VLSI Array Processors

S. Y. Kung*

High speed signal processing depends critically on parallel processor technology. In most applications, general-purpose parallel computers cannot offer satisfactory real-time processing speed due to severe system overhead. Therefore, for real-time digital signal processing (DSP) systems, special-purpose array processors have become the only appealing alternative. In designing or using such array processors, most signal processing algorithms share the critical attributes of regularity, recursiveness, and local communication. These properties are effectively exploited in innovative systolic and wavefront array processors. These arrays maximize the strength of very large scale integration (VLSI) in terms of intensive and pipelined computing, and yet circumvent its main limitation on communication. The application domain of such array processors covers a very broad range, including digital filtering, spectrum estimation, adaptive array processing, image/vision processing, and seismic and tomographic signal processing. This article provides a general overview of VLSI array processors and a unified treatment from algorithm, architecture, and application perspectives.

1. INTRODUCTION

VLSI and Digital Signal Processing—A Symbiotic Relationship

The practicality of algorithms for many Digital Signal Processing (DSP) applications will ultimately be determined by their computational feasibility. It depends critically—particularly, for real-time signal processing—on the parallel processing capabilities, in both speed and volume, offered by state-of-the-art computing machines. The availability of low-cost, high-density, fast-speed VLSI devices, and of the emerging computer-aided design facilities, presages a major breakthrough in the future design of massively parallel processors. Current parallel computers incur severe system overhead; therefore, VLSI-oriented array processors are most appealing for high-speed signal processing. The design of such systems requires, however, a broad knowledge of the relationship betweeen parallel computing algorithms and the structures of array hardware and software.

The basic discipline in a top-down design methodology (as depicted in Figure 1-1) depends on a fundamental understanding of algorithm, architecture, and application. The boundary between software and hardware has become increasingly vague in the environment of VLSI system design. This enhances the already prevailing roles of the algorithm analyses and the mappings of algorithms to architectures. Therefore, a very broad spectrum of innovations will be required for obtaining highly parallel array processing. These innovations will include new ideas on communication/computation trade-offs, parallelism extractions, array architectures, programming techniques, processor/structure primitives, and numerical performances of DSP algorithms.

This article presents the major principles of the design of VLSI array processors for real-time signal/image processing requirements. In Section 1, the major impacts of VLSI technologies are reviewed. A special focus is placed on communication problems inherent in VLSI. VLSI-oriented array processors that circumvent the communication constraints are discussed. In Section 2, several types of array processors important to DSP applications are proposed. In Section 3, mappings of algorithms onto array architectures are addressed, together with the issues of coping with the communication constraints and improving pipelining rates. In Section 4, new array algorithmic criteria and analyses tailored to the array processing environment are presented. In Section 5, the implementation and design considerations of DSP array processor systems are discussed. Finally, an example of DSP applications of array processors is given in Section 6, followed by a concluding remark in Section 7.

1.1. Impacts of VLSI device technology

VLSI architecture enjoys the major advantage of being very scalable technologically [17]. This means that the efforts of architecture redesign will be very minor when the device technology is scaled down to ultra-submicron level. However, as chip size is increased, the interconnection problems will become very severe. Before long, chip cost, performance and speed will be determined primarily by interconnect delay and area. Therefore, VLSI device technology does not simply offer a promising future, but also creates some new design constraints. For example, the modularity of building blocks and the alleviation of the burden of global interconnection are often essential requirements in VLSI design.

Scaling effects

In the scaling of geometry, we often assume that all the

*This research was supported in part by the National Science Foundation under Grant ECS-82-13358, and by the Army Research Office under Grant DAAG29-79-C-0054.

dimensions, as well as the voltages and currents on the chip, are scaled down by a factor α. (A value of α greater than 1 implies that sizes or levels are shrinking). When scaling down the linear dimensions of a transistor by α, the number of transistors that can be placed on a chip of given size scales up by α^2. Figure 1–2 depicts the effect of scaling down a conductor and a MOSFET transistor by a factor α.

If the average interconnection length is not scaled down with the same factor α, the interconnection delay may actually increase. When the delay time of the circuit depends largely on the interconnection delay (instead of the logic gate delay), minimality and localization of interconnections will become essential factors for an effective realization of the VLSI circuits.

1.2. VLSI Architectural Design Principles

VLSI architectures should exploit the potential of the VLSI technology and also take into account (i) the layout constraint and the resultant interconnection costs in terms of area and time, and (ii) the cost of VLSI processors as measured by silicon area and pin count. VLSI architecture design strategies stress modularity, regularity of data and control paths, local communication, and massive parallelism. Some design principles are summarized below.

Principle of Homogeneity

In VLSI, there is an emphasis on keeping the overall architecture as *regular* and *modular* as possible, thus reducing the overall complexity. For example, memory and processing power will be relatively cheap as a result of high regularity and modularity. Even in the communication or wiring, a careful algorithmic study may help create some form of regularity. This depends on special arrangements, realized in the course of topological mappings from algorithms to architectures.

Principle of Pipelining

In many DSP applications, throughput rate often represents the overriding factor dictating system performance. In order to optimize throughput, a different design choice is often made than that of minimizing the total processing time (latency). Pipeline techniques fit naturally in our aim of improving throughput rate. Suitable pipelining techniques are now well-established, particularly for most signal processing algorithms. A prominent example is the systolic/wavefront array discussed in Section 2.

For signal processing arrays, pipelining at all levels should be pursued. It may bring about an extra order of magnitude in performance with very little additional hardware. Although most of the current array processors stress only word level pipelining, the new trend is to exploit the potential of multiple-level pipelining (i.e., combined pipelining in all the bit-level, word-level, and array-level granularities).

Principle of Locality

The principle of locality is seen at every level of VLSI design [18]. In systolic arrays, both spatial locality and temporal locality are stressed [9]. The notion of locality can have two meanings in array processor designs: *localized data transactions* and *localized control flow*. In fact, most recursive signal processing algorithms permit both locality features; and they are fully exploited in the design of wavefront arrays, as we shall elaborate in Section 2.

Communication is the Key Factor

In VLSI technology, computations per se are becoming easily affordable. Therefore, the most critical factor in VLSI design is *communication*. Architectures that balance communication and computation, and that circumvent communication bottlenecks with minimum hardware cost, will eventually play a dominating role in VLSI systems.

2. VLSI ARRAY PROCESSORS

Until recently, computation-intensive tasks were handled by high performance supercomputers, including pipelined computers, array processors, and multiprocessor systems. The development of these computer systems has involved a thorough exploration of parallel computing, efficient programming, and resource optimization. However, the general-purpose nature of these machines has led to a complicated system organization and severe system overheads. These machines are not suitable for real-time signal processing where a very high throughput rate is absolutely essential.

A solution to the real-time requirement of signal processing is to use special-purpose array processors, and to maximize the processing concurrency by either pipeline processing or parallel processing or both. As long as communication in VLSI remains restrictive, locally-interconnected arrays will be of great importance. An increase of efficiency can be expected if the algorithm arranges for a balanced distribution of work load while observing the requirement of locality, i.e. short communication paths. These properties of load distribution and information flow serve as a guideline to the designer of VLSI algorithms, and eventually lead to new designs of architecture and language.

The first such special-purpose VLSI architectures are *systolic* and *wavefront* arrays, which boast tremendously massive concurrency. The concurrency in the systolic/wavefront arrays is derived from *pipeline processing* or *parallel processing* or both. These types of processing are illustrated in Figure 2–1. The notion of combined pipeline and parallel processing will become more evident when we demonstrate in a moment how parallel processing "computational wavefronts" are pipelined successively through processor arrays.

Array Processors and Algorithm Expressions

A fundamental issue in mapping algorithms onto an array is *to express parallel algorithms in a notation than can be easily understood and compiled into efficient VLSI array processors*. Thus a powerful expression of array algorithms will be essential to the design of arrays. This paper proposes primarily three ways of array algorithm expression: *signal-flow-graph* (SFG), *systolic* and *wavefront* expressions.

2.1. Signal Flow Graph (SFG)

A Signal Flow Graph, consisting of *nodes* and *edges* [14, 9], is illustrated in Figure 2–2. The SFG representation has been popularly used for signal processing flow diagrams, such as FFT, digital filters and many other domains of signal and system applications.

The descriptions of array processing activities, in terms of the SFG representation, are often easy to comprehend. A typical example used for illustrating a two-dimensional array operation is matrix multiplication, as discussed in the **BOX 1**, Figure 2–3.

The abstraction provided by the SFG is very powerful and easy to use, and yet the transformation of an SFG description to a wavefront or systolic array can be accomplished automatically, as discussed in Section 3.1. In fact, a *systolic array can be considered an SFG array in combination with pipelining and retiming*. This is the reason why we first map parallel algorithms onto SFG arrays and then convert them into systolic arrays.

2.2. Systolic Array

The systolic array is very amenable to VLSI implementation. It is especially suitable to a special class of computation-bound algorithms, taking advantage of their regular, localized data flow. "A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word 'systole' to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network" [10].

For example, it is shown in [10] that some basic "inner product" *processing elements* (PE's)—each performing the operation $Y \leftarrow Y + A * B$—can be *locally* connected to perform digital filtering, matrix multiplication, and other related operations. In general, the data movements in a systolic array are prearranged and described in terms of the "snapshots" of the activities. (For examples, see BOX 1, Figure 2–4.)

A systolic array often represents a direct mapping of computations onto processor arrays. It will be used as an attached processor of a host computer (cf. Section 5). The systolic array features the important properties of modularity, regularity, local interconnection, as well as a high degree of pipelining and highly synchronized multiprocessing. It is also scalable architecturally, i.e. the size of the array may be indefinitely extended as long as the system synchronization can be maintained. There is extensive literature on the subject of systolic array processing, and the reader is referred to [6] and the references therein.

One problem, however, is that the data movements in a systolic array are controlled by global timing-reference

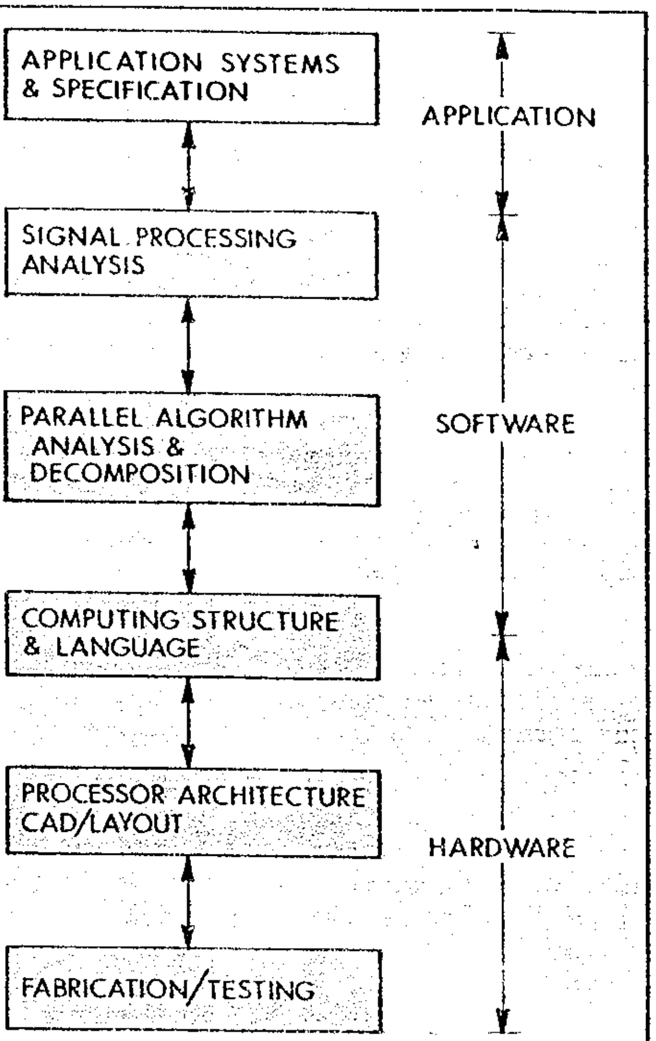

APPLICATION SYSTEMS & SPECIFICATION
SIGNAL PROCESSING ANALYSIS
PARALLEL ALGORITHM ANALYSIS & DECOMPOSITION
COMPUTING STRUCTURE & LANGUAGE
PROCESSOR ARCHITECTURE CAD/LAYOUT
FABRICATION/TESTING
APPLICATION
SOFTWARE
HARDWARE

Figure 1—1 Top-down Design Integration: In striving for a cohesive exploration of the overall implications of VLSI, cross-disciplinary discussion on application, algorithm, and architecture is necessary. In fact, *integration* will be the keyword in VLSI. This means that innovations will be needed in a very broad spectrum of disciplines, including algorithm analyses, parallelism extractions, array architectures, programming techniques, functional primitives, structural primitives, and numerical performance of DSP algorithms.

"beats." In order to synchronize the activites in a systolic array, extra delays are often used to ensure correct timing. More critically, the burden of having to synchronize the entire computing network will eventually become intolerable for very-large-scale or ultra-large-scale arrays.

2.3. Wavefront Array

A simple solution to the above-mentioned problems is to take advantage of the control-flow locality, in addition to the data-flow locality, inherently possessed by most algorithms of interest. This permits a data-driven, self-timed approach to array processing. Conceptually, this approach substitutes the requirement of correct "timing" by correct "sequencing." This concept is used extensively in dataflow computers and wavefront arrays.

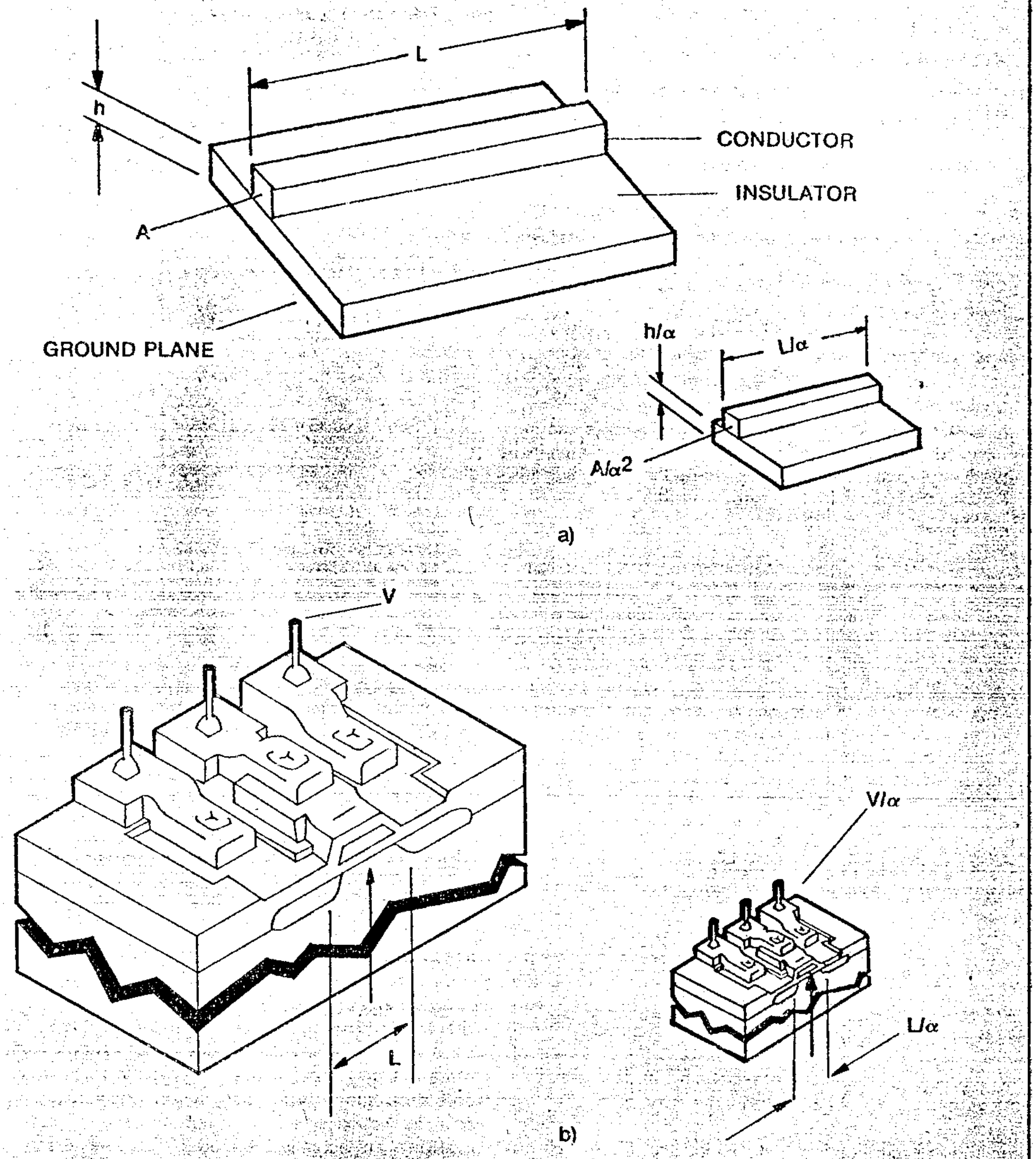


Figure 1–2. (a) Scaling of a MOSFET Transistor by a factor α. The switching delay of a transistor is scaled down at least by α, due to the fact that the channel length is decreased by a factor α. (b) Scaling of a conductor by a factor α: Since the cross-sectional area of the conductor is decreased by a factor α^2, the resistance per unit length will increase by a similar factor. If the length of the conductor is scaled by α (as simple scaling implies), then the net increase of resistance is in proportion to α. However, scaling down also implies that the capacitance of a fixed interconnection scales down by α. The scaling up of resistance and the scaling down of capacitance cancel exactly, leaving the RC time constant and the interconnect delay unchanged. Since gate delays decrease while interconnect delays remain constant with scaling, the speed at which a circuit can operate is eventually dominated by interconnect delays rather than device delays.

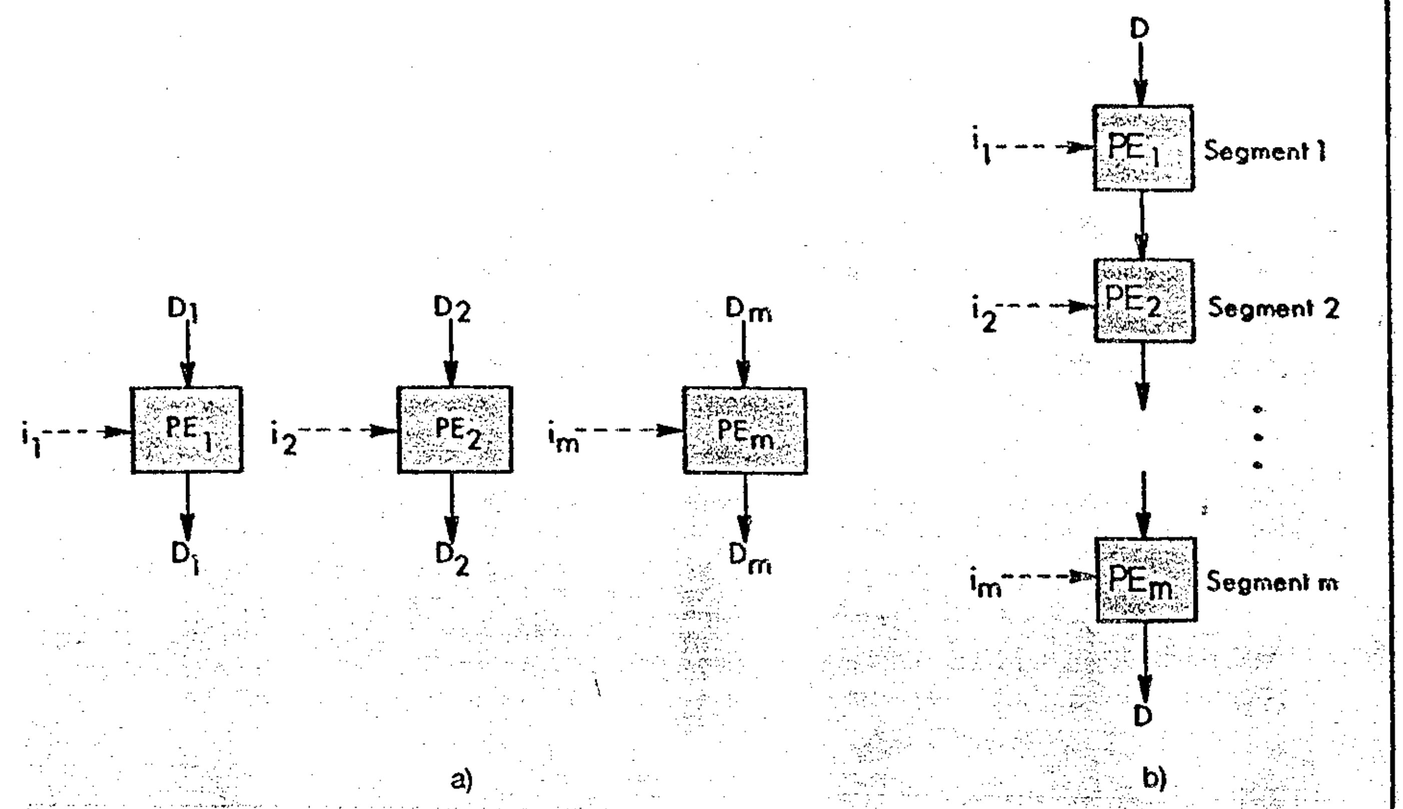


Figure 2—1. Array processors derive a massive concurrency from both parallel and pipeline processing schemes [8]. (a) Parallel processing means that all the processes defined in terms of the data (D_i) and the instructions (i_i)—may directly access the m processing in parallel and keep all the processors busy. (b) Pipeline processors means that a process is decomposed into many subprocesses, which are pipelined through m processors, i.e., m segments aligned in a chain, and each subprocess will be processed in succession. For each subprocess coming out of the array, there will be a processor vacant and ready to receive and handle a new subprocess immediately. Therefore, all the m processors can again be kept busy all the time by the pipeline technique.

A dataflow multiprocessor [5] is an asynchronous, data-driven multiprocessor that runs programs expressed in data-flow graph form. Since the execution of its instructions is "data driven," i.e., the triggering of instructions depends only upon the availability of operands and resources required, unrelated instructions can be executed concurrently without interference. The principal advantages of data-flow multiprocessors are simple representation of concurrent activity, relative independence of individual PE's, greater use of pipelining, and reduced use of centralized control and global memory.

However, for a general-purpose dataflow multiprocessor, the interconnection and memory conflict problems remain very critical. Such problems can be greatly alleviated if *modularity* and *locality* are incorporated into dataflow multiprocessors. This motivates the concept of the Wavefront Array Processors (WAP).

The derivation of a wavefront process consists of three steps: (i) the algorithms are expressed in terms of a sequence of recursions; (ii) each of the recursions is mapped to a corresponding computational wavefront; and (iii) the wavefronts are successively pipelined through the processor array. (A simple matrix multiplication example is discussed in BOX 1.)

As a justification for the name "wavefront array," we note that the computational wavefronts are similar to electromagnetic wavefronts, since each processor acts as a secondary source and is responsible for the propagation of the wavefront. The pipelining is feasible because the wavefronts of two successive recursions will never intersect (by Huygen's' wavefront principle), thus avoiding any contention problems. It is even possible to have wavefronts propagating in several different fashions. For example, in the extreme case of non-uniform clocking, the wavefronts are actually crooked. What is necessary and sufficient is that the order of task sequencing be correctly followed. The correctness of the sequencing of the tasks is ensured by the wavefront principle [11].

The wavefront processing utilizes both the localities of data flow *and* control flow inherent in many signal processing algorithms. Since there is no need for synchronizing the entire array, a wavefront array is truly architecturally scalable. In fact, it may be stated that a *wavefront array is a systolic array in combination with the dataflow principle*.

ARRAY PROCESSORS FOR MATRIX MULTIPLICATIONS

In this BOX, a simple matrix multiplication example is used to illustrate SFG, systolic, and wavefront array processors.

Let $A = \{a_{ij}\}$, $B = \{b_{ij}\}$, and $C = A \times B = \{c_{ij}\}$ all be $N \times N$ matrices. The matrix A can be decomposed into columns A_i and matrix B into rows B_j, and therefore,

$$C = A_1 * B_1 + A_2 * B_2 + \ldots\ldots + A_N * B_N \qquad (1)$$

where the product $A_i * B_i$ is termed "outer product." The matrix multiplication can then be carried out in N recursions (each executing one outer product).

$$C^{(k)} = C^{(k-1)} + A_k * B_k \qquad (2)$$

There will be N sets of computational "wavefronts" involved, one for each recursion. More explicitly,

$$c_{i,j}^{(k)} = c_{i,j}^{(k-1)} + a_i^{(k)} b_j^{(k)} \qquad (3)$$

for $k = 1, 2, \ldots, N$. For simplicity, let

$$a_i^{(k)} = a_{ik} \quad \text{and} \quad b_j^{(k)} = b_{kj}$$

SFG Array Processing

The notion of SFG array processing allows an extensive use of broadcasting, since a node or a zero-delay edge is considered to be delay-free. As a result, a very straightforward SFG array for the matrix multiplication algorithm is derived in Figure 2–3.

Systolic Array Processing

For the matrix multiplication algorithm, a square systolic array and the corresponding data arrangement can be proposed, as shown in Figure 2–4. The question now is: How does such a scheme actually produce the desired multiplication results? A popular way to give a demonstration is to display the space-time activities in the first few consecutive "beats," like those displayed in Figure 2–4. If the design is correct, the pre-arranged data will meet the designated partners, perform the appropriate operations, and yield desired "products." The complete activities, as well as the general rule, can then be derived by induction. A simple (and possibly automatic) conversion from an SFG array into a systolic array, which may alleviate the burden of verification, is discussed later in this section.

Wavefront Array Processing

The notion of a computational wavefront is also well illustrated by the example of the matrix multiplication algorithm. The topology of the matrix multiplication algorithm can be mapped naturally onto the square, orthogonal $N \times N$ matrix array of the Wavefront Array Processor (WAP), as in Figure 2–5. To create a smooth data movement in a localized communication network, we propose the notion of the computational wavefront. A wavefront in a processsor array corresponds to a mathematical recursion in the algorithm. Successive pipelining of the wavefronts through the computational array will accomplish the computation of all recursions.

The computational wavefront for the first recursion in matrix multiplication is now examined more elaborately. Suppose that the registers of all the processing elements (PE's) are initially set to zero:

$$c^{(0)}_{ij} = 0 \quad \text{for all } (i, j).$$

The entries of A are stored in the memory modules to the left (in columns), and those of B in the memory modules on the top (in rows). The process starts with PE (1, 1) and

$$c^{(1)}_{11} = c^{(0)}_{11} + a_{11} * b_{11}$$

is computed. The computational activity then propagates to the neighboring PE's (1, 2) and (2, 1), which will execute:

$$c^{(1)}_{12} = c^{(0)}_{12} + a_{11} * b_{12}$$

and

$$c^{(1)}_{21} = c^{(0)}_{21} + a_{21} * b_{11}$$

The next front of activity will be at PE's (3, 1), (2, 2) and (1, 3), thus creating a computation wavefront traveling down the processor array. Once the wavefront sweeps through all the cells, the first recursion is complete. As the first wave propagates, we can execute an *identical* second recursion in parallel by *pipelining* a second wavefront *immediately after* the first one. For example, the (1, 1) processor will execute

$$c^{(2)}_{11} = c^{(1)}_{11} + a_{12} * b_{21}$$

$$\vdots$$

$$c^{(k)}_{ij} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \ldots + a_{ik} * b_{kj}$$

and so on.

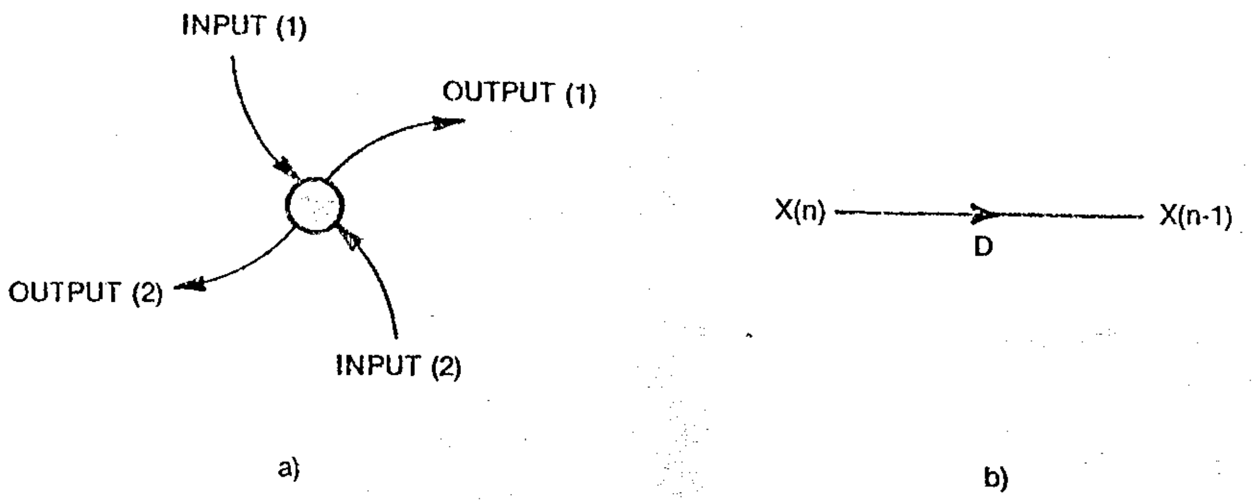


Figure 2–2. Examples of Signal Flow Graph notation: (a) An Operation Node with two inputs and two outputs; (b) An Edge as a Delay Operator. In general, a *node* is often denoted by a circle representing an arithmetic or logic function such as multiply, add, etc. *performed with zero delay,* (see Figure 2–2(a)). A node is considered to be delay-free, unless otherwise specified. In fact, the SFG representation derives its power from the assumption that the computations in the node are delay free, warranting simpler snapshot descriptions than the systolic counterpart. Consequently, the task of tracing the detailed space-time activities associated with pipelining is simplified. An *edge,* on the other hand, denotes either a function or a delay. Unless otherwise specified, for a large class of signal processing SFG's, the following conventions are adopted for convenience. When an edge is labeled with a capital letter *D* (or *D'*, *2D*, etc.), it represents a time-delay operator with delay-time *D* (or *D'*, *2D*, etc.) (see Figure 2–2(b)).

3. MAPPING ALGORITHMS ONTO ARRAYS

3.1. Systolization of SFG Computing Networks

A Cut-set Systolization Procedure [9]

A *cut-set* in an SFG is a minimal set of edges that partitions the SFG into two parts. The systolization procedure is based on two simple rules:

Rule (i). Time-scaling: All delays D may be scaled, i.e., $D \rightarrow \alpha D$, by a single positive integer α. Correspondingly, the input and output rates also have to be scaled down by a factor α. The time-scaling factor (or, equivalently, the slow-down factor) α is determined by the slowest (i.e. maximum) loop delay in the SFG array.

Rule (ii). Delay-Transfer: Given any cut-set of the SFG, we can group the edges of the cut-set into *in-bound edges* and *out-bound edges,* depending upon the directions assigned to the edges. Rule (ii) allows advancing k time-units on all the out-bound edges and delaying k time-units on the in-bound edges. It is clear that, for a (time-invariant) SFG, the general system behavior is not affected because the effects of lags and advances cancel each other in the overall timing. Note that the input-input and input-output timing relationships will also remain exactly the same only if they are located on the same side. Otherwise, they should be adjusted by a lag of +k time-units or an advance of −k time-units.

As an illustration, with reference to the BOX, Figure 2–3, the dashed lines indicate a set of possible cuts. A simple procedure involving delay-transfer of one time-unit (D) yields the systolic array shown in Figure 2–4.

Example: Multiplication of a Banded Matrix and a Full Rectangular Matrix

Let us look at a slightly different but commonly encountered type of matrix multiplication problem. This involves a banded Matrix A that has nonzero elements only on a finite "band" along the diagonal. Consider the product of a banded Matrix A, of size N × N, and bandwidth P, and a rectangular Matrix **B** of size N × Q:

$$AB = \left[\begin{array}{ccccccccc} x & x & x & & & & & & \\ x & x & x & x & & & & \Phi & \\ x & x & x & x & x & & & & \\ & x & x & x & x & x & & & \\ & & x & x & x & x & x & & \\ \Phi & & & x & x & x & x & x & \\ & & & & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & & & & \bullet & \bullet & \bullet & \bullet \\ & & & & & & \bullet & \bullet & \bullet \end{array}\right] \left[\begin{array}{ccccc} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{array}\right]$$

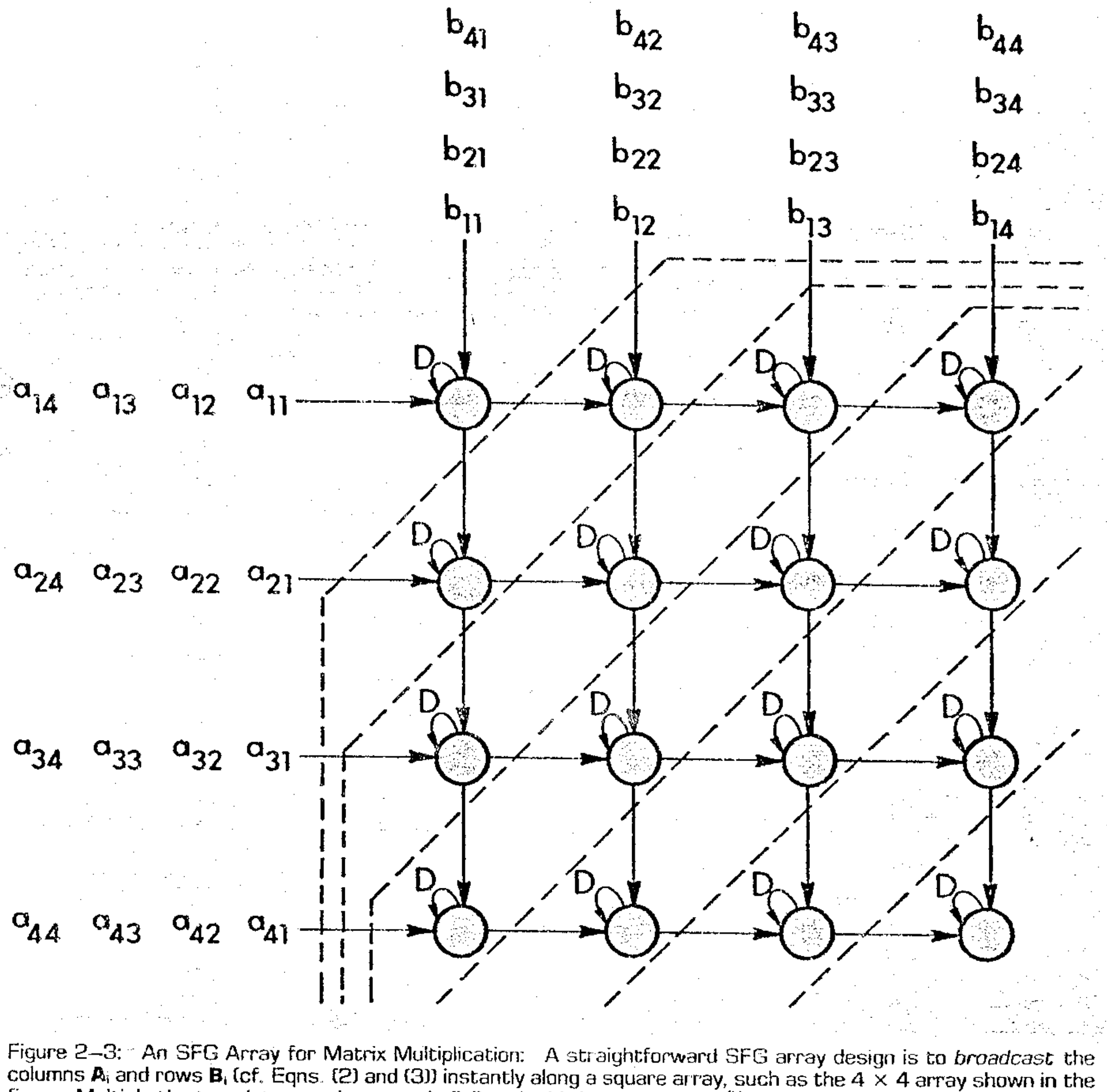
(A banded Matrix A is one which has non-zero elements only on a Finite "band" along the diagonal.)

This situation arises in many application domains, such as DFT and time-varying (multi-channel) linear filtering. In most applications, N >> P and N >> Q; therefore, *it is very uneconomical to use $N \times N$ arrays for computing $C = A \times B$.*

Fortunately, with a slight modification to the SFG in Figure 2–3, the same speed-up performance can be achieved with only a $P \times Q$ rectangular array (as opposed to an $N \times N$ array). This is shown in Figure 3–1(a).

A Systolization Example

If local interconnection is preferred, the proposed procedure can then be used to systolize the SFG array in Figure 3–1(a) and yield the data array as depicted in Figure



b_{41} b_{42} b_{43} b_{44}
b_{31} b_{32} b_{33} b_{34}
b_{21} b_{22} b_{23} b_{24}
b_{11} b_{12} b_{13} b_{14}
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
a_{44} a_{43} a_{42} a_{41}
D

Figure 2–3: An SFG Array for Matrix Multiplication: A straightforward SFG array design is to *broadcast* the columns $\mathbf{A}_i$ and rows $\mathbf{B}_i$ (cf. Eqns. (2) and (3)) instantly along a square array, such as the 4 × 4 array shown in the figure. Multiply the two data meeting at node (i, j) and add the product to $c_{ij}^{(k)}$, the data value currently residing in a register in node (i, j). Finally, the new result will update the register via a loop with a delay D and get ready to interact with the new arriving operands. As all the column and row input data continue to arrive at the nodes, all the outer products will be sequentially summed. Although this design is not directly suitable for a VLSI circuit design due to the use of global communication, it may be converted to a systolic array, as shown in Figure 2–4, or a wavefront array, shown in Figure 2–5. A simple conversion strategy will be provided in Section 3.

3-1(b). The systolization procedure is detailed below:

(i) Time-scaling: According to Rule (i) above, the slow-down factor α is determined by the maximum loop delay in the SFG array. Referring to Figure 3-1(a), any loop consisting of one up-going and one down-going edges yields a (maximum) delay of two. This is why the final pipelined systolic array has to bear a slow-down factor $\alpha = 2$. The pipelining rate is 0.5 word per unit-time for each channel.

(ii) Delay Transfer: Apply Rule (ii) above to the cut-sets shown in Figure 3-1(a). The systolized SFG will have one delay assigned to each edge and thus represents a localized network. Also based on Rule (ii), the inputs from

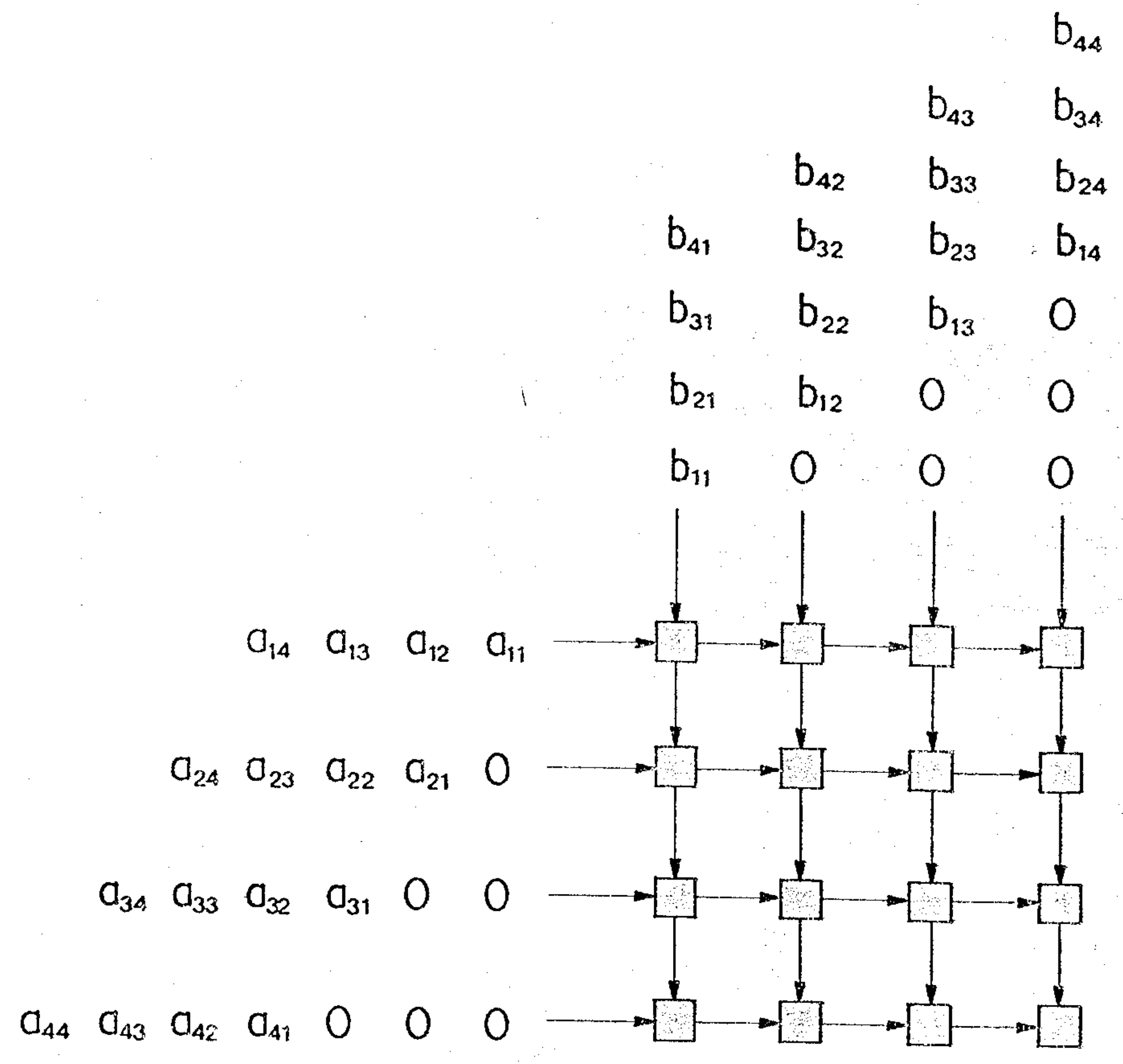

b_{44}
b_{43} b_{34}
b_{42} b_{33} b_{24}
b_{41} b_{32} b_{23} b_{14}
b_{31} b_{22} b_{13} 0
b_{21} b_{12} 0 0
b_{11} 0 0 0
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21} 0
a_{34} a_{33} a_{32} a_{31} 0 0
a_{44} a_{43} a_{42} a_{41} 0 0 0

Figure 2-4: A systolic array for matrix multiplication. For this example, a two-dimensional square array forms a natural topology for the matrix multiplication problem. The figure specifically shows a 4 × 4 array of processing elements (PE's). All the PE's (represented by the square boxes) uniformly consume and produce data in one single time-unit. In terms of one "snapshot" of the activities— the input data (from matrices **A** and **B**, appearing at the left and top parts of the figure) are pre-arranged in an orderly sequence. The **C** data stay temporarily within the PE's and will be pumped out from one side of the array. Due to the systolization rules discussed in Section 3, the inputs from different columns of **B** and rows of **A** will have to be adjusted by a certain number of delays before arriving at the array. This is why some extra zeros are introduced here. (To become more convinced, the reader might want to try it out with several consecutive shapshots of the data movements.) In general, the major characteristics of systolic arrays are [9] (i) synchrony: the data is rhythmically computed (timed by a global clock) and pumped through the network; (ii) regularity, modularity and spatial locality of interconnections; (iii) temporal locality, and (iv) effective pipelinability.

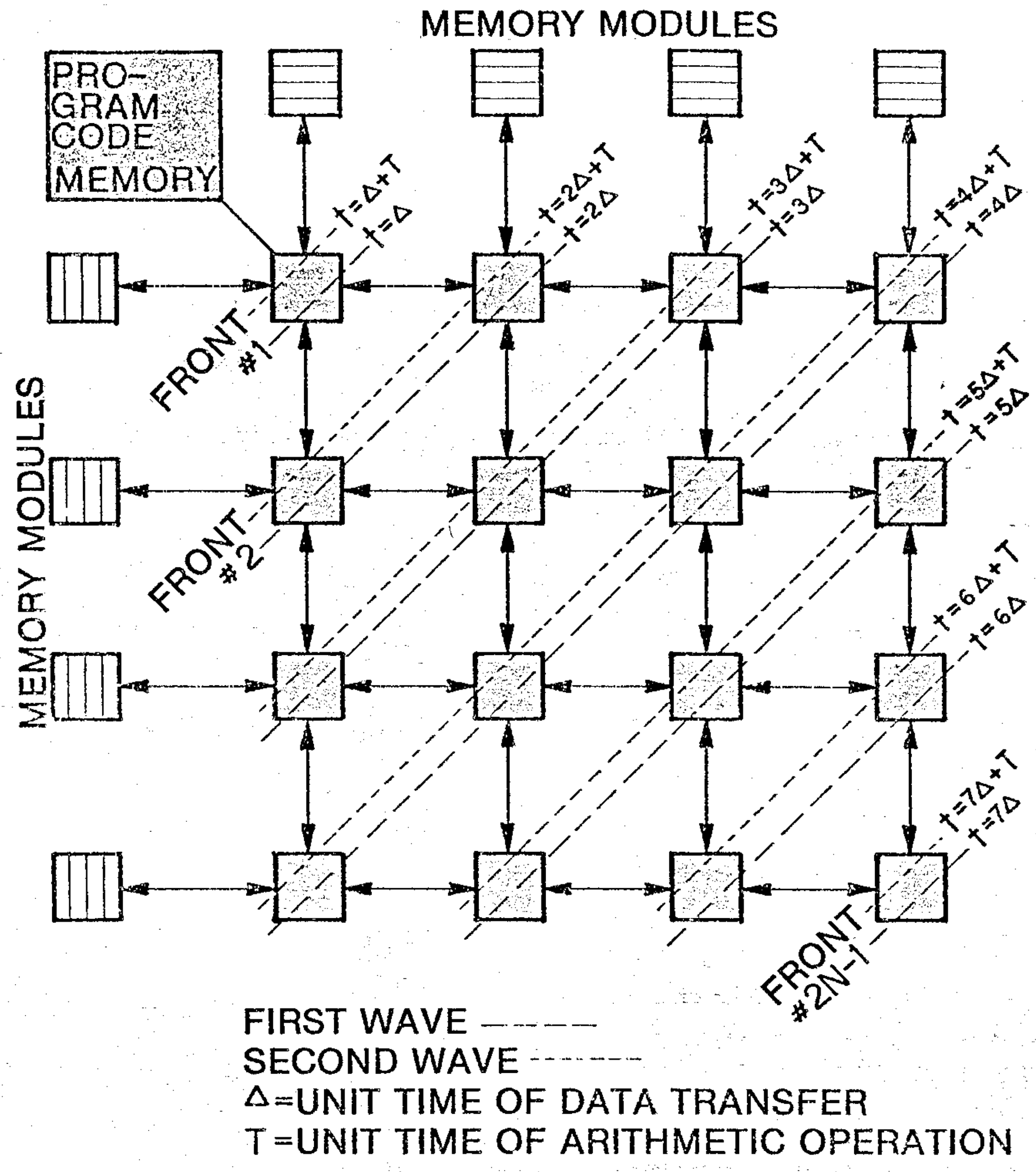
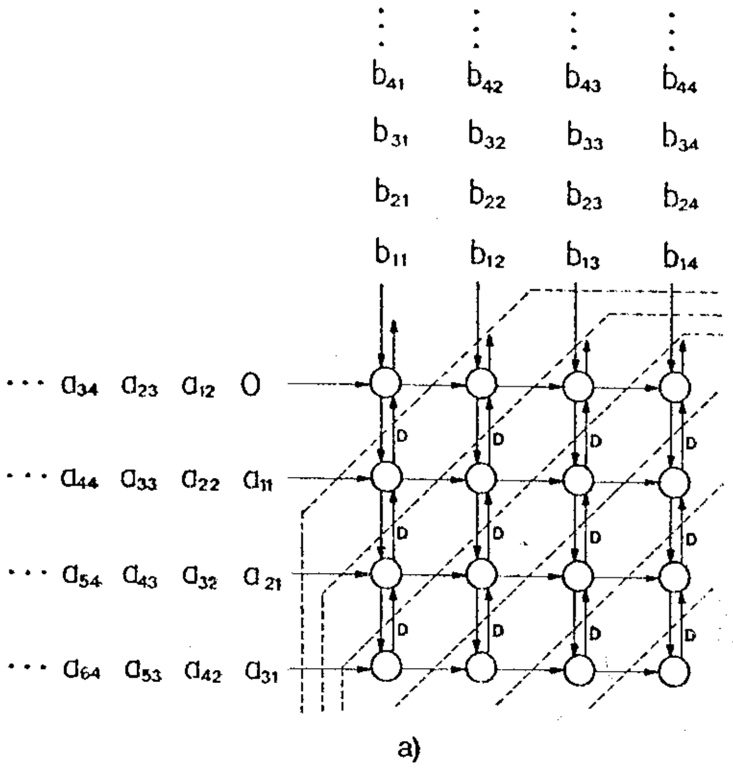
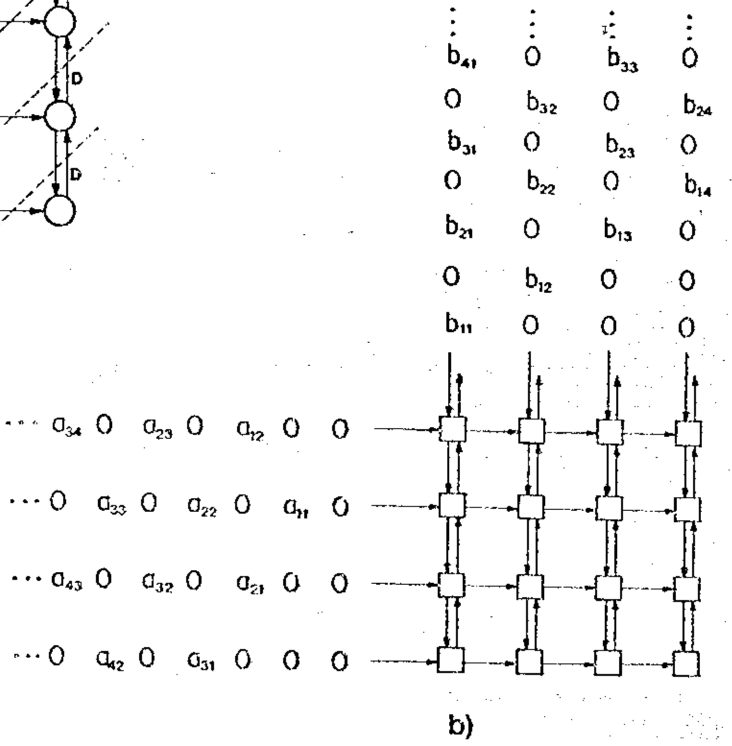


Figure 2–5: Wavefront processing for Matrix Multiplication. In this example, the wavefront array consists of N × N processing elements with regular and local interconnections. The figure shows the first 4 × 4 processing elements of the array. The computing network serves as a (data) wave propagating medium. Hence the hardware will have to support pipelining of the computational wavefronts as fast as resource and data availability allow, which can often be accomplished simply by means of a handshaking protocol, such as that proposed in [11]. The (average) time interval (T) between two separate wavefronts is determined by the availability of the *operands and operators*. In this case, T is equal to the time needed for the arithmetic operations: multiply-and-add. The speed of wavefront propagation is determined by the time interval Δ, which in this case is equivalent to the data transfer time. In general, the major characteristics of wavefront arrays are (i) self-timed, data-driven computation, meaning that no global clock is needed; (ii) regularity, modularity and spatial locality of interconnections; and (iii) effective pipelinability.



b_{41} b_{42} b_{43} b_{44}
b_{31} b_{32} b_{33} b_{34}
b_{21} b_{22} b_{23} b_{24}
b_{11} b_{12} b_{13} b_{14}
$\cdots a_{34}\ a_{23}\ a_{12}\ 0$
$\cdots a_{44}\ a_{33}\ a_{22}\ a_{11}$
$\cdots a_{54}\ a_{43}\ a_{32}\ a_{21}$
$\cdots a_{64}\ a_{53}\ a_{42}\ a_{31}$
D
a)



b_{41} 0 b_{33} 0
0 b_{32} 0 b_{24}
b_{31} 0 b_{23} 0
0 b_{22} 0 b_{14}
b_{21} 0 b_{13} 0
0 b_{12} 0 0
b_{11} 0 0 0
$\cdots a_{34}\ 0\ a_{23}\ 0\ a_{12}\ 0\ 0$
$\cdots 0\ a_{33}\ 0\ a_{22}\ 0\ a_{11}\ 0$
$\cdots a_{43}\ 0\ a_{32}\ 0\ a_{21}\ 0\ 0$
$\cdots 0\ a_{42}\ 0\ a_{31}\ 0\ 0\ 0$
b)

Figure 3–1: (a) An SFG Array for matrix multiplication involving a banded matrix. The left memory module will store the matrix **A** along the band-direction, (see Figure 2–3) and the upper-module will store **B** the same as before. Note that the major modification to the array is that, between the recursions of outer products, there should be an upward shift of the partial sums. This is because the input matrix **A** is loaded in a skewed fashion. The final result (**C**) will be output from the I/O ports of the top-row PE's.

(b) The systolic array design as a result of applying the systolization procedure to the Figure 3–1(a). Note that the pipelining rate is proportional to α^{-1}, and that $\alpha = 2$ in this example.

different columns of B and rows of A will have to be adjusted by a certain number of delays before arriving at the array. By counting the cut-sets involved in Figure 3–1(a), it is clear that the first column of B needs no extra delay, the second column needs one delay, the third needs two (attributed to the two cut-sets separating the third column input and the adjacent top-row processor), etc. Therefore, the B matrix will be skewed, as shown in Figure 3–1(b). A similar arrangement can be applied to the input matrix A.

3.2. Converting an SFG Array into a Wavefront Array

The wavefront propagation is very similar to the previous case. Since in wavefront processing the (average) time interval (T) between two separate wavefronts is determined by the availability of the *operands and operators*. In this case, there is a feedback loop involved, shown by the edge-pairs (an up-going arrow and a down-going arrow) in Figure 3–1. For example, in the node (1,1) the second front has to wait till the first front completes all the follow-

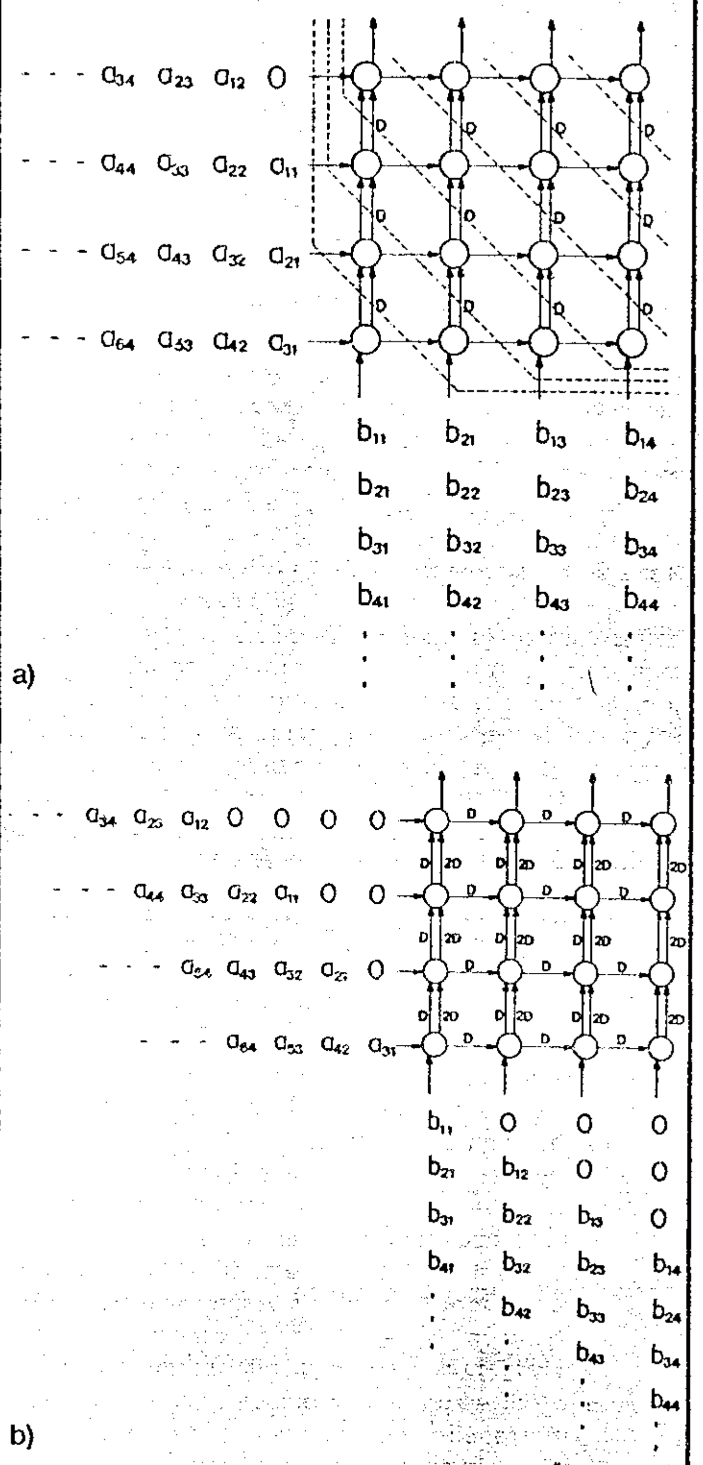


Figure 3–2: (a) Another SFG Array for banded-matrix-and-matrix multiplication. Note that the additional modification to Figure 3–1(a) is that the input **B** matrix is now loaded from the bottom and travels upwards, instead of from the top and traveling downwards.

(b) The systolic array as a result of applying the systolization procedure to the Figure 3–2(a). As the pipelining rate is proportional to α^{-1}, and since α is now reduced to 1, this systolic array can run twice faster than the $\alpha = 2$ design of Figure 3–1(b).

ing steps: (i) propagate data downwards (processing time: Δ), (ii) perform the arithmetic operations at node (2, 1) (processing time: T_{MA}), and (iii) return the result upwards to PE(1, 1) (processing time: Δ). Once the result is returned to PE(1, 1), the second front can be immediately activated. The activations of all the later fronts follow exactly the same procedure; therefore, the (average) time separating two consecutive fronts is $T = T_{MA} + 2\Delta$.

In fact, every regular and modular SFG array can be converted into a wavefront array. In a self-timed system, the exact timing reference is ignored; instead, the central issue is sequencing. Getting a data token in a self-timed system is equivalent to incrementing the clock by one time-unit in a synchronous system. Therefore, the conversion of an SFG into a data-driven system involves substituting the delay operators D by "handshaked delay" registers. (A "handshaked delay" register is a device that prevents any incoming data from directly passing through *until* the handshaking flag signals a "pass." For example, applying this conversion process to the SFG array in Figure 2–3 for matrix multiplication yields the wavefront array shown in Figure 2–5 [9]. The conversion process is equally applicable to the SFG array of Figure 3–1(a).

3.3 How to improve the pipelining rate?

In the above case, the maximum loop delay is 2. Thus, the slow-down factor α equals 2 and the pipelining rate is reduced by one half. Now the question is: What is the best pipelining rate achievable for the algorithm? To improve the rate, we would like to reduce the slow-down factor down to $\alpha = 1$. This is possible only when all the loops are eliminated. For the banded-matrix-and-matrix multiplication problem, one has to resort to a major modification on the SFG array. The trick is to load the input B matrix from the bottom and travel upwards, as illustrated in Figure 3–2(a), instead of from the top and traveling downwards, as in Figure 3–1(a). Here it is a valid change, because as long as broadcasting is assumed in the SFG model, it does not matter whether the (same) data are loaded top-down or bottom-up. But the effects on α, and therefore the achievable pipelining rates, are totally different. Note that in the modified SFG array there are no loops. This implies that $\alpha = 1$. Consequently, the cuts as shown in Figure 3–2(a) do not call for any time rescaling, but only need a delay transfer of D. After the cut-set delay transfer procedure, the resultant systolic array is depicted in Figure 3–2(b). Note that this new version offers a throughput-rate of 1 word per unit-time for each channel, which is *twice faster* than the previous systolic design.

3.4. Spiral Systolic Arrays

There are many systolic arrays and algorithms that are more complicated than the ones for the matrix multiplication example discussed above. These include systolic algorithms for convolution, correlation, FIR, IIR, and lattice filtering for one- or two-dimensional signals. Also worth mentioning are a number of important matrix operations, such as linear system solution, least square solution, Toeplitz system solution, and eigenvalue and singular value decompositions.

Due to limited space, it is impossible to cover all the algorithms in this article. Instead, the author will focus upon a very interesting class of algorithms that can be regarded as a generalization of the matrix multiplication and LU decomposition problems. These algorithms share a common recursive formulation:

$$C_{ij}^{k} = C_{ij}^{k-1} + C_{ik}^{k-1} \cdot (C_{kk}^{k-1})^{*} \cdot C_{kj}^{k-1} ;$$

where $+$, $\cdot$, and $*$ denote certain algebraic operators defined by the specific application. This formulation covers a broad and useful application domain that includes transitive closure, shortest path, LU decomposition, and many other problems. As an example, for LU-decomposition problem, the operation $(C_{kk}^{k-1})^{*}$ is inversion, i.e. $(C_{kk}^{k-1})^{-1}$.

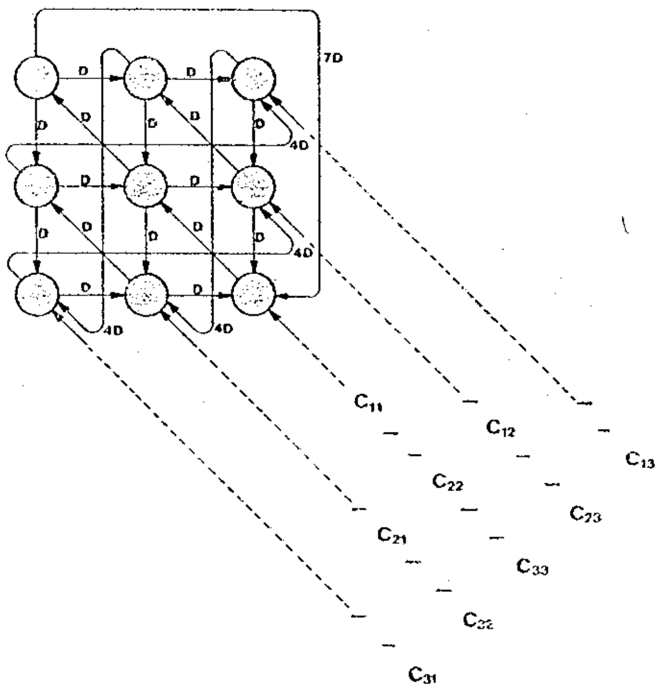


Figure 3–3: Mapping algorithms onto a spiral systolic array involves a careful bookkeeping of recursion indices. A complicated mapping is exemplified by the mathematical formulation shown in Eq. (4), which embraces many important applications. To cope with the index change, it is a common practice to move the whole data-array northwestwards along the diagonal direction after each recursion. As a result, the first row and first column PE's will require special operations, while the remaining PE's will all be the same. If N is the array size, such a diagonal movement can be expressed mathematically as:

$$(i, j) \text{ element} \rightarrow [(i-1) \bmod N, (j-1) \bmod N] \text{ cell},$$

after each recursion. To support this data movement, we need diagonal as well as wrap-around connections (for boundary PE's). Since the data propagate only to the right or downwards, it is straightforward to incorporate pipelining into the array. Note that $(C_{kk}^{k-1})^{*}$ should be sent to every PE; we choose to send it down along the first column (and perform $(C_{kk}^{k-1})^{*} \cdot C_{ik}^{k-1}$). The result is then sent along each respective row of PE's, where the remaining multiply-and-add operations are performed.

A systolic array for this class of algorithms is proposed in Fig. 3–3. We call this configuration a *spiral systolic array*, since this array configuration is basically an Illiac IV spiral configuration used in Illiac IV, augmented by diagonal connections [13].

4. ALGORITHM DESIGN CRITERIA

An effective design of algorithms for array processing hinges upon a full understanding of the problem specification, mathematical analysis, parallelism analysis, and the practicality of mapping the algorithms onto real machines.

Parallel array algorithm design is a new area of research study that has profited from the theory of signals and systems and has been influenced by linear algebraic numerical methods. In a conventional algorithm analysis, the complexity of an algorithm depends on the computation and storage required. The modern concurrent computation criteria should include one more key factor: *communication*. In the design of array algorithms, the major factors therefore are computation, communication, and memory.

The key aspects of parallel algorithms under VLSI architectural constraints are presented below:

1. **Maximum Concurrency:** The algorithm should be structured to achieve maximum concurrency and/or maximum throughput. (Two algorithms with equivalent performance in a sequential computer may fare very differently in parallel processing environments.) An algorithm will be favored if it expresses a higher parallelism that is exploitable by the computing arrays.
   Example: A very good example is the problem of solving Toeplitz systems, for which the major algorithms proposed in the literature are the Schur algorithm and the Levinson algorithm [12]. The latter is by far more popular in many spectrum estimation techniques, such as the maximum entropy method [1] or maximum likelihood method [2]. In terms of sequential processing, both the algorithms require the same number of operations. However, in terms of the achievable concurrency when executed in a linear array processor, the Schur algorithm displays a clear-cut advantage over the Levinson algorithm. More precisely, using a linear array of N processing elements, the Schur algorithm will need only O(N) computation time, compared with O(NlogN) required for the Levinson algorithm. Here N represents the dimension of the Toeplitz matrix involved. For a detailed discussion, see [12].
2. **Maximum pipelinability and the balancing of computation and I/O:** Most signal processing algorithms

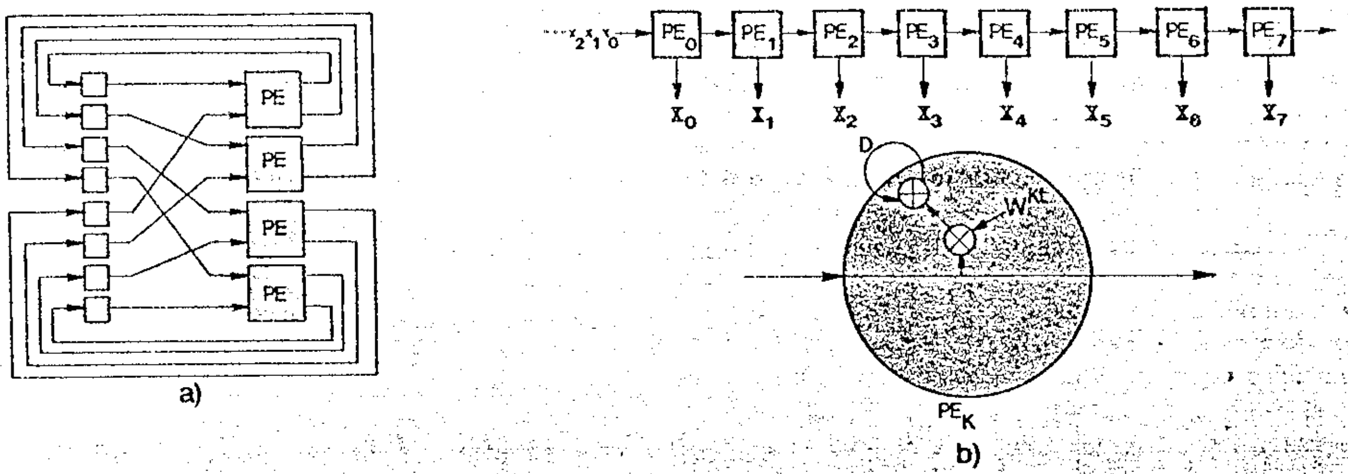


Figure 4–1: Array Processor Architectures for (a) FFT Algorithm; and (b) DFT Algorithm. Note that, in terms of total processing times, the ratio is log N vs. N in favor of FFT. On the other hand, the FFT array requires a *global* (Perfect-Shuffle) communication network. In contrast, the DFT array can be easily systolized and implemented in a modular processor array with *local* communication. Note that the weighting factors W^{kt} are time-varying; and the Fourier transform output $\{X_k\}$ stays in the node and will eventually be pumped out.

demand very high throughput and are computation-intensive compared with the input/output (I/O) requirement. Pipelining is essential to the throughput of array processors. The exploitation of the pipeline technique is often very natural in regular and locally-connected networks; therefore, a major part of concurrency in array processing will be derived from pipelining. In general, the pipelining rate is determined by the "maximum" loop delay in the SFG array. To maximize the rate, one must select the best among all possible SFG arrays for any algorithm. The pipeline techniques are especially suitable for balancing computation and I/O because the data tend to engage as many processors as possible before they leave the array. This helps reduce I/O bandwidth for outside communication.

Example: Note that, for the banded-matrix-and-matrix multiplication algorithm, the systolic array shown in Figure 3-2(b) offers a throughput-rate twice as fast as the design in Figure 3–1(b).

3. **Trade-off between communication and computation costs:** To make the interconnection network practical, efficient and affordable, regular communication should be encouraged. Key issues affecting the regularity include local vs. global, static vs. dynamic, and data-independent vs. data-dependent interconnection modules. The criterion should maximize the tradeoff between interconnection cost and throughput. For example, to conform with the communication constraints imposed by VLSI, a lot of emphasis has recently been placed on a special class of local and recursive algorithms.

   Example: A comparison between the costs of DFT and FFT algorithms will be discussed momentarily in Example 1.

4. **Numerical performance, quantization effects, and data dependency:** Numerical behavior depends on many factors, such as the word-length of the computer and the algorithms used. Unpredictable data dependency may severely jeopardize the processing efficiency of a highly regular and structured array algorithm. Effective VLSI arrays are inherently highly pipelined, and therefore require well structured algorithms with predictable data movements. Iterative methods with dynamic branching, which are dependent on data produced in the middle of the process, are less suited

for pipelined architecture. A comparison between several major linear system solvers will be discussed in Example 2.

Example 1: Trade-off between computation and communication costs

When the communication in VLSI systems is emphasized, the trade-off between computation and communication becomes a central issue. (cf. Figure 4–1.) The preference on regularity and locality will have a major impact in designing parallel and/or pipelined algorithms. Comparing the two Fourier transforming techniques, DFT and FFT, the computations are $O(N^2)$ vs. $O(N\log N)$ in favor of FFT. However, the DFT enjoys a simple and local communication, while the FFT involves a global interconnection, i.e. the nodes retrieve their data from far away elements. In the trade-off of computation vs. communication costs, the choice is no longer obvious.

Example 2: Comparison of Linear System Solvers

It is well known that there are three major numerical algorithms for solving a linear system of equations; namely, the LU decomposition, the Householder QR (HQR) and the Givens QR (GQR) decomposition algorithms [19]. From a numerical performance point of view, a HQR or a GQR decomposition is often preferred over an LU decomposition for solving linear systems. As for the *maximum concurrency achievable by array processing,* the GQR algorithm achieves the same 2-D concurrency as the LU decomposition with the same complexity in a modular, streamlined fashion. They both hold a decisive advantage over the HQR method in terms of maximum concurrency achievable. Therefore, the Givens QR method is superior to the LU decomposition and the Householder QR method when both numerical performance and massive parallelism are considered. Note that the price, however, is that the GQR method is computationally more costly than the other two methods, (cf. the third column of Table 1.)

TABLE 1: Comparison of linear system solvers. The key issues are data dependency, numerical performance, maximum concurrency, and total number of computations. (*The numerical performance of LU decomposition may be improved by using a pivoting scheme [19]; but this necessitates control on the magnitude of a pivot, jeopardizing the otherwise smooth data flow in the array).

Algorithm	Numerical Performance	Maximum Concurrency Achievable	Number of Operations
LU Decomposition	Bad*	2-D array	$(N^3/3)$
HQR Decomposition	Good	1-D array	$(2N^3/3)$
GQR Decomposition	Good	2-D array	$(4N^3/3)$

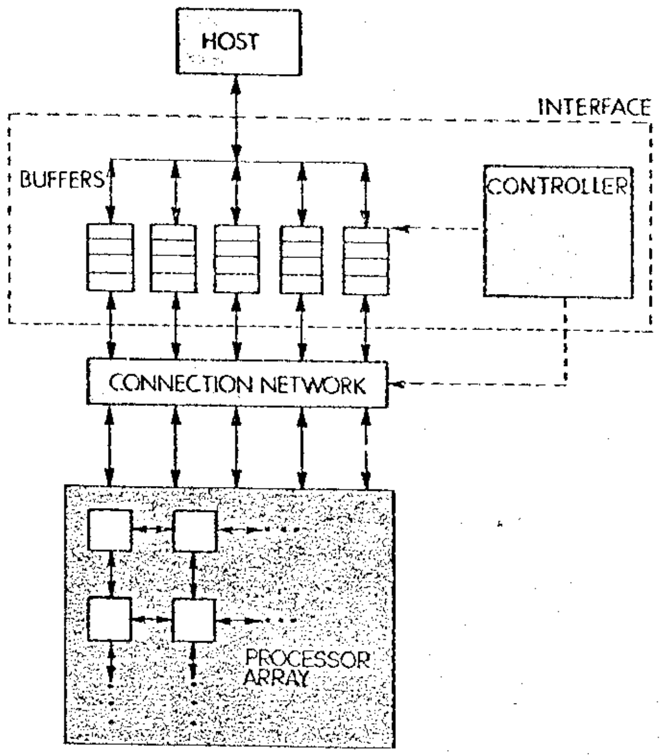


Figure 5–1: An example of Array Processing System Configuration: An array processor is often used as an attached processor linked with a host through an interface system.
Host Computer: The host computer should provide batch data storage, management, and formatting; determine the schedule program that controls the interface system and connection network, and generate and load object codes to the PE's. A very challenging task to the system designer is to identify a suitable host machine for interfacing with high-speed array processor units.
Interface System: The interface system, connected to the host via the host bus, has the functions of downloading and up-loading data. Based on the schedule program, the controller monitors the interface system and array processor. The interface system should also furnish an adequate hardware support for many common data management operations. Other challenging tasks for the system designer are managing blocks of data and making sure the memory (buffer) unit is able to balance the low bandwidth of system I/O and the high bandwidth of array processors.
Connection Network: Connection networks provide a set of mappings between processors and memory modules to accommodate certain common global communication needs. Incorporating certain structured interconnections may significantly enhance the speed performance of the processor arrays.
Processor Arrays: For simplicity, only one processor array is physically depicted in the figure. However, the concept of networking several processor arrays has now attracted a good deal of attention. For example, when a problem is reducible to several subproblems that can be executed one after the other, it will be useful to have each subproblem executed in its own processor array, while utilizing the network to facilitate the data pipelining between the arrays. This suggests a pipelining scheme at the array-level, which may increase the processing speed-up by one more order of magnitude.

5. IMPLEMENTATION CONSIDERATIONS OF ARRAY PROCESSOR SYSTEMS

5.1. Design of Array Processor Systems

The major components of an array processor system are:

1) the *host computer* 2) the *interface system*, including buffer memory and control unit 3) the *connection networks* (for PE-to-PE and PE-to-memory connections) 4) the *processor array*, comprising a number of processor elements with local memory.

A possible overall system configuration is depicted in Figure 5–1, where the design considerations for the four major components are further elaborated. In general, in an overall array processing system, one seeks to maximize the following performance indicators: computing power using multiple devices; communication support, to enhance the performance; flexibility, to cope with the partitioning problems; reliability, to cope with the fault-tolerance problem; and practicality and cost-effectiveness.

5.2. DSP-Oriented Array Processor Chips

The implementation of VLSI chips and the structure of array computing systems depend largely on the established switching technologies. Another important design consideration is the appropriate level of granularity of the processor elements (PE's) composing the array, (cf. Figure 5–2[a]).

For some low-precision digital and image processing applications, it is advisable to consider very simple processing primitives. A good example of a commercial VLSI chip is NCR's Geometric Arithmetic Parallel Processor, or GAPP, which is composed of a 6-by-12 arrangement of single bit processor cells. Each of the 72 processor cells in the NCR45CG72 device contains an ALU, 128 bits of RAM, and bi-directional communication lines connected to its four nearest neighbors: one each to the North, East, South, and West [4]. Each instruction is broadcast to all the PE's, causing the array to perform like a SIMD (single-instruction-multiple-data) machine. The GAPP array, however, is mostly programmed in a low-level (macro-assembly-level) language, posing some programming difficulties for general users.

Many DSP applications require the PE's to include more complex primitives, such as multiply-and-add modules. An example of a commercial chip with a larger granularity is INMOS' Transputer [21]. Transputer is an Occam-language based design, which provides hardware support for both concurrency and communication—the heart of array computing. It has a 32-bit processor capable of 10 MIPS, 4 Kbytes of 50 ns static RAM, and a variety of communications interfaces. It adopts the now popular RISC (reduced-instruction-set-computer) architecture design. The INMOS links, with built-in handshaking circuits, are the hardware representation of the channels for communications. Furthermore, its programming language, Occam, is very suitable for programming wavefront-type array processing. Therefore, the transputer can be readily adapted for the construction of (asynchronous) wavefront arrays.

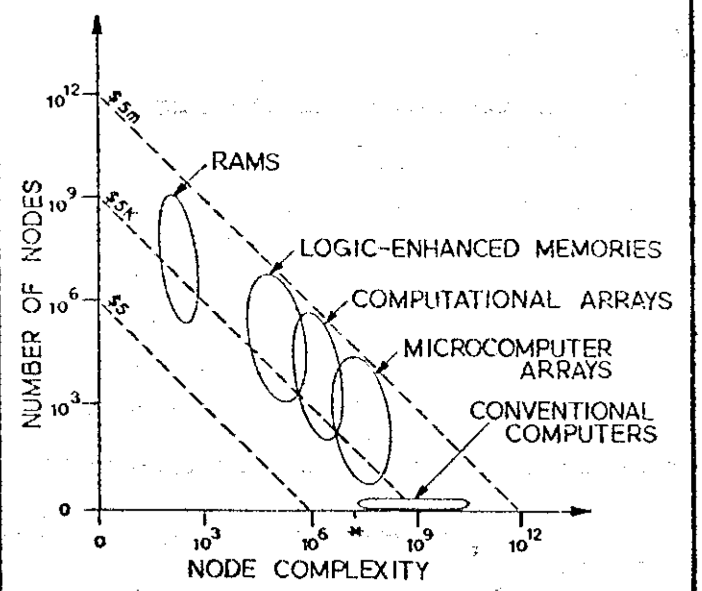


Figure 5–2: Different levels of granularity of the processor element (PE) in array systems, adapted from [18]. An example of smaller PE granularity is NCR's GAPP. The GAPP array lies in the intersection domain between the logic-enhanced-memory group and the computational array group shown in the figure. Such kinds of simple processor primitives are often preferred in low-precision image processing applications. An example of a larger PE granularity is INMOS' Transputer. Many DSP applications require fast multiply/accumulate, high-speed RAM, fast coefficient table addressing; a possible choice appears to be an enhanced version of the transputer discussed in the text. In the figure, a transputer array belongs largely to the micro-computer array domain. Because of built-in asynchronous communication hardware, transputers are very suitable for the implementation of wavefront arrays.

Other examples of commercially available VLSI chips worthy of consideration for array processor implementations are NEC's data flow chip μpd7281 [3], TI's programmable DSP chip TMS320, and recent 32-bit processors such as AMD 29325, Motolora 68020, and Weitek's 32-bit (or the new 64-bit) floating-point chips [20]. Many DSP applications require very special features such as fast multiply/accumulate, high-speed RAM, fast coefficient table addressing, and others. Therefore, the development of a series of customized special-purpose chips for DSP array processors should be given a high priority by the VLSI and DSP research community.

5.3. Comparisons between Systolic and Wavefront Arrays

The main difference between the two array processing schemes is that the wavefront arrays operate asynchronously (at the price of handshaking hardware), while the systolic arrays pulse globally in synchronization with a global clock.

The systolic array features the very important advantages of modularity, regularity, local interconnection, highly pipelined multiprocessing, and continuous flow of data between the PE's. It has a good number of DSP applications. However, the wavefront design offers some additional useful advantages:

Maximum Pipelining: A major thrust of the wavefront array derives from its maximizing the pipelinability by exploring the data-driven nature inherent in many parallel algorithms. This becomes especially useful in the case of uncertain processing times used in individual PE's. As reported in [16], wavefront pipelining may yield a significant speed-up, compared with pure systolic pipelining. (In the special simulation example used in [16], the improvement is a factor of almost two.)

Architectural Extendibility: The wavefront array also highlights the extendibility of the array size, since it can get around the global synchronization requirements. Whereas the asynchronous model in the wavefront arrays incurs a fixed time delay overhead due to the handshaking processes, the synchronization time delay in the systolic arrays is primarily due to the clock skew which changes dramatically with the size of the array. This latter phenomenon will be a potential barrier in the design of ultra-large-scale synchronous computing systems.

Programming Simplicity: The notion of computational wavefront also facilitates the programmability of array processors. By tracing the wavefronts, the description of the space-time activities in the array may be significantly simplified. The previously mentioned parallel processing language, Occam, is very suitable for programming wavefront arrays.

Fault-Tolerance of Array Processors: To enhance the reliability of computing systems, real-time signal processing architectures demand a special attention in run-time fault tolerance. However, 2-D systolic arrays are in general not feasible for run-time fault tolerance design, since it requires a global stoppage of PE's when any failure occurs. It is known that certain fault tolerance issues (roll-back, suspension of execution, etc.) are simpler to handle in data-flow architecture than in other multi-processors [5]. Since wavefront arrays incorporate the data-driven feature into the arrays, they pose similar advantages in dealing with time uncertainties in the fault tolerance environment. For example, once a fault is detected, further propagation of the wavefront will be automatically suspended, according to the wavefront principle. More specifically, due to its asynchronous nature, one only needs to stop the faulty PE, and all subsequent PE's will automatically stop as a ripple. Systolic arrays, by comparison, would probably require a global "error-halt" signal to be broadcast, and the corresponding roll-back problem would be far more cumbersome.

In summary, a systolic array is useful when the PE's are simple primitive modules, since the handshaking hardware in a wavefront array would represent a non-negligible overhead for such applications. On the other hand, a wavefront array is more favored when the PE's involve more complex modules (such as multiply-and-add and lattice or rotation operations), or when a robust and reliable computing environment (such as fault-tolerance) is essential.

6. APPLICATION TO ADAPTIVE NOISE CANCELLATION

From the applicational system perspective, an illustrative problem is the upgrading of modern passive sonar systems by introducing high-speed array processors for front-end signal processing and spectrum estimation computations. The new systems are expected to possess real-time- and high-performance-processing capabilities. This article will use the specific signal processing example of adaptive noise cancellation. More specifically, the author will discuss the McWhirter's algorithm based on least square minimization using QR decomposition [15].

Given any $N \times p$ matrix $\mathbf{X}$ with $N > p$ and an N-element vector $\mathbf{y}$, find the p-element vector of weights $\mathbf{w}$ which minimizes $\|\mathbf{e}\|$, where

$$\mathbf{e} = \mathbf{X}\mathbf{w} + \mathbf{y}$$

and $\|\cdot\|$ denotes the usual Euclidean norm. The problem may be solved by the method of orthogonal triangularization (QR decomposition), which is numerically well-conditioned and described below. An orthogonal matrix $\mathbf{Q}$ is generated such that

$$\mathbf{QX} = \begin{bmatrix} \mathbf{R} \\ \hline \mathbf{O} \end{bmatrix} \quad \text{and} \quad \mathbf{Qy} = \begin{bmatrix} \mathbf{u} \\ \hline \mathbf{v} \end{bmatrix}$$

where $\mathbf{R}$ is a $p \times p$ upper trianglular matrix and $\mathbf{u}$ is a p-element vector. It follows that the least squares weight vector $\mathbf{w}$ must satisfy the equation

$$\mathbf{Rw} + \mathbf{u} = \mathbf{O}$$

which may readily be solved by back-substitution.

The processor of recursive least-squares minimization may be carried out using a triangular wavefront processor, based on pipelining a sequence of Givens rotations, resulting in an elementary orthogonal transformation of the form:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} 0\ldots 0, r_i \ldots r_k \ldots \\ 0 \ldots 0, x_i \ldots x_k \ldots \end{bmatrix} = \begin{bmatrix} 0 \ldots 0, r_i' \ldots r_k' \ldots \\ 0 \ldots 0, 0 \ldots x_k' \ldots \end{bmatrix} \tag{4}$$

The elements c and s may be regarded as the cosine and sine, respectively, of a rotation angle that is chosen to eliminate the leading element of the lower vector. This is illustrated in Figure 6–1 for the case $p = 4$, (cf. [16]).

The Givens rotation method is recursive in the sense that the data from the matrix $\mathbf{X}$ is introduced row by row, and as soon as each row $\mathbf{x}_n^T$ has been absorbed into the computation, the resulting triangular matrix $\mathbf{R}(n)$ represents an exact QR decomposition for all data processed up to that stage.

The boundary cell in each row (indicated by a large circle in Figure 6–1) computes the rotation parameters c and s appropriate to the internally stored components and the vertically propagating data vector. These rotation parameters are then passed horizontally to the right. The internal cells (indicated by squares) are subsequently used to apply the same transformation to all other elements of the received data vector. Note that when the triangular

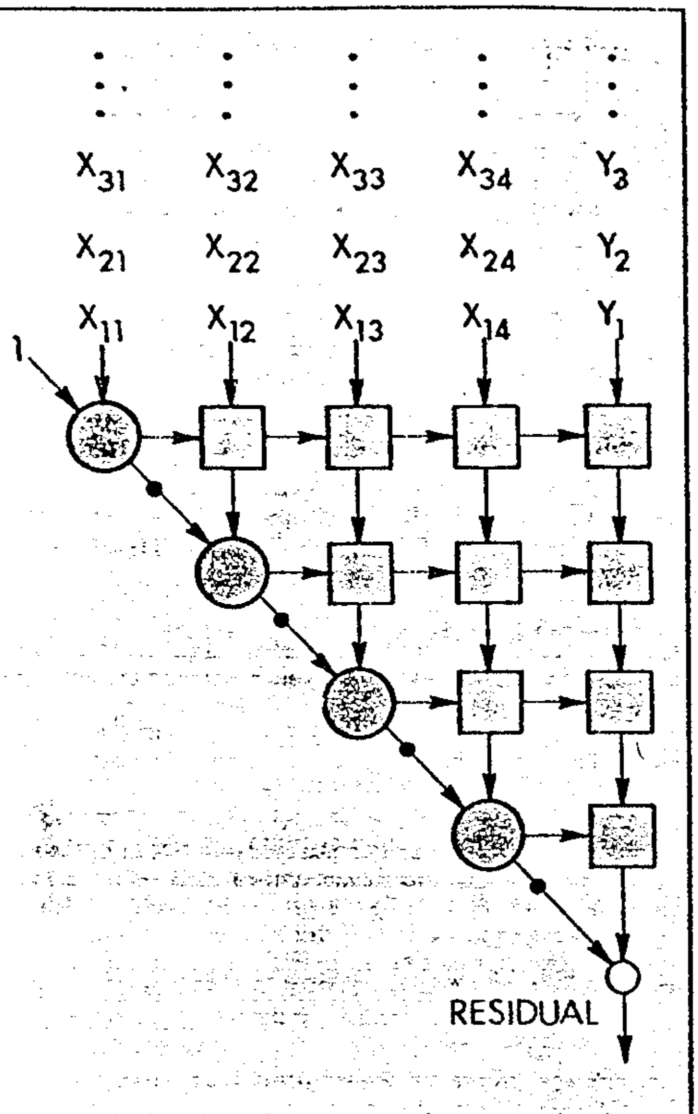

X_{31} X_{32} X_{33} X_{34} Y_3
X_{21} X_{22} X_{23} X_{24} Y_2
X_{11} X_{12} X_{13} X_{14} Y_1
1
RESIDUAL

Figure 6–1: Wavefront array for recursive least-squares minimization, adapted from [16]. As a least-square solver array, each of the non-diagonal PE's performs a basic Givens rotation as shown in (4). Note that the diagonal (circle) PE's are responsible for generating the rotation angle parameters c and s. The parameters are then propagated rightwards to all the PE's along the same row to be used for the rotation operations.

array illustrated in Figure 6–1 is operated in the wavefront processor mode, the operation of PE is no longer controlled by a globally distributed clock signal. Instead, each PE receives its input data from the specified directions, performs the specified functions, and delivers the appropriate output values to neighboring PE's. The operation of each cell is controlled locally and depends on the necessary input being available, and on its previous outputs having been accepted by the appropriate nearest neighbors. As a result, it is not necessary to impose a temporal skew on the input data matrix. For example, let us consider the top processor in the right hand column of the figure. In the wavefront case, this will not operate on its first input data sample y until the required rotation parameters c and s are available from the neighboring processor on the left.

As each row $\mathbf{x}_n^T$ of the matrix **X** moves down through the array, it interacts with the previously stored triangular matrix **R**(n − 1) and is eliminated by the sequence of Givens rotations. As each element y_n moves down through the right hand column of processors, it undergoes the same sequence of Givens rotations, interacting with the previously stored vector **u**(n − 1) and generating the updated vector **u**(n) in the process. It follows that the exact least-squares solution could be derived at every stage of the process by solving the corresponding triangular linear system for **w**(n) [7].

In many least-squares applications, the primary objective is to compute the sequence of residuals

$$e_n = x_n^T w(n) + y_n$$

while the associated weight vectors w(n) are not of direct interest. It has recently been shown [15] how the residual e_n at each stage of the computation may be generated quite simply by using the array in Figure 6–1, eliminating the need to solve the associated triangular linear system for w(n). The parameter x_{out} produced by the bottom processor in the right hand column of cells is simply multiplied by the parameter y_{out} that emerges from the final boundary processor. This produces the residual directly [16].

7. SUMMARY

We have witnessed the rapid growth of signal processing and computing technology that followed the inventions of the transistor in the 1940's and integrated circuits in the late 1950's. The emergence of new VLSI technology, along with modern engineering workstations, CAD tools, and other hardware and software advances in computer technology, virtually assure a revolutionary information processing era in the near future. The signal processing community will very soon face a prevailing impact of modern VLSI technologies regarding future integration of computers, communications, control, command, intelligence and information.

VLSI technology, starting as a device research area, provides opportunities and constraints that will open up new areas of research in computer architecture. From a scientific research perspective, a close interaction between *VLSI* and *array architecture* research areas will be essential. This paper has identified several novel architectures that maximize the strength of VLSI in terms of intensive and pipelined computing, and yet circumvent its main limitations in reliability and communication. In the author's opinion, research and development in the array processors will not only benefit from the revolutionary VLSI technology; it will also play a central role in shaping the course of algorithmic, architectural, and applicational trends of future supercomputer technology.

REFERENCES

[1] Burg, J. P. *Maximum Entropy Spectral Analysis.* PhD thesis, Stanford University, 1975.

[2] Capon, J. (1969). "High-Resolution Frequency-Wavenumber Spectrum Analysis," *Proc. IEEE,* 57, 1408–1418.

[3] Chase, M. (1984). "A Pipelined Data Flow Architecture for Digital Signal Processing the NEC μPD7281," *IEEE Workshop on VLSI Signal Processing,* Los Angeles, Nov. 1984.

[4] Davis, R. H., and Thomas, D. (1984). "Systolic array chip matches the pace of high-speed processing," *Electronic design,* Vol.: 207–218, October, 1984.

[5] Dennis, J. B. (1980). "Data Flow Supercomputers," *IEEE Computer,* Nov, 1980, 48–56.

[6] Fisher, A. L., and Kung, H. T. (1985). "Special-purpose VLSI Architectures: General Discussions and a Case Study," In *VLSI and Modern Signal Processing,* Kung, S. Y., Whitehouse, H. J., and Kailath, T. K. (eds.) Prentice-Hall, Inc.

[7] Gentleman, W. M., and Kung, H. T. (1981). "Matrix Triangularization by Systolic Array," *SPIE Proc. Real-Time Signal Processing IV* 298.

[8] Hayes, J. P. (1978). *Computer Architecture and Organization.* McGraw Hill, N.Y.

[9] Kung, S. Y., (1984). "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE,* Vol. 72(7).

[10] Kung, H. T., and Leiserson, C. E. (1978). "Systolic Arrays (for VLSI)," *Sparse Matrix Symposium,* SIAM, 256–282.

[11] Kung, S. Y., Arun, K. S., Gal-Ezer, R. J., and Bhaskar Rao, D. V. (1982). "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Transactions on Computers, Special Issue on Parallel and Distributed Computers* C-31(11): 1054–1066.

[12] Kung, S. Y., and Hu, Y. H. (1983). "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," *IEEE Transactions on ASSP,* ASSP-31(No. 1) 66–76.

[13] Kung, S. Y., and Lo, S. C. (1985). "A Spiral Systolic Architecture/Algorithm for Transitive Closure Problems," *Submitted to IEEE Trans. on Computers, Special Issue on Distributed Computing,* Vol. C-34(12).

[14] Mason, S. J. (1953). "Feedback Theory—Some Properties of Signal Flow Graphs," *Proceedings, IRE,* 41: 920–926.

[15] McWhirter, J. G. (1983). "Recursive least-squares minimization using a systolic array," *Real Time Signal Processing VI,* 105.

[16] Broomhead, D. S., et al. (1984). "A Practical Comparison of the Systolic and Wavefront Array Processing Architectures," *Proceedings,* IEEE Workshop on VLSI Signal Processing, Los Angeles, November, 1984.

[17] Mead, C., and Conway, L. (1980). *Introduction to VLSI Systems,* Addison-Wesley.

[18] Seitz, C. (1984). "Concurrent VLSI Architectures," *IEEE Transactions on Computer,* C-33.

[19] Stewart, G. W. (1973). *Introduction to Matrix Computations,* Academic Press.

[20] Ware, F., et al. (1984). "Fast 64-bit chip set gangs up for double-precision floating-point work," *Electronics,* 99–103, July, 1984.

[21] Wilson, P., (1984). "Thirty-two bit micro supports multiprocessing," *Computer Design,* 143–150. June, 1984.

Sun-Yuan Kung received his B.S. Degree in Electrical Engineering in 1971 from the National Taiwan University, Taipei, Taiwan; M.S. Degree in Electrical Engineering in 1974 from the University of Rochester, in Rochester, New York and Ph.D. Degree in Electrical Engineering in 1977 from the Stanford University, Stanford, California.

In 1974, he joined the Amdahl Corporation, Sunnyvale, as an Associate Engineer in LSI design and simulation. From 1974 to 1977, he was a Research Assistant of Information Systems Laboratories, Stanford University. Since July 1977, he has joined the faculty of Electrical Engineering-Systems in the University of Southern California, Los Angeles, California, where he is presently an Associate Professor. In 1984, he was a Visiting Professor at the Stanford University; and later in the same year a Visiting Professor at the Delft University of Technology, The Netherlands.

Dr. Kung's research interests are in the areas of approximation theory in linear systems, digital signal processing, modern and high-resolution spectrum analysis, parallel array processors and VLSI supercomputing for signal processing. Since 1981, he has been in charge of the ONR Selected Research Project (SRO-II) on the development of massively parallel signal processors. He served as the General Chairman of the USC Workshop on VLSI Signal Processing, Los Angeles, November 1982. He was a U.S. delegate to the US-Japan Joint Seminar on Mathematical Systems Theory, Gainesville, Florida, April 1983; and was an editor of an advanced research book on "VLSI and Modern Signal Processing," Prentice-Hall, Inc., August, 1984. He currently serves on the IEEE ASSP Technical Committee on VLSI and is the Associate Editor for the VLSI area in the IEEE Transactions on Acoustics, Speech and Signal Processing.

Dr. Kung is a Member of ACM and a Senior Member of IEEE.