$						$\vdots$
$(M/2 - 2, 0)$	$(M - 1, 0)$	$(M/2 - 1, 1)$	$(M - 1, 1)$	$\cdots$	$(M/2 - 1, M/2 - 1)$	$(M - 1, M/2 - 1)$

To do this, PE (i, j) is renumbered (k, l) where $k = i \bmod (M/2)$ and $l = 2j + [i/(M/2)]$. Effectively, this renumbering takes each column of the original configuration, divides it in half, and aligns the halves to form two columns, as shown in Table 5.5. In terms of the binary representation of the physical address, the binary representation of k is $p_{2\mu-3} \cdots p_{\mu-1}$, and the binary representation of l is $p_{\mu-2} \cdots p_0 p_{2\mu-2}$.

After the renumbering, G is distributed to each pair of PE columns PE$(k, 2\lambda)$ and PE$(k, 2\lambda + 1)$, where $0 \le k, \lambda < M/2$. Within such a pair of PE columns, PE$(k, 2\lambda)$ has two points from the kth row of G, and PE$(k, 2\lambda + 1)$ has two corresponding points from the $(k + M/2)$th row of G, for $0 \le k < M/2$. Within each of the two columns of G, these are precisely the points which must be paired at the start of the one-dimensional M-point, $M/2$ PE algorithm. Using the $C_{2\mu-2}$ function, PE$(k, 2\lambda)$ and PE$(k, 2\lambda + 1)$, where $0 \le k, \lambda < M/2$, can exchange data so that each column of PEs gets a different column of G with each PE holding the two elements of G needed to start the one-dimensional FFT. In terms of physical PE addresses, $p_{2\mu-2}$ corresponds to the high order position and the routing function $C_{2\mu-2}$ equals C_{n-1}. In terms of logical numbering, $p_{2\mu-2}$ is the low order bit in the binary representation of the logical column index, so the $C_{2\mu-2}$ routing function effects the exchange of elements between the corresponding rows in columns 2λ and $2\lambda + 1$, for $0 \le \lambda < M/2$.

Each column of PEs now performs the one-dimensional FFT on its column of G. However, to perform the FFT on a column, it is necessary to perform data transfers based on the row index, which is given by k and represented by $p_{2\mu-3} \cdots p_{\mu-1}$ of the physical PE address. Therefore, whenever C_i is executed in the original algorithm, $C_{i+\mu-1}$ is executed in this algorithm. In this way, the cube routing functions allow communication within a column instead of within a row.

The complexity of this SIMD algorithm has been derived from the one-dimensional FFT case. The number of multiplication steps required is $2\log_2 M$; the number of addition steps required is $2(2\log_2 M)$; and the number of data transfer steps required is $2\log_2(M/2) + 1 = 2\log_2 M - 1$. The complexity of a serial FFT algorithm is $M^2 \log_2 M$ multiplications and $2M^2 \log_2 M$ additions. To sum

up, the speedup of a parallel two-dimensional FFT algorithm is $M^2/2$ over a serial FFT algorithm. Without surprise, this speedup equals the number of PEs in the SIMD array processor.

5.3.4 Connection Issues for SIMD Processing

SIMD array processors allow explicit expression of parallelism in user programs. The compiler detects the parallelism and generates object code suitable for execution in the multiple processing elements and the control unit. Program segments which cannot be converted into parallel executable forms are executed in the control unit; program segments which can be converted into parallel executable forms are sent to the PEs and executed synchronously on data fetched from parallel memory modules under the control of the control unit. To enable synchronous manipulation in the PEs, the data is permuted and arranged in vector form. Thus, to run a program more efficiently on an array processor, one must develop a technique for vectorizing the program codes. The interconnection network plays a major role in vectorization. Several connection issues in using SIMD interconnection networks are addressed below.

Permutation and connectivity In array processing, data is often stored in parallel memory modules in skewed forms that allow a vector of data to be fetched without conflict. However, the fetched data must be realigned in a prescribed order before it can be sent to individual PEs for processing. This alignment is implemented by the routing functions of the interconnection network, which also realigns the data generated by individual PEs into skewed form for storage in the memory modules.

A rearrangeable network and the nonblocking network can realize every permutation function, but using these networks for alignment requires considerable effort to calculate control settings. A recursive routing mechanism has been suggested for a few families of permutations needed for parallel processing; however, the problem remains for the realization of general permutations. Many attempts have been made on the permutation capabilities of single-stage networks and blocking multistage networks. These networks cannot realize arbitrary permutations in a single pass. Recent results show that the baseline network can realize arbitrary permutations in just two passes while other blocking multistage networks, such as the Omega network, need at least three passes. Wu and Feng (1981) have proved that the shuffle-exchange network can realize arbitrary permutations in $3(\log_2 N) - 1$ passes where N is the network size.

Partitioning and reconfigurability A configuration concept has been proposed to better use the interconnection network. Under this concept, a network is just a configuration in the same topologically equivalent class. To configure a permutation function as an interconnection network, we can assign input-output link names in a way that realizes the permutation function in one conflict-free pass. Assigning logical names that realize various permutation functions without conflicts is called a reconfiguration problem. Through the reconfiguration process, the baseline network can realize every permutation in one pass without conflicts.

This means that concurrent processing throughout could be enhanced by the proper assignment of tasks to processing elements and data to memory modules.

When dividing an SIMD interconnection network into independent subnetworks of different sizes, each subnetwork must have all the interconnection capabilities of a complete network of the same type and size. Hence, with a partitionable network, a system can support multiple SIMD machines. By dynamically reconfiguring the system into independent SIMD machines and properly assigning tasks to each partition, we can use resources more efficiently. Siegel (1980) has proved that single-stage networks such as the shuffle-exchange and Illiac networks cannot be partitioned into independent subnetworks, but blocking multistage networks such as a data manipulator can be partitioned.

Reliability and bandwidth The reliable operation of interconnection networks is important to overall system performance. The reliability issue can be thought of as two problems: *fault diagnosis* and *fault tolerance*. The fault-diagnosis problem has been studied for a class of multistage interconnection networks constructed of switching elements with two valid states. The problem is approached by generating suitable fault-detection and fault-location test sets for every fault in the assumed fault model. The test sets are then trimmed to a minimal or nearly minimal set. The second reliability problem concerns mainly the degree of fault tolerance. It is important to design a network that combines full connection capability with graceful degradation in spite of the existence of faults.

A high network bandwidth is often desired at reasonably low network cost. The *network bandwidth* is defined as the number of PE requests honored per unit of time. Several analytical methods have been used to estimate network bandwidth. We shall treat some of them in Section 7.2.4. Most analytical models suggested by researchers are too simplified and closed-form solutions are not attainable. Numerical experiments can simulate actual PE-connection requests by tracing the program to be executed. Continued research efforts are needed in this area to accurately estimate the bandwidth of various interconnection networks.

The cost of a network is primarily determined by the switching complexity. To achieve a cost-effective network design, one must find the optimal trade-off between performance (bandwidth) and cost (complexity). The main difficulty lies in the fact that bandwidth analysis depends on unpredictable program behavior and cost varies rapidly with progress in technology. Cost effectiveness must also insure flexibility in application programming and reliability in achieving fault tolerance. These are wide open areas of further research on interconnection networks for both SIMD and MIMD computers.

5.4 ASSOCIATIVE ARRAY PROCESSING

Two SIMD computers, the Goodyear Aerospace STARAN and the Parallel Element Processing Ensemble (PEPE), have been built around an *associative memory* (AM) instead of using the conventional *random-access memory* (RAM). The fundamental distinction between AM and RAM is that AM is content-

addressable, allowing parallel access of multiple memory words, whereas the RAM must be sequentially accessed by specifying the word addresses. The inherent parallelism in associative memory has a great impact on the architecture of *associative processors*, a special class of SIMD array processors which update with the associative memories.

In this section, we describe the functional organization of an associative array processor and various parallel processing functions that can be performed on an associative processor. We classify associative processors based on associative-memory organizations. Finally, we identify the major searching applications of associative memories and associative processors. Associative processors have been built only as special-purpose computers for dedicated applications in the past.

5.4.1 Associative Memory Organizations

Data stored in an associative memory are addressed by their contents. In this sense, associative memories have been known as *content-addressable memory*, *parallel search memory*, and *multiaccess memory*. The major advantage of associative memory over RAM is its capability of performing parallel search and parallel comparison operations. These are frequently needed in many important applications, such as the storage and retrieval of rapidly changing databases, radar-signal tracking, image processing, computer vision, and artificial intelligence. The major shortcoming of associative memory is its much increased hardware cost. Presently, the cost of associative memory is much higher than that of RAMs.

The structure of a basic AM is modeled in Figure 5.32. The associative memory array consists of n words with m bits per word. Each bit cell in the $n \times m$ array consists of a flip-flop associated with some comparison logic gates for pattern match and read-write control. This logic-in-memory structure allows parallel read or parallel write in the memory array. A *bit slice* is a vertical column of bit cells of all the words at the same position. We denote the jth bit cell of the ith word as B_{ij} for $1 \le i \le n$ and $1 \le j \le m$. The ith word is denoted as:

$$W_i = (B_{i1}B_{i2}\cdots B_{im}) \qquad \text{for } i = 1, 2, \ldots, n$$

and the jth bit slice is denoted as:

$$B_j = (B_{1j}B_{2j}\cdots B_{nj}) \qquad \text{for } j = 1, 2, \ldots, m$$

Each bit cell B_{ij} can be written in, read out, or compared with an external interrogating signal. The parallel search operations involve both comparison and masking and are executed according to the organization of the associative memory. There are a number of registers and counters in the associative memory. The *comparand* register $C = (C_1, C_2, \ldots, C_m)$ is used to hold the key operand being searched for or being compared with. The *masking* register $M = (M_1, M_2, \ldots, M_m)$ is used to enable or disable the bit slices to be involved in the parallel comparison operations across all the words in the associative memory.

The *indicator* register $I = (I_1, I_2, \ldots, I_n)$ and one or more *temporary* registers $T = (T_1, T_2, \ldots, T_n)$ are used to hold the current and previous match patterns,

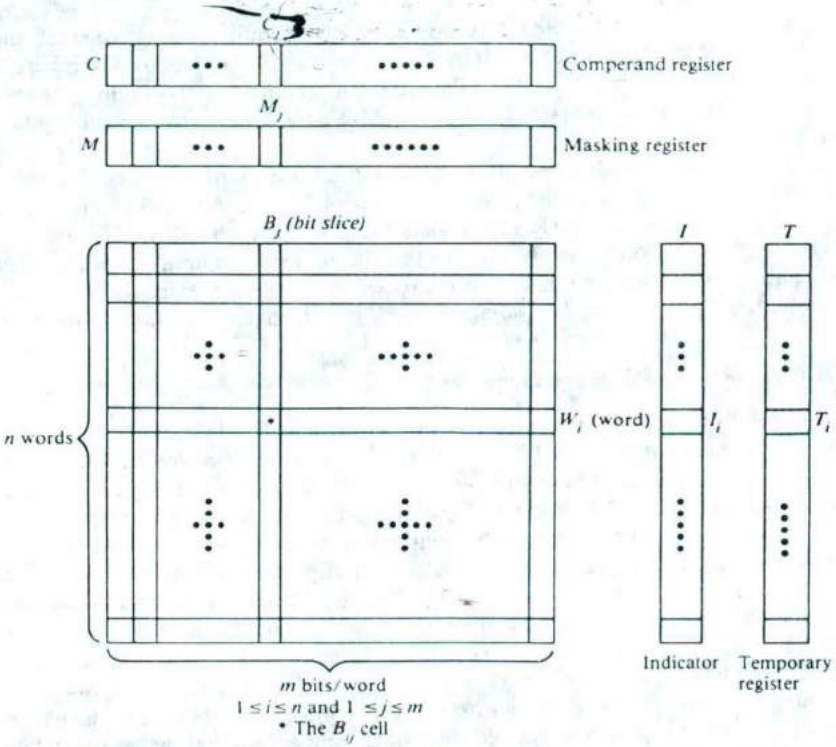


Figure 5.32 An associative memory array and working registers.

respectively. Each of these registers can be set, reset, or loaded from an external source with any desired binary patterns. The counters are used to keep track of the i and j index values. There are also some match detection circuits and priority logic, which are peripheral to the memory array and are used to perform some vector boolean operations among the bit slices and indicator patterns.

The search key in the C register is first masked by the bit pattern in the M register. This masking operation selects the effective fields of bit slices to be involved. Parallel comparisons of the masked key word with all words in the associative memory are performed by sending the proper interrogating signals to all the bit slices involved. All the involved bit slices are compared in parallel or in a sequential order, depending on the associative memory organization. It is possible that multiple words in the associative memory will match the search pattern. Therefore, the associative memory may be required to tag all the matched words. The indicator and temporary registers are mainly used for this purpose. The interrogation mechanism, read and write drives, and matching logic within a typical bit cell are depicted in Figure 5.33. The interrogating signals are associated with each bit slice, and the read-write drives are associated with each word. There are

Interrogation information	Mask*	Information stored	
		0	1
0	0	0	0
1	0	0	0
0	1	0	1
1	1	1	0

*Mask = 0 means that no comparison is performed at that bit position for all words.

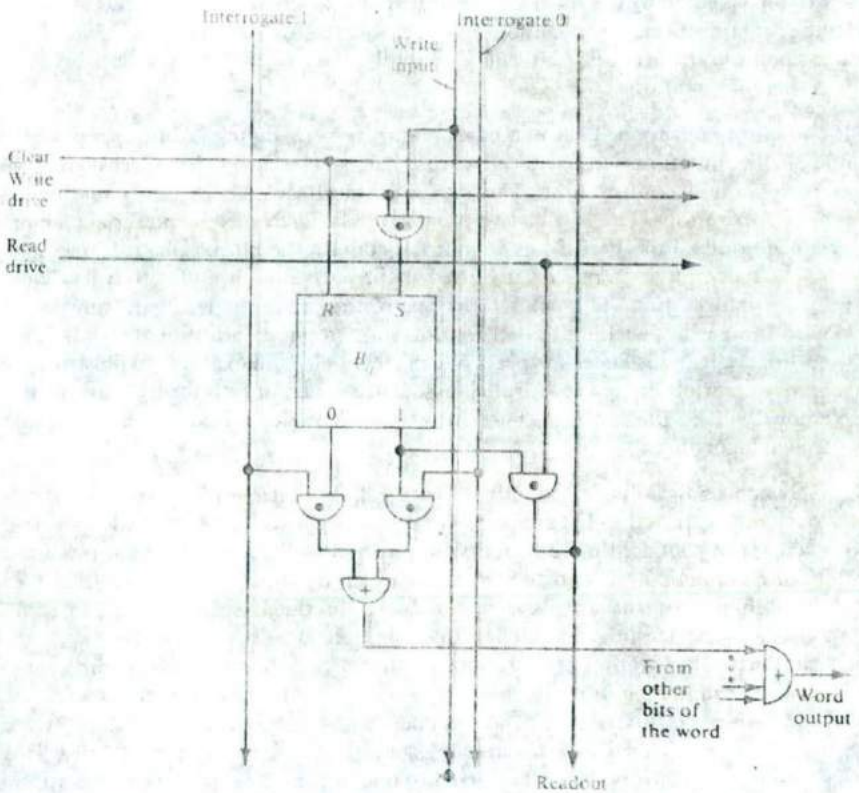


Figure 5.33 The schematic logic design of a typical cell in an associative memory.

two types of comparison readouts: the bit-cell readout and the word readout. The two types of readout are needed in two different associative memory organizations.

In practice, most associative memories have the capability of *word parallel* operations; that is, all words in the associative memory array are involved in the parallel search operations. This differs drastically from the *word serial* operations encountered in RAMs. Based on how bit slices are involved in the operation, we consider below two different associative memory organizations:

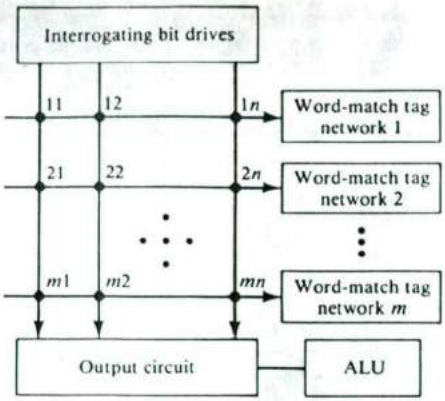
The bit parallel organization In a bit parallel organization, the comparison process is performed in a parallel-by-word and parallel-by-bit fashion. All bit slices which are not masked off by the masking pattern are involved in the comparison process. In this organization, word-match tags for all words are used (Figure 5.34*a*). Each cross point in the array is a bit cell. Essentially, the entire array of cells is involved in a search operation.

Bit serial organization The memory organization in Figure 5.34*b* operates with one bit slice at a time across all the words. The particular bit slice is selected by an extra logic and control unit. The bit-cell readouts will be used in subsequent bit-slice operations. The associative processor STARAN has the bit serial memory organization and the PEPE has been installed with the bit parallel organization.

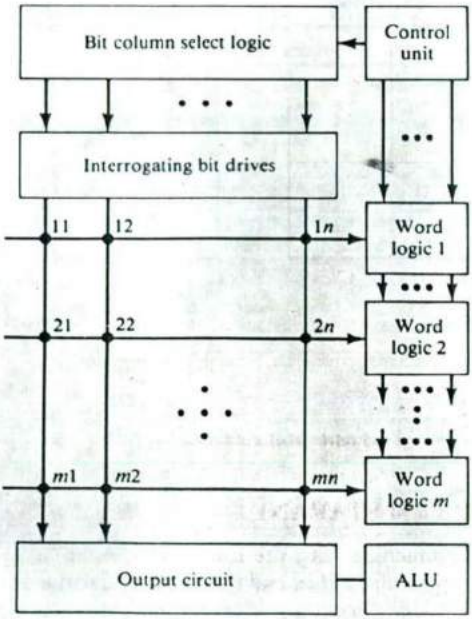
The associative memories are used mainly for search and retrieval of nonnumeric information. The bit serial organization requires less hardware but is slower in speed. The bit parallel organization requires additional word-match detection logic but is faster in speed. We present below an example to illustrate the search operation in a bit parallel associative memory. Bit serial associative memory will be presented in Section 5.4.3 with various associative search and retrieval algorithms.

Example 5.8 Consider a student-file search in a bit parallel associative memory, as illustrated in Figure 5.35. The query needs to search all students whose age is not younger than 21 but is younger than 31. This requires performing the *not-less-than* search and the *less-than* search on the age field of the file. Two matching patterns are used in the two subsequent searches. The masking pattern selects the age field. The lower-bound 21 is loaded into the C register as the first key word. Parallel comparisons are performed on all student records (words) in the file. Initially, the indicator register is cleared to be zero.

After the first search, those students who are not younger than 21 are marked with a *1* in the indicator register, one bit per each student word. This matching vector is then transferred to one of the T registers. Then the upper-bound 31 is loaded into C as the second matching key. After the second search, a new matching vector is sent to the I register. A bitwise ANDing operation is then performed between the I and T registers with the resulting vector residing in the I register as the final output of the search process. The whole search process requires only two accesses of the associative memory. An output circuit (shown in Figure 5.34) is used to control the reading out of the result.



(*a*) Bit-parallel organization



(*b*) Bit-serial organization

Figure 5.34 Associative memory organizations.

Query: Search for those students whose ages are in the range (21, 31)

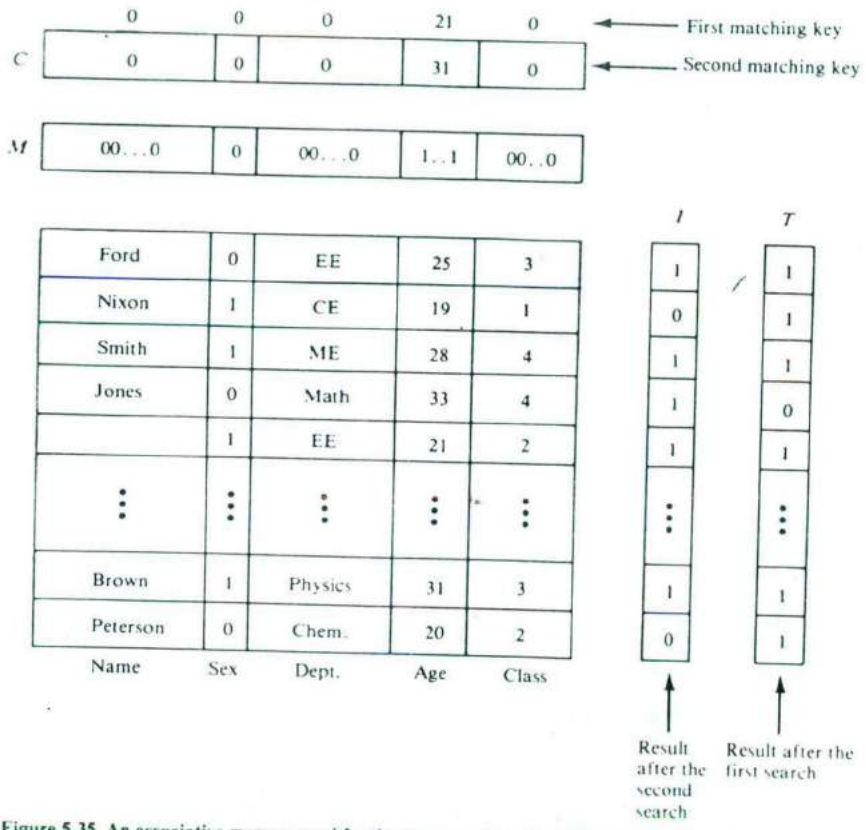


Figure 5.35 An associative memory used for the storage and retrieval of a student file.

5.4.2 Associative Processors (PEPE and STARAN)

An associative processor is an SIMD machine with the following special capabilities: (1) Stored data items are content-addressable, and (2) arithmetic and logic operations are performed over many sets of arguments in a single instruction. Because of these content-addressable and parallel processing capabilities, associative processors form a special subclass of SIMD computers. Associative processors are effective in many special application areas. We classify associative processors into two major classes, the *fully parallel* versus the *bit serial organizations*, depending on the associative memory used. Two associative processors are functionally described below along with their potential applications.

The PEPE architecture There are two types of fully parallel associative processors: *word-organized* and *distributed logic*. In a word-organized associative processor, the comparison logic is associated with each bit cell of every word and the logical decision is available at the output of every word. In a distributed-logic associative processor, the comparison logic is associated with each character cell of a fixed number of bits or with a group of character cells. The most well-known example of a distributed-logic associative processor is the PEPE. Because of the requirement of additional logic-per-cell, a fully parallel associative processor may be cost prohibitive. A distributed-logic associative processor is less complex and thus less expensive. The PEPE is based on a distributed-logic configuration developed at Bell Laboratories for radar signal-processing applications.

A schematic block diagram of PEPE is given in Figure 5.36. PEPE is composed of the following functional subsystems: an output data control, an element memory control, an arithmetic control unit, a correlation control unit, an associative output control unit, a control system, and a number of processing elements. Each processing element (PE) consists of an arithmetic unit, a correlation unit, an associative output unit, and a 1024 × 32-bit element memory. There are 288 PEs organized into eight element bays. Selected portions of the work load are loaded from a host computer CDC-7600 to the PEs. The loading selection process is determined by the inherent parallelism of the task and by the ability of PEPE's unique architecture to manipulate the task more efficiently than the host computer. Each processing element is delegated the responsibility of an object under observation by the radar system, and each processing element maintains a data file for specific objects within its memory and uses its associative arithmetic capability to continually update its respective file.

PEPE represents a typical special-purpose computer. It was designed to perform real-time radar tracking in the antiballistic missile (ABM) environment. No commercial model was made available. It is an attached array processor to the general-purpose machine CDC 7600, as demonstrated in Figure 5.36.

The bit-serial STARAN organization The full parallel structure requires expensive logic in each memory cell and complicated communications among the cells. The bit serial associative processor is much less expensive than the fully parallel structure because only one bit slice is involved in the parallel comparison at a time. Bit serial associative memory (Figure 5.34*b*) is used. This bit serial associative processing has been realized in the computer STARAN (Figure 5.37). STARAN consists of up to 32 associative array modules. Each associative array module contains a 256-word 256-bit *multidimensional access* (MDA) memory, 256 processing elements, a flip network, and a selector, as shown in Figure 5.38*a*. Each processing element operates serially bit by bit on the data in all MDA memory words. The operational concept of a STARAN associative array module is illustrated in Figure 5.38*b*.

Using the flip network, the data stored in the MDA memory can be accessed through the I/O channels in bit slices, word slices, or a combination of the two. The flip network is used for data shifting or manipulation to enable parallel

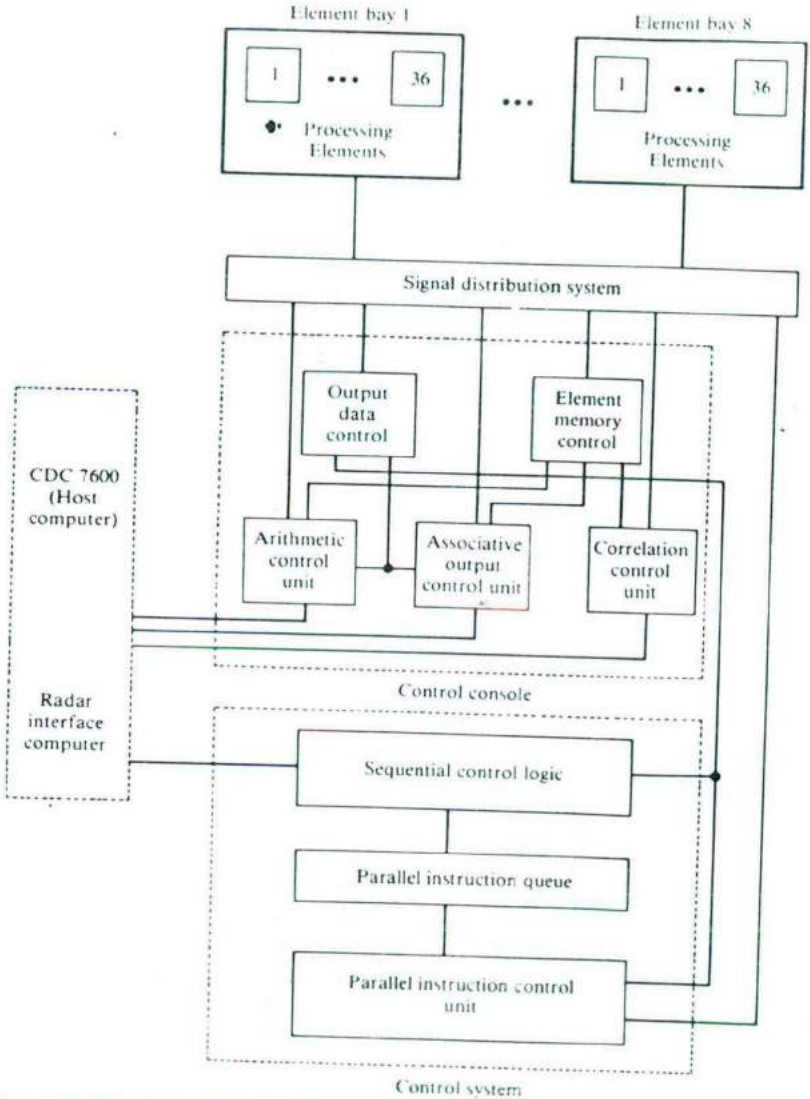


Figure 5.36 The Parallel Element Processing Ensemble (PEPE). (Courtesy of *Proc. of National Electronics Conference*, Berg et al. 1972.)

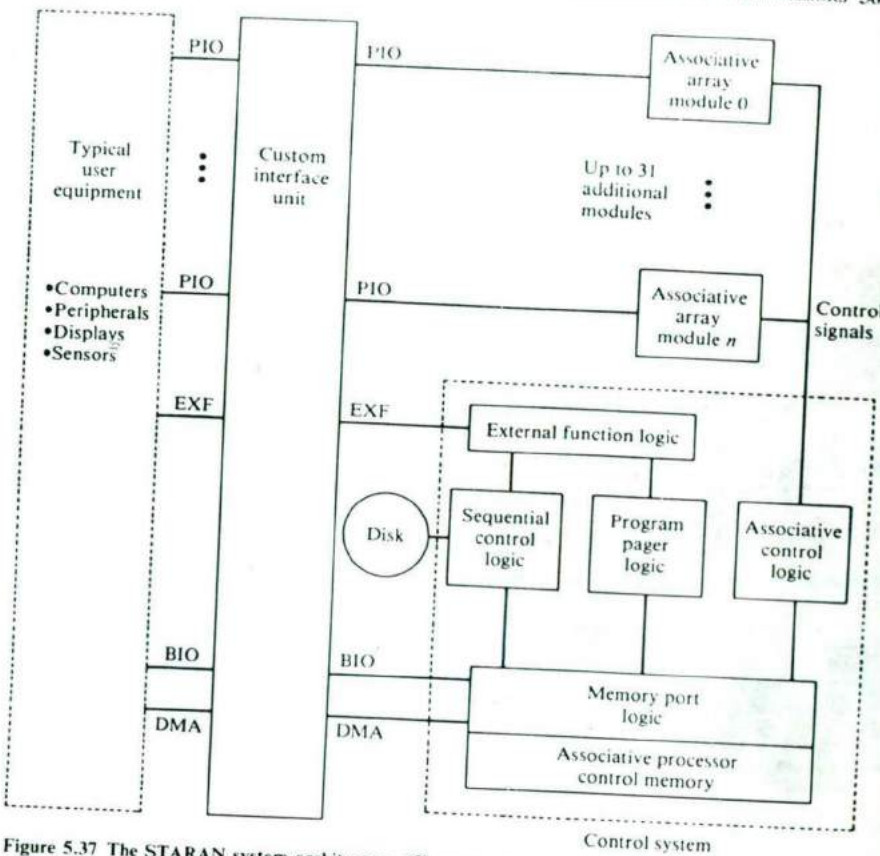
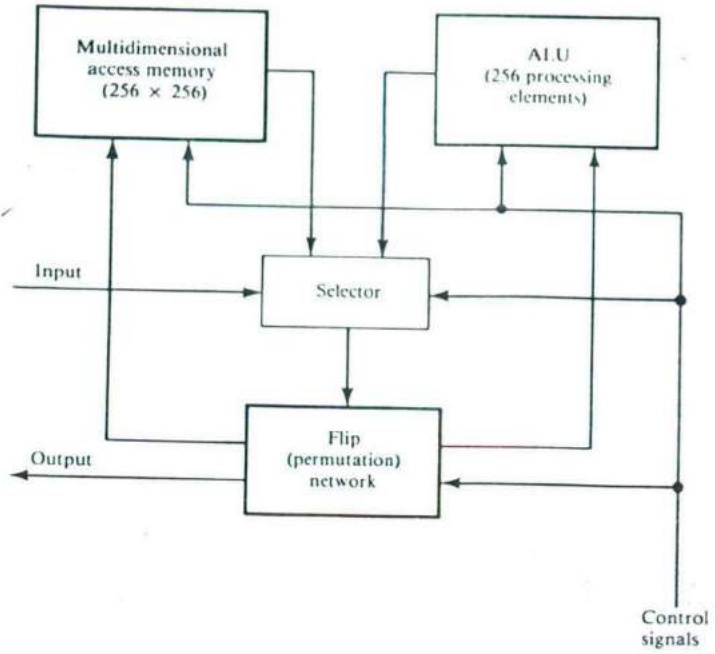


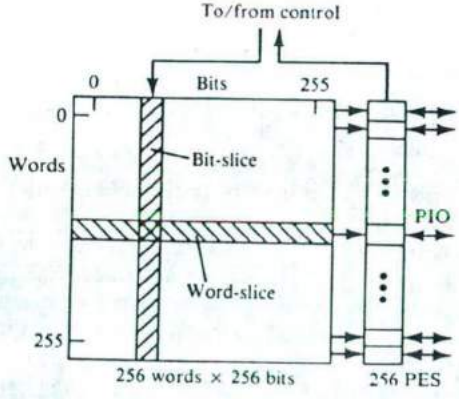
Figure 5.37 The STARAN system architecture. (Courtesy of *Proc. of National Computer Conference*, AFIPS Press, Batcher 1974.)

search, arithmetic or logical operations among words of the MDA memory. The MDA was implemented by Goodyear Aerospace using random-access memory chips with additional XOR logic circuits. The first STARAN was installed for digital image processing in 1975. Since then, Goodyear Aerospace has announced some enhanced STARAN models. The size of the MDA memory has increased to 9216 × 256 per module in the enhanced model with improved I/O and processing speed.

To locate a particular data item, STARAN initiates a search by calling for a match specified by the associative control logic. In one instruction execution, the data in all the selected memories of all the modules is processed simultaneously by the simple processing element at each word. The interface unit shown in Figure 5.37 involves interface with sensors, conventional computers, signal processors, interactive displays, and mass-storage devices. A variety of I/O options are



(*a*) STARAN associative array module



(*b*) Bit-slice versus word-slice operations in the STARAN

Figure 5.38 Associative memory and parallel operations in the STARAN.

implemented in the custom-interface unit, including the direct-memory access, buffered I/O channels, external function channels and parallel I/O.

Each associative array module can have up to 256 inputs and 256 outputs into the custom-interface unit. They can be used to increase the speed of inter-array data communication to allow STARAN to communicate with a high-bandwidth I/O device and to allow any device to communicate directly with the associative array modules. In many applications, such as matrix computation, air-traffic control, sensor-signal processing and data-management systems, a hybrid system composed of an associative processor and a conventional sequential processor can increase the throughput rate, simplify the software complexity, and reduce the hardware cost.

STARAN has high-speed input-output capabilities and the ability to interface easily with conventional computers. In such a hybrid system, each associative array module performs the tasks best suited to its capabilities. STARAN handles the parallel processing tasks and the conventional computer handles the tasks that must be processed in a single sequential data stream.

PEPE and STARAN are two large-scale associative processors that have been implemented for practical applications. The high cost-performance ratio of associative processors has limited them to be used mainly in military applications. Associative memories and processors have also been suggested for the design of text retrieval computers and back-end database machines. Several exploratory database machines are based on the use of associative memory, as reported in the book by Hsiao (1982).

5.4.3 Associative Search Algorithms

Associative memories are mainly used for the fast search and ordered retrieval of large files of records. Many researchers have suggested using associative memories for implementing relational database machines. Each relation of records can be arranged in a tabular form, as illustrated in Example 5.8. The tabulation of records (relations) can be programmed into the cells of an associative memory. Various associative search operations have been classified into the following categories by T. Y. Feng (1976).

Extreme searches

The *Maxima*: The largest among a set of records is searched.
The *Minima*: The smallest among a set of records is searched.
The *Median*: Search for the median according to a certain ordering.

Equivalence searches

Equal-to: Exact match is searched under a certain equality relation.
Not-equal-to: Search for those elements unequal to the given key.
Similar-to: Search for a match within the masked field.

Proximate-to: Search for those records that satisfy a certain proximity (neighborhood) condition.

Threshold searches

Smaller-than: Search for those records that are strictly smaller than the given key.
Greater-than: Search for those records that are strictly greater than the given key.
Not-smaller-than: Search for those records that are equal to or greater than the given key.
Not-greater-than: Search for those records that are equal to or smaller than the given key.

Adjacency searches

Nearest below: Search for the nearest record which is smaller than the key.

Between-limits searches

$[X, Y]$: Search for those records within the closed range $\{z | X \leq z \leq Y\}$.
(X, Y): Search for those records within the open range $\{z | X < z < Y\}$.
$[X, Y)$: Search for those records within the range $\{z | X \leq z < Y\}$.
$(X, Y]$: Search for those records within the range $\{z | X < z \leq Y\}$.

Ordered retrievals

Ascending sort: List all the records in ascending order.
Descending sort: List all the records in descending order.

Listed above are primitive search operations. Of course, one can always combine a sequence of primitive search operations by some boolean operators to form various query conjunctions. For examples, one may wish to answer the queries *equal-to-A* but *not-equal-to-B*; the *second largest from below*; *outside the range* $[X, Y]$; etc. Boolean operators AND, OR, and NOT can be used to form any query conjunction of predicates. A predicate consists of one of the above relational operators plus an attribute such as the pairs $\{\leq, A\}$ or $\{\neq, A\}$. The above search operators are frequently used in text retrieval operations.

Two examples are given below to show how to perform associative search operations. We consider the use of a bit serial associative memory (Figure 5.34) in which all the memory words can be accessed (read) in parallel, but where bit slices of all words or within a specified field of all words must be processed sequentially one slice after another from let to right in the AM array. The following notations are used to designate any specific *field* of a word:

- s: The starting bit address of a field, where $1 \leq s \leq n$.
- f: The field length in bits.

- k: The index within the field, where $1 \le k \le f$.
- i: The index for different bit slices, where $1 \le i \le n$.
- j: The index for successive words, where $1 \le j \le m$.

Furthermore, we use $\mathbf{1} = (1, 1, \ldots, 1)$ and $\mathbf{0} = (0, 0, \ldots, 0)$ to denote the binary vectors of all 1s or of all 0s, respectively. The indicator register I is formed with S-R flip-flops I_i for $i = 1, 2, \ldots, n$. The reset and set signals of flip-flop I_i are denoted as R_i and S_i, respectively. In the specification of the following algorithms, we use $I_i(k)$ to refer to the contents of flip-flop I_i at the kth step. The same convention applies to $T_i(k)$, $R_i(k)$, and $S_i(k)$. The initial contents of the working registers are indicated by $I(0)$ and $T(0)$.

Example 5.9: The MINIMA search This algorithm searches for the smallest number among a set of n positive numbers stored in a bit serial AM array. Each number has f bits stored in a field of a word from the bit position s to the bit position $s + f - 1$:

1. Initialize

$$C \leftarrow I;\ I(0) \leftarrow \mathbf{1};\ T(0) \leftarrow \mathbf{0};\ k = 1,\ j = s + k - 1,\ M \leftarrow (0 \ldots \underbrace{11 \ldots 1}_{f \text{ bits of 1s}} 0 \ldots 0).$$

2. Load $T_i(k) = I_i(k-1) \cap (C_j \oplus B_{ij})$ for all $i = 1, 2, \ldots, n$.

3. Detect $Q(k) = \bigcup_{i=1}^{n} T_i(k)$.

4. Reset T by applying $r_i(k) = \overline{T_i(k)} \cap Q(k)$ for all $i = 1, 2, \ldots, n$.
5. Increment $k \leftarrow k + 1$. Then proceed to step 2 if $k \le f - 1$, or read out the work W_i indicated by $I_i(f) = 1$ if $k = f$.

Example 5.10: The NOT-SMALLER-THAN search This algorithm searches for those numbers that are greater than or equal to a given number N. Assume the same field format as in Example 5.9.

1. Initialize

$$C \leftarrow N;\ I(0) \leftarrow \mathbf{1};\ T(0) \leftarrow \mathbf{1},\ k = 1,\ j = s + k - 1,\ M \leftarrow 0 \ldots \underbrace{011 \ldots 1}_{f \text{ bits of 1s}} 0 \ldots 0.$$

2. If $C_j = 0$, then load $T_i(k)$ with $T_i(k-1) \cap (C_j \oplus B_{ij})$; else modify I by applying $R_i(k) = T_i(k) \cap (C_j \oplus B_{ij})$ for all $i = 1, 2, \ldots, n$.
3. Increment $k \leftarrow k + 1$. Then test if $k = f$. If no, proceed to step 2. If yes, read out those qualified numbers indicated by $I_i(f) = 1$.

In steps 2 and 4 of Example 5.9 and in step 2 of Example 5.10, all n words are involved in the specified operations. The bit-slice operations are governed by the increment of index k in the loop. When a bit parallel associative memory is used,

the above algorithms have to be modified. The bit parallel associative memories were only used in nonnumeric text retrieval operations. The bit serial associative memories were mostly used to perform associative numeric computations.

5.5 BIBLIOGRAPHIC NOTES AND PROBLEMS

General discussions of array processors can be found in the texts by Thurber (1976), Kuck (1978), and Stone (1980). Interconnection networks for SIMD networks have been surveyed in Feng (1981) in a comparative manner. The SIMD computer model was proposed by Siegel (1979). A comprehensive treatment of interconnection networks can be found in the book by Siegel (1984). Multistage cube-type networks and their capabilities are discussed in Batcher (1976), Pease (1977), and Wu and Feng (1980). The barrel shifter and data manipulator are based on the work of Feng (1974) and Bauer (1974). The shuffle-exchange networks are discussed in Stone (1971) and Lang and Stone (1976). The Omega network was introduced by Lawrie (1975). Wu and Feng (1981) have studied the universality of shuffle-exchange networks. Network partitioning has been studied by Siegel (1980). Network reliability and fault tolerance were studied in Wu and Feng (1979) and in Shen and Hayes (1980).

A comparative study of large-scale array processors was given in Paul (1978), Thurber (1979), and Hwang et al. (1981). Associative search algorithms are based on the unpublished notes by Feng (1977). Stone (1980) has described the $O(n^2)$ algorithm for matrix multiplication. The parallel sorting method for mesh-connected array processors is developed by Thompson and Kung (1977). The FFT algorithms on SIMD computer are based on the work of Mueller, et al. (1980). The $O(n \log_2 n)$ algorithm for matrix multiplication on a cube network is based on the work of Thomas (1981).

Associative memories and associative processors have been surveyed in Yau and Fung (1977). There are also two published books, Thurber (1976) and Foster (1976), which are devoted to associative processors. More detailed information of PEPE can be found in Vick and Merwin (1973). The STARAN system architecture and applications are discussed in Batcher (1974) and Rohrbacher and Potter (1977). Additional SIMD computer algorithms can be found in Stone (1980), Kuck (1977), Thurber and Masson (1979), and Hockney and Jesshope (1981).

Problems

5.1 Explain the following terminologies associated with SIMD computers:

(*a*) Lock-step operations
(*b*) Masking of processing elements
(*c*) Routing functions for Illiac network
(*d*) Recirculating networks
(*e*) Cube-routing functions
(*f*) Barrel-shifting functions
(*g*) Shuffle-exchange functions
(*h*) Associative memory
(*i*) Bit serial associative processor
(*j*) Adjacency search

5.2 You are asked to design a data routing network for an SIMD array processor with 256 PEs. Barrel cyclic shifters are used so that a route from one PE to another requires only one unit of time per integer-power-of-two shift in either direction.

(*a*) Draw the interconnection barrel shifting network, showing all directly wired connections among the 266 PEs. In the drawing, at least one node (PE) must show all its connections to other PEs.

(*b*) Calculate the minimum number of routing steps from any PE_i to any other PE_{i+k} for the arbitrary distance $1 \leq k \leq 255$. Indicate also the upper bound on the minimum routing steps required.

5.3 Consider 64 PEs (PE_0 to PE_{63}) in the Illiac-IV. Determine the minimum number of data-routing steps needed to perform the following inter-PE data transfers: PE_i to $PE_{(i+k) \bmod 64}$ where $0 \leq i \leq 63$ and $0 \leq k \leq 63$.

5.4 Consider the use of a four-PE array processor to multiply two 3×3 matrices. The interconnection structure of the four PEs is shown in Figure 5.39. Wraparound connections appear in all rows and columns of the array. You need to map the matrix elements initially one to each of the processors. All the 3 multiplications needed for each output element c_{ij} must be performed in the same PE in order to accumulate the sum of products. Of course, you are allowed to shift the matrix elements around if needed.

(*a*) Show the initial mapping of the *A* and *B* matrix elements to the processors before the first multiplication is carried out. (You may have to wrap around the matrix.)

(*b*) What are the initial multiplications to be carried out in each processor (there may be more than one multiplication in each processor) without any data shifting?

(*c*) Parallel shifts are carried out in the horizontal and vertical directions. Show the mapping of the *A* and *B* matrix elements to the processors before the second group of multiplications can be carried out.

(*d*) What are the multiplications to be carried out in each processor without any further shifting? (Don't bother with summing with the previous terms. Summation operations in dot product operations are embedded in the multiply hardware automatically.)

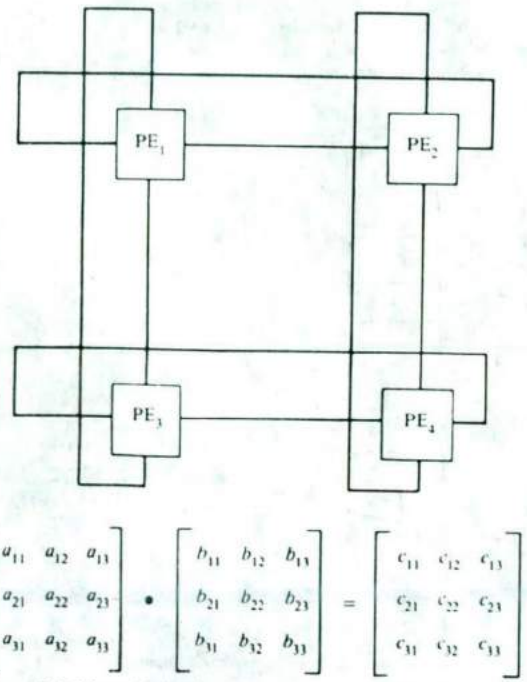


Figure 5.39 The multiplication of two 3×3 matrices on a mesh of 4 PEs in Problem 5.4.

(*e*) Suppose the two matrices have already been allocated as in part (*a*). Assume each processor can perform one multiplication per unit time or one shift in a single direction per unit time for each number. (If you shift two numbers, it takes two units of time.) Determine the time units needed to complete parts (*b*), (*c*), and (*d*), respectively. Minimizing the total time delay is the design goal.

5.5 Let A be a $2^m \times 2^m$ matrix stored in row-major order in the main memory. Prove the transposed matrix A^T can be obtained by performing m perfect shuffles on A.

5.6 Given an $n \times n$ matrix $A = (a_{ij})$, we want to find the n column sums:

$$S_j = \sum_{i=0}^{n-1} a_{ij} \qquad \text{for } j = 1, 2, \ldots, n-1$$

using an SIMD machine with n PEs. The matrix is stored in a skewed format, as shown in Figure 5.40. The jth column sum S_j is stored in location β in PEM_j at the end of the computation. Use the machine organization as shown in Figures 5.1*a* and 5.2. Write an SIMD algorithm and indicate the successive memory contents in the execution of the algorithm.

5.7 Consider the use of the associative memory array in Figure 5.32 for implementing a Not-Equal-To search. Assume bit-slice parallel-word operations similar to those described in Examples 5.9 and 5.10. Write out the detailed steps. Initial conditions in registers and the intermediate and final indicator pattern must be interpreted.

5.8 (*a*) Benes binary network is a type of multistage network which is *rearrangeable* and *nonblocking* because it can perform all possible connections between inputs and outputs by rearranging its existing connections so that a new path for a new input-output pair can always be established. Develop a routing algorithm to realize the following permutation in an 8×8 Benes network:

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 4 & 0 & 2 & 6 & 1 & 5 \end{pmatrix}$$

Control setting of the input and output switching elements is shown in Figure 5.41 from the first iteration of the algorithm. The algorithm to be developed is *recursive* in nature. It can be applied recursively to the two middle subnetworks, labelled *a* and *b* in the figure.

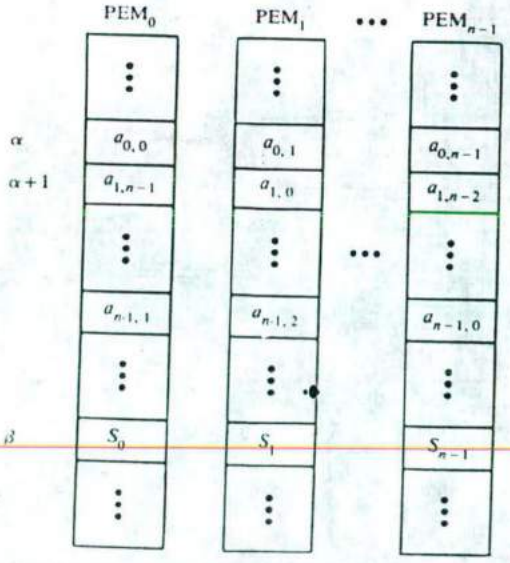


Figure 5.40 Memory allocation for the matrix computations in Problem 5.6.

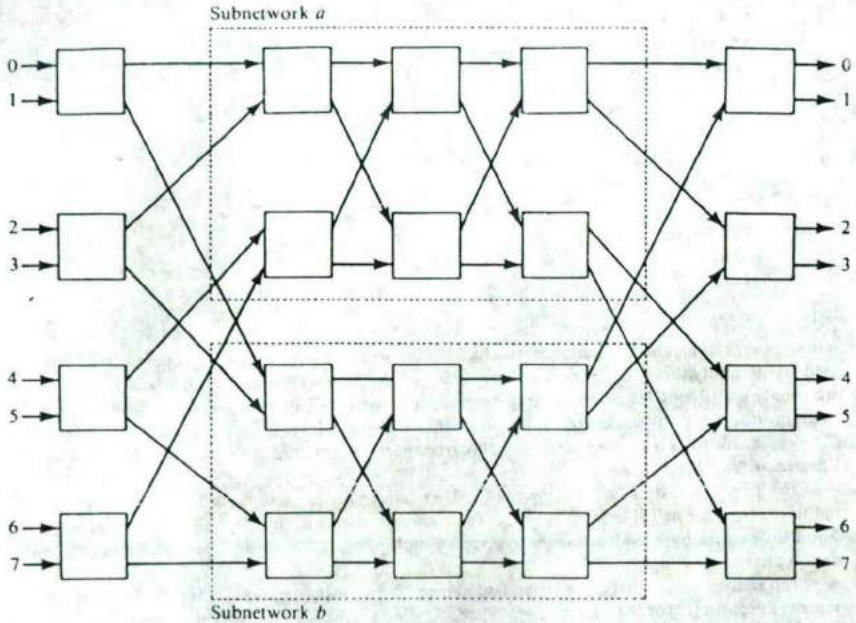


Figure 5.41 An 8 × 8 Benes network for Problem 5.8.

(*b*) Classify those multistage SIMD interconnection networks you have studied according to the three distinct features *blocking*, *rearrangeable*, and *nonblocking*.

5.9 Consider an N-input Omega network where each switch cell is individually controlled and $N = 2^n$.

(*a*) How many different permutation functions (one-to-one and onto mapping) can be defined over N inputs?

(*b*) How many different permutation functions can be performed by the Omega network in one pass? If $N = 8$, what is the percentage of permutation functions that can be performed in one pass?

(*c*) Given any source-destination ($S - D$) pair, the routing path can be uniquely controlled by the destination address. Instead of using the destination address (D) as the routing tag, we define $T = S \oplus D$ as the routing tag. Show that T alone can be used to determine the routing path. What is the advantage of using T as the routing tag?

(*d*) The Omega network is capable of performing broadcasting (one source and multiple destinations). If the number of destination PEs is a power of two, can you give a simple routing algorithm to achieve this capability?

5.10 How many steps are required to broadcast an information item from one PE to all other PEs in each of the following single-stage interconnection networks? ($N = 2^n$ PEs).

(*a*) A shuffle-exchange network. Each step could be either a shuffle or an exchange but not mixed.

(*b*) A cube network. The C_i routing is performed for each step i, $0 \le i \le n - 1$.

5.11 Prove or disprove that the Omega network can perform any shift permutation in one pass. The *shift permutation* is defined as follows: given $N = 2^n$ inputs, a shift permutation is either a circular left shift or a circular right shift of k positions, where $0 \le k < N$.

5.12 A polynomial, $p(x) = \sum_{i=0}^{N-1} a_i x^i$ can be evaluated in $2 \log_2 N$ steps in an SIMD computer with N PEs, where N is a power of 2. Assume that each PE can perform either an *add* or a *multiply* under masking control. A shuffle interconnection exists among the N PEs. Each PE has a data register and

Table 5.6 The shuffle-multiply sequence for SIMD polynomial evaluation in Problem 5.12.

Data	Mask	Data	Mask	Data	Mask	Data
a_0	0	a_0	0	a_0	0	a_0
a_1	1	a_1x	0	a_1x	0	a_1x
a_2	0	a_2	1	a_2x^2	0	a_2x^2
a_3	1	a_3x	1	a_3x^3	0	a_3x^3
a_4	0	a_4	0	a_4	1	a_4x^4
a_5	1	a_5x	0	a_5x	1	a_5x^5
a_6	0	a_6	1	a_6x^2	1	a_6x^6
x	1	x^2	1	x^4	1	x^8
	First iteration Multiply by x		Second iteration Multiply by x^2		Third iteration Multiply by x^4	

a mask flag. The machine is equipped with both *broadcasting* and *masking* capabilities (instructions). To evaluate the polynomial requires first to generate all the product terms: a_ix^i for $i = 0, 1, \ldots, N-1$, by a sequence of $\log_2 N$ shuffle-multiply operations as listed in Table 5.6, and then to generate the sum of all product terms by a sequence of $\log_2 N$ shuffle-add operations.

(*a*) Show the major components and interconnection structure of the desired SIMD machine for the size of $N = 8$.

(*b*) Figure out the exact sequence of SIMD machine instructions needed to carry out the shuffle-multiply sequence in Table 5.6. The *shuffle* instruction is used to generate the successive mask vectors. The PE operates by broadcasting the successive multipliers, x, x^2, and x^4, retrieved from the eighth data register.

(*c*) Before entering the shuffle-add sequence, the eighth data register should be reset to zero. Repeat question (*b*) for the summing sequence. At the end, the final sum can be retrieved from any one of the eight PE registers.

(*d*) Explain the advantages of using the shuffle interconnection network for the implementation of the polynomial evaluation algorithm, as compared with the use of the Illiac mesh network for the same purpose.

CHAPTER

SIX

SIMD COMPUTERS AND PERFORMANCE ENHANCEMENT

This chapter is devoted to array-structured SIMD computer systems. Three milestone array processors, Illiac-IV, BSP, and MPP, will be studied in detail. These systems represent two decades of development of array processors. The three systems differ not only in their hardware-software structural features but also in their programming and application requirements. The Illiac-IV uses local memories attached to the processing elements. The BSP has parallel memory modules shared by all arithmetic elements. The Illiac-IV uses the mesh network and the BSP uses the crossbar network. The MPP is a bit-slice array processor built with VLSI technology.

After studying these SIMD computers, we discuss general methods to enhance the performance of SIMD array processors. These include parallel memory allocation, language extensions for array processing, and improvement of the system throughput. Finally, multiple-SIMD computer organizations are presented for parallel vector processing in multiarray processors with shared resources.

6.1 THE SPACE OF SIMD COMPUTERS

In this section, we review major SIMD computers that have been constructed, designed, or proposed up to early 1983. We use the term *array processor* exclusively for SIMD computers using conventional (nonassociative) random-access memory and the term *associative processor* for SIMD computers using associative memory.

We divide the space of SIMD computers into five subspaces, based on word-slice and bit-slice processing and the number of control units used:

- Word-slice array processors
- Bit-slice array processors
- Word-slice associative processors
- Bit-slice associative processors
- Multiple-SIMD computers

Known SIMD computers in each class will be briefly surveyed below, among which only the Illiac-IV, the BSP, and the MPP will be described in detail. SIMD computer architectures and example systems are summarized in Table 6.1. Section numbers are identified within the parentheses for SIMD computers covered in this book. For those systems that are not covered, major reference books and articles are identified by the author's name and the year of publication.

6.1.1 Array and Associative Processors

In 1958, Unger conceived a computer structure for spatial problems. The Unger spatial computer is conceptually shown in Figure 6.1. A two-dimensional array of PEs is controlled by a common master. Unger's machine was proposed for pattern-recognition applications. The concept of lock-step SIMD operation was further consolidated in the Solomon computer proposed by Slotnick, et al., in 1962. The Solomon computer, though never built, motivated the development of the Illiac series and many later SIMD machines. In 1965, Senzig and Smith designed a *vector arithmetic multiprocessor* (VAMP) which consists of a linear array of PEs with shared memory modules and a shared arithmetic pipeline, as illustrated in Figure 6.2. Each PE is a virtual processor, having only a few working registers in it. This pipeline-array processor was designed to save hardware in vector processing.

The Illiac-IV evolved from several predecessors in the Illiac series. It was the first major array supercomputer developed in the late 1960s. We shall study the Illiac-IV system and its successor system, the BSP, in Section 6.2. Both systems are no longer operational. However, we wish to learn the design methodologies and the application experiences accumulated with the development of these two systems. In 1979, two proposals were made to extend the Illiac IV-BSP architecture to meet the future demand of gigaflop machines. The Phoenix project suggested a multiple-SIMD computer consisting of 16 Illiac-IV arrays with a total of 1024 PEs. Burroughs proposed an array architecture which upgrades the BSP to 512 PEs sharing 521 memory modules. This Burroughs computer proposal was for the Numerical Aerodynamic Simulation Facilities demanded by the NASA Ames Research Center in the United States.

Several bit-slice array processors have been developed in Europe and the United States. The latest CLIP-4 is a bit-slice array processor built with a cellular mesh of 96 × 96 PEs with eight neighbors per PE. The CLIP series is designed for bit-slice image-processing applications. The Distributed Array Processor (DAP)

Table 6.1 SIMD computer systems

Computer	Architecture[1]	Developer and references
Unger	wos array	Proposed by Unger (1958)
Solomon	wos array	Proposed by Slotnick (1962)
VAMP	wos array	Proposed by Senzig and Smith (1965)
ILLIAC	wos array[2]	Illiac-IV operational 1972 (Section 6.2)
BSP	wos array	Developed by Burroughs and suspended in 1979 (Section 6.2)
CLIP	bis array	Developed at University College, London, See Duff (1976)
DAP	bis array	Developed by ICL, England, section 3.3 in Hockney and Jesshope (1981)
MPP	bis array	Developed by Goodyear Aerospace (Section 6.3)
PEPE	wos ass	Developed by Burroughs Corp. and System Dev. Corp. (Section 5.4.2)
STARAN	bis ass	Developed by Goodyear Aerospace Corp. (Section 5.4.2)
OMEN	bis ass	Developed by Sanders Associates, chapter 7 in Thurber (1976)
RELACS	bis ass	Proposed for database machine in Berra and Oliver (1979)
MAP	wos MSIMD	Proposed by Nutt (1977) (Section 6.4.4)
PM[4]	wos MSIMD	Proposed by Briggs and Hwang et al. (1979) (Section 6.4.4)
Phoenix	wos MSIMD	Proposed by Feierbach and Stevenson (1979)
NASF	wos array	Proposed in Stevens (1979)

[1] wos (word slice), bis (bit slice), ass (associative), array (array processor).
[2] Original Illiac design had MSIMD with 4 CUs and 256 PEs.

was developed by International Computer Limited in England. The DAP can be constructed in groups of 16 PEs in various sizes, such as 32×32, 64×64, 128×128 and 256×256. The MPP is a 128×128 bit-slice array processor, to be described in Section 6.3. The MPP represents state-of-the-art construction of large-scale SIMD computers in the early 1980s. It will be used to process satellite images for NASA applications.

Four associative processors are listed in Table 6.1. The PEPE is the only word-slice associative processor that we know of. The rest are bit-slice array processors. In Thurber (1976), details of the PEPE, the STARAN, and the OMEN

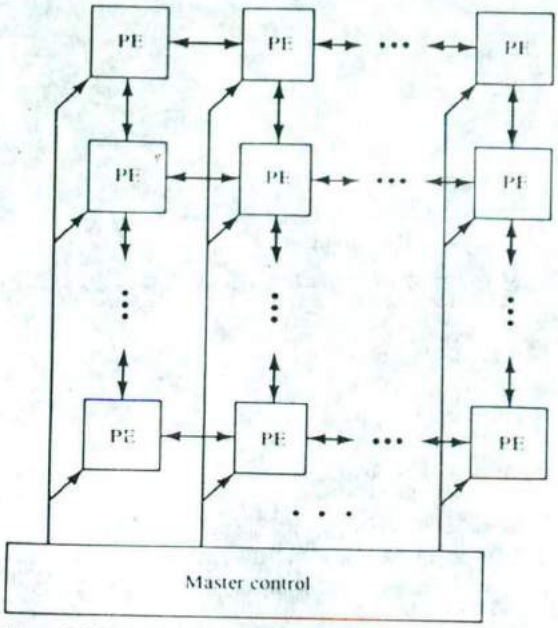


Figure 6.1 Unger's spatial computer concept. (Courtesy of *Proc. of IRE*, October 1958.)

are given. At present, most associative processors are designed to perform fast information retrieval and database operations. The RELACS is an associative database machine proposed by researchers at Syracuse University in 1979. It is based on using staged memory between the disks and the host processor. Associative memories are used to implement relational database operations. We have already studied the PEPE and the STARAN architecture in Section 5.4.2. The STARAN is the only commercial associative processor that has several installations in operation at present.

6.1.2 SIMD Computer Perspectives

It is quite clear that SIMD computers are special-purpose systems. For a specific problem environment, they may perform impressively. However, array processors have some programming and vectorization problems which are difficult to solve. The reality is that array processors are not popular among commercial computer manufacturers. The performances of several array processors are compared in Figure 6.3. Only the peak performance is indicated under ideal programming and resource allocation conditions. As the size of the PE-array increases, the performance should increase linearly. Of course, the peak speed is also a function of the word length especially for bit-slice operations. For vector processing, the performance depends also on the vector length.

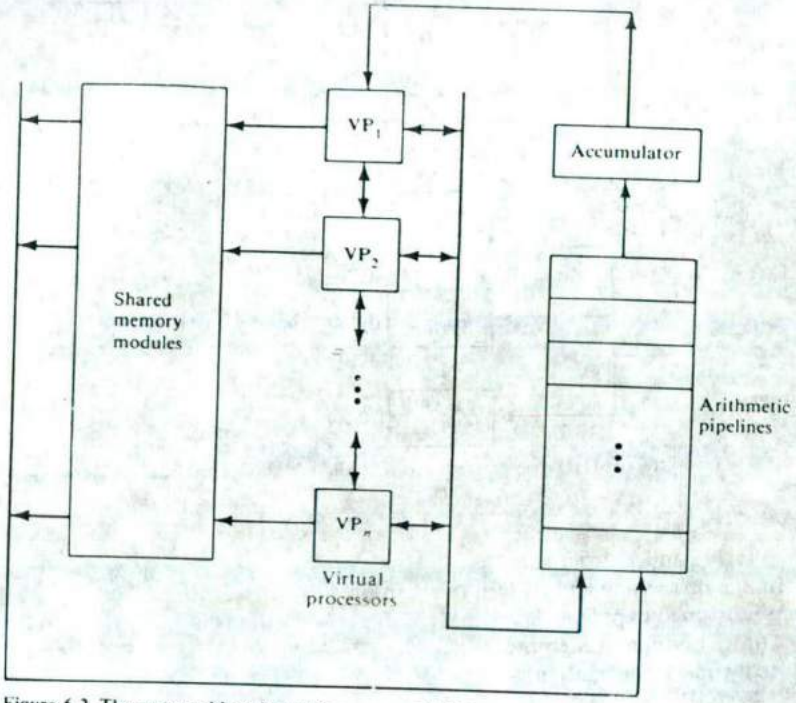


Figure 6.2 The vector arithmetic multiprocessor (VAMP) proposed by Senzig and Smith. (Courtesy of AFIPS *Proc. FJCC*, 1965.)

Multiple-SIMD (MSIMD) computers form a special subclass of MIMD computers. Multiple instruction streams exist in a multiple-array processor. Each instruction stream handles multiple data sets, as does an SIMD array. The Illiac-IV was originally proposed as an MSIMD machine. There are also other MSIMD computers, such as the Pheonix project and the PM^4 proposed in the literature. We shall briefly introduce these systems in Section 6.4.4. The MSIMD computers can offer higher application flexibility than can a single SIMD machine. So far, none of the proposed MSIMD machines has been built.

Listed below are some application areas that have been challenged or suggested for array processors and, in particular, for the Illiac-IV, the BSP, the MPP, and the STARAN systems:

- Matrix algebra (multiplication, decomposition, and inversion)
- Matrix eigenvalue calculations
- Linear and integer programming
- General circulation weather modeling
- Beam forming and convolution

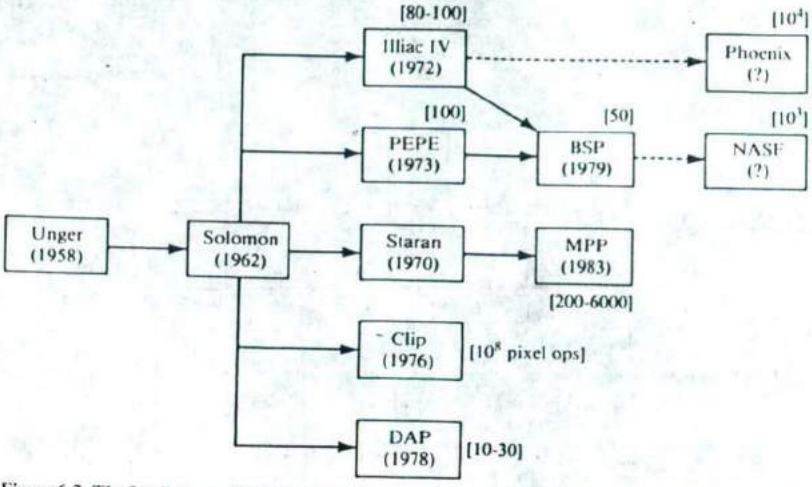


Figure 6.3 The family tree of SIMD array processors (numbers in brackets are estimated peak performance in mflops).

- Filtering and Fourier analysis
- Image processing and pattern recognition
- Wind-tuned experiments
- Automated map generation
- Real-time scene analysis

Some of these applications will be studied with the architectural descriptions in subsequent sections. The above listing is by no means exhaustive. Most of these applications need to process spatially distributed data.

6.2 THE ILLIAC-IV AND THE BSP SYSTEMS

Presented below are the system architecture, hardware and software features, and application requirements of the Illiac-IV and the BSP computer systems. The Illiac-IV system was developed at the University of Illinois in the 1960s. The system was fabricated by the Burroughs Corporation in 1972. The original objective was to develop a highly parallel computer with a large number of arithmetic units to perform vector or matrix computations at the rate of 10^9 operations per second. In order to achieve this rate, the system was to employ 256 PEs under the supervision of four CUs. Due to cost escalation and schedule delays, the system was ultimately limited to one quadrant with 64 PEs under the control of one CU. The speed of the 64-PE quadrant is approximately 200 million operations per second. The Illiac-IV computer has been applied in numerical weather forecasting and in nuclear engineering research, among many other scientific applications.

6.2.1 The Illiac-IV System Architecture

The 64 PEs in the Illiac computer are interconnected as a two-dimensional mesh network, shown in Figure 6.4. The PEs are numbered from 0 to 63. The data flow through the Illiac-IV array includes the CU bus for sending instructions or data in blocks of eight words from the PEMs to the CU. Data is represented in either 64- or 32 bit floating-point, 64-bit logical, 48- or 24-bit fixed point, or 8-bit character mode. By utilizing these data formats, the PEs can hold vectors of operands with 64, 128, or 512 components. The instructions to be executed are distributed throughout the PEMs. The operating system supervises the execution of instructions fetched from the PEMs.

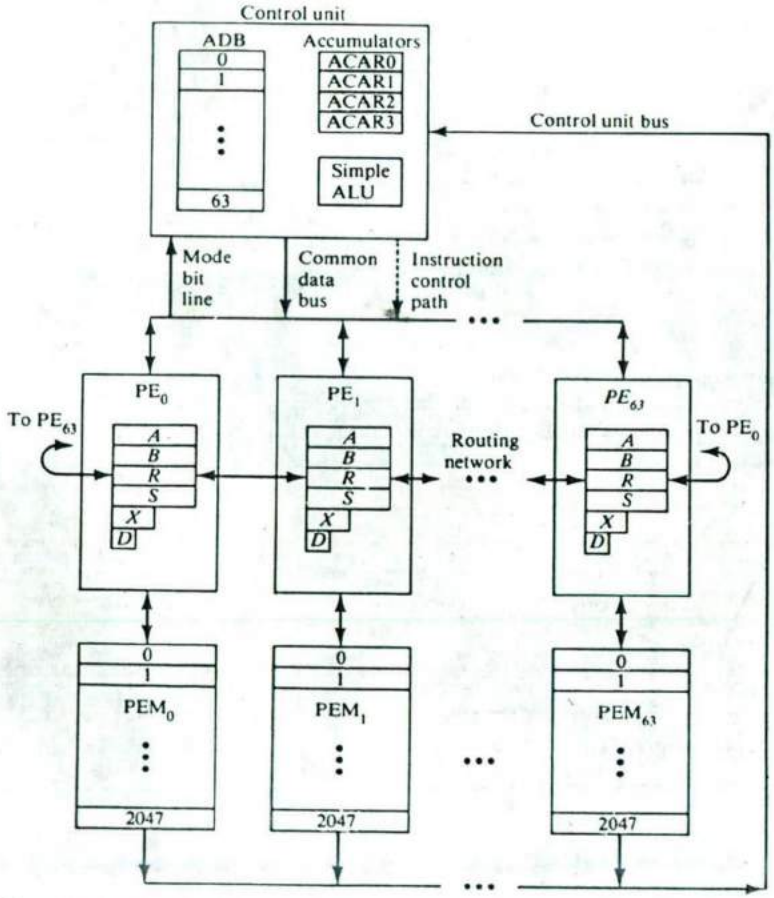


Figure 6.4 A 64-PE Illiac IV array. (Courtesy of *IEEE Proc.* Bouknight et al., April 1972.)

The common data bus is used to broadcast information from the CU to the entire array of 64 PEs. For example, a constant multiplier need not be stored 64 times in each PEM; instead, the constant can be stored in a CU register and then broadcast to each enabled PE. Special routing instructions are used to send information from one PE register to another PE register via the routing network. Standard load or store instructions are used to transfer information from PE registers to PEM. At most, seven routing steps are needed to transfer information among the PEs via the mesh network. The software figures out the shortest routing path in each data-routing operation. The *mode-bit line* consists of one line coming from the *A* register of each PE in the array. These lines can transmit the mode bits of the *D* register in the array to the accumulator register in the CU. There are CU instructions which can test the mask vector and branch on a zero or nonzero condition.

The Illiac-IV communicates with the outside world through an I/O subsystem (Figure 6.5), a disk file system, and a B6500 host computer which supervises a large laser memory (10^{12} bits) and the ARPA network link. The disk has 128 heads, one per track, with a 40-ms rotation speed and an effective transfer rate of 10^9 bits per second. The B6500 manages all programmer requests for system resources. The operating system, including compilers, assemblers, and I/O service routines, are residing in the B6500. As a total system, the Illiac-IV array is really a special-purpose back-end machine of the B6500. The ARPA net linkage makes the Illiac-IV available to all members of the ARPA network.

The control unit (CU) of the Illiac-IV array performs the following functions needed for the execution of programs:

1. Controls and decodes the instruction streams.
2. Transmits control signals to PEs for vector execution.
3. Broadcasts memory addresses that are common to all PEs.

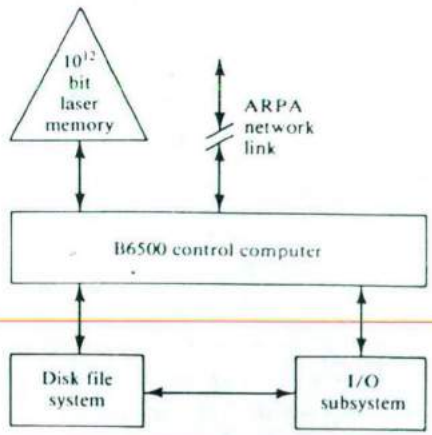


Figure 6.5 The Illiac IV I/O System.

4. Manipulates data words common to the calculations in all PEs.
5. Receives and processes trap or interrupt signals.

A block diagram of the CU is shown in Figure 6.6. The CU by itself is a scalar processor, in addition to its capability of concurrently controlling the PE-array operations. The *instruction buffer* (PLA) and *local data buffer* (LDB) are 64-word fast-access buffers. The PLA is associatively addressed to hold current and pending instructions. The LDB is a data cache with 64 bits per word. There are four *accumulator registers* (ACAR). The CU *arithmetic unit* performs 64-bit scalar

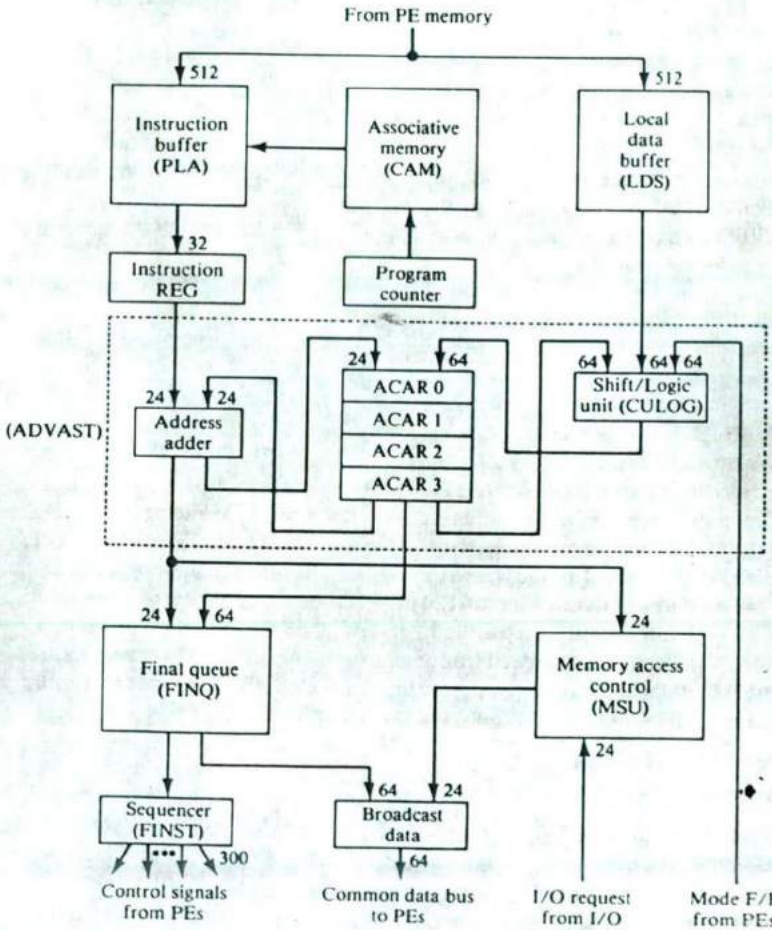


Figure 6.6 Functional block diagram of the Illiac IV control unit. (Courtesy of *IEEE Proc.* Bouknight et al., April 1972.)

addition, subtraction, shift and logic operations. More complex and vector arithmetic logic operations are relegated to the PEs. Address arithmetic is performed by the 24-bit address adder. The *final queue* is used to stack the addresses and data waiting to be transmitted to the PEs.

All instructions are 32 bits wide and classified as either CU instructions or PE instructions. CU instructions are for program control (indexing, jumps, etc.) and scalar operations. PE instructions are decoded by the *adv*anced instruction *s*tation (ADVAST) and then transmitted via control signals to all PEs. In fact, the ADVAST decodes all instructions and executes the CU instructions. The ADVAST constructs the necessary address and data operands after decoding a PE instruction. The PLA instruction buffer can hold 128 instructions, sufficient to hold the inner loop of many programs.

A block diagram of the processing element is shown in Figure 6.7. The PE computes with the distributed data and reforms local indexing for skewed memory fetch. Major components in a PE include:

1. Four 64-bit registers: A is an accumulator, B is the operand register, R is the data-routing register, and S is a general-storage register.
2. An adder/multiplier, a logic unit, and a barrel switch for arithmetic, boolean, and shifting functions, respectively.
3. A 16-bit index register and an adder for memory address modification and control.
4. An 8-bit mode register to hold the results of tests and the PE masking information.

Each PE has a 64 bit wide routing path to four neighbors. To minimize the physical routing distance, the PEs are grouped as shown in Figure 6.7. This drawing has been logically described in Figure 6.8 for a smaller network size. Routing by a distance of plus or minus eight occurs interior to each group of eight PEs. The CU data and instruction fetches require blocks of eight words, which are accessed in parallel. The individual PEM is a thin-film memory with a cycle time of 240 ns and an access time of 120 ns. Each has a capacity of 2048 words. Each PEM is independently accessible by its attached PE, the CU, or other I/O connections. The computing speed and memory of the Illiac-IV arrays require a substantial secondary storage for program and data files. A backup memory is used for programs with data sets exceeding the fast-memory capacity.

6.2.2 Applications of the Illiac-IV

The Illiac-IV was primarily designed for matrix manipulation and for solving partial differential equations. Many ARPA net users attempt to use the Illiac-IV for their own applications. The main difficulties in programming the Illiac-IV are the exploitation of identical arithmetic computations in user programs and the proper distribution of data sets in the PEMs to allow parallel accesses. In this section, we examine several programming problems of the Illiac-IV.

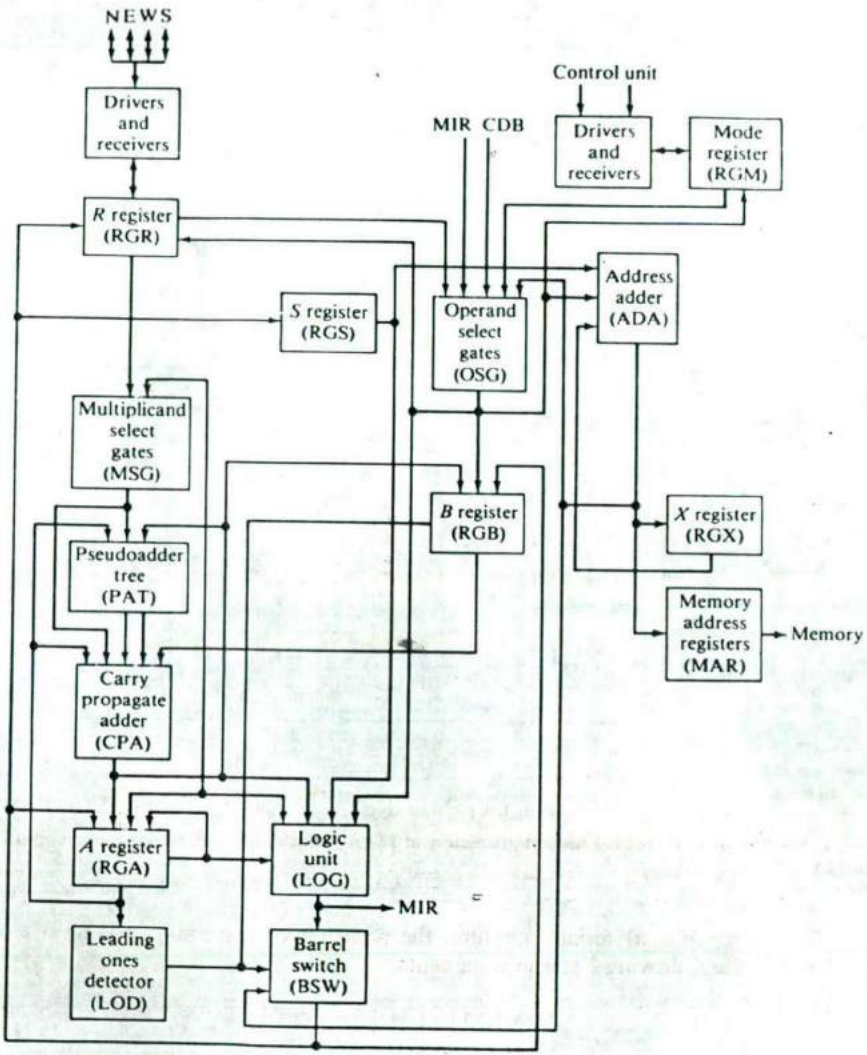
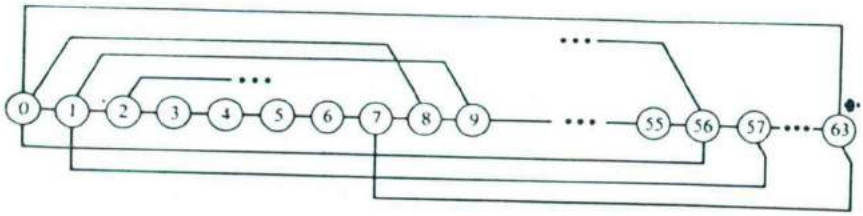
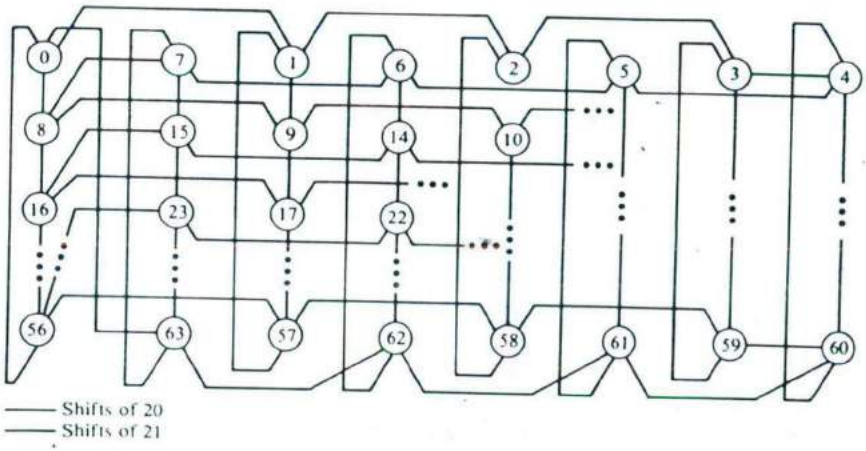


Figure 6.7 The internal structure of the processing element in Illiac IV. (Courtesy of *IEEE Proc.* Bouknight et al., April 1972.)



(*a*) Electrical connectivity



(*b*) The physical layout

Figure 6.8 The Illiac IV routing network. (Courtesy of *IEEE Trans. Computers*, Barnes et al., August 1968.)

In a conventional serial computer, the addition of two arrays (vectors) is realized by the following Fortran statements:

```
DO 100 I=1,N
100 A(I)=B(I)+C(I)                    (6.1)
```

These two Fortran statements will be compiled into a sequence of machine instructions which include the initialization of the loop, the looping-control instructions, the component-addition instruction, and the storage of the result. The initialization instructions are outside the loop. All the remaining machine instructions must be executed N times in the loop.

The Illiac-IV can perform the additions in the loop simultaneously by involving all 64 PEs in synchronous lock-step fashion. The data must be allocated in the

PEMs to support parallelism in the PEs. We consider below the programming of the computations in Eq. 6.1 on the Illiac-IV.

Example 6.1

Case 1: $N = 64$ **(The array matches the problem size)** Only three Illiac-IV machine instructions are needed to implement the Eq. 6.1 loop. The 64 components of the A, B, and C arrays are allocated in memory locations α, $\alpha + 1$, and $\alpha + 2$ of the PEMs, respectively, as shown in Figure 6.9. The machine instructions are:

LDA $\alpha + 2$ (*Load* the accumulators of all PEs with the C array).
ADRN $\alpha + 1$ (*Add* to the accumulators the contents of the B array)
STA α (*Store* the result in the accumulators to the PEMs)

Note that all the 64 *loads* in LDA, the 64 *adds* in ADRN, and the 64 *stores* in STA are performed in parallel in only three machine instructions. This means a speedup 64 times faster than a conventional serial computer.

Case 2: $N < 64$ **(The problem size is smaller than the array size)** In this case, only a subset of the 64 PEs will be involved in the parallel operations. The same memory allocation and machine instructions as in case 1 are needed, except some of the memory space and PEs will be masked off. The smaller the value

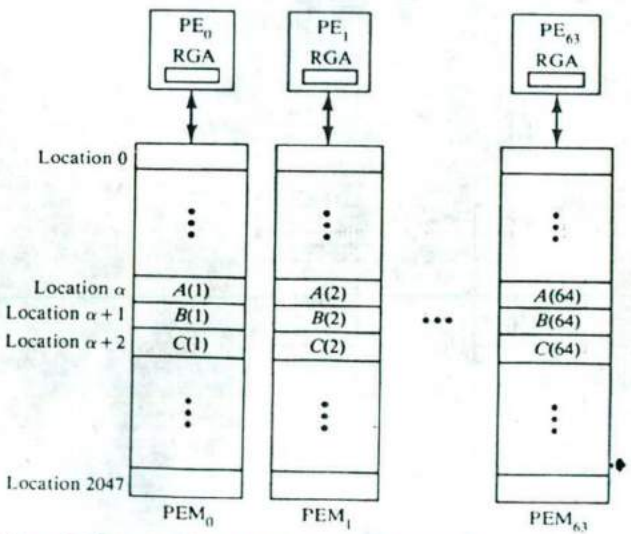


Figure 6.9 Data allocation in PEMs to execute the program:

```
DO 10 I = 1, 64
10 A(I) = B(I) + C(I)
```

N compared to 64, the severer the idleness of the disabled PEs and PEMs in the array.

Case 3: $N > 64$ (The problem size is greater than the array size) The memory allocation problem becomes much more complicated in this case. The case of $N = 66$ is illustrated in Figure 6.10. The first 64 elements of the A, B, and C arrays are stored from locations α, $\alpha + 2$, and $\alpha + 4$, respectively, in all PEMs. The two residue elements A(65), A(66); B(65), B(66); and C(65), C(66) are stored in locations $\alpha + 1$, $\alpha + 3$, and $\alpha + 5$, respectively, in PEM_0 and PEM_1. The unused memory locations are indicated by question marks. Six machine-language instructions are needed to perform the 66 load, add, and store operations:

1. *Load* the accumulator from location $\alpha + 4$.
2. *Add* to the accumulator the contents of location $\alpha + 2$.
3. *Store* the result to location α.
4. *Load* the accumulator from location $\alpha + 5$.
5. *Add* to the accumulator the contents of location $\alpha + 3$.
6. *Store* the result to location $\alpha + 1$.

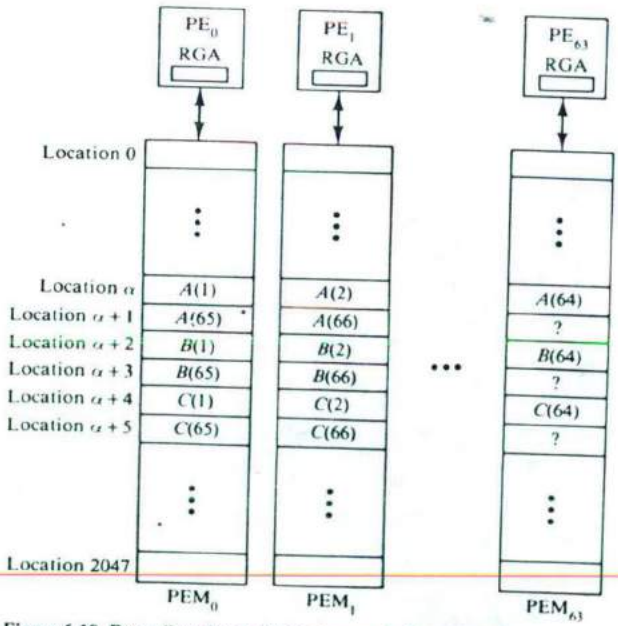


Figure 6.10 Data allocation in PEMs to execute the program.

```
DO 10 I = 1, 66
10 A(I) = R(I) + C(I)
```

The two residue data items in the A, B, and C arrays require three additional Illiac instructions. In fact, the above six instructions can be used to perform any vector addition of dimensions $65 \leq N \leq 128$ in Illiac-IV. The particular storage scheme shown in Figure 6.10 wastes almost three rows of storage ($62 \times 3 = 186$ words).

Next, we consider the implementation of a linear recurrence in the Illiac-IV:

```
    DO 100 I=2,64
100 A(I)=B(I)+A(I-1)
```
(6.2)

This recursive loop demands the following set of Fortran statements to be executed sequentially:

```
A(2)=B(2)+A(1)
A(3)=B(3)+A(2)
  ⋮
A(63)=B(63)+A(62)
A(64)=B(64)+A(63)
```

We can rewrite the above sequential statements as follows:

```
A(2)=B(2)+A(1)
A(3)=B(3)+B(2)+A(1)
A(4)=B(4)+B(3)+B(2)+A(2)
  ⋮
A(N)=B(N)+B(N-1)+···+B(2)+A(1)
```

The above 63 computations can be computed independently and simultaneously in the Illiac-IV. We write

$$A(k) = A(1) + \sum_{I=2}^{k} B(I) \qquad \text{for } 2 \leq k \leq 63 \tag{6.3}$$

A Fortran code is given below to perform the above expanded computations:

```
    S=A(1)
    DO 100 K=2,64
    S=S+B(K)
100 A(K)=S
```

The implementation of this decoupled Fortran program on the Illiac-IV requires the following machine instructions, based on the memory allocation shown in Figure 6.11.

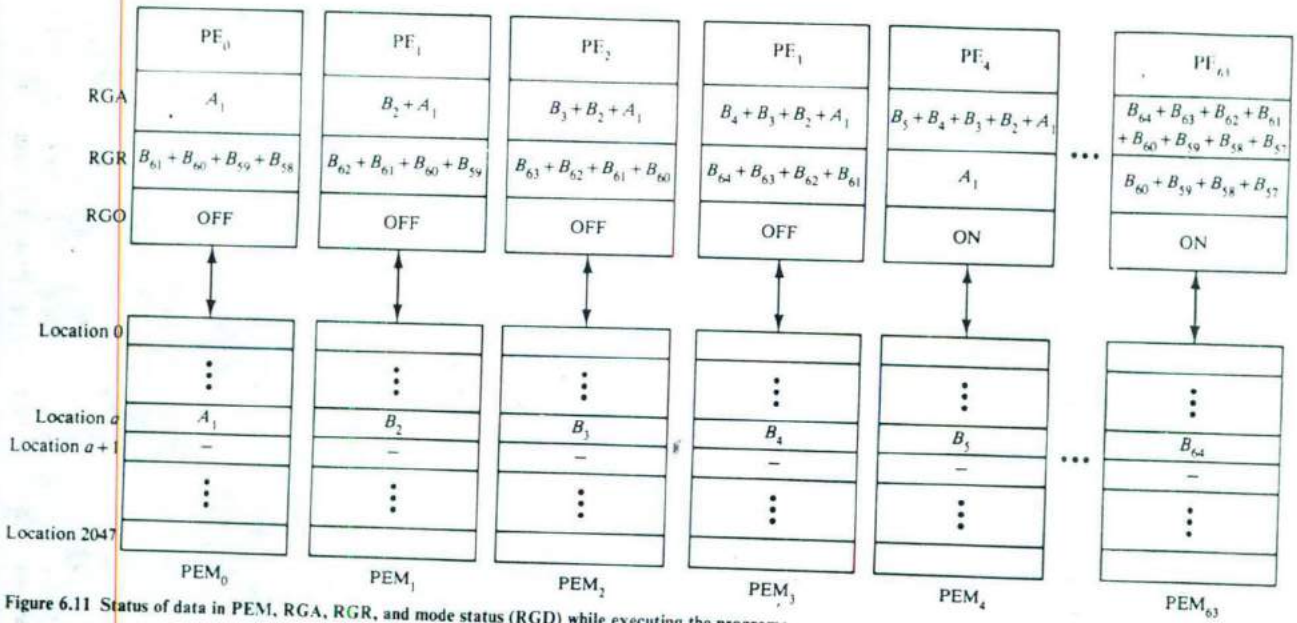


Figure 6.11 Status of data in PEM, RGA, RGR, and mode status (RGD) while executing the program:

```
DO 10 I = 2, 64
10 A(I) = B(I) + A(I - 1)
```

1. Enable all PEs.
2. All PEs load accumulators from memory location α.
3. Set index $i \leftarrow 0$.
4. All PEs load registers from their accumulators.
5. All PEs route their register contents to the right for a distance of 2^i.
6. Set index $j \leftarrow 2^i - 1$.
7. Disable PEs from 0 through j.
8. All enabled PEs add to accumulators the contents of routing registers.
9. Reset $i \leftarrow i + 1$.
10. If $i < 6$, go to step 4.
11. Enable all PEs.
12. All PEs store results in accumulators to memory location $\alpha + 1$.

Figure 6.11 shows the status of data in the PEM, the accumulator, the routing register, and the mode-status registers from those enabled and disabled instructions after step 8 is executed, when $i = 2$. Parallelism has been revealed after decoupling in this example.

Most of the Illiac-IV system software is executed by the host processor B6500. This system manager performs the standard B6500 operations, handles user-seeking Illiac-IV services, and implements the necessary features to support the operation of the PE array, the disk-file system, and the I/O subsystem. The Illiac-IV array can be visualized as the highest priority time-sharing user of the B6500 among many users connected via the ARPA net. Results produced by the B6500 or by the Illiac-IV programs may be printed locally or transmitted over the ARPA net to remote output devices local to the user. The Illiac-IV array has a small resident operating system executed by its CU which will allow the fast processing of traps and other special loading operations.

The Illiac-IV operating system runs between a *diagnostic mode* and a *normal mode*. The main task of the diagnostic mode is the testing and diagnosis of possible faults in the I/O subsystem and in the Illiac-IV array itself. The Illiac-IV operating system consists of a set of asynchronous processes which run under the control of the B6500 master-control program. The following events may take place when a user submits an Illiac-IV job to the B6500:

1. The B6500 translates Algol or Fortran programs into binary input files to be used by the Illiac-IV array processor.
2. The Illiac-IV programs written in Ask, Glypnir, or Illiac-IV Fortran will operate on the files prepared by the B6500 programs and prepare binary output files.
3. The B6500 transforms the binary files from the Illiac-IV to the required external form for use or storage.
4. An Illiac control-language program controls the operating system for the job which it defines.

The B6500 programs and the Illiac-IV programs communicate via the disk files (for data) and via the 48-bit path for CU interrupt signals. The protocol for

these signals over the 48-bit path is administered by two modules. The first is a small executive program residing in the Illiac-IV itself (called OS4) which processes all interrupts for the array, handles all communications between the user programs and the rest of the operating system, and provides a few standard functions for use in the array. The OS4 communicates with a module (known as the *job partner*) in the B6500, which acts as a clearing house for all communication between the OS4 and thus the user program running on the Illiac-IV. The job partner thus initiates all data transfers between the B6500 and the Illiac-IV disk. This arrangement emphasizes the B6500 as an I/O processor for the Illiac-IV or, conversely, the Illiac-IV as a peripheral processor for the B6500.

The Illiac-IV is very difficult to program properly if one does not banish nearly all serial machine preconceptions and habits. It is worth pointing out the differences between the Illiac-IV high-level languages and the existing languages:

1. The natural method of addressing PEMs is by rows of 64 words, since the words of PEMs may be addressed in parallel. However, a column of words in one PEM may not be addressed at once.
2. The vector elements are operated upon based on the mode pattern. The Illiac-IV language should allow efficient manipulation of the mode patterns.
3. The Illiac language should allow reasonable expression of routing and indexing independently in each PE.

The design experiences of the Illiac-IV are very useful in developing later SIMD array processors. The performance of the Illiac-IV is about two to four times faster than the CDC-7600. The Illiac-IV has limited scalar capability; it uses a recirculating mesh network with fixed size. Some of these difficulties have been overcome in later array processors like the BSP and the MPP. We shall discuss some performance enhancement methods, including the skewed-memory allocations and some language extensions, in Section 6.4.

6.2.3 The BSP System Architecture

The BSP was a commercial attempt made by the Burroughs Corporation beyond the Illiac-IV in order to meet the increasing demand of large-scale scientific and engineering computers. It improves in many aspects on the Illiac-IV design. We describe below the parallel architecture of the BSP and its conflict-free memory organization. Even though the BSP has been suspended by Burroughs, it is a well-designed array supercomputer that we can still learn much from. The BSP extends the array processing capability of the Illiac-IV to a vectorizing Fortran machine. With a maximum speed of 50 megaflops, the BSP was designed to perform large-scale computations in the fields of numerical weather prediction, nuclear energy, seismic signal processing, structure analysis and econometric modeling.

The BSP is not a stand-alone computer. It is a back-end processor attached to a host machine, a *system manager*, such as the B7800 depicted in Figure 6.12. The motivation for attaching the BSP to a system manager is to free the BSP from

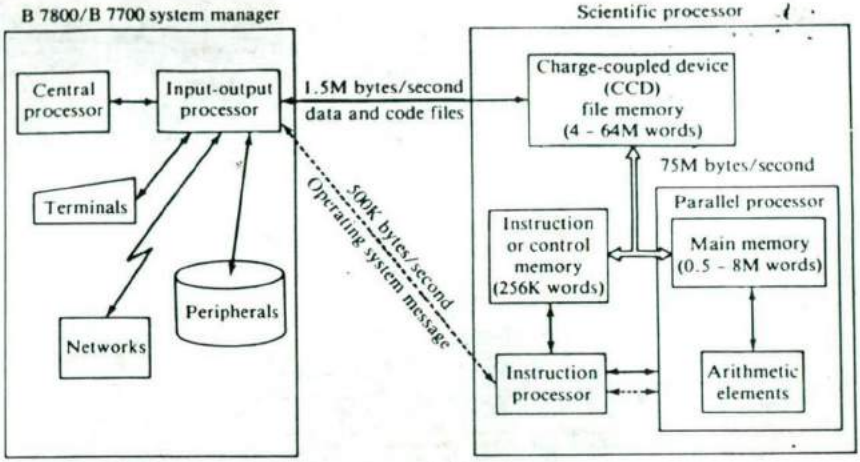
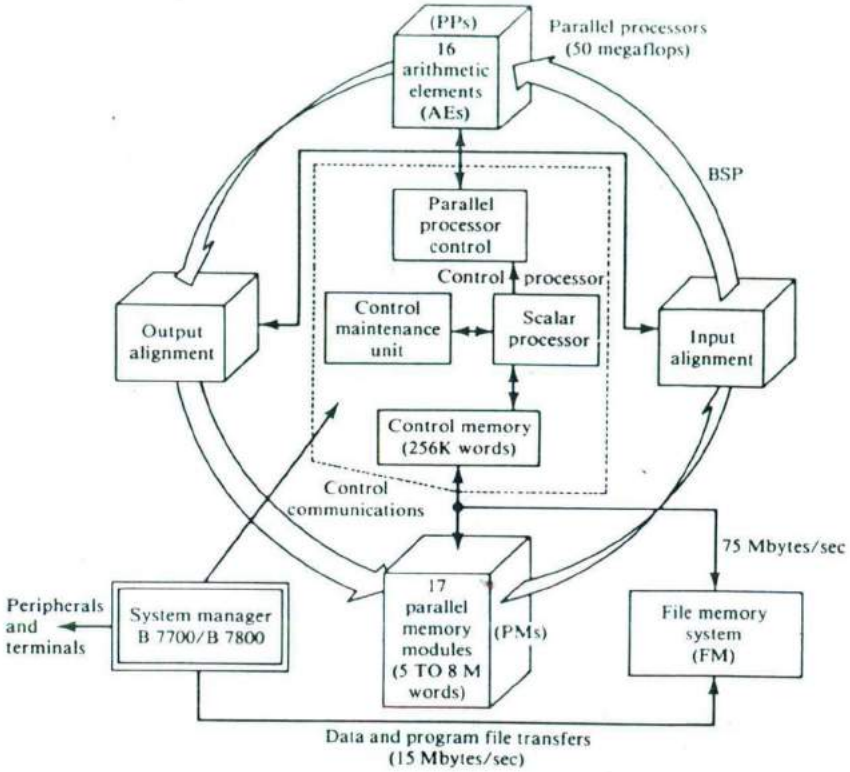


Figure 6.12 The Burroughs scientific processor attached to a host processor.

routine management and I/O functions in order to concentrate on arithmetic computations. The system manager provides time-sharing services, data and program-file editing, data communication to remote job-entry stations, terminals and networks, vectorizing compiling and linking of the BSP programs, long-term data storage, and database-management functions. Major components in the BSP include the control processor, the parallel processors, a file memory, parallel-memory models, and the alignment network shown in Figure 6.13.

The *control processor* provides the supervisory interface to the system manager in addition to controlling the parallel processor. The *scalar processor* processes all operating system and user-program instructions, which are stored in the *control memory*. It executes some serial or scalar portions of user programs with a clock rate of 12 MHz and is able to perform up to 1.5 megaflops. All vector instructions and certain grouped scalar instructions are passed to the *parallel processor controller*, which validates and transforms them into microsequences controlling the operation of the 16 *arithmetic elements* (AEs). The bipolar control memory has 256K words with a 160-ns cycle time. Each word has 48 bits plus 8 parity-check bits to provide the SECDED capability. The *control and maintenance unit* is an interface between the system manager and the rest of the control processors for initiation, communication of supervisory command and maintenance purposes.

The parallel processors perform vector computations with a clock period of 160 ns. All 16 AEs must execute the same instruction (broadcast from the parallel processor controller) over different data sets. Most arithmetic operations can be completed in two clock periods (320 ns). The BSP is capable of executing up to 50 megaflops. Data for the vector operations are stored in 17 parallel *memory modules*, each of which contains up to 512 K words with a cycle time of 160 ns. The data transfer between the memory modules and the AEs is 100 M words per



CLOCK 1	FETCH		
CLOCK 2	ALIGN	FETCH	
CLOCK 3	PROCESS	ALIGN	FETCH
CLOCK 4	ALIGN	PROCESS	ALIGN
CLOCK 5	STORE	ALIGN	PROCESS
CLOCK 6		STORE	ALIGN
CLOCK 7			STORE

Figure 6.13 Functional structure and pipelined processing in the BSP. (Couresy of *IEEE Trans. Computers*, Lawrie and Vora, 1982.)

second. The organization of the 17 memory modules provides a conflict-free memory that allows access to vectors of arbitrary length and with a skip distance which is not a multiple of 17.

Memory-to-memory floating-point operations are pipelined in BSP. The pipeline organization of the BSP consists of five functional stages. First, 16 operands are fetched from the memory modules, routed via the *input alignment network* into

the AEs for processing, and routed via the *output alignment network* into the modules for storage. These steps are overlapped, as illustrated in Figure 6.13. Note that the input alignment and the output alignment are physically in one alignment network. The division shown here presents only a functional partition of the pipeline stages. In addition to the spatial parallelism exhibited by the 16 AEs and the pipeline operations of the fetch, align, and store stages, the vector operations in the AEs can overlap with the scalar processing in the scalar processor. This results in a powerful and flexible system suitable for processing both long and short vectors and isolated scalars as well.

Both alignment networks contain full crossbar switches as well as hardware for broadcasting data to several destinations and for resolving conflicts if several sources seek the same destination. This permits general-purpose interconnectivity between the arithmetic array and the memory-storage modules. It is the combined function of the memory-storage scheme and the alignment networks that supports the conflict-free capabilities of the parallel memory. The output alignment network is also used for interarithmetic element switching to support special functions such as the data compress and expand operations and the fast Fourier transform algorithm.

The *file memory* is a semiconductor secondary storage. It is loaded with the BSP task files from the system manager. These tasks are then queued for execution by the control processor. The file memory is the only peripheral device under the direct control of the BSP; all other peripheral devices are controlled by the system manager. Scratch files and output files produced during the execution of a BSP program are also stored in the file memory before being passed to the system manager for output to the user. The file memory is designed to have a high data-transfer rate, which greatly alleviates the I/O-bound problem.

In summary, concurrent computations in the BSP are made possible by four types of parallelism:

1. The parallel arithmetic performed by the 16 arithmetic elements
2. Memory fetches and stores, and the transmission of data between memory and arithmetic elements
3. Indexing, vector length, and loop-control computations in the parallel processor controller
4. The generation of linear vector operating descriptions by the scalar processor

The 16 AEs operate synchronously under the control of a single micro-sequence in SIMD mode. Each AE has only the most primitive operators hard-wired. The control word is 100 bits wide. Besides being a floating-point machine, the AE has substantial nonnumeric capability as well.

Floating-point add, subtract, and *multiply* each take two memory clocks. The use of two clocks balances the memory bandwidth with the AE bandwidth for triadic operations. A *triadic operation* is defined as having three operands and one result. The *floating-point divide* (requiring 1200 ns) is implemented by generating the reciprocol in a Newton-Raphson iteration. ROMs exist in each AE to give

the first approximations for the divide and the square root iterations. The floating-point format is 48 bits long. It has 36 bits of a significant fraction and 10 bits of a binary exponent. This gives 11 decimal digits of precision. The AE has double-length accumulators and double-length registers in key places. This permits the direct implementation of double-precision operators in the hardware. The AE also permits software implementations of triple-precision arithmetic operations. It has been estimated that 20 to 40 megaflops could be achieved for a broad range of Fortran computations in the BSP.

6.2.4 The Prime Memory System

The BSP parallel memory consists of 17 memory modules, each with a 160-ns cycle time. Since we access 16 words per cycle, this provides a maximum effective 10-ns memory-cycle time. This is well balanced with the arithmetic elements which perform floating-point addition and multiplication at the rate of 320 ns/16 operations = 20 ns per operation, since each operation requires two arguments and temporary registers are provided in the arithmetic elements.

Only array accessing (including I/O) uses parallel memory, since programs and scalars are held in the control memory. Thus, perfect balance between parallel memory and floating-point arithmetic may be achieved for trial vector forms since three arguments and one result (four memory accesses) are required for two arithmetic operations. For longer vector forms, since temporaries reside in registers, only one operand is required per operation, so there is substantial parallel-memory bandwidth remaining for input and output of information.

The main innovation in the parallel memory of the BSP is its 17 modules. In past supercomputers it has been common to use a number of parallel-memory modules, but such memory systems are vulnerable to serious bandwidth degradation due to conflicts. For example, if 16 memories were used and a 16×16 array were stored with rows across the units and one column in each memory unit, then column access would be sequential.

The BSP offers a linear vector approach to parallelism. Memory addressing methods of achieving such parallelism are described in this section. The basic quantity susceptible to parallelism in the BSP is the *linear vector*. A linear vector is a vector whose elements are mapped into the main memory in a linear fashion. The linear-vector components are separated by a constant distance d. For example, in a Fortran columnwise mapping, columns have $d = 1$, rows have $d = n$, and forward diagonals have $d = n + 1$. The manipulation of linear vectors in a BSP utilizes both spatial and temporal parallelism.

A unique feature of the BSP is its conflict-free memory system which delivers a useful operand to each AE per each memory cycle. The distance between elements of a vector need not be unity. Therefore, DO loops may contain nonunity increments, or the program may access rows, columns, or diagonals of matrices without penalty. Supercomputer designers have elected either to use memories with severe access restrictions or have used expensive fast-memory parts to attain a degree of conflict-free access through memory-bandwidth overkill.

The hardware techniques used to ensure conflict-free access in the BSP include a prime number of memory ports, full crossbar switches between the memory ports and the AEs, and a special memory address generation which computes the proper addresses for a particular address pattern. This address pattern is the one used by orthodox serial computers. That is, each higher memory address refers to the "next" word in memory. With this pattern, the parallel memory is completely compatible with all the constructs of present programming languages. In particular, Fortran EQUIVALENCE, COMMON, and array-parameter passing can be implemented in the same way as on a conventional computer.

Consider a BSP-like machine with N AEs and M memory modules, where M is a prime number. The *modular number* μ specifies which memory unit a data element associated with a linear address a is stored. This module number can be computed by

$$\mu = a(\text{mod } M) \tag{6.4}$$

The *address offset* i within the assigned memory module is calculated by

$$i = \left\lfloor \frac{a}{N} \right\rfloor \tag{6.5}$$

Figure 6.14 shows a 4-by-5 matrix mapped columnwise into the memory of a serial machine. For simplicity of illustration, assume $N = 6$ and $M = 7$ in the hypothetical machine. (The BSP has $N = 16$ and $M = 17$.) The module and offset calculations are shown in the illustration. The module number will remain constant for a cycle equal to the number of AEs, then it will increment by one value. The offset corresponds to repeated cycles of the same module with no value repeating in one cycle and the length of the cycle equal to the number of memory banks.

Example 6.2 As long as the number of AEs is less than or equal to the number of memory banks, the sequence of offset values will cause a different memory bank to be connected to each AE. Thus, each AE may receive or send a unique data object. The particular storage pattern produced in this six AE, seven memory bank system for the 4-by-5 example array is shown in Figure 6.14. The module and offset calculations for the second row of the array is explained below. The starting address is 1 and the skip distance is $d = 4$. We obtain the following module numbers and address offsets:

$$\begin{aligned} \mu &= 1(\text{mod } 7),\ 5(\text{mod } 7),\ 9(\text{mod } 7),\ 13(\text{mod } 7),\ 17(\text{mod } 7) \\ &= 1, \qquad 5, \qquad 2, \qquad 6, \qquad 3 \end{aligned} \tag{6.6}$$

The offsets within each memory module are obtained accordingly as

$$\begin{aligned} i &= \left\lfloor \tfrac{1}{6} \right\rfloor, \left\lfloor \tfrac{5}{6} \right\rfloor, \left\lfloor \tfrac{9}{6} \right\rfloor, \left\lfloor \tfrac{13}{6} \right\rfloor, \left\lfloor \tfrac{17}{6} \right\rfloor \\ &= 0, \quad 0, \quad 1 \quad 2, \quad 2 \end{aligned} \tag{6.7}$$

The decision of using $M = 17$ memory modules for the $N = 16$ AEs will provide conflict-free array access to most common array partitions and yet have

A 4 × 5 matrix

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

Physical mapping of the 4 × 5 matrix

	0	1	2	3	4	5	6
0	0 a_{11}	1 a_{21}	2 a_{31}	3 a_{41}	4 a_{12}	5 a_{22}	
1	7 a_{42}	8 a_{13}	9 a_{23}	10 a_{33}	11 a_{43}		6 a_{32}
2	14 a_{34}	15 a_{44}	16 a_{15}	17 a_{25}		12 a_{14}	13 a_{24}
3	21	22	23		18 a_{35}	19 a_{45}	20
4	28	29		24	25	26	27
5	35		30	31	32	33	34

For example, if $M = 7$, $N = 6$, the 4×5 array is mapped

Array elements	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}
Linear address: $a =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
memory module number $\mu =$	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5
offset within module $i =$	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3

Figure 6.14 The BSP memory mapping for a linearized vector of matrix elements. (Courtesy of Burroughs Corp. 1978.)

little redundant memory bandwidth, since only one memory module is unused per cycle. It is clear that conflict-free access to one-dimensional arrays is possible for any arithmetic sequence index pattern except every 17th element. For two-dimensional arrays with a skewing distance of four, conflict-free access is possible for rows, columns, diagonals, back-diagonals, and other common partitions, including arithmetic-sequence indexing of these partitions. The method can be extended to higher numbers of dimensions in a straightforward manner.

The unused memory cells are the result of having one less AE than there are memory banks. However, the example case is not a useful occurrence. For the real BSP with 16 AEs and 17 memory banks, division by 16 is much simpler and faster than division by 17. One then pays the penalty of supplying some extra memory to reach a given usable size. The above equations yield an AE-centrist vantage point. As long as the same set of equations is always applied to the data from the first time it comes in as I/O onward, then the storage pattern is completely invisible to the user. This applies to program dumps as well because the hardware always obeys the same rules.

Conflict does occur if the addresses are separated by an integer multiple of the number of memory banks. In this case, all the values one wants are in the same memory bank. For the BSP, this means that skip distances of 17, 34, 51, etc., should be avoided. In practice, 51 is a likely problem skip. This is because it is the skip of a forward diagonal of a matrix with column length 50. If conflict occurs in the BSP, the arithmetic is performed correctly, but at $\frac{1}{16}$ the normal speed. The system logs the occurrence of conflicts and their impact on the total running time. This information is given to the programmer for corrective action if the impact was significant.

6.2.5 The BSP Fortran Vectorizer

The BSP has a total of 64 vector instructions, which can be grouped in four types:

- Array expression statements
- Recurrence and reduction statements
- Expand, compress, random store, and fetch
- Parallel data transmissions between control memory and file memory

Array expression statements include indexing and evaluating righthand side array expressions ranking from monad to pentad (five righthand side operands), plus the assignment of the resulting values to parallel memory. The array operations are performed in an element-by-element fashion and allow scalars and array variables of one or two dimensions to be mixed on the righthand side. For example:

```
      DO 5 I=1,30
        DO 5 J=7,25
5         X(I,J)=(A(I,J+1)*0.5+B(I+1,J))
                 *X(I,J+1)+C(J)                    (6.8)
```

would be compiled as a single vector form. This vector form can be regarded as a six-address instruction that contains the four array arithmetic-operation specifications and the assignment operation.

Recurrence vector instructions correspond to assignment statements with data-dependence loops. For example:

```
      DO 3 I=1,25
3     Y(I)=F(I)*Y(I-1)+G(I)                    (6.9)
```

has a righthand side that uses a result computed on the previous iteration. This recurrence produces an array of results while others lead to a scalar result. For example, a polynomial evaluation by Horner's rule leads to the following reduction:

```
      P=C(0)
      DO 5 I=1,25
5     P=C(I)+Y*P.                              (6.10)
```

The third type of vector instructions involves various sparse-array operations. For example, in the case of a Fortran variable with subscripted subscripts, e.g., A(B(I)), no guarantee can be made concerning conflict-free access to the array A. In this case, the indexing hardware generates a sequence of addresses that allows access to one operand per clock. These are then processed in parallel in the arithmetic elements. Such accesses are called *random store* and *random fetch* vector forms.

Sparse arrays may be stored in memory in a compressed form and then expanded to their natural array positions using the input-alignment network. After processing, the results may be compressed for storage by the output-alignment network. These are called *compressed vector operand* and *compressed vector result* vector forms and use control-bit vectors that are packed in such a way that one 48-bit word is used for accesses to three 16-element vector slices.

The fourth class of vector instructions is used for I/O. Scalar and array assignments are made to control memory and parallel memory, depending on whether they are to be processed in the scalar processor unit or the parallel processor, respectively. However, it is occasionally necessary to transmit data back and forth between these memories. Transmissions to file memory are standard I/O types of operations. In Table 6.2, representative vector instructions in the BSP are listed. These four types of vector instructions comprise the entire array functions performed by the BSP.

In ordinary Fortran programs, it is possible to detect many array operations that can easily be mapped into BSP vector instructions. This is accomplished in the BSP compiler by a program called the Fortran vectorizer. We will not attempt a complete description of the vectorizer here, but we will sketch its organization, emphasizing a few key steps.

First, consider the generation of a program graph based on data dependencies. Each assignment statement is represented by a graph node. Directed arcs are drawn between nodes to indicate that one node is to be executed before another. The BSP does a detailed subscript analysis and builds a high-quality graph with few redundant arcs, thereby leading to more array operations and fewer recurrences.

Example 6.3 Consider the following program which explains the problem of scoping and data dependence:

```
      DO 5 I=1,25
1       A(I)=3*B(I)
3         DO 3 J=1,35
5           X(I,J)=A(I)*X(I,J-1)+C(J)
        B(I)=2*B(I+1)
```
(6.11)

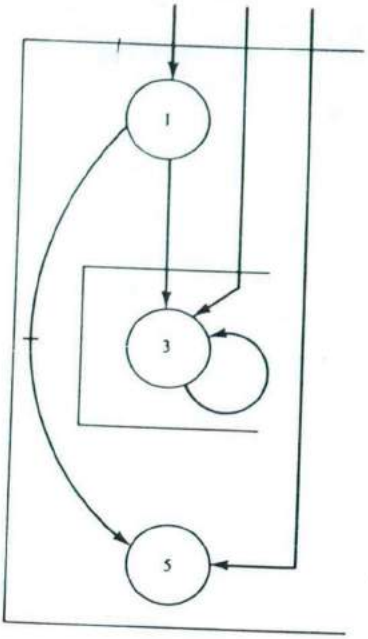
A dependence graph for this problem is shown in Figure 6.15*a*, where nodes are numbered according to the statement label numbers of the program. Node 1 has an arc to node 3 because of the $A(I)$ dependence, and node 3 has a self-loop because $X(I, J-1)$ is used one J iteration after it is generated. The crossed arc from node 1 to node 5 is an antidependence arc indicating that statement 1 must be executed before statement 5 to ensure that $B(I)$ on the righthand side of statement 1 is an initial value and not one computed by statement 5. Arcs from above denote initial values being supplied to each of the three statements: array B to statements 1 and 5, and array C to statement 3. The square brackets denote the scope of loop control for each of the DO statements.

Given a data-dependence graph, loop control can be distributed down to individual assignment statements or collections of statements with internal loops of data dependence. In our example, there is one loop (containing just one statement) and two individual assignment statements. After the distribution of loop control, the graph of Figure 6.15*a* may be redrawn, as shown in Figure 6.15*b*, which can easily be mapped into BSP vector instructions. Statements 1 and 5 go directly into array-expression vector forms since they are both dyads.

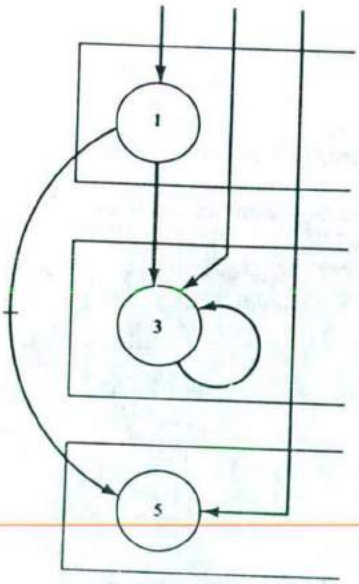
Besides vectorizing loops which do not contain branches or cyclic dependencies, the BSP vectorizer (executed by the system manager) can issue vector instructions even for BRANCH and IF statements as long as the branch paths are known to be under the control of bit vectors. The vectorizer also detects cyclic dependencies and converts them to vector recurrence statements. The following example shows the vectorization of a program containing an IF statement in a loop:

```
      DO 1 I=1,92,
        DO J=1,46
1         IF(A(I,J).LT.0)B(I,J)=A(I,J)*3.5.
```
(6.12)

This loop can be mapped into a single array-expression-statement vector form with bit-vector control that performs the parallel tests and makes the appropriate assignments to $B(I, J)$. By using loop distribution, many of the IFs found in ordinary Fortran programs can be transformed into vector operations that



(*a*) A data dependence graph



(*b*) Graph with distributed loop control

Figure 6.15 Dependence graph description of the Loop program in Eq. 6.11.

Table 6.2 Vector forms in the BSP

MONAD	It accepts one vector set operand, does one monadic operation on it, and produces one vector set result.	$Z \leftarrow op\ A$
DYAD	It accepts two vector set operands, does one operation on them, and produces one vector set result.	$Z \leftarrow A\ op\ B$
VSDYAD	It is similar to the DYAD except operand B is a scalar.	Z ← A op B
EXTENDED DYAD	It accepts two vector set operands, does one operation, and produces two vector set results.	(Z1, Z2) ← A op B
DOUBLE PRECISION DYAD	It accepts four vector set operands (i.e., two double-precision operands), performs one operation, and produces two vector set results.	(Z1, Z2) ← (A1, A2) op (B1, B2)
DUAL-DYAD	It accepts four vector set operands, does two operations, and produces two vector set results.	$Z \leftarrow A\ op_1\ B\ \ Y \leftarrow C\ op_2 D$
TRIAD	It accepts three vector set operands, does two operations, and produces one vector set result.	$Z \leftarrow (A\ op_1\ B)\ op_2\ C$
TETRADI	It accepts four set operands, does three operations, and produces one vector set result.	$Z \leftarrow ((A\ op_1\ B)\ op_2\ C)\ op_3\ D$
TETRAD2	It is similar to the TETRADI except for the order of operations.	$Z \leftarrow (A\ op_1\ B)\ op_2\ (C\ op_3\ D)$
PENTADI	It accepts five vector set operands, does four operations, and produces one vector set result.	$Z \leftarrow (((A\ op_1\ B)\ op_2\ C)\ op_3\ D)\ op_4\ E$
PENTAD2	It is similar to the PENTADI except for the order of operations.	$Z \leftarrow ((A\ op_1\ B)\ op_2\ (C\ op_3\ D))\ op_4\ E$
PENTAD3	It is similar to the PENTADI except for the order of operations.	$Z \leftarrow ((A\ op_1\ B)\ op_2\ C)\ op_3\ (D\ op_4\ E)$
AMTM	It is similar to the MONAD and is used to transmit from parallel memory to control memory.	Z ← op A
TMAM	It accepts 6 vector set operands from control memory to transmit to parallel memory.	Z ← A1(0, 0), A2(0, 0), A3(0, 0), A4(0, 0), A5(0, 0), A6(0, 0)
COMPRESS	It accepts a vector set operand, compresses it under a bit vector operand control, and produces a vector set result.	X ← A, BVO
EXPAND	It accepts a vector operand, expands it under a bit vector control, and produces a vector set result.	X ← V, BVO

Table 6.2 (*Cont.*)

MERGE	It is the same as the EXPAND except that the vector set result elements corresponding to a zero bit in BV are not changed in the parallel memory.	X ← V, BV()
RANDOM FETCH	It performs the operation $Z(j, k) \leftarrow U(I(j, k))$, where U is a vector and I is an index vector set.	
RANDOM STORE	It performs the operation $X(I(j, k)) \leftarrow A(j, k)$, where X is a vector and I is an index vector set.	
REDUCTION	It accepts one vector set operand and produces one vector result given by $X(i) \leftarrow A(i, 0)$ op $A(i, 1)$ op $A(i, 2)$ op $A(i, 3) \ldots A(i, L)$, where op must be a commutative and associative operator.	
DOUBLE PRECISION REDUCTION	It accepts two vector set operands (one double-precision vector set) and produces two vector results (one d.p. vector) given by $(X_1(i), X_2(i)) \leftarrow (A_1(i, 0), A_2(i, 0))$ op $(A_1(i, 1), A_2(i, 1))$ op $\ldots (A_1(i, L), A_2(i, L))$, where op must be a commutative and associative operator.	
GENERALIZED DOT PRODUCT	It accepts two vector set operands and produces one vector result given by $X(i) \leftarrow \{A(i, 0)\ \mathrm{op}_2\ B(i, 0)\}\ \mathrm{op}_1\ \{A(i, 1)\ \mathrm{op}_2\ B(i, 1)\}\ \mathrm{op}_1 \ldots \{A(i, L)\ \mathrm{op}_2\ B(i, L)\}$, where op_1 must be a commutative and associative operator.	
RECURRENCE-1L	It accepts two vector set operands and produces one vector result given by $X(i) \leftarrow (\{\ldots\{(B(i, 0)\ \mathrm{op}_1\ A(i, 1))\ \mathrm{op}_2\ B(i, 1)\}\ \mathrm{op}_1 \ldots\}\ \mathrm{op}_1\ A(1, L))\ \mathrm{op}_2\ B(i, L)$ where op_2 can be ADD or IOR and op_1 can be MULT or AND.	
PARTIAL REDUCTION	It accepts one vector set operand and produces one vector set result given by $Z(i, j) \leftarrow Z(i, j - 1)$ op $A(i, j)$, where op must be a commutative and associative operator.	
RECURRENCE-1A	It accepts two vector set operands and produces one vector set result given by $Z(i, j) \leftarrow \{Z(i, j - 1)\ \mathrm{op}_1\ A(i, j)\}\ \mathrm{op}_2\ B(i, j)$, where op_1 can be MULT or AND and op_2 can be ADD or IOR.	

allow substantial speedups on the BSP. Of course, there is also a residual set of IFs that must be compiled as serial code. Fortran language extensions have also been made in the BSP to facilitate vector processing.

6.3 THE MASSIVELY PARALLEL PROCESSOR

A large-scale SIMD array processor has been developed for processing satellite imagery at the NASA Goddard Space Flight Center. The computer has been named *massively parallel processor* (MPP) because of the $128 \times 128 = 16{,}384$ microprocessors that can be used in parallel. The MPP can perform bit-slice arithmetic computations over variable-length operands. The MPP has a micro-programmable control unit which can be used to define a quite flexible instruction

set for vector, scalar, and I/O operations. The MPP system is constructed entirely with solid-state circuits, using microprocessor chips and bipolar RAMs.

6.3.1 The MPP System Architecture

In 1979, NASA Goddard awarded a contract to Goodyear Aerospace to construct a massively parallel processor for image-processing applications. The major hardware components in MPP are shown in Figure 6.16. The *array unit* operates

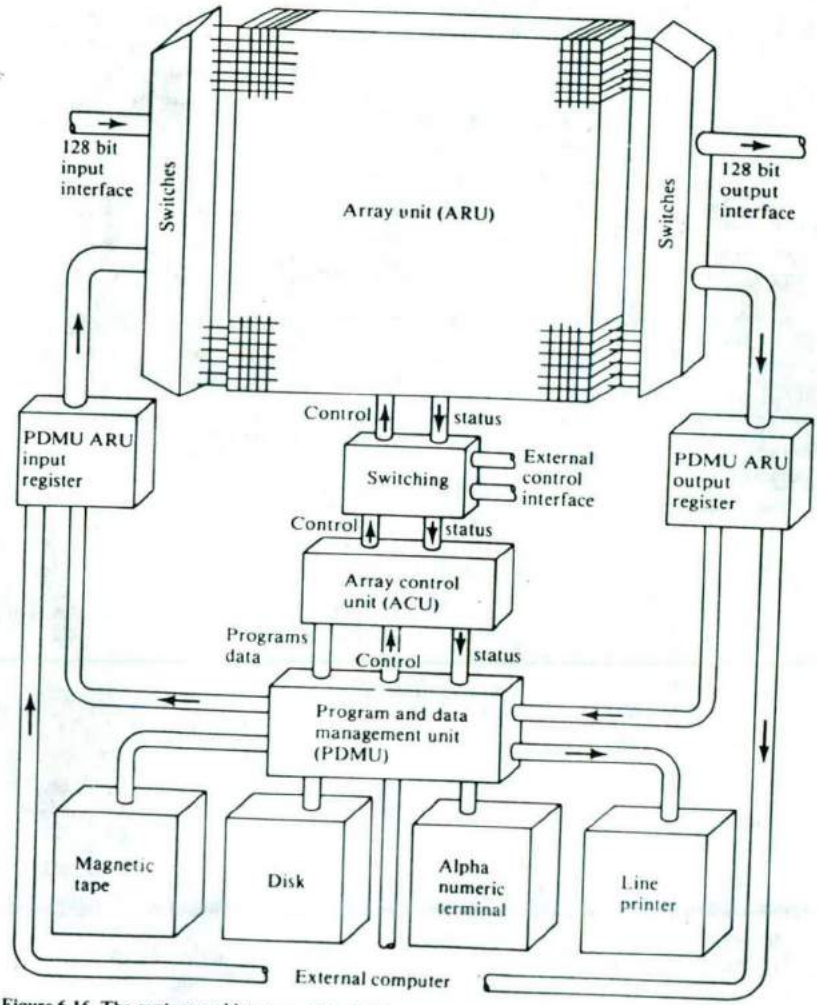


Figure 6.16 The system architecture of the MPP system. (Courtesy of *IEEE Trans. Computers*, Batcher, 1980.)

with SIMD mode on a two-dimensional array of 128 × 128 PEs. Each PE is associated with a 1024-bit random-access memory. Parity is included to detect memory faults. Each PE is a bit-slice microprocessor connected to its nearest neighbors. The programmer can connect opposite array edges or leave them open so that the array topology can change from a plane to a horizontal cylinder, a vertical cylinder, or a torus. This feature reduces routing time significantly in a number of imaging applications.

For improved maintainability, the array has four redundant columns of PEs. The physical structure of the PE array is 132 columns by 128 rows. Hardware faults are masked out with circuitry to bypass a faulty column and leave a logical array structure of 128 × 128. Arithmetic in each PE is performed in bit serial fashion using a serial-by-bit adder and a shift register to recirculate operands through the adder. This increases the speed of multiplication, division, and floating-point operations significantly. The PE array has a cycle time of 100 ns. The *array control unit* (ACU) is microprogrammable. It supervises the PE array processing, performs scalar arithmetic, and shifts data across the PE array.

The *program and data management unit* is a back-end minicomputer. It manages data flow in the array, loads programs into the controller, executes system-test and diagnostic routines, and provides program-development facilities. A Digital Equipment PDP-11/34 minicomputer is used with interfaces to the array and the MPP external computer interface. Peripherals include a magnetic tape drive (9-track, 800/1600 BPI), two 67-megabyte disks, a line printer, and an alphanumeric terminal with CRT display. The I/O interface can reformat the data so that images are transferred in and out of the array in specific formats. These registers are built with the *multidimensional access* memories developed earlier in the STARAN computer. The MDA memory provides data buffering as well as performing some data manipulations between the PE array, the database-management machine, and the external host computer.

The MPP system has more than one operational mode. In the *stand-alone* mode, all program development, execution, test, and debug is done within the MPP system and controlled by operator commands on the user terminal. The array can transfer data in and out through the disks and tape units or through the 128-bit MPP interfaces. In the *on-line* mode, the external computer can enter array data, constants, programs and job requests. It will also receive the output data and status information about the system and the program. Data can be transferred between the MPP and the external computer at 6M bytes per second. In the *high-speed data* mode, data is transferred through the 128-bit external interfaces at a rate of 320M bytes per second.

The PEs are bit-slice processors for processing arbitrary-length operands. The array clock rate is 10 MHz. With 16,384 PEs operating in parallel, the array has a very high processing speed (Table 6.3). Despite the bit-slice nature of each PE, the floating-point speeds compare favorably with other fast number-crunching machines. Figure 6.17 shows the array unit, which includes the PE array, the associated memory, the control logic, and I/O registers. The PE array performs all logic, routing, and arithmetic operations. The *Sum-OR* module provides a zero test of any bit plane. Control signals from the array controller are routed to all

Table 6.3 Speed of typical operations in MPP

	Peak speed (mops*)
Addition of arrays	
8-bit integers (9-bit sum)	6553
12-bit integers (13-bit sum)	4428
32-bit floating-point numbers	430
Multiplication of arrays (element-by-element)	
8-bit integers (16-bit product)	1861
12-bit integers (24-bit product)	910
32-bit floating-point numbers	216
Multiplication of array by scalar	
8-bit integers (16-bit product)	2340
12-bit integers (24-bit product)	1260
32-bit floating-point numbers	373

* Million operations per second (mops)

PEs by the *fan-out* module. The *corner-point* module selects the 16 corner elements from the array and routes them to the controller. The I/O registers transfer array data to and from the 128-bit I/O interfaces, the database machine, and the host computer. Special hardware features of the array unit are summarized below:

1. Random-access memory of 1024 bits per PE
2. Parity on all array-processor memory
3. Extra four columns of PEs to allow on-line repairing
4. Program-controlled edge interconnections
5. Hardware array resolver to isolate array errors
6. A buffer memory with corner-turning capability

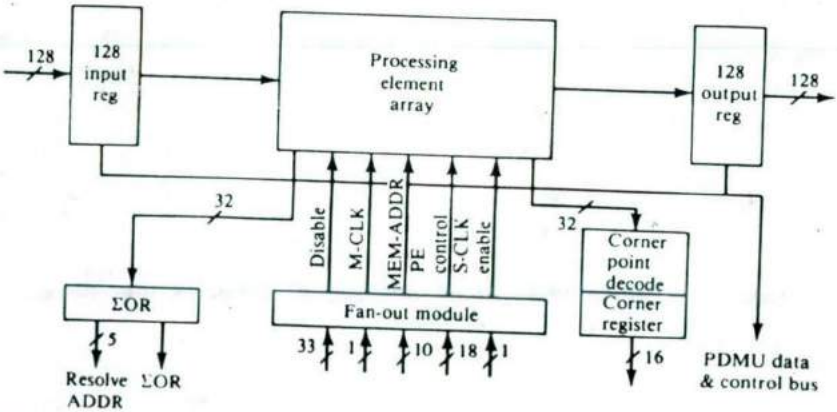


Figure 6.17 The PE array and supporting devices of the array unit. (Courtesy of Goodyear Aerospace Corp.)

Each PE in the array communicates with its nearest neighbor up, down, right, and left—the same routing topology used in the Illiac-IV. The ability to access data in different directions can be used to reorient the arrays between the bit-plane format of the array and the pixel format of the image. The edges of the array can be left open to have a row of zeros enter from the left edge and move to the right or to have the opposite edges wrap around. Since cases have been found where open edges were preferred and other cases have been found where connected edges were preferred, it was decided to make edge-connectivity a programmable function.

A topology register in the array control unit defines the connections between opposite edges of the PE array. The top and bottom edges can either be connected or left open. The connectivity between the left and right edges has four states: *open* (no connection), *cylindrical* (connect the left PE of each row to the right PE of the same row), *open spiral* (for $1 \leq n \leq 127$, the left PE of row n is connected to the right PE of row $n - 1$), and *closed spiral* (like the open spiral, but also connects the left PE of row 0 to the right PE of row 127). The spiral modes connect the 16,384 PEs together in a single linear-circuit list.

The PEs in the array are implemented with VLSI chips. Eight PEs are arranged in a 2×4 subarray on a single chip. The PE array is divided into 33 groups, with each group containing 128 rows and 4 columns of PEs. Each group has an independent group-disable control line from the array controller. When a group is disabled, all its outputs are disabled and the groups on either side of it are joined together with 128 bypass gates in the routing network.

6.3.2 Processing Array, Memory, and Control

Each PE has six 1-bit flags (A, B, C, G, P, and S), a shift register with a programmable length, a random-access memory, a data bus (D), a full adder, and some combination logic (Figure 6.18). The P register is used for logic and routing operations. A logic operation combines the state of the P register and the state of the data bus (D) to form the new state of the P register. All 16 boolean functions of the two variables P and D are implementable. A routing operation shifts the state of the P register into the P register of a neighboring PE (up, down, right, or left). The G register can hold a mask bit that controls the activity of the PE. The data-bus states of all 16,384 enabled PEs are combined in a tree or inclusive-OR elements. The output of this tree is fed to the ACU and used in certain operations such as finding the maximum or minimum value of an array in the array unit.

The full adder, shift register, and registers A, B, and C are used for bit serial arithmetic operations. To add two operands, the bits of one operand are sequentially fed into the A register, least-significant-bit first; corresponding bits of the other operand are fed into the P register. The full adder adds the bits in A and P to the carry bits in the C register to form the sum and carry bits. Each carry bit is stored in C to be added in the next cycle, and each sum bit is stored in the B register. The sum formed in B can be stored in the random-access memory and/or in the shift register. Two's complement subtaction is performed.

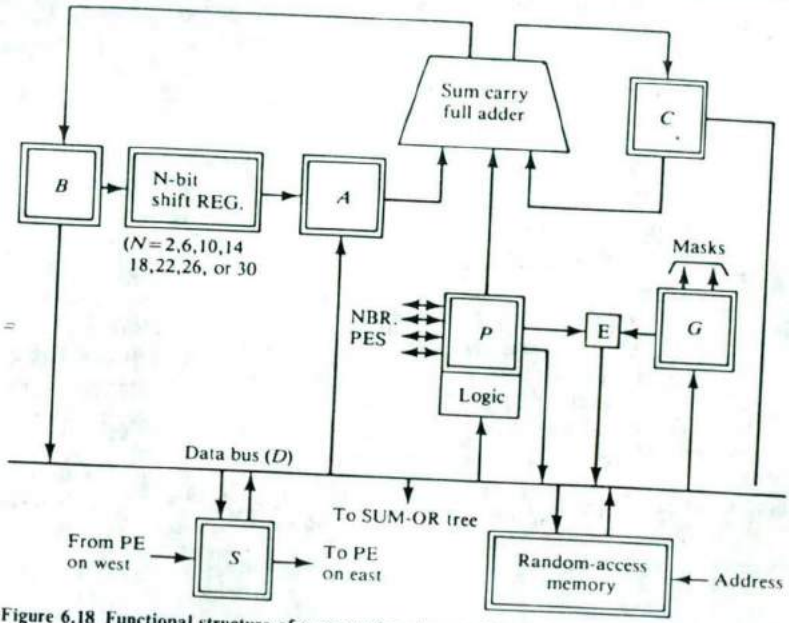


Figure 6.18 Functional structure of a processing element (PE) in the MPP. (Courtesy of *IEEE Trans. Computers*, Batcher, September 1980.)

Multiplication in the MPP is a series of addition steps where the partial product is recirculated through the shift registers A and B. Appropriate multiples of the multiplicand are formed in P and added to the partial product as it recirculates. Division in the MPP is performed with a nonrestoring division algorithm. The partial dividend is recirculated through the shift register and registers A and B while the divisor or its complement is formed in P and added to it. The steps in floating-point addition include comparing exponents, placing the fraction of the operand with the least exponent in the shift register, shifting to align the fraction with the other fraction, storing the sum of the fractions in the shift register and normalizing it. Floating-point multiplication includes the multiplication of the fractions, the normalization of the product, and the addition of the exponents.

The S register is used to input and output array data. While the PEs are processing data in the random-access memories, successive columns of input data can be shifted from the left into the array via the S registers. Although S registers in the entire plane are loaded with data, the data plane can be dumped into the random-access memories by interrupting the array processing in only one cycle time. Planes of data can move from the memory elements to the S registers and then be shifted from left to right column by column. Up to 160 megabytes/s can be transferred through the array I/O ports. Processing is interrupted for only 100 ns for each bit plane of 16,384 bits to be transferred.

The random-access memory stores up to 1024 bits per PE. Standard RAM chips are available to expand the memory planes. Parity checking is used to detect memory faults. A parity bit is added to the eight data bits of each 2 × 4 subarray of PEs. Parity bits are generated and stored for each memory-write cycle and checked when the memories are read. A parity error sets an error flip-flop associated with each 2 × 4 subarray. A tree of logic elements gives the array controller an inclusive-OR of all error flip-flops. By operating the group-disable control lines, the controller can locate the group containing the error and disable it.

Standard 4 × 1024 RAM chips are used for the PE memories. As shown in Figure 6.19, 2 × 4 subarrays of PEs are packaged on a custom VLSI CMOS-SOS chip. The VLSI chip also contains the parity tree and the bypass gates for the subarray. Each printed circuit board contains 192 PEs in an 8 × 24 array. Sixteen boards make up an array slice of 128 × 24 PEs. Five array slices (80 boards) make up the bulk of the entire PE array. The remaining 12 PE columns are packaged on 16 I/O-processor boards, which also contain the topology switches, the I/O switches, and the I/O interface registers. The 96 boards of the array are packaged in one cabinet with forced-air cooling.

Like the control unit of other array processors, the array controller of the MPP performs scalar arithmetic and controls the operation of the PEs. It has three sections that can operate in parallel, as depicted in Figure 6.20. The *PE*

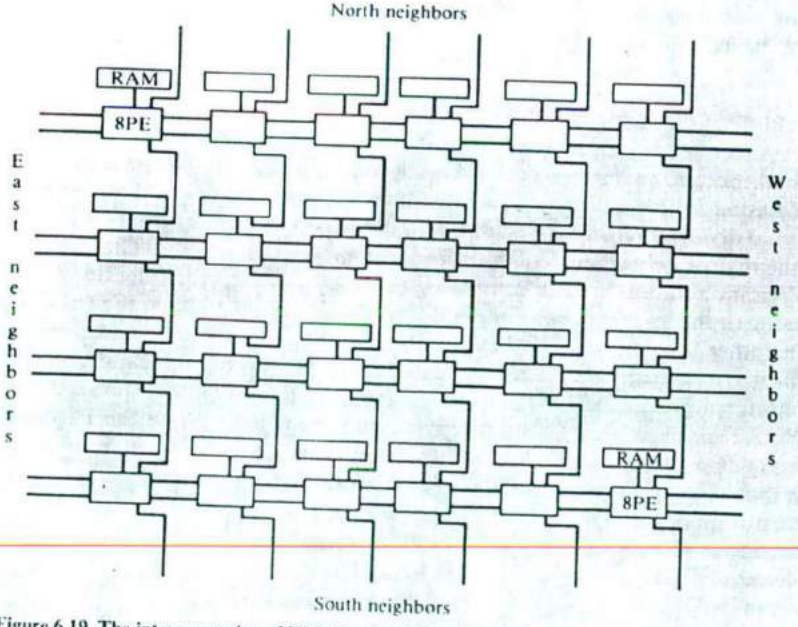


Figure 6.19 The interconnection of VLSI PE and RAM chips in the MPP array. (Courtesy of Goodyear Aerospace Corp. 1980.)

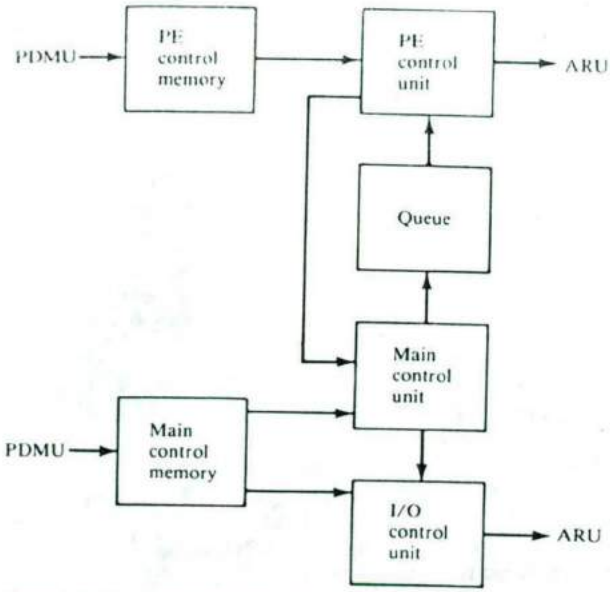


Figure 6.20 The control unit of the PE array in MPP.

control performs all array arithmetic in the application program. The *I/O control* manages the flow of data in and out of the array. The *main control* performs all scalar arithmetic of the application program. This arrangement allows array arithmetic, scalar arithmetic, and input-output to take place concurrently.

The PE control generates all array control signals except those associated with the I/O. It contains a 64-bit common register to hold scalars and eight 16-bit index registers to hold the addresses of bit planes in the PE memory elements to count loop executions and to hold the index of a bit in the common register. The PE control reads 64-bit-wide microinstructions from the PE control memory. Most instructions are read and executed in 100 ns. One instruction can perform several PE operations, manipulate any number of index registers, and branch conditionally. This reduces the overhead significantly so that PE processing power is not wasted.

The PE control memory contains a number of system routines and user-written routines to operate on arrays of data in the array. The routines include both array-to-array and scalar-to-array arithmetic operations. A queue between the PE control and the main control queues up to seven calls to the PE control routines. Each call contains up to eight initial index-register values and up to 64 bits of scalar information. Some routines extract scalar information from the array (such as a maximum value) and return it to the main control.

The I/O control shifts the S registers in the array, manages the flow of information in and out of the array ports, and interrupts PE control momentarily to

transfer data between the S registers and buffer areas in the PE memory elements. Once initiated by the main control, the I/O control can chain through a number of I/O commands. The main control is a fast scalar processor which reads and executes the application program in the main control memory. It performs all scalar arithmetic itself and places all array arithmetic operations on the PE control call queue.

The MPP being delivered to NASA uses a DEC VAX-11/780 computer as the host. The interface to the host has two links: a high-speed data link and a control link. The *high-speed data link* connects the I/O interface registers of the MPP to a DR-780 high-speed user interface of the VAX-11/780. Data can be transferred at the rate of 6 megabytes/s. The *control link* is the standard DECNET link between a PDP-11 and a VAX-11/780. The DECNET hardware and software allow the VAX users to transfer their program requests to the MPP from remote stations.

6.3.3 Image Processing on the MPP

In this section, some proposed image processing applications are described for the MPP. The intent is to familiarize our readers with the MPP instruction set, the data access in the staging memory, and parallel processing potentials of the MPP system. The speed power of the MPP promises the development of new image processing techniques, such as for real-time time-varying scene analysis. We shall restrict ourselves to the computation aspects of image processing rather than the statistical or syntactic theories behind image processing and pattern recognition.

The MPP organization described in previous sections can be functionally simplified to consist of only three major components, as shown in Figure 6.21. The PE array and the staging memory are connected by a high-speed I/O bus

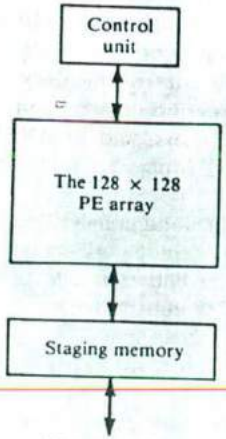


Figure 6.21 The staging memory concept in the MPP.

capable of transferring 320M bytes/s. The staging memory acts as a data buffer between the parallel and the outside world. The staging memory can accept data from conventional and special-purpose peripherals at rates up to 320M bytes/s. Its internal controller allows it to pack and reformat data so that the parallel array can more efficiently process them.

The staging memory can randomly address any individual datum but, for any given address, it fetches a block of 16K data elements and sends it to the array, where each microprocessor memory receives one datum. The exact configuration of the block of data fetched by the staging memory is under program control. For example, if k is the specific address, data are fetched from the following addresses:

$$\begin{matrix} k, & k+n, & k+2n, & \dots k+127n, \\ k+m, & k+n+m, & k+2n+m, & \dots k+127n+m, \\ \vdots & \vdots & \vdots & \vdots \\ k+127m, & k+n+127m, & k+2n+127m, & \dots k+127n+127m \end{matrix}$$

The parameters n and m as well as the address k can be specified by the programmer. Figure 6.22 illustrates the accessing of a block of 128 × 128 pixels, starting at (x, y), from a 512 × 512 image stored in the staging memory.

Instruction set of the MPP The instruction set for the MPP can be divided into three subsets: sequential, parallel, and interface. The sequential instructions are similar to those of any other sequential computer. They consist of *load*, *store*, *add*, *subtract*, *compare*, *branch*, *logical operations*, etc. Executed by the sequential controller alone, these instructions are used primarily to direct program flow and to calculate individual parameters and constants that will be broadcast to the parallel array.

The parallel instructions, also similar to conventional sequential instruction sets, consist of *load*, *store*, *add*, *subtract*, *compare*, and *logical operations*, but not *branch*. The parallel instructions are stored in the sequential controller's memory

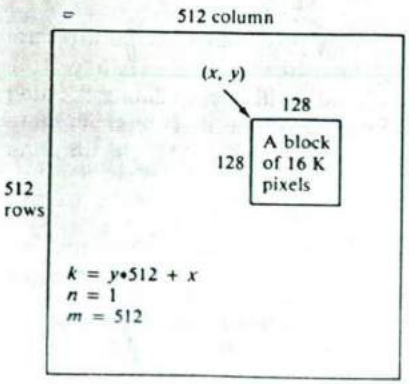


Figure 6.22 Parallel access to the staging memory. (Courtesy of *IEEE* Computer, Potter, January 1983.)

intermingled with the other instructions. When the sequential controller detects a parallel instruction, it passes it via the interface registers to the parallel array, where it is executed by all 16K processors simultaneously. This fundamental form of parallelism provides the incredible computing power of the MPP.

The MPP, like most SIMD processors, has a set of interface instructions that allows the movement of data between the sequential and parallel portions. Constants and parameters can be broadcast to each of the parallel processors by the sequential controller with a special version of the parallel instructions. But the inverse operation—moving parallel results to the sequential portion—can be more complex. The key is the ability to select a unique processor of the 16K to be active.

Each of the PEs is assigned a unique identification number. The STEP instruction selects the lowest-numbered PE to be active and thus enables it to communicate with the sequential controller. The STEP instruction can be combined with a previously executed comparison or other logical operation so that only those PEs satisfying the logical conditions are involved in the operation.

The STEP instruction can also be combined with other instructions to sequence through specified subsets of PEs. This allows the data from each PE to be processed in turn by the sequential controller and by the PE array that is under program control. Described below are several planned image processing applications of the MPP. Performance results on the MPP were not available at the time this book was produced.

Feature extraction The first step in many image-pattern recognition problems is to extract features such as edges, regions, and texture measurements on which to base classifications. Two-dimensional or areal functions such as two-dimensional convolution and correlation are frequently used to extract this information. For these situations, the 16K processors are interconnected into a grid in which each processor can communicate with its four neighbors, as illustrated in Figure 6.23. Using these interconnections, feature extraction functions can be efficiently executed.

Pattern classification The statistical classification of pixels based on multispectral data is quite straightforward in an SIMD computer like the MPP. Data for 16K pixels are input to the parallel array so that all the multispectral data associated with one pixel are stored in the memory of one processor, as illustrated in Figure 6.24. One iteration of the classification algorithm then uses the 16K classification results to be calculated.

Syntactic pattern analysis In addition to feature extraction, the parallel array can be used very effectively to guide linguistic techniques. In general, these techniques consist of a large number of production or reduction rules that must be selectively applied. If one rule is stored in each PE, then 16K rules can be updated and searched in parallel without being ordered.

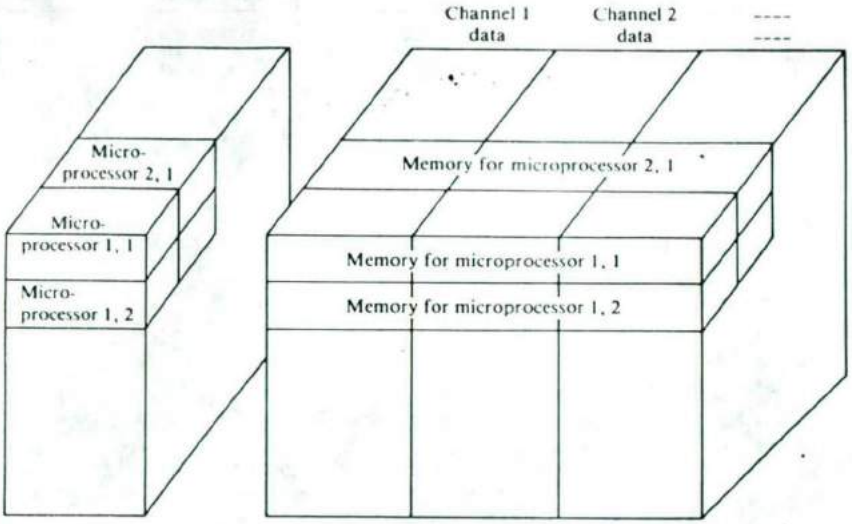


Figure 6.23 Image data storage in the bit-plane addressable MPP memory system. (Courtesy of *IEEE Computer*, Potter, January 1983.)

In the MPP and other SIMD processors, a different rule can be assigned to each PE. Consequently, when a feature is found, all of the rules can be considered in parallel to determine those that apply; the STEP instruction can then be used to sequence through the rules that require processing. Since the addition and deletion of rules in the array memory is a simple operation requiring no sorting, data packing, or garbage collection, this approach will be extremely useful in situations where the grammar is undergoing modification.

Real-time scene analysis The existence of new special-purpose image processing hardware like the MPP allows the development of new techniques for scene analysis. Since imagery can be processed in real time by these machines, the rich information content of time-varying imagery can be trapped for scene analysis. Real-time interaction with the three-dimensional world scene itself means that scene analysis need no longer involve the difficult task of producing detailed exact models of the real world from a single frame of imagery.

Capable of over 6 billion operations per second, the MPP is useful for pattern-recognition tasks such as image processing where large numbers of values must be calculated. The parallel-search aspect of SIMD computers promises to be extremely useful in more complex algorithm areas, too, such as linguistic scene analysis, where large grammars are used. The speed of the MPP not only makes real-time scene analysis possible, but also offers the prospect of real-time time-varying scene analysis with an interactive moving sensor.

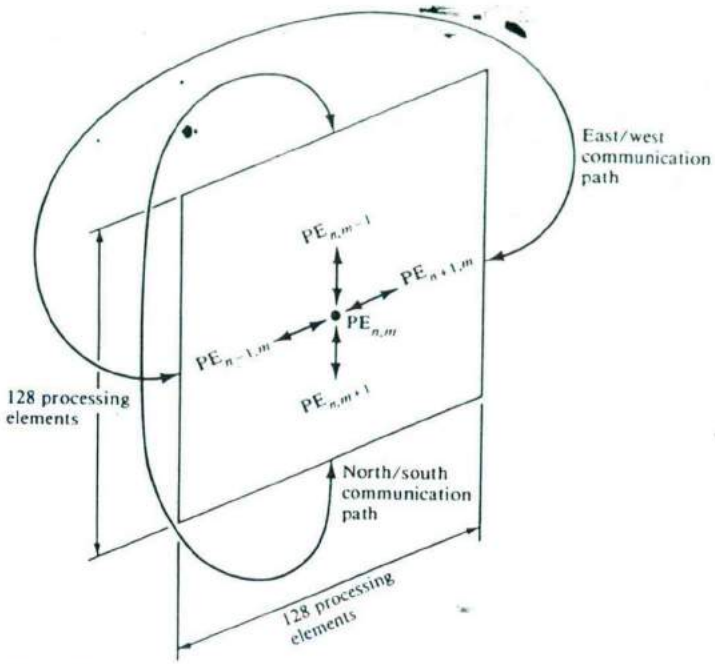


Figure 6.24 The PE-grid and end-around connection in the MPP.

6.4 PERFORMANCE ENHANCEMENT METHODS

The effectiveness of an SIMD array processor depends on the computation types, the interconnection network, the data storage schemes, the vectorization of programs, the language features, and the compiling techniques. Several performance enhancement methods are introduced below for array processors. The performance of SIMD array processors is then evaluated from the viewpoint of system throughput versus work-load distributions. Finally, we study multiple-SIMD computer organizations and related design issues.

6.4.1 Parallel Memory Allocation

An array processor is effective only for those computations that can be vectorized. The program must try to take advantage of the spatial parallelism exhibited by multiple PEs. A high-speed interconnection network is needed to route data among the PEs quickly. Furthermore, vector operands to be used by the PE must be properly stored in the data memories to allow for parallel fetch of certain specific array patterns. Array processors have been considered special-purpose computers in the sense that only matrix computations of fixed sizes can fully

explore the hardware parallelism. In order to increase application flexibility and enhance performance, efficient memory allocation schemes have been sought by many researchers and users of SIMD machines.

Consider a parallel memory system of M memory modules. Each of the M memories contains K words and has its own index register. The desirability of fetching matrix elements in rows, columns, or diagonals is clear from most matrix operations. While many computations on arrays may be formulated as row and column operations, operations on squares and blocks (submatrices) are often performed. Matrix multiplication by partitioning is an obvious example. Square (or nonsquare) submatrices become very important when an array is much larger or smaller than the number of available memory modules. One would like to have a parallel memory with as fine a resolution as possible. For example, an M module memory system can be used to access $M \times M$ arrays one row at a time or one $\sqrt{M} \times \sqrt{M}$ block at a time. In the latter case, the subarray $\sqrt{M} \times \sqrt{M}$ should be fetched in one memory cycle. Large arrays of dimension $p\sqrt{M} \times q\sqrt{M}$ can also be handled in parallel steps if p and q are integers. If p and q are not integers, then some fetches yield less than M useful array elements.

Consider the storage of a two-dimensional array of, at most, $M \cdot K$ elements. The dimensions of the original array are $P \times Q$, such that $P > M$ and $Q > K$. The standard arrangement of matrix elements is called a *matrix space*, as illustrated in Figure 6.25. We are interested in fetching M words in parallel, one word from each memory. We call any vector of M words (not necessarily one from each memory) an M vector. Among all possible M vectors, we want to determine those that can be accessed in a single memory cycle. The total number of possible M vectors equals $[MK/M]$. Any particular mapping of the MK elements into the M memory modules will allow the access of any one of K^M different M vectors in a single memory cycle. Thus, the ratio of all possible M vectors to those one can access with a particular allocation scheme is approximated below using Stirling's formula:

$$\begin{bmatrix} MK \\ M \end{bmatrix} \cdot K^M \cong \sqrt{\frac{1}{2\pi M}} \cdot e^M \tag{6.13}$$

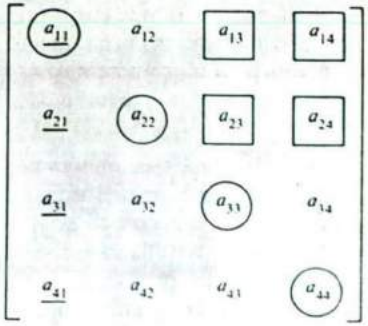


Figure 6.25 Matrix space: column 1 is underlined, a square block is in squares, and the main diagonal is circled.

This ratio means that there is no guarantee we will find one or a limited number of allocation schemes that will allow us to access an arbitrary M vector in a single memory cycle.

Define d as the distance (mod M) measured in columns between adjacent elements of a subarray in the $P \times Q$ space. For example, the elements of a row or of a diagonal have $d = 1$, the elements of a column have $d = 0$, and the even-subscripted elements of a row or diagonal have $d = 2$. If $Q \leq M$, stored arrays can be mapped into memory modules by placing the adjacent elements of rows in the same relative locations of adjacent memory modules (i.e., rows across the memory). If $Q > M$, the above mapping can be performed on partitions of Q.

Furthermore, we define s, the *skewing degree* of a storage scheme, as the distance measured in columns that each row has been shifted with respect to the row above it. All shifting must be done mod M. One value of s is used for an entire array. Figure 6.26 illustrates the storage of an array with skewing degrees $s = 0$ and $s = 1$.

The elements of an M vector fetched in one memory cycle will have the same order they had in matrix space if, and only if, $1 = d + (j - 1)s(\text{mod } M)$ for all $i \leq j$, where i and j are the subsequent rows in $P \times Q$ space from which elements are fetched. Let $j - i = r$. Note that $d + r \times s(\text{mod } M)$ is the displacement between successive elements to be fetched. If this is not one, the elements are not in successive memory modules. For each stored M vector, we define an ordered set as

$$S = \{k_i | 1 \leq k_i \leq M, 1 \leq i \leq M\} \tag{6.14}$$

The ith element of the M vector is stored in memory module k_i. Let $s' = d + r \times s(\text{mod } M)$ be the displacement between successive M-vector elements. If an M vector is relatively prime to s', then we can access the M vector in one memory cycle. It follows that if some stored M vector is not relatively prime to g, then M/g elements may be accessed at once and g fetches are required to complete the access of that vector.

If we restrict ourselves to $M = 2^L$ for some integer L, we can access rows and diagonals with $s = 0$ or rows and columns with $s = 1$. However, if $s = 1$ and $d = 1$ for diagonals and $d + s = 2$, we cannot access diagonals (Figure 6.25) in parallel. In fact, for any s, s or $s + 1$ must be even. This proves the impossibility of fetching rows, columns, and diagonals using one storage scheme when M is an even integer.

With the following nonuniform skewing scheme, it is possible to access rows, columns, and some square blocks. Suppose we choose $t = \sqrt{M} + \delta_i(\text{mod } M)$, where $\delta_i = 1$ if row index $i = k\sqrt{M} + 1$ for $k \geq 1$, and $\delta_i = 0$ otherwise. Within a strip of width $\sqrt{M}$, we can clearly access a $\sqrt{M} \times \sqrt{M}$ block in one cycle. Across these strip boundaries, conflicts arise in fetching square blocks. However, due to the additional skewing by one at strip boundaries, it is possible to access columns because t is relatively prime to M, as shown in Figure 6.26. Sometimes an access to rows or columns of a square block may be desired. Then blocks may be regarded as one memory access and the above method can be applied. This

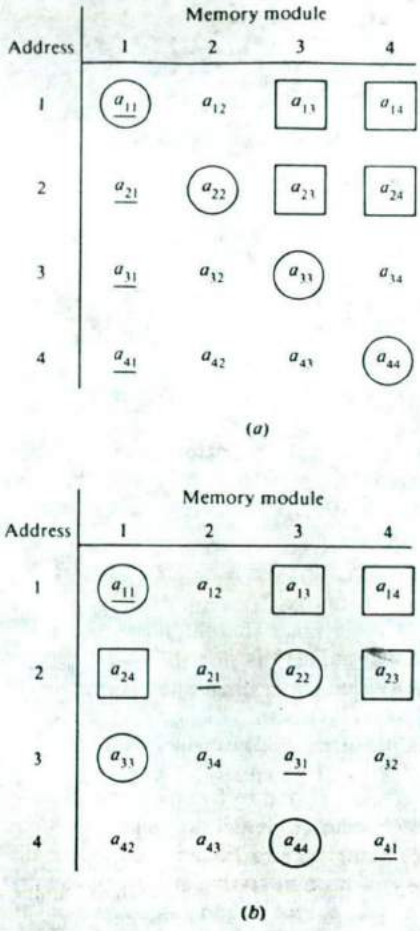


Figure 6.26 A stored array with skewing degree (a) $S = 0$ and (b) $S = 1$.

strategy may be useful in parallel memories in which each memory module produces several processor words as one superword.

In some programs, not every element of a row or column is to be accessed at each step. Instead, some index set is used to produce a partition. Let us restrict such index sets to arithmetic sequences with c being the difference between successive elements of the arithmetic sequence. Then to fetch indexed rows, for example, we have $r = 0$ and $d = c$. By the accessibility condition, c must be relatively prime to M. The only safe value for M in this case is a prime number. Consider several values of $M = 2^{2L} + 1$, such as $M = 17$ and $M = 257$, which are both prime, and $M = 65$, which is not. Thus 17, 67 (the next prime greater than 65), or 257 may be considered as good candidates for the selection of M. The number 127 is

Table 6.4 A skewed memory allocation for the prime memory

Module Address	M_0	M_1	M_2	M_3	M_4
0	00	10	20	30	x
1	50	60	70	x	40
2	21	31	x	01	11
3	71	x	41	51	61
4	x	02	12	22	32
5	42	52	62	72	x
6	13	23	33	x	03
7	63	73	x	43	53
8	34	x	04	14	24
9	x	44	54	64	74
10	05	15	25	35	x
11	55	65	75	x	45
12	26	36	x	06	16
13	76	x	46	56	66
14	x	07	17	27	37
15	47	57	67	77	x

also interesting, being an odd power of two minus one and not much larger than a perfect square.

A *storage scheme* is a set of rules which determines the module number and address within that module where a given array element is stored. We will restrict our attention to two-dimensional arrays. However, generalization of these storage schemes is simple for higher-dimensioned arrays. Described below is the storage scheme successfully developed by Lawrie and Vora (1982) for the BSP.

Example 6.4 Table 6.4 shows an 8×8 array stored in five memory modules using column major storage. Any five consecutive elements of a row, column, diagonal, etc., lie in separate modules and thus can be accessed in parallel without conflict. For example, the second through sixth elements of the first row are stored in module numbers 3, 1, 4, 2, 0, and at addresses 2, 4, 6, 8, 10, respectively.

Let M be the number of memory modules and P be the number of processors, where we assume $P < M$ and M is prime. The two storage equations $f(i, j)$ and $g(i, j)$ determine the module number and address, respectively, of element (i, j) of the array. In our case we have the following equations:

$$f(i, j) = [j \times I + i + \text{base}] \bmod M \tag{6.15}$$

$$g(i, j) = \frac{[j \times I + i + \text{base}]}{P} \tag{6.16}$$

where we assume the array is dimensioned (I, J), *base* is the base address of the array, and the number of processors P is the greatest power of two less than M.

Notice that these equations require a mod M operation where M is a prime number. They also require an integer divide by P operation. However, P is a power of two, which makes this divide easily implementable. This simplification is made possible by the "holes" shown in the table.

The number of holes in each row of the memory is equal to $M - P$ in general. For example, if $M = 37$ and $P = 32$, then $\frac{5}{37}$ of the memory is wasted. These holes could be filled with other data, e.g., scalar data, but a cleaner solution is available at the expense of an increase in the complexity of the indexing equations.

A *P vector* is defined as a set of elements of the array formed by the linear subscript equations:

$$V(a, b, c, e) = \{A(i, j): i = ax + b, j = cx + e, 0 \leq x < P \leq M\} \quad (6.17)$$

where the array is dimensioned $A(I, J)$. Thus, if $a = b = 0$ and $c = e = 1$, then the P vector ($P = 5$) is the second through sixth elements of the first row of A: $A(0, 1), A(0, 2), \ldots, A(0, 5)$. If $a = c = 2$ and $b = e = 0$, then the P vector ($P = 4$) is every other element of the main diagonal of $A(0, 0), A(2, 2), \ldots, A(6, 6)$..Note that the elements of the P vector are ordered with index x.

The index equation is defined below for the P vector V. We define $\alpha(x)$ to be the address in module $\mu(x)$ of the xth element of the P vector. Thus, combining Eqs. 6.15 through 6.17, we obtain

$$\begin{aligned} \mu(x) &= f(ax + b, cx + e) \\ &= [(cx + e) * I + (ax + b) + base] \bmod M \\ &= [rx + B] \bmod M \end{aligned} \quad (6.18)$$

where $r = a + cI$ and $B = b + eI + base$. We define r to be the *order* of the P vector and B to be the *base* address. Next, we define

$$\begin{aligned} \alpha(x) &= g(ax + b, cx + e) \\ &= \frac{[(cx + e) * I + (ax + b) + base]}{P} \\ &= \frac{[rx + B]}{P} \end{aligned} \quad (6.19)$$

It is easy to show that if r is relatively prime to the number of memory modules, then access to the P vector can be made without memory conflict.

Since it is most convenient to generate the address $\alpha(x)$ in memory $\mu(x)$, we solve for x in terms of μ and get

$$x(\mu) = [(\mu - B)r'] \bmod M \quad (6.20)$$

where $r \times r' = 1 \bmod M$. Substituting this into Eq. 6.19, we get

$$\begin{aligned} \alpha(\mu) &= \frac{\{(a + Ic)[(\mu - B)r' \bmod M] + b + eI + base\}}{P} \\ &= \frac{\{r[(\mu - B)r' \bmod M] + B\}}{P} \end{aligned} \quad (6.21)$$

Example 6.5 Consider the P vector $V(0, 0, 1, 1)$, i.e., the second through sixth elements of the first row of $A(8 \times 8)$. We have $B = 8$ and $d = 8$; thus

$$\mu(x) = [(x) \times 8 + 8] \bmod 5$$

$$\alpha(x) = \frac{[(x) \times 8 + 8]}{4}$$

Since $r' = 2$ (i.e., $2 \times 8 = 1 \bmod 5$), $B = 8$, and $M = 5$, the following addresses are obtained as shown in Table 6.4:

$$\alpha(\mu) = \frac{\{8[2(\mu - 8) \bmod 5] + 8\}}{4}$$

$$\mu(x) = (3, 1, 4, 2, 0)$$

$$\alpha(x) = (2, 4, 6, 8, 10)$$

$$\alpha(\mu) = (10, 4, 8, 2, 6)$$

The proper addresses in memories $M_0, M_1, \ldots, M_4$ are 10, 4, 8, 2, 6, respectively.

We use the $\mu(x)$ equation in the xth processor to determine the module number of the memory containing the xth element of the desired N vector. At the same time, addressing hardware in memory μ uses the $\alpha(\mu)$ equation to determine the necessary address of the desired element. We use $\alpha(\mu)$ instead of $\alpha(x)$ because this eliminates the need to route the addresses from the processors through the switch.

The design of the access conflict-free memory is based on the use of a prime number of memories. Crucial to this design is the simplification of the offset equations. Most of the mod M operations and offset calculations can be done with ROMs or with some indexing hardware. The design of this memory system fits nicely in the context of the BSP. The indexing hardware carries out the necessary addressing and alignment calculations automatically once the initial vector-set descriptors have been set up. The problem of indexing overhead and memory-access conflicts may seriously deteriorate the system performance if not properly controlled.

6.4.2 Array Processing Languages

Three high-level languages have been proposed for Illiac-IV: *Tranqual*, *Glypnir*, and *Illiac-IV Fortran*. The Tranqual is the first Algol-like language proposed for Illiac-IV. It was designed to allow programmers to manipulate arrays of data in a parallel fashion, independent of the machine organization. However, the development of Tranqual was halted by the demand for an extended Fortran for Illiac-IV. Glypnir is also an Algol-like block-structured language, but it is specially designed to be compatible with B6500 Algol in the sense that it was written to better exploit the parallelism in the Illiac-IV architecture.

All arithmetic operations are carried out under the control of a mask pattern. The mask provides 64 true-false values of a boolean vector to be associated with

each of the PEs. When a bit of the mask pattern is true, the corresponding PE is enabled and may thus deliver the results of an operation. Consider the Glypnir expression

$$A := B * C \qquad (6.22)$$

When A, B, and C are vector-valued PE variables (vectors), each may have up to 64 components. The above multiplication means that each component of B is multiplied by the corresponding element of C and the resulting product vector is stored in A. However, when C is a CU variable (a scalar), the multiplication will be repeated 64 times in an invisible PE variable.

Special facilities exist to allow the rotation and shifting of rows to the right and left in a way similar to the more familiar operations conventionally carried out on words, thus allowing the Route instruction to be utilized. **For** and **if** statements are also provided, but often give unconventional results. For instance, given the PE variables A, B, and C, the statement

$$\textbf{If } A > B \textbf{ then } C := A \textbf{ else } C := B \qquad (6.23)$$

will deliver the maximum elements of A and B to C and may result in both the **then** and **else** statements to be executed.

Blocks of assembler language can be explicitly embedded in a Glypnir program for the optimization of any section of code. It has also facilities to refer to selected hardware registers for lower-level code optimization. However, the language demands that the programmer undertakes the detailed supervision of storage allocation and be constrained to only Illiac-IV rows (64 components) or vectors of rows. To remove these restrictions, the Illiac-IV Fortran allows the user to program with vectors of any length in either "straight" or "skewed" storage allocations. Skewed allocation allows equal accessibility of rows or columns in an array.

In Illiac-IV Fortran, the binary data type can be used to specify bit-control vectors for masking purpose. The DO statement has been extended to allow parallel execution of arithmetic expressions, and extra constructs have been added to the language to allow the shifting and rotation of vectors and array rows. The only significant change are the EQUIVALENCE and COMMON statements, where the two-dimensional STORE of Illiac-IV imposes reactions on the usual serial definition.

A parallel-processing programming language *Actus* has been introduced by R. H. Perrott (1979) for array processors. Most parallel computers use extensions of existing languages, such as the extended Fortran for the Star-100, the CFT for Cray-1, and Glypnir language for the Illiac-IV. The language SL-1 is one of the few languages that has tried to bring some of the benefits of structured programming to the Star-100 system. More recently, *Vectran* has been developed by the IBM research group to facilitate the application of vector-array processing algorithms. Actus offers a theoretical extension of the language Pascal. Actus attempts to redress the technology imbalance between hardware and software development

for synchronous parallel machines. It is aimed at exploiting parallelism and incorporating some software engineering approaches. The main features in Actus are briefly introduced. The algorithmic and data constructs in Actus are of sufficient generality to make efficient use of parallel resources.

The array is declared in Actus by indicating the maximum extent of parallelism. The syntax can support any number of dimensions. For example, a scalar array is represented by:

$$\textit{var}\ \textbf{scalara}: \textit{array}[1..m, 1..n]\ \text{of integer}; \tag{6.24}$$

i.e., **scalara** contains $m \times n$ (predefined) integer numbers. The low indices are restricted to one for convenience. The maximum extent of parallelism is introduced by replacing only one pair of sequential dots ".." by a parallel pair ":"

$$\textit{var}\ \textbf{para}: \quad \textit{array}[1:m, 1..n]\ \text{of real}; \tag{6.25}$$

indicates an array **para** of $m \times n$ real numbers for which the maximum extent of parallelism is m. The array **para** can be manipulated m elements at a time since it has been declared as a parallel variable with that extent of parallelism. The array is thus the main data structure to indicate variables which can be manipulated in parallel. Thus $\alpha[1:4, 2]$ is equivalent to referencing in parallel, $\alpha[1, 2]$, $\alpha[2, 2]$, $\alpha[3, 2]$, $\alpha[4, 2]$, and $\alpha[2:3, 1]$ is equivalent to referencing in parallel $\alpha[2, 1]$ $\alpha[3, 1]$.

Identifiers can be used to represent a sequence of integer numbers. They are used to assign values to parallel variables with an extent of parallelism equal to the number of values. The form of a parallel constant is

$$\textit{const}\ \text{identifier} = \textbf{start}: (\text{increment})\ \text{finish}; \tag{6.26}$$

where the values of **start**, increment, and finish must be integers and the sequence is "**start**, **start** + increment, **start** + 2 × increment, . . . , finish." If the increment is unity, it may be omitted, e.g., "const $n = 50$; **seq** = $1:n$; **oddseq** = $1:(2)31$." Parallel constants can be used to assign values to parallel variables; for example, "**seq**" with an extent of parallelism 50 and "**oddseq**" with an extent 16.

The extent of parallelism can be changed by the use of an index set which identifies the data elements that are to be altered. The members of an index set are (ordered) integer values, each of which identifies a particular element of a data type that can be accessed in parallel. An index set is defined with the data declarations

$$\textit{index}\ \text{index} = i:j; \tag{6.27}$$

where i and j are constant integer values such that $i \leq j$. The elements i to j inclusive will be accessed whenever the index-identifier index is used as a parallel-array index.

The advantages of using index sets are that (*i*) statements become more readable since they use the identifier name, and the (*ii*) extent of parallelism involved can be evaluated before the statement is encountered. Index-set identifiers cannot be redefined, but they can be operated upon by union (+), intersection (*), and difference (−) in order to facilitate parallel computations. The complement (−) gives the other members of the declared extent of parallelism.

In order to enable the movement of data between elements of the same or different parallel variables, two primitive data-alignment operators are included in the Actus language. These are

1. The *shift operator*, which causes movement of the data within the range of the declared extent of parallelism
2. The *rotate operator*, which causes the data to be shifted circularly with respect to the extent of parallelism

A single extent of parallelism can be associated with each simple or structured statement of the language which involves one, or more than one, parallel variable; this must be less than or equal to the declared extent of parallelism for the parallel variables involved. Hence, during program execution, the smallest unit for which the extent of parallelism can be defined is the single assignment statement. This does not exclude the use of scalar and parallel variables in the same statement, but facilitates testing and data alignment of the parallel variables.

In order to avoid repeatedly indicating the extent of parallelism for a series of assignment statements in which the extent will not change, the **within** construct has been introduced. This, in turn, will avoid a calculation of the extent of parallelism for each of the statements individually. It takes the form:

$$\textbf{within}\ \textit{specifier}\ \textbf{do}\ \textit{statement} \tag{6.28}$$

where "*specifier*" is either an index-set identifier or an explicit extent of parallelism. The specifier defines the extent of parallelism for the "*statement*." This construct avoids a calculation of the extent of parallelism until another extent-setting construct is encountered or the construct is exited. If another extent-setting construct is encountered, the current extent of parallelism is stacked and the new extent evaluated and applied. This is the rule which governs the nesting of all extent-setting constructs. The **within** construct also serves another purpose when it is embedded in a loop: the *specifier* can consist of variables which are changed each time through the loop, thus, for example, enabling the examination of various subgrids within a larger grid.

To allow for those situations where selection or repetition is concerned, the structured programming concepts of **if**, **case**, **while**, and **for** were expanded to enable the test or loop variables to contain parallel as well as scalar variables. Selective statements are used to spread the extent of parallelism between two or more execution paths, as determined by a test expression in the **if** or **case** constructs.

If a test expression involves parallel variables, the test is evaluated for each indicated element of the variables. For an example:

$$\textbf{if}\ a[0{:}49] > b[0{:}49]\ \textbf{then}\ a[\#]^{*} := a[\#]- \tag{6.29}$$

In this example, 50 elements of a are tested to see which are greater than the corresponding elements of b; those elements that are greater are decremented by 1.

There are two types of repetitive statements, depending upon whether the number of times the statement is to be executed is known before the statement is

encountered or whether the number is dependent on conditions generated by the statement.

The parallel quantifiers **any** and **all** can be used with parallel test variables, in which case the extent of parallelism must be explicitly defined in the statements of the construct (as with a test which involves scalar variables only). For an example:

$$\textbf{while any } (a[1:50] < b[1:50]) \textbf{ do } a[1:50] := a[1:50] + 1 \qquad (6.30)$$

all the elements of a are incremented by one until none of the elements of a are less than their corresponding element in b.

Functions and *procedures* can be declared using the data declarations and statements previously defined; the maximum extent of parallelism of all variables must be known at compile time. The Pascal scope rules for procedures and functions apply; hence, local variables cannot have their extent of parallelism altered by a *function* or *procedure* call.

The formal parameter list for both functions and procedures was expanded to allow for parameters which are parallel variables. The actual parameters can then be either of the same extent of parallelism or a section of the same extent of a larger parallel variable. Only procedures and functions involving scalar variables may be parameters. In the case of a function, either a scalar or parallel variable can be returned as a result of its execution; the extent of parallelism can be different from that of the parameter(s). Procedures can be used to return one or more results which can be either scalar or parallel variables or a mixture of both.

The features of Actus have been described using a syntax similar to that of Pascal; this was due to a plan to use an existing Pascal compiler for its implementation at the Institute of Advanced Computation, NASA/Ames Research Center in California. The Pascal P compiler was used in the creation of the Actus compiler. This P compiler is being modified and enhanced with the new features to form an Actus P compiler which also generates code for a hypothetical stack computer. Since this code is machine independent, the Actus P compiler can be used as a basis for the implementation of Actus on other parallel machines. Preliminary results of the implementation indicate that the features of Actus can be mapped onto the instruction set of the Illiac-IV.

Another consideration in implementing the Actus language is to automate the management of the memory. It is important to determine either from the user or by the compiler the size of the working set; the *working set* is the minimum amount of the database required to be resident in the fast store so that processing can continue without excessive interruptions. On the basis of such information, the fast store can be divided into buffers and processing can be overlapped with backing store transfers. Thus, the compiler rather than the user is responsible for the organization of data transfers.

In summary, two research objectives were achieved by the Actus language and by its compiler. The first objective was achieved by introducing the concept of the extent of parallelism, whose maximum size is defined in the data declarations

and subsequently manipulated (in parallel) either in part or in total by the statements and constructs of the language. Using this concept, it was found to be possible to adapt a unified approach for both types of computers. The second objective was achieved by modifying existing data and program-structuring constructs of Pascal to accommodate the special demands of a parallel environment.

6.4.3 Performance Analysis of Array Processors

A space-time approach is used below to evaluate the performance of SIMD array processors. The execution of a vector job in an array processor occupies the equipment space of PEs over a period of the time space. An analysis of an SIMD machine with m PEs is presented below. Ideally, data operands in an array processor should be uniformly distributed among the PEMs. This effectively creates m separate data streams. Theoretically, the maximum speedup cannot exceed m when compared with a functionally equivalent SISD computer. We use the following notations to formulate expressions in measuring the *speedup*, the *throughput*, and the *utilization* of an array processor:

m: The number of PEs in an array processor
t_a: The time required by a PE to complete the execution of a broadcast instruction from the CU
n: The number of instructions to be executed in a specific job
N_i: The length of a vector operand in the ith instruction ($1 \le i \le n$)
W_a: The throughput of an array processor
t_i: The time required to finish the ith instruction in an array processor
T_a: The total time required to finish the execution of a job in an array processor
S_m: The speedup of an array processor (with m PEs) over a serial computer
ϕ: The efficiency of an array processor

Note that this list of performance parameters is very similar to those for pipeline computers studied in Section 4.5.4. The m PEs correspond to the k pipeline segments. In fact, the vector job parameters n and N_i are the same in analyzing both types of vector processors. The instruction execution time t_a of a PE is assumed to be a constant equal to the average time for typical instructions. Figure 6.27 shows the space-time diagram for the execution of the ith vector instruction on an array processor with $m = 3$ PEs, where $N_i = 10$ operands are contained in the ith vector instruction. The PE arrays are used four times to complete the execution of the 10 operations. During the fourth iteration, two PEs are disabled.

In an array processor, the time required to finish the ith instruction is computed by

$$t_i = \left\lceil \frac{N_i}{m} \right\rceil \cdot t_a \tag{6.31}$$

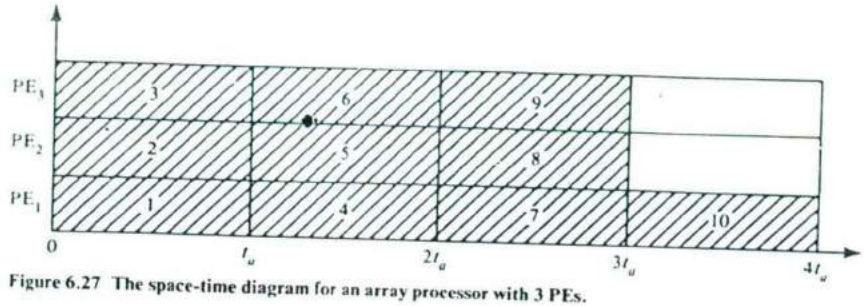


Figure 6.27 The space-time diagram for an array processor with 3 PEs.

for a vector job consisting of n vector instructions having various numbers N_i of operands for $i = 1, 2, \ldots, n$. The total time required to complete the execution of the vector job in an array processor is

$$T_a = \sum_{i=1}^{n} t_i = t_a \cdot \sum_{i=1}^{n} \left\lceil \frac{N_i}{m} \right\rceil \tag{6.32}$$

The same job if executed on a serial SISD computer requires

$$T_s = t_a \cdot \sum_{i=1}^{n} N_i \tag{6.33}$$

where t_a was assumed to be independent of instruction types (in an average sense). The improvement in speed is represented by the following speedup ratio:

$$S_m = \frac{T_s}{T_a} = \frac{\sum_{i=1}^{n} N_i}{\sum_{i=1}^{n} \lceil N_i/m \rceil} \tag{6.34}$$

The *efficiency* ϕ of an array processor is displayed by the utilization rate of the available resources (PEs). The PE utilization rate shows the percentage of PEs being actively involved in the execution of the vector job. The space-time product offers a measure of this utilization of resources. Formally, we have

$$\phi = \frac{t_a \cdot \sum_{i=1}^{n} N_i}{m \cdot t_a} = \frac{S_m}{m} \tag{6.35}$$

This shows that the efficiency ϕ corresponds to the ratio of the actual speedup S_m to the maximum speedup m. In the ideal case $\phi \to 1$, when $S_m \to m$.

Example 6.6 To illustrate the above performance measures, we choose the same vector distribution as in Figure 4.39. The mean vector length of the job distribution is 4.4. On the average, 4.4 PEs are needed to execute a vector instruction among a set of 10 instructions. Similar to that plotted in Figure 4.39, we calculate the efficiency (using Eq. 6.35) and the speedup (Eq. 6.34) of

an array processor having m PEs ($1 \leq m \leq 10$). The numerical results are plotted in Figure 6.28. The speedup increases monotonically with respect to the number of available PEs, whereas the efficiency declines with the increase of PEs.

For an array processor, not only the vector length but also the residue of the vector length will affect the system performance. For example, to execute a vector instruction of length 65 on the Illiac-IV computer with 64 PEs requires one additional instruction cycle to execute the residue of one component operation. The system performance will be degraded due to small residues. Degradation results mainly from many idle PEs. Figure 6.29 shows the speedup (utilization) against vector length on an array processor with eight PEs. Here we consider only the execution of a single vector instruction ($n = 1$). The maximum speedup is achieved when N_1 is a multiple of $m = 8$.

For small residues, the speedup drops rapidly. When N_1 approaches infinity, the ill effect of residue becomes less severe. The PE utilization will approach one when the vector length goes to infinity, as shown by the envelope of the saw-tooth curve. Parallel algorithms, good memory-allocation schemes, and optimizing compilers are needed in array processors. Whether a large array processor will perform as ideally projected depends heavily on the skill of the users. The failure of promoting the Illiac-IV and the BSP into the commercial market was mainly due to user reluctance in accepting SIMD computers for general-purpose applications.

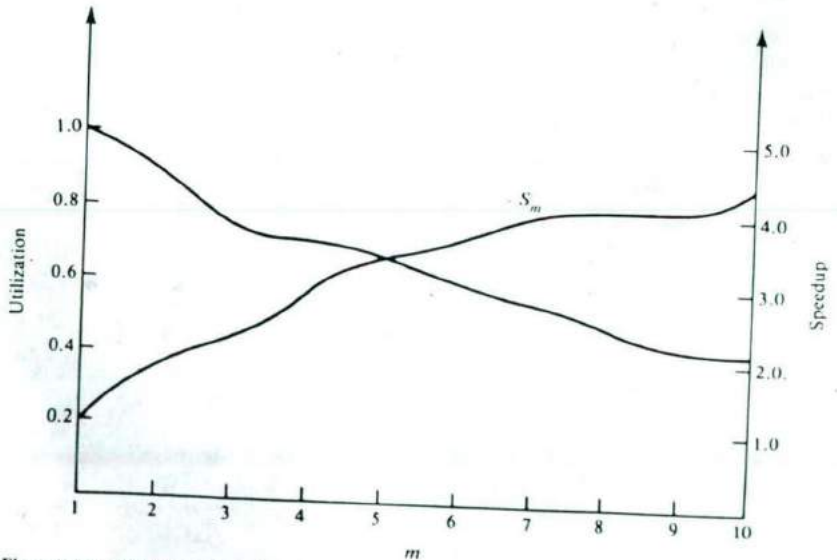


Figure 6.28 Efficiency and speedup of an array processor with various number of PEs.

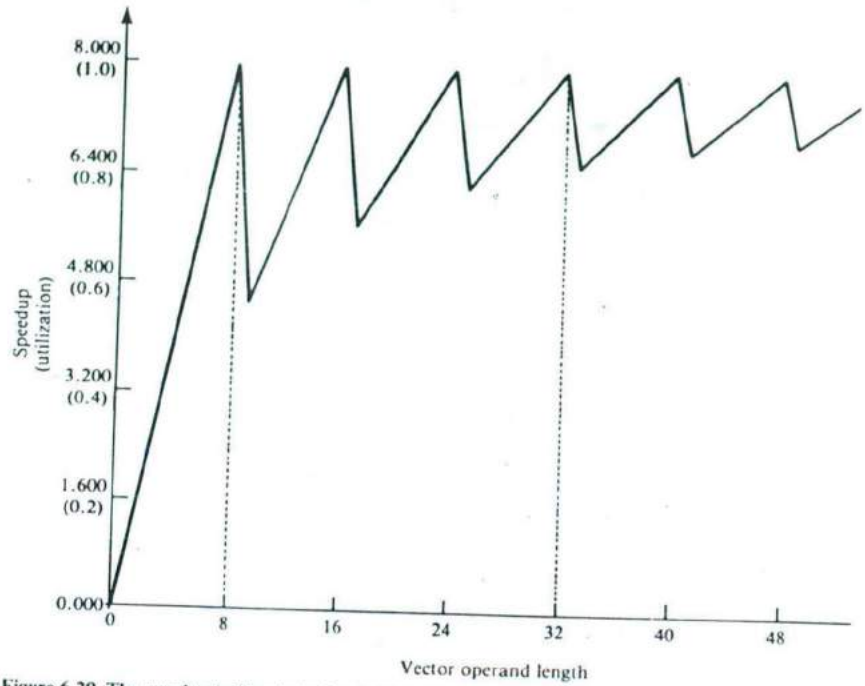


Figure 6.29 The speedup (utilization) of an 8-PE array versus vector length.

6.4.4 Multiple-SIMD Computer Organization

A shared-resource array processor consists of two or more CUs sharing a pool of dynamically allocatable PEs. Such a parallel computer is depicted in Figure 6.30. The system operates with *multiple single-instruction multiple-data* (MSIMD) streams. Each CU must be allocated with a subset of PEs for the execution of a single vector job (an SIMD process). The only way vector jobs can interact with each other is through their independent use of the same PE resources.

Examples of MSIMD array processors include the original Illiac-IV design with four CUs sharing 256 PEs (Figure 6.31), the Multi-Associative Processor (MAP) with eight CUs sharing 1024 PEs, and the PM^4 system proposed at Purdue University (Figure 6.32). The PM^4 was proposed as a reconfigurable computer system which can operate in either MSIMD mode, multiple SISD mode, or in MIMD mode. A typical configuration of the PM^4 consists of 16 CUs with, say, 1024 PMUs. The system was intended to be used for computer imaging applications. The PM^4 has been upgraded to a generalized research multiprocessor system, called PUMPS, at Purdue University. In the Phoenix computer project, the coupling of sixteen 64-PE arrays is being considered to extend the Illiac-IV design for MSIMD array processing.

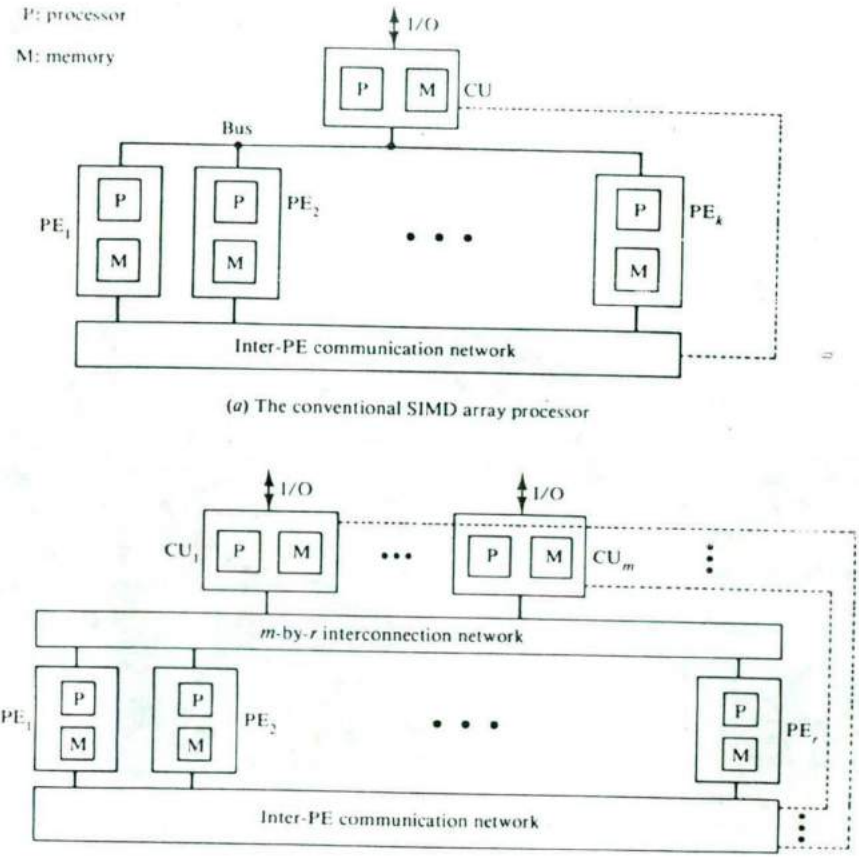
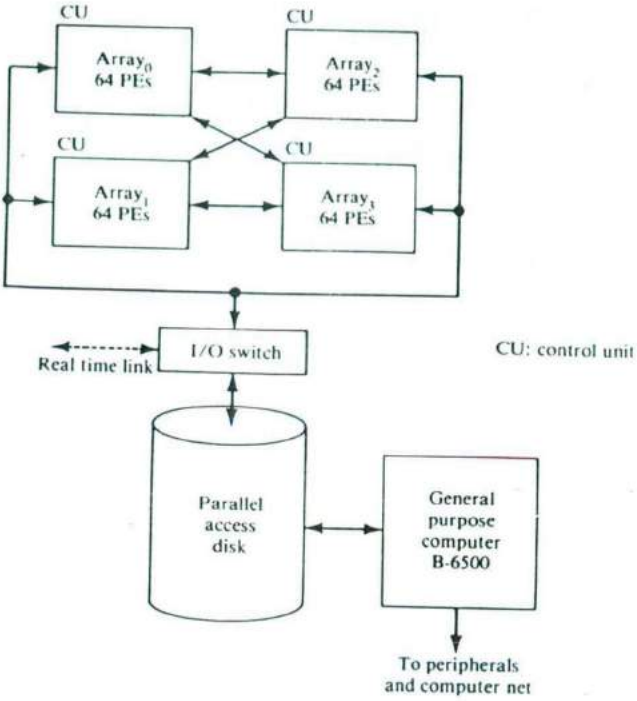
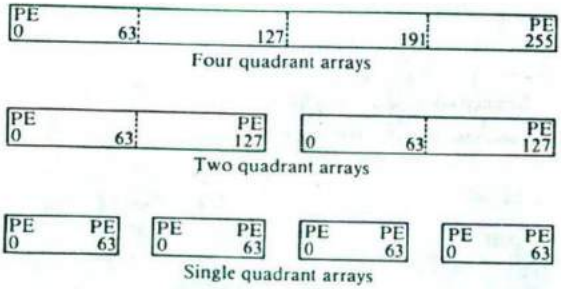


Figure 6.30 SIMD versus MSIMD computer organization. (Courtesy of *IEEE Trans. Computers*, Hwang and Ni, September 1980.)

There are three possible multiarray Illiac configurations, as shown in Figure 6.31. This provides some flexibility in matching array size to problem (matrix) size. The end connection of the PEs in each array can be reconnected to the ends of other arrays to form dual two-quadrant arrays with 128 PEs each, or to form a single four-quadrant array of 256 PEs. For multiarray configurations, all CUs receive the same instruction string and any data centrally accessed. The CUs execute the instructions independently. Inter-CU synchronization takes place only on those instructions in which data on control information must cross the array boundaries. The multiplicity of array configurations introduces additional complexities in program control and memory addressing.



(*a*) The system structure



(*b*) Multiconfigurations

Figure 6.31 The original Illiac IV design and multiple configurations. (Courtesy of *IEEE Trans. Computers*, Barnes, et al., August 1968.)

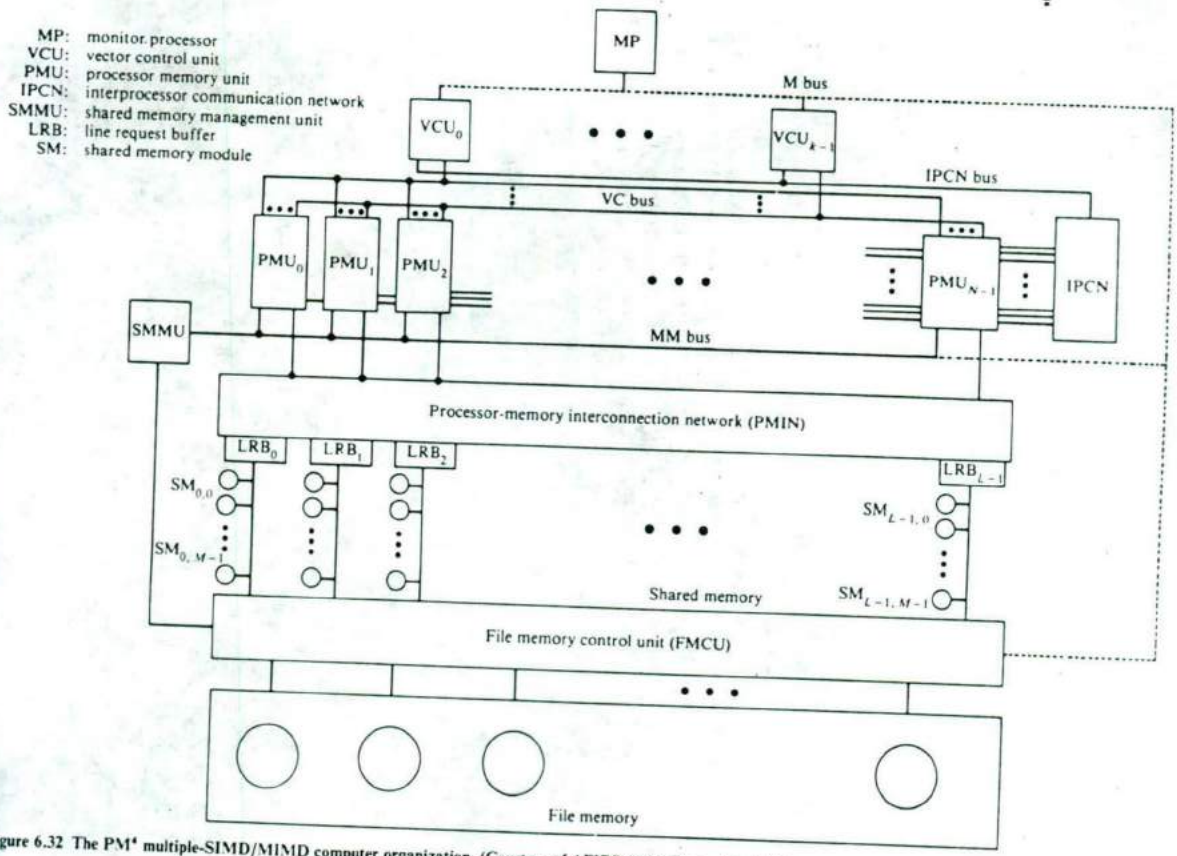


Figure 6.32 The PM⁴ multiple-SIMD/MIMD computer organization. (Courtesy of AFIPS *NCC Proc.*, June 1979.)

The scheduling of PE resources in an MSIMD computer can be modeled by the queuing network in Figure 6.33. There are m identical CUs in the system, each of which handles an instruction stream. An available CU enters the busy state only if there is a vector job and the PEs demanded by that job are available from the PE pool, otherwise it remains in the wait state. There are n identical PEs shared by all the CUs. Each subarray of PEs can be allocated to one CU through a partitionable interconnection network. All instructions are stored in the CU memory and decoded by the CUs. Only the vector-type instructions are broadcast to the allocated subarray of PEs for execution. Control-type or scalar instructions are directly executed through the CUs. All the PEs can independently accept one assignment at a time. The local memory of each PE is used to store only the distributed data sets to be used for SIMD processes.

The subset of PEs allocated to a CU may vary in size for different jobs. In the case of the original Illiac-IV design, the three multi-array configurations (i) {64, 64, 64, 64} (4 active CUs), (ii) 128, 128 (2 active CUs), and (iii) 256 (1 active CU) corresponds to four, two, and one instruction streams, respectively. Note that some of the CUs may be left idle due to the limited number of available PEs in the resource pool. In the Phoenix extension, the increased reconfiguration capability offers better application flexibility than the Illiac-IV. The PE array partitioning in the Illiac-IV is called *block* reconfigurable. Each block of PE corresponds to one quadrant of the array. In a generalized MSIMD computer, all the PEs should be

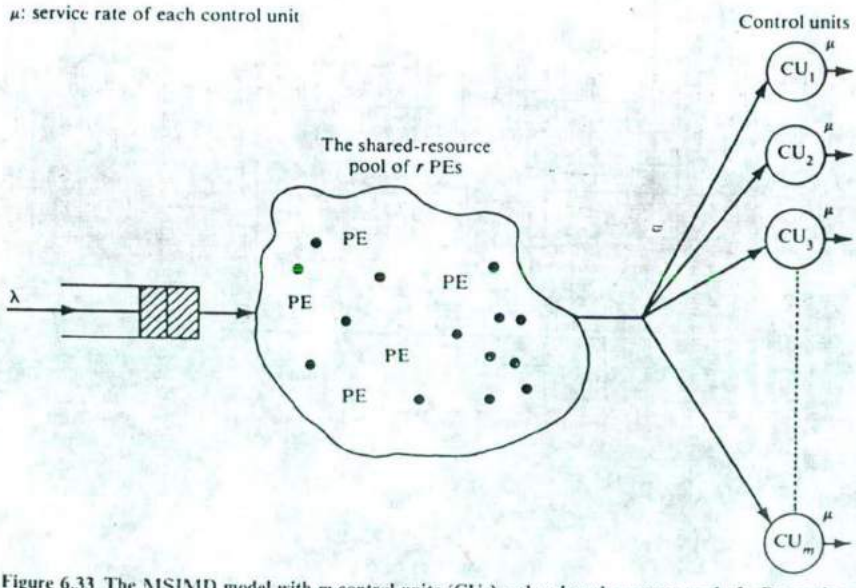


Figure 6.33 The MSIMD model with m control units (CUs) and a shared-resource pool of r Processing Elements (PEs). (Courtesy of *IEEE Trans. Computers*, Hwang and Ni, September 1980).

dynamically partitionable into subarrays of various sizes. The PM^4 was proposed to operate in this fashion.

Several research issues of MSIMD computers are identified below. The goal is to design MSIMD computers for multiple array processing in an interactive manner. These design issues must be properly addressed in order to achieve high performance at reasonable system cost. The operating system of an MSIMD computer is much more complicated than that of a single SIMD machine.

Performance optimization The performance of an MSIMD computer is measured by the utilization rates of the CUs and PEs and by the average job response time. The computer is modeled in Figure 6.30 as a multiserver queueing system. Based on reasonable assumptions of the job arrival distribution, the CU service rate, and the queueing discipline used, one can predicate the expected performance.

The optimal choice of the size of the PE resource pool is a fundamental issue for a given number of CUs in the system. The selection depends on the input workload distribution, the cost of the PEs and their interconnection networks, and the queueing discipline used in allocating the shared PEs to the CUs. The resource allocation can be optimized by promoting the CU-PEs utilization rates at reduced system cost. The results obtained from theoretical queueing analysis need to be verified with machine simulation experiments before meaningful system sizes can be decided for specific application problem environments.

Partitioning the routing network The allocation of PE blocks to multiple CUs demands the use of some partitionable inter-PE routing networks. With a large number of PEs, the network size may become too large to be cost effective. With partitioning, smaller networks may be interconnected to route the data at much lower hardware cost. Of course, the partitionability from large permutation networks to multiple but disjoint subnetworks depends on the network type, the increased delay and blocking rate after partitioning, and the emulation capabilities among reconfigured network topologies. Without a partitionable interconnection network, the MSIMD computer cannot operate flexibly and efficiently.

Algorithms and system control Multiple SIMD operations need to decompose a large computation problem by algorithmic partitioning. Of course, the algorithm decomposition must be constrained by the hardware resource configuration and availability. Furthermore, none of the proposed array-processing languages can handle multiple data arrays. Designing an effective operating system to coordinate the multiple CU operations is a very challenging effort. The experience in developing an MSIMD operating system will help develop an intrinsic MIMD operating system, especialy for scientific applications.

Due to inexperience in the aforementioned problem areas, MSIMD machines are still in the proposal stage among computer architects. Extensive research and development efforts are necessary towards the production of multiple array processing systems. MSIMD computers are more suitable for regularly structured scientific computations.

6.5 BIBLIOGRAPHICAL NOTES AND PROBLEMS

The original SIMD concept can be traced to Unger (1958) and Slotnick, et al. (1962). The structure of the original 256-PE Illiac-IV computer was first reported in Barnes, et al. (1968). The Illiac-IV software and applications programming was reported by Kuck (1968). A more recent report of the Illiac-IV system was given in Bouknight, et al. (1972). The Actus language extensions for array processors are based on the work of Perrott (1979). Assessments of the first-generation vector processors, including the Illiac-IV, were given by Stokes (1977) and Theis (1974). The language Glypnir was described in Lawrie, et al. (1975). The Fortran-like language CFD was reported by Stevens (1975). Programming experiences on the Illiac-IV are summarized in Stevenson (1980). Possible extension of the Illiac-IV to the Phoenix array processor was discussed in Feierbach and Stevenson (1978).

The Burroughs Corporation has published a series of technical notes on the BSP architecture (1978a, b, c). The BSP was comprehensively reported in Kuck and Stokes (1982). Arithmetic design of the BSP was reported in Gajski and Rubinfield (1978). The prime memory system is based on the work of Lawrie and Vora (1982). The MPP has been reported in Batcher (1980). Detailed design features of the MPP are described in a final report by Goodyear Aerospace Corporation (1979). The image processing applications of the MPP are reported by Potter (1983). The skewed allocation of parallel memories is based on the work of Budnik and Kuck (1971). A good summary of BSP features is given in Kozdrowicki and Theis (1980), where comparisons of the BSP to Cyber-205 and Cray-1 are provided. The throughput analysis of array processors is based on the comparative study by Hwang, et al. (1981). Multiple SIMD computer organizations are modeled in Hwang and Ni (1979, 1980). The partitioning of permutation network for MSIMD machine has been studied in Siegel (1980).

Problems

6.1 Explain the following system features associated with the Illiac-IV, the BSP, and the MPP array processors.

(*a*) Multi-array configurations of the Illiac-IV
(*b*) The prime memory for the BSP
(*c*) The bit-slice operations in the MPP
(*d*) Concurrent scalar-array operations in the BSP
(*e*) Concurrent I/O and arithmetic logic operations in the MPP array
(*f*) The staging memory configurations in the MPP
(*g*) Host computers for the Illiac-IV, the BSP, and the MPP
(*h*) The I/O facilities in the Illiac-IV, the BSP, and the MPP

6.2 Prove that the Illiac recirculating network cannot be partitioned into independent subnetworks, each of which would have the properties of a complete Illiac network. (Hint: Use the rotating functions defined in Section 5.2.2.)

6.3 Devise an SIMD algorithm for finding the inverse of an 8×8 triangular matrix $A = (a_{ij})$ on the Illiac-IV computer with 64 PEs. Show the memory allocation and CU instructions needed to implement the algorithm. Minimizing the number of instruction steps and the required data memory words is the primary design goal. The given matrix A is assumed to be nonsingular. The successive contents of the involved PE registers and of the PEMs should be demonstrated along with the instruction steps. Masking can be used to enable and disable the PEs.

6.4 Using the vector instruction forms specified in Table 6.2, devise a BSP program for solving a linear triangular system of algebraic equations with n unknowns. You can assume $n \gg p = 16$, the number of arithmetic elements in the BSP. A "block" back substitution method is suggested to solve the triangular system. Memory allocation for the characteristic matrix must be specified among the $m \times 17$ memory modules.

6.5 Given an $n \times m$ image, the gray level for each pixel (picture element) is between 0 and $b - 1$. Let $A[i, j]$ denote the gray level at the pixel (i, j). An algorithm to construct a histogram in an SISD computer is shown below:

```
For i = 0 to b − 1 do
    Histog(i)←0;
For i = 1 to n do
  For j = 1 to m do
    Histog(A[i, j])←Histog(A[i, j]) + 1;
```

Suppose we want to use an SIMD machine with p PEs to construct the histogram. Assume $n, m \gg p$, n, m, and $p(p = 2^y)$ are powers of 2, and $n/p = k$ is an integer. Each PEM stores k rows of image data, e.g., PEM_0 stores rows one to k, etc. The storage formats are shown in Figure 6.34, where

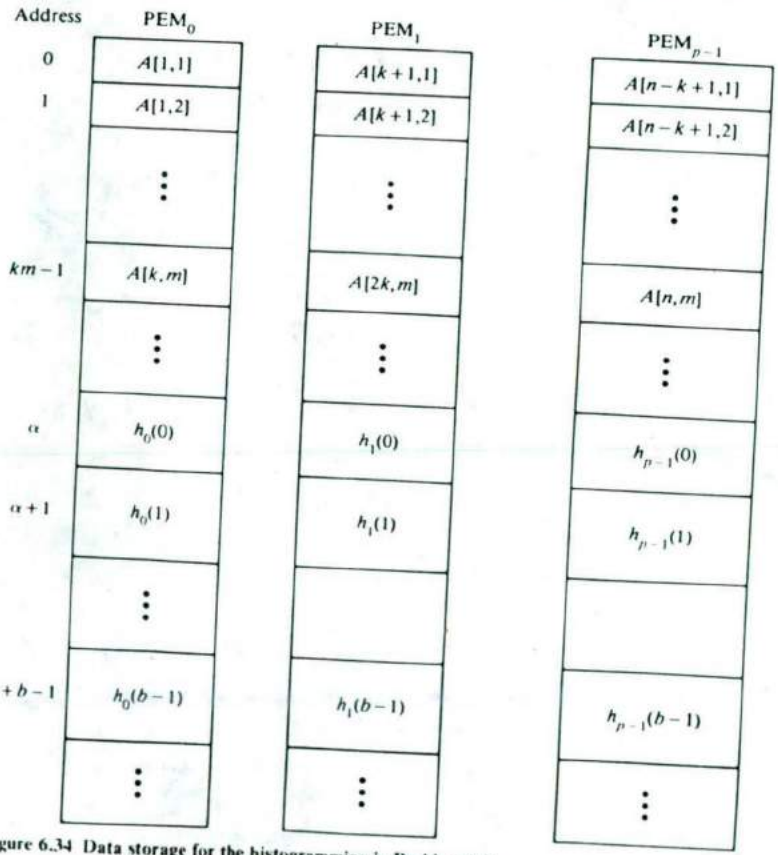


Figure 6.34 Data storage for the histogramming in Problem 6.5.

Table 6.5 Sample vector-scalar instruction set for Problem 6.5

VLOAD r (3 cycles)	Indexed addressing $(r) \leftarrow ((D_i) + (I_i))$ $r \in \{A_i, B_i, C_i, R_i\}$	VADDI r, # (3 cycles)	Immediate adding $(r) \leftarrow (r) + \# \, r \in U_i$
VMOV r_1, r_2 (1 cycle)	Register transfer $(r_1) \leftarrow (r_2)$ $r_1, r_2 \in U_i$ where $U_i = \{A_i, B_i, C_i, R_i, I_i, R_i\}$	LCYCLE r (s cycles)	Left cyclic shift $(R_{i-s}) \leftarrow (R_i)$ where $s = 2^d$, $d = (r)$ $r \in \{A_i, B_i, C_i\}$
VMVI r, # (2 cycle)	Immediate operand $(r) \leftarrow \#$ $r \in U_i$	MVI INX, # (1 cycle)	Immediate operand (INX) $\leftarrow \#$ INX $\in Z$, where $Z = \{\text{INX}_1, \text{INX}_2, \ldots, \text{INX}_s\}$
VSTORE r (3 cycles)	Indexed addressing $((D_i) + (I_i)) \leftarrow (r)$ $r \in \{A_i, B_i, C_i, R_i\}$	ICR INX, 1 (1 cycle)	Increment index register (INX) $\leftarrow$ (INX) + 1 INX $\in Z$
VADD r_1, r_2 (2 cycles)	Register adding $(r_1) \leftarrow (r_1) + (r_2)$ $r_1 r_2 \in U_i$	JLT $\text{INX}_i, \text{INX}_j$, LOOP (2 cycles)	Conditional branch if $(\text{INX}_i) < (\text{INX}_j)$ then go to LOOP $\text{INX}_i, \text{INX}_j \in Z$

the memory locations from $\alpha + b - 1$ in each PEM are used to store a local histogram. The method to construct the histogram consists of forming a local histogram in each PE and then combining local histograms to get a global histogram.

The organization of each PE_i is shown in Figure 5.2. Each PE_i can communicate with its neighboring PE_{i+1} and PE_{i-1} in one routing step. You are allowed to use the vector instructions and scalar instructions listed in Table 6.5. The number below each instruction indicates the instruction cycle time. Also, we assume that there are five global index registers in the control unit.

(*a*) Using these instructions, write a program to construct a histogram in the SIMD machine. The resulting data should be stored in the memory locations $\alpha, \alpha + 1, \ldots, \alpha + b - 1$ in PEM_0.

(*b*) Compute the total number of cycles required in your program. What is the speedup of your program over a conventional one in an SISD computer, which consists of the CU and one PE? Assume the scalar counterpart of each vector instruction consumes the same time as each vector instruction. Note that there is no communication cost in an SISD computer.

6.6 Consider the evaluation of the following inner-product expression in an SISD machine with one PE or in an SIMD machine with m PEs connected by a linear circular ring:

$$S = \sum_{i=1}^{n} A_i \cdot B_i \tag{6.36}$$

It is assumed that each ADD operation requires two time units and each MULTIPLY operation requires four time units. Data shifting along the bidirectional ring between any adjacent PEs requires one time unit.

(*a*) What is the evaluation time of S on the SISD computer?

(*b*) What is the evaluation time of S on the SIMD computer?

(*c*) What is the speedup of using the SIMD machine over the SISD machine for the evaluation of S?

6.7 Consider K couples of vectors. The ith couple consists of a row vector R_i and a column vector C_i, each of dimension $N = 2^n$. To compute the pairwise inner product for the ith couple, we perform the following:

$$IP[i] = \sum_{j=1}^{N} R_i[j] * C_i[j] \tag{6.37}$$

Below is the algorithm to perform $IP[i]$ for all $i = 1, 2, \ldots, K$.

```
For i ← 1 to K do
begin
IP[i]←0;
  For j← 1 to N do
  IP[i]←IP[i] + R_i[j]*C_i[j].
end
```

(*a*) Neglecting the initialization step, index updating and testing, find the total compute time on a uniprocessor as a function of K and N. Assume that multiplication and addition take the same unit time to complete.

(*b*) To speedup this computation, an SIMD machine can be used by exploiting the parallelism in the computation. Two different implications are suggested below. Find the compute time in each case.

(i) Use $P = N$ processing elements (PEs) to compute $IP[i]$ successively for each couple of vectors R_i, C_i.

(ii) A couple of vectors are allocated to each PE, which computes one inner product. The number of PEs is $P = K$ in this case.

6.8 Consider an SIMD machine with 256 PEs using a perfect shuffle interconnection network. If the shuffle interconnection function is executed 10 times, where will the data item originally in PE_{123} be located?

6.9 We have learned that an SIMD machine with $P = 2^{2n}$ PEs can access without conflict the rows, columns, diagonal, and reverse diagonal of a matrix from $M = 2^{2n} + 1$ parallel memory modules, if the skewing distance $S = 2^n$. Prove that it is also possible to access any 2^n-by-2^n square block in one memory cycle under the same condition.

6.10 Table 6.4 shows the skewed memory allocation for an 8×8 matrix in an array processor with $M = 5$ memory modules and $P = 4$ processors.

(*a*) List all patterns that can be accessed in one memory cycle.

(*b*) Give a P vector $V(1, 1, 1, 1)$. Calculate the word addresses in the memory modules involved.