On Supercomputing with Systolic/Wavefront Array Processors

SUN-YUAN KUNG, SENIOR MEMBER, IEEE

Invited Paper

In many scientific and signal processing applications, there are increasing demands for large-volume and/or high-speed computations which call for not only high-speed computing hardware, but also for novel approaches in computer architecture and software techniques in future supercomputers. Tremendous progress has been made on several promising parallel architectures for scientific computations, including a variety of digital filters, fast Fourier transform (FFT) processors, data-flow processors, systolic arrays, and wavefront arrays. This paper describes these computing networks in terms of signal-flow graphs (SFG) or data-flow graphs (DFG), and proposes a methodology of converting SFG computing networks into synchronous systolic arrays or data-driven wavefront arrays. Both one- and two-dimensional arrays are discussed theoretically, as well as with illustrative examples. A wavefront-oriented programming language, which describes the (parallel) data flow in systolic/wavefront-type arrays, is presented. The structural property of parallel recursive algorithms points to the feasibility of a Hierarchical Iterative Flow-Graph Design (HIFD) of VLSI Array Processors. The proposed array processor architectures, we believe, will have significant impact on the development of future supercomputers.

I. INTRODUCTION

The increasing demands for high-performance signal processing and scientific computations indicate the need for tremendous computing capability, in terms of both volume and speed. The availability of low-cost, high-density, fast processing/memory devices will presage a major breakthrough in future supercomputer designs, especially in the design of highly concurrent processors.

Current parallel computers can be characterized into three structural classes: vector processors, multiprocessor systems, and array processors [13]. The first two classes belong to the general-purpose computer domain. The development of these systems requires a complicated design of control units and optimized schemes for allocation of machine resources. The third class, however, belongs to the domain of special-purpose computers. The design of such systems requires a broad knowledge of the relationship between parallel-computing algorithms and optimal-computing hardware and software structures.

It is this last class that we shall focus upon, since it offers a promising solution to meet real-time processing requirements. Especially, locally interconnected computing networks, such as systolic and wavefront arrays, are well suited to efficiently implement a major class of signal processing algorithms due to their massive parallelism and regular data flow [15], [17]. Such architectures promise real-time solutions to a large variety of advanced computational tasks.

This paper first discusses some important design considerations for massively parallel VLSI array processors and the algorithmic background of these array processors. This will lead to a systematic software/hardware design approach. Specifically, we discuss the methodology of imposing systolic architectures and/or data-flow computing onto signal-flow graph computing networks. The concept of computational wavefronts, which leads to systolic-type and wavefront-type architectures, is reviewed. A wavefront programming language is proposed to broaden the applications of the wavefront/systolic type arrays.

A. Architectural Considerations in Array Processor Design

There are many important issues in designing array processor systems, such as processor interconnection, system clocking, and modularity [19].

Interconnection in massively parallel array processors is the most critical issue of the system design, since communication is very expensive in terms of area, power, and time consumption [26]. Therefore, communication has to be restricted to *localized interconnections*. To avoid global interconnections, local and regular data movements are strongly preferred. This is the most salient characteristic of systolic and wavefront arrays.

The clocking scheme is also a very critical issue. In the globally synchronous scheme, there is a global clock network which distributes the clock signal over the entire array. For very large systems, the clock skew incurred in global clock distribution is a nontrivial factor, causing unnecessary slowdown in the clock rate [18], [10]. Under this circumstance, the self-timed scheme is more preferable.

Large design of layout costs suggest using repetitive modular structures, i.e., a few different types of simple (and often standard) cells. Thus we have to identify the primitives that can be implemented efficiently and optimally realize the potential of new device technologies.

Programmable processor modules (as opposed to dedicated modules) are favored due to cost-effectiveness considerations. The high cost of designing such modules may be amortized over a broader applicational domain. Indeed, a major portion of scientific computations can be reduced to a basic set of matrix operations and other related algo-

Manuscript received February 9, 1984; revised March 5, 1984. This research was supported in part by the Office of Naval Research under Contracts N00014-81-K-0191 and N00014-83-C-0377 and by the National Science Foundation under Grant ECS-82-12479.

The author is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089, USA.

0018-9219/84/0700-0867$01.00 ©1984 IEEE

rithms. These should be exploited in order to simplify the hardware module.

B. Mapping Parallel Algorithms onto Locally Interconnected Computing Networks

There are quite a few software packages for scientific computation and image/signal processing algorithms available today. For example, LINPAK [9] and EISPAK [29] are popular packages for many scientific computations, especially those using various types of matrix operations (such as). However, the execution time of these algorithms, running on conventional computers, is often too slow for real-time applications. Fortunately, VLSI and other new techniques have made high-speed parallel processing economical and feasible. When mapping these algorithms onto parallel processors, typical questions often raised are: "How to fully utilize the inherent concurrency in these algorithms?" "How are these algorithms best implemented in hardware?" "What kind of array processor(s) should one turn to for a specific application?"

After examining most of the algorithms collected in the aforementioned packages, some prominent traits surface, such as localized operations, intensive computation, and matrix operands. The common features of these algorithms should be exploited to facilitate the design of array processors for signal processing applications.

An array processor is composed of an array of processor elements (PE) with direct (static) or indirect (dynamic) interconnections, including linear, orthogonal, hexagonal, tree, perfect-shuffle, or other types of structures. The most critical issue is communication, i.e., moving data between PE's in a large-scale interconnection network.

Correspondingly, a communication-oriented analysis on parallel algorithms will be most useful for mapping algorithms onto the arrays. To conform with the constraints imposed by VLSI, this paper will emphasize a special class of algorithms, i.e., *recursive* and *locally dependent* algorithms.[1]

For proper communication in an interconnected computing network, each PE in the array should know

a) **where** to send (or fetch) data, and
b) **when** to send (or fetch) data.

When mapping a locally recursive algorithm onto a computing network, it allows a simple solution to the question "a) **where** to send the data?" because the data movements can be confined to nearest neighbor PE's. Therefore, locally interconnected computing networks will suffice to execute the algorithm with high performance.

The conventional approach to the second question "b) **when** to send the data?" is to use a globally synchronous scheme, where the timing is controlled by a sequence of "beats" [30]. A prominent example is the systolic array [16], [15] However, locality can have two meanings in array processor designs: localized data transactions and/or localized timing scheme (i.e., using self-timed, data-driven control). In fact, the class of locally recursive algorithms permits both locality features; these should be exploited in the architectural designs. An example for such a design is the wavefront array [17].

II. Signal-Flow Graph (SFG) Computing Networks

The most useful graphical representation for scientific and signal processing computations is the signal-flow graph (SFG). While the graphical representations are most popularly used for signal processing flow diagrams, such as FFT and digital filters, etc., the SFG representations in fact cover a broad domain of applications, including linear and nonlinear, time-varying and time-invariant, and multidimensional systems. For convenience, this paper will treat only time-invariant SFG systems.[2]

Notations: In general, a **node** is often denoted by a circle representing an arithmetic or logic function *performed with zero delay*, such as multiply, add, etc. (cf. Fig. 1(a)). An **edge**, on the other hand, denotes either a function

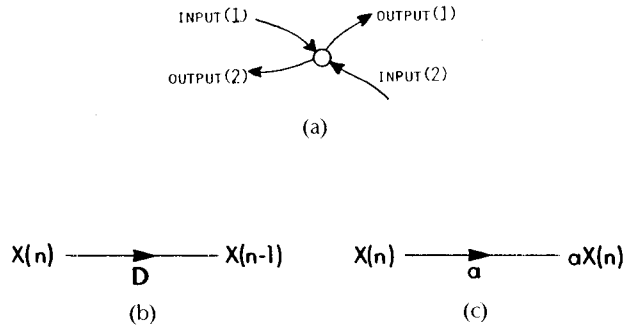


Fig. 1. Examples of SFG graphical denotations. (a) An operation node with (two) inputs and (two) outputs. (b) An edge as a delay operator. (c) An edge as a multiplier.

or a delay. Unless otherwise specified, for a large class of signal processing SFGs, 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 Fig. 1(a)). On the other hand, if an edge is labeled with a lower case letter, such as a, a_i, b_i, it represents a multiplication by a constant a, a_i, b_i (see Fig. 1(b)).

When the concept of the SFG was originally conceived, there was little consideration given to the locality preferences in parallel-computing network design. Hence this paper addresses the issue of systematic approaches of mapping SFGs into locally interconnected parallel-array processors.

There are two major classes of SFGs: those with *local interconnections*, and those with *global interconnections*. A typical example of a global SFG is one representing the FFT algorithm. The principle of the (decimation-in-time) FFT is based on successively decomposing the data, say $\{x(i)\}$, into *even* and *odd* parts. This partitioning scheme will result in global communication between PEs. More precisely, the FFT recursions can be written as (using the "in-place" computing scheme [27])

[1] In a recursive algorithm, all processors do nearly identical tasks and each processor repeats a fixed set of tasks on sequentially available data.

[2] This incurs no loss of generality, since any internal time-varying parameters can be equivalently represented by an (external) input signal.

$$x^{(m+1)}(p) = x^{(m)}(p) + w_N^r x^{(m)}(q)$$
$$x^{(m+1)}(q) = x^{(m)}(p) - w_N^r x^{(m)}(q)$$

with p, q, r varying from stage to stage. The "distance" of the global communication involved will be proportional to $|p - q|$. An SFG for the (decimation-in-time) FFT, with space and time indices properly labeled, is shown in Fig. 2. Note

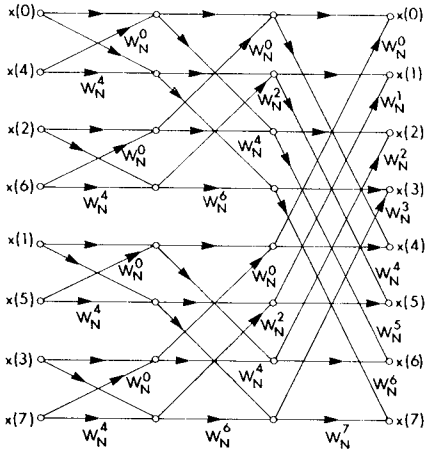


Fig. 2. A signal-flow graph (SFG) for the decimation-in-time FFT algorithm.

that in the last stage the maximum distance is $|p - q| = N/2$ (see [27]). For example, the maximum distance will be 512 units for a 1024-point FFT. Thus the FFT algorithm is a *global one*, since the recursion involves globally separated indices. Therefore, FFT computing structures will call for spatially global interconnections, and cannot be easily mapped onto systolic or wavefront arrays.

In contrast to the FFT algorithm, most other recursive signal processing algorithms are *local*, i.e., the spatial separations between nodes are within a certain limit. Therefore, the corresponding SFGs are "localizable." For examples, the SFGs for FIR and IIR filters can be easily implemented with spatially local SFGs. Generally, an IIR (infinite impulse response) filter is defined by the difference equation

$$y(k) = \sum_{m=1}^{N} x(k-m)b(m) + \sum_{m=1}^{N} y(k-m)a(m). \quad (1)$$

(Note that FIR filtering, linear convolution, and transversal filterings are simply special cases when $a(m) = 0$.)

A popular SFG[3] [27] for (1) is shown in Fig. 3.

We note that this SFG has spatially local interconnections. But it is not *temporally local*, since according to the SFG, propagating a datum "X" from, say, the left-most node to the right-most node uses "zero" time. More precisely, the SFG imposes the requirement that the datum "X" has to be **broadcast** to all the nodes on the upper path. This is certainly undesirable from a systolic design perspective

[3]For this section, a node (circle) commonly denotes addition when there are multiple outputs and single input to the node. It denotes a branching node if there is one input and multiple outputs.

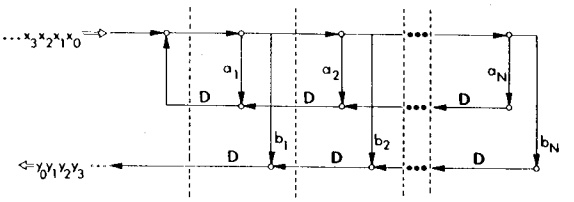


Fig. 3. Direct form design of ARMA (IIR) filter. The SFG is spatially localized but *not temporally localized*.

(and unrealistic for a circuit implementation). This will be the focus point of the next section.

III. Systolization of SFG Computing Networks

A. Systolic Array

Systolic processors [16], [15] are a new class of "pipelined" array architectures. According to Kung and Leiserson [16], "A systolic system is a network of processors which rhythmically compute and pass data through the system." For example, it is shown in [16] that some basic "inner product" PEs ($Y \leftarrow Y + A*B$) can be *locally* connected together to perform digital filtering, matrix multiplication, and other related operations. The systolic array features the important properties of modularity, regularity, local interconnection, a high degree of pipelining, and highly synchronized multiprocessing. The data movements in a systolic array are often described in terms of the "snapshots" of the activities [16].

There are no formal or coherent definitions of the systolic array in literature. In order to have a formal treatment of the subject, however, we shall adopt the following definition:

1) Definition: Systolic Array: A systolic array is a computing network possessing the following features:

a) *Synchrony:* The data are rhythmically computed (timed by a global clock) and passed through the network.

b) *Regularity (i.e., Modularity and Local Interconnections):* The array should consist of modular processing units with regular and (spatially) local interconnections. Moreover, the computing network may be extended indefinitely.

c) *Temporal Locality:* There will be at least one unit-time delay allotted so that signal transactions from one node to the next can be completed.

d) *Pipelinability (i.e., O(M) Execution-Time Speed-Up):* A good measure for the efficiency of the array is the following

$$\text{Speed-Up Factor} = \frac{\text{Processing Time in a Single Processor}}{\text{Processing Time in the Array Processor}}$$

A systolic array should exhibit a *linear-rate pipelinability*, i.e., it should achieve an $O(M)$ speed-up, in terms of processing rates, where M is the number of processor elements (PEs).

We note that a *regular* SFG, such as the canonic SFG for ARMA filters, is already very close to a systolic array. The major difference being that most SFGs are not given in temporally localized form. Therefore, it is important to be able to convert them into localized ones. The topic of imposing temporal locality into a computing network has been a focus point of several researchers, including a series

of publications by Fettweis [31], Leiserson [22]–[24], etc., and, more recently, the works reported in [20], [25], [14], [2]. The main advantage of the (cut-set) scheme proposed here (largely based on [20]) lies in its simplicity to use and its straightforward proof. Our proof is based on a graph-theoretical result—the *colored arc lemma*, which will be discussed momentarily.

2) Systolic Array for ARMA (IIR) Filter: Before discussing the general procedure, let us first take a look at an example. The canonic SFG for an ARMA Filter in Fig. 3 can be easily converted into a local-type one. Our first step in Fig. 4(a) is to rescale the time unit by setting $D = 2D'$. After shifting one of the two delays from the upper edges to the corresponding lower edges, a modified design can be derived as in Fig. 4(a).

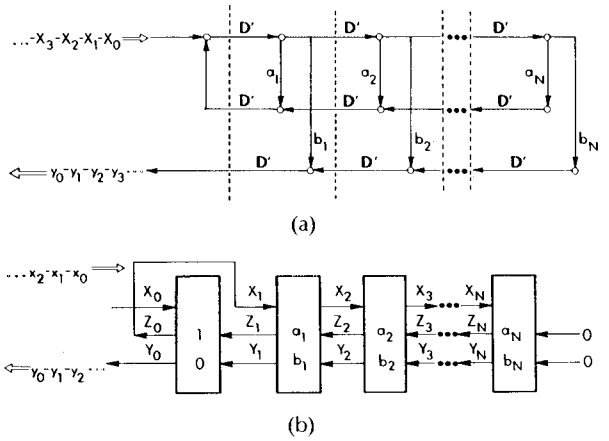


Fig. 4. (a) A modified SFG for an ARMA filter—a systolized version. (b) A systolic array for an ARMA (IIR) filter.

To verify that Fig. 4(a) yields the same transfer function as Fig. 3, one can simply check that the transfer function remains the same [20]. A more general proof will be discussed in a moment.

Let us now demonstrate how the modified form can be trivially converted into a systolic design. To do this, the operation time for one multiplication, one addition, and data transfer are merged with the uni-time delay D'. Therefore, the delay D', the multiplier, and the adder in each single section (defined by means of dashed lines) in Fig. 4(a) are all merged into an "inner product" processor. This leads to an overall systolic array configuration for IIR filter as shown in Fig. 4(b).

Now we are ready to discuss a general systematic procedure for converting SFGs into systolic arrays. First, we need a procedure to convert an SFG into a temporally localized SFG, which contains only nonzero-delay edges between modular sections.

Definition: Cut-Set:
A *cut-set* in an SFG is a minimal set of edges which partitions the SFG into two parts.

B. A Cut-Set (Temporal) Localization Procedure

The localization 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 by a factor α (with respect to the new time unit D') (see Fig. 4(a)).

Rule (ii) Delay-Transfer: Given any cut-set of the SFG, we can group the edges of the cut-set into *inbound edges* and *outbound edges*, depending upon the directions assigned to the edges. Rule (ii) allows advancing k (D') time units on all the outbound edges and delaying k time units on the inbound edges, and *vice versa*. 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[4] by a lag of $+k$ time units or an advance of $-k$ time units.

We shall refer to these two basic rules as the (cut-set) *localization rules*. Based on these rules, we assert the following.

Theorem:
All computable[5] SFG's are temporally localizable.

Proof of the Theorem: We claim that the localization Rules (i) and (ii) can be used to "localize" any (targeted) zero-delay edge, i.e., to convert it into a nonzero-delay edge. This is done by choosing a "good" cut-set and apply the rules upon it. A good cut-set including the "target edge" should not include any "bad edges," i.e., those zero-delay edges in the opposite direction of the target edge. This means that the cut-set will include only i) the target edge, ii) nonzero delay edges going in either direction, and iii) zero-delay edges going in the same direction. Then, according to Rule (ii), the nonzero delays of the opposite-direction edges can "give" one or more spare delays to the target edge (in order to localize it). If there are no spare delays to give away, simply scale all delays in the SFG according to Rule (i) to create enough delays for the transfer needed.

Therefore, the only thing left to prove is that such a "good" cut-set always exists. For this, we refer to Fig. 5., in which we have kept only all of the zero-delay *successor edges* and the zero-delay *predecessor edges* connected to the target edge, and removed all the other edges from the graph. In other words, Fig. 5 depicts the bad edges which should not be included in the cut-set. As shown by the dashed lines in Fig. 5, there must be "openings" between these two sets of bad edges—otherwise, some set of zero-delay edges would form a zero-delay loop, and the SFG would not be computable. Obviously, any cut-set "cutting" through the openings is a "good" cut-set, thus the existence proof is completed. (The author was later advised by a colleague that the existence proof discussed above is in fact a result known as the *colored arc lemma* in graph theory.) It is clear that repeatedly applying the localization Rule (ii) (and (i), if necessary) on the cut-sets will eventually lead to a temporally localized SFG.

[4]If there is more than one cut-set involved, and if the input and output are separated by more than one cut-set, then such adjustment factors should be accumulated.

[5]An SFG is meaningful only when it is computable, i.e., there exists no zero-delay loop in SFG.

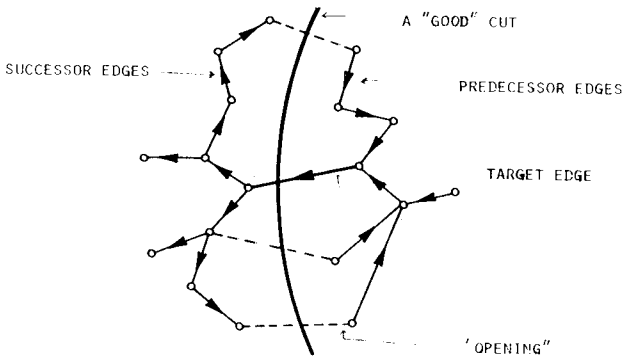


Fig. 5. "Openings" between bad edges ensure the existence of a "good" cut-set. This may be used as a clue for selecting a "good" cut-set.

C. Systolization Procedure

As we have claimed earlier, a regular SFG is *almost* equivalent to a systolic array and can be easily systolized. The systolization procedure, essentially based on the cut-set localization rules, is outlined below:

1) *Selection of Basic Operation Modules.* The choice may not be unique. In general, the finer the granularity of the basic modules, the more efficient (in speed) a systolic array will be. (A comparison of two possible lattice modules for a systolic array will be discussed in the next subsection.)

2) *Applying Localization Rules.* If the given SFG is regular, i.e., modular and spatially local, then regular cut-sets can be selected and the above rules can be used to derive a regular and temporally localized SFG.[6]

3) *Combination of Delay and Operation Modules:* To convert such an SFG into a systolic form, we need only to successfully introduce a delay into each of the operation modules, such as $A + B*C$. Combine the delay with the module operation to form a basic systolic element. All the extra delays will be modeled as pure delays without operations.

4) *Verification of the Arrays:* An SFG representation for linear, time-invariant systems is directly verifiable by the Z-transform technique. The correctness of the transformation to a systolic design is also guaranteed by the cut-set rules. Therefore, there is no need to display snapshots for the verification purpose. Nevertheless, one may find snapshots a simple and useful tool for a better appreciation of the movement of data.

IV. Examples of the Systolization Procedure

In this section, we shall apply the systolization procedure to some one- and two-dimensional SFGs.

A. Systolic Lattice Filters

For the one-dimensional case, a very interesting example is the systolic implementation of a digital lattice filter, which should have many important applications to speech and seismic signal processing. For this example, let us now apply the transformation rules to the SFG for an autoregressive (AR) lattice filter, as shown in Fig. 6(a). There are two possible choices of basic operation modules for the lattice array: (A) a lattice operation module,[7] and (B) a multiply/add (AM) basic module. Note that in each lattice operation there are two MA operations—implying that the lattice operation uses twice the time of MA.

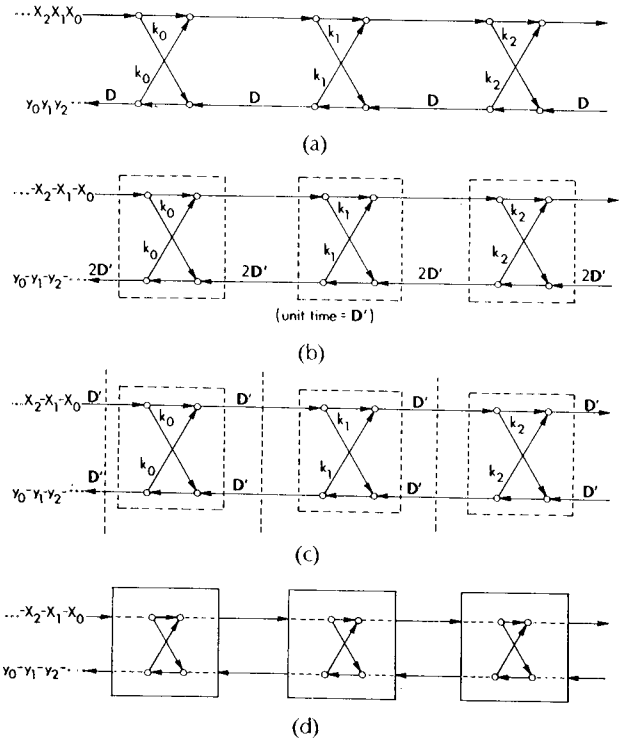


Fig. 6. (a) An SFG for AR lattice filters. (b) Time-rescaled SFG for AR lattice filters (type A). (c) "Localized" SFG for AR lattice filters (type A). (d) Systolic array for AR lattice filters (type-A).

1) Lattice Systolic Array (Type-A): By localization Rule (i), we first double each delay, i.e., $D \rightarrow 2D'$ as shown in Fig. 6(b). Apply (uniformly) the cuts to the SFG and subtract one delay from each of the left-bound edges and, correspondingly, add one delay to each of the right-bound edges in the cut-sets. This yields Fig. 6(c). Finally, by combining the delays with the lattice module, we have the final systolic structure, as in Fig. 6(d). Note that, because of the time-scaling, the input sequence $\{x(i)\}$ will be interleaved with "blanks" to match the adjusted delays. It is clear that $\alpha = 2$ and the array can yield an $M/2$ execution-time speedup.

2) Lattice Systolic Array (Type-B): By the localization Rule (i), we first triple each delay; i.e., $D \rightarrow 3D'$. (The resultant SFG is the same as what is shown in Fig. 6(b), but substituting $2D'$ by $3D'$.) Apply uniform cut-sets to the SFG as shown in Fig. 7(a). Subtract two delays from each left-bound edge, and, correspondingly, add two delays to every right-bound edge in the cut-sets. This yields Fig. 7(a). Now let us use a cut-set partitioning the upper edges from the lower edges as shown in Fig. 7(b). Transfer one delay from the down-going edges to the up-going ones. The result is depicted in Fig. 7(b). Finally, by combining the delays with

[6]In order to preserve the modular structure of the SFG (a basic feature of systolic design), the cut-set localization should be applied uniformly across the network. Otherwise, the resultant array may not be systolic.

[7]That is, the lattice operation is now treated as a single module—a desirable choice of CORDIC implementation.

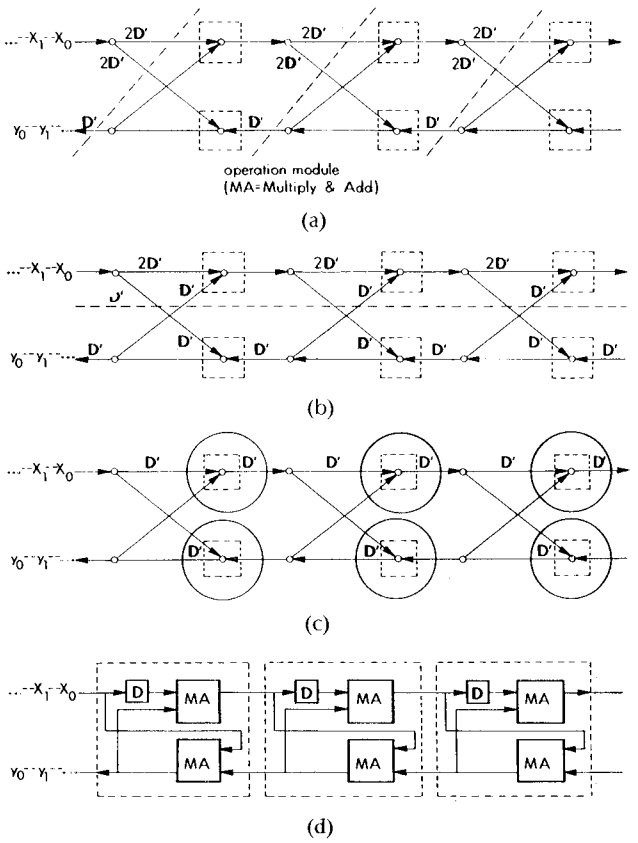


Fig. 7. (a) Time-rescaled SFG (and cut-sets) for AR lattice filters (type-B). (b) Partially localized SFG—first step. (c) "Localized" SFG—all operations are ready to merge with the corresponding uni-time delays D'. (d) Systolic array for AR lattice filters (type-B) with the small squares denoting pure delays.

the Multiply–Add Module (cf. Fig. 7(c)), a systolic structure is obtained as shown in Fig. 7(d). Note that because of time scaling the input sequence $\{x(i)\}$ will be interleaved with two "blanks" to match the adjusted rates. It is clear that the array can achieve a $2M/3$ execution-time speedup.[8] In terms of speed, this systolic design is superior to the Type-A design.

B. Two-Dimensional Systolic Arrays

The systolization procedures can be applied to two-dimensional networks. Although the descriptions of two-dimensional activities are often cumbersome; fortunately, the SFG representations of two-dimensional algorithms (especially, the (temporally) nonlocalized version) are often much easier to comprehend. With the procedures discussed in the previous section, the conversion of a two-dimensional SFG to a systolic array is straightforward.

A typical example used for illustrating a two-dimensional array operation is matrix multiplication. Let

$$A = [a_{ij}] \qquad B = [b_{ij}]$$

and

$$C = A \times B.$$

[8]Note that since the upper and lower MA modules in each PE are never simultaneously active, only one MA hardware module suffices to serve both functional needs.

Suppose that both A and B are nonsparse $N \times N$ matrices. The matrix A can be decomposed into columns A_i and the matrix B into rows B_j and, therefore,

$$C = [A_1B_1 + A_2B_2 + \cdots + A_NB_N]$$

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

$$c_{i,j}^{(k)} = c_{i,j}^{(k-1)} + a_i^{(k)} b_j^{(k)} \tag{2}$$

$$a_i^{(k)} = a_{ik}$$

$$b_j^{(k)} = b_{kj}$$

for $k = 1, 2, \cdots, N$ and there will be N sets of wavefronts involved.

1) Systolizing an SFG for Matrix Multiplication: The simplest matrix multiplication array design is one letting columns A_i and rows B_i be broadcast instantly along the square array as shown in Fig. 8(a). All outer products will then be sequentially summed via a loop with single delay. This design is not suitable for VLSI circuit design since it needs to use global communication. However, there is a rapidly growing interest in the developments of optical array processors, [12], [3]. From an optical interconnection perspective this SFG may be directly implementable.

If local interconnection is preferred, the proposed procedure in Section III can again be used to systolize the SFG. Let us apply Rule (ii) to the cut-sets shown in Fig. 8(a). The systolized SFG will have one delay assigned to each edge and thus represent a localized network. According to Rule (ii), 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. By counting the cut-sets involved in Fig. 8(a), it is clear that the first column of B needs no extra delay, the second column needs one delay, the third needs two (i.e., attributing 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 Fig. 8(c). A similar arrangement can be applied to A.

2) Multiplication of a Banded Matrix and a Full Matrix: Let us look at a slightly different, but commonly encountered, type of matrix multiplication problem. This involves a banded-matrix A, $N \times N$, with bandwidth P, and a rectangular matrix B, $N \times Q$. This situation arises in many application domains, such as DFT and time-varying (multichannel) linear filtering, etc. In most applications, $N \gg P$ and $N \gg Q$, and this makes the use of $N \times N$ arrays for computing $C = A \times B$ very uneconomical.

Fig. 9(a) shows that, with slight modification to the SFG in Fig. 8(a), the same speedup performance can be achieved with only a $P \times Q$ rectangular array (as opposed to an $N \times Q$ array). Now, the left memory module will store the matrix A along the band-direction (see Fig. 9(a)) 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 also be outputted from the I/O ports of the top-row PEs.

Applying the systolization procedure leads to the data array as depicted in Fig. 9(b).

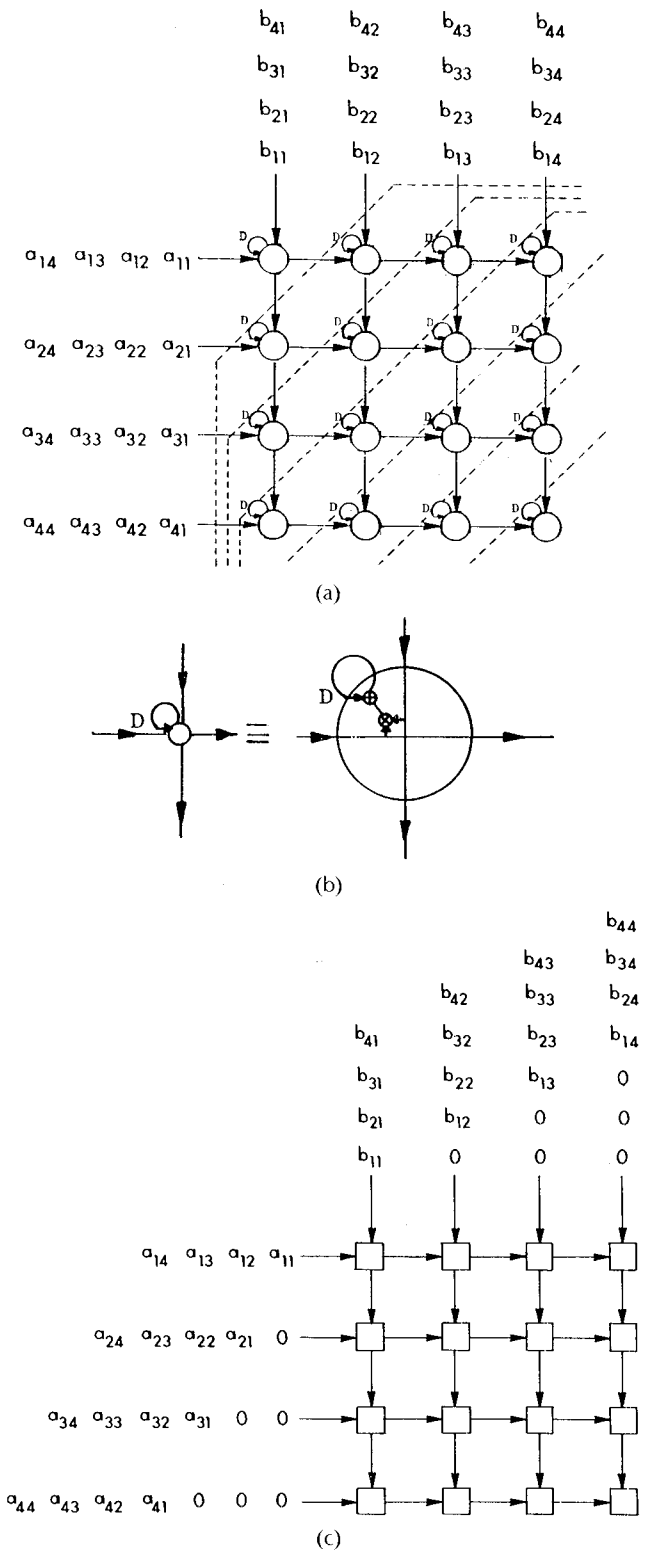


Fig. 8. (a) An SFG for matrix multiplication. (b) The detailed diagram of the processing nodes. (c) A systolic array for matrix multiplication.

3) Multiplication of Two Banded Matrices: Another interesting case is the situation when both A and B are banded matrices, with bandwidths P and Q, respectively. Let us assume that $N \gg P$ and $N \gg Q$, where P and Q are bandwidths for A and B, respectively. Then it is possible to achieve full parallelism with only a $P \times Q$ rectangular array (as opposed to an $N \times N$ array).

Now, the left- and upper memory modules will store the matrices A and B (respectively) along the band direction (see Fig. 10(a)). The delayed feedback edge (with partial sum of the outer products) will be along the diagonal direction (to the N–W direction). This is because both A and B are stored in the skewed version of Fig. 10(a). Applying the systolization procedure to the cut-sets as shown in Fig. 10(a) will call for a triple scaling of $D \rightarrow 3D'$. (This is because each north–west-bound delay edge is "cut" twice.) The procedure leads to an array configuration depicted in Fig. 10(b), which is topologically equivalent to the two-dimensional hexagonal array proposed in [16], [15]. Similarly to what happens in the hexagonal array, the output data (of the matrix C) are also pumped from the both sides.

4) Systolizing an SFG for LU Decomposition: In LU decomposition, a given matrix C is decomposed into

$$C = A \times B$$

where A is a lower- and B an upper-triangular matrix. The recursions involved are

$$C_{ij}^{(k)} = C_{ij}^{(k-1)} - a_i^{(k)} * b_j^{(k)} \tag{3}$$

where

$$a_i^{(k)} = \frac{1}{c_{kk}^{(k)}} C_{i,k}^{(k-1)}$$

$$b_j^{(k)} = C_{k,j}^{k-1}$$

for $k = 1, 2, \cdots, N$; $k \leqslant i \leqslant N$
$k \leqslant j \leqslant N$.

The SFG representation for the above iterations is shown in Fig. 11. Note that it bears a great similarity to the SFG for multiplication of two banded matrices. Therefore, by almost the same systolization procedure as shown in Fig. 10, a systolic array for LU decomposition can be obtained. The resultant configuration of the array (not shown here) is very similar to Fig. 10(b).

C. Linear-Rate Pipelinability

Although the above systolization procedure is essentially complete, it is useful to find out how well the operations of an algorithm can be pipelined through the array (i.e. the pipelinability). The answer is rather straightforward:

We assert that the array has a linear-rate pipelinability *if and only if* α remains constant with respect to M, where M is the number of processor elements (PEs). It is clear that if the total time-scaling factor is α (i.e., $D = \alpha D'$), then the data input rate is slowed down by α. Consequently, in average only one of α PEs can be active. This implies that the full processing rate speedup M reduces to $\alpha^{-1}M$. If α remains constant with respect to M, then the array is pipelinable with a linear-rate speedup.

For most practical computational models the scaling factor α is 1, 2, or 3. For example: in the ARMA and lattice (type-A) systolic arrays, $\alpha = 2$, and in the lattice (type-B) array, $\alpha = 3$, regardless of how large M is. The same is true for most two-dimensional graphs, e.g., the SFGs for matrix multiplications, and LU decomposition, etc. However, there are examples of SFGs that, when localized, lead to nonconstant α. The arrays then cannot exhibit $O(M)$ execution-time speedups.

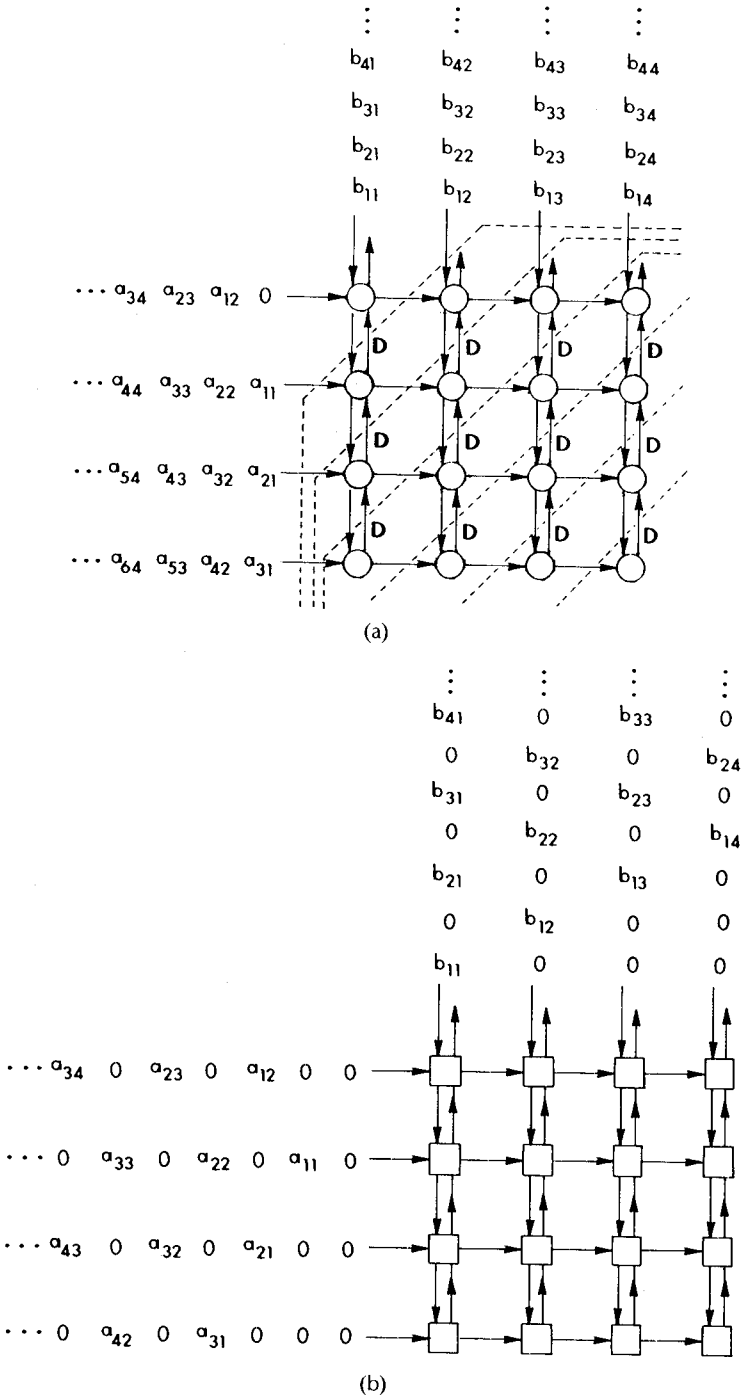


Fig. 9. (a) An SFG for matrix multiplication with one banded matrix. (b) Systolic array for matrix multiplication with one banded matrix.

1) An Example of Nonpipelinable SFGs: As an example of a regular but nonpipelinable SFG, let us look at the SFG shown in Fig. 12(a), which is originally due to Dewilde [8], [14]. Note that there exist no simple, uniform cuts with which to proceed the conversion. In fact, applying the localization rules with nonuniform cut-sets as shown in Fig. 12(a) (where the numbers in parentheses indicate the order of the cut-sets to apply the localization rules) will lead to a temporally localized array as depicted in Fig. 12(b). Note that the final time-scaling factor α turns out to be linearly proportional to M. This means that the speed performance of the "parallel" array processor is basically no different from a sequential computer, because no efficient pipelining is possible. Therefore, the array is said to be nonsystolic.

D. Improving Processing Speed and Utilization Efficiency

1) Multirate Systolic Array—Improving Processing Speed [20]: Note that due to the recaling of time units, the input data $\{x_i\}$ have to be interleaved with "blank" data (see Figs. 4, 6, 7), and the throughput rate becomes $(\alpha T)^{-1}$. This

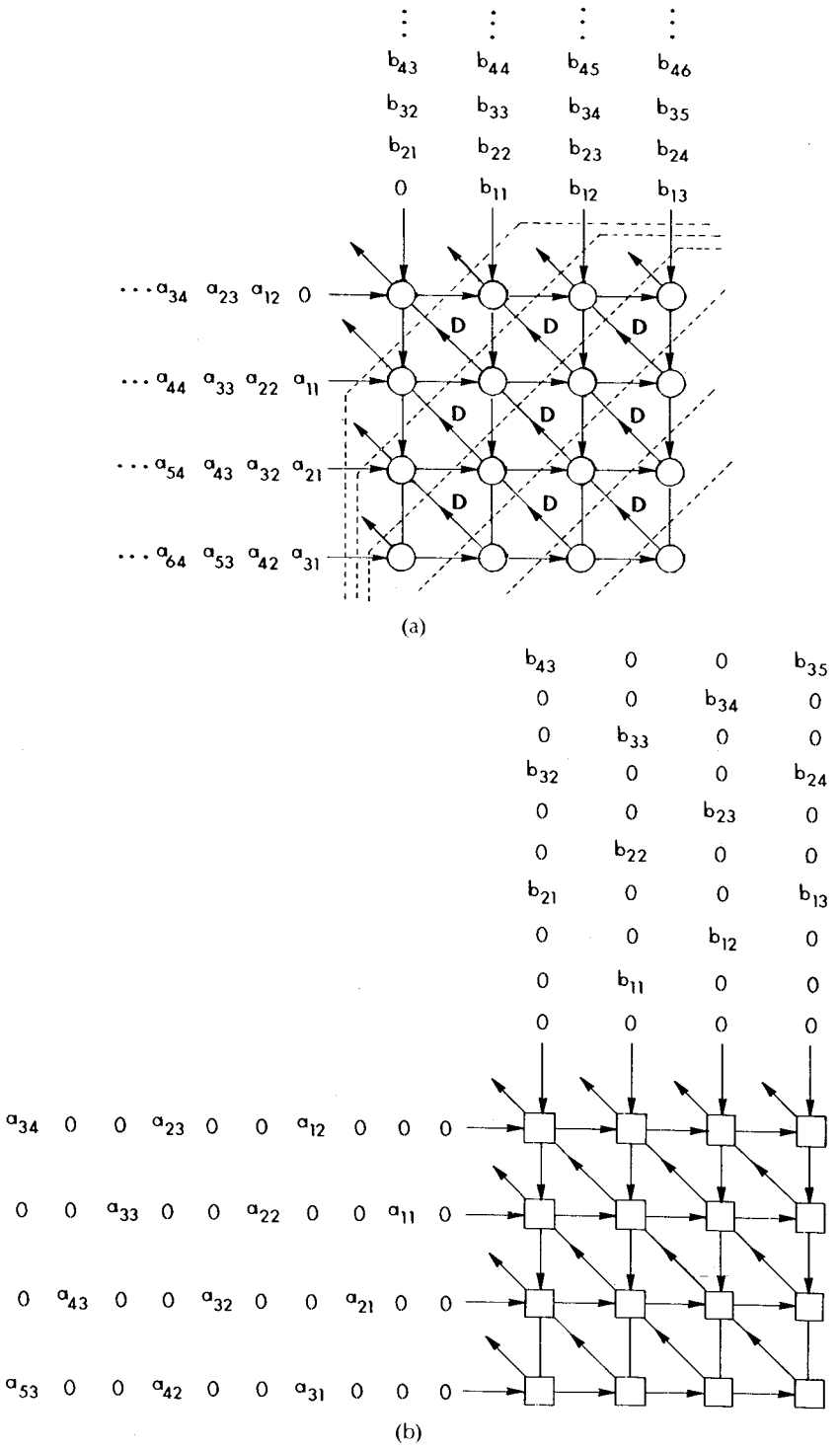
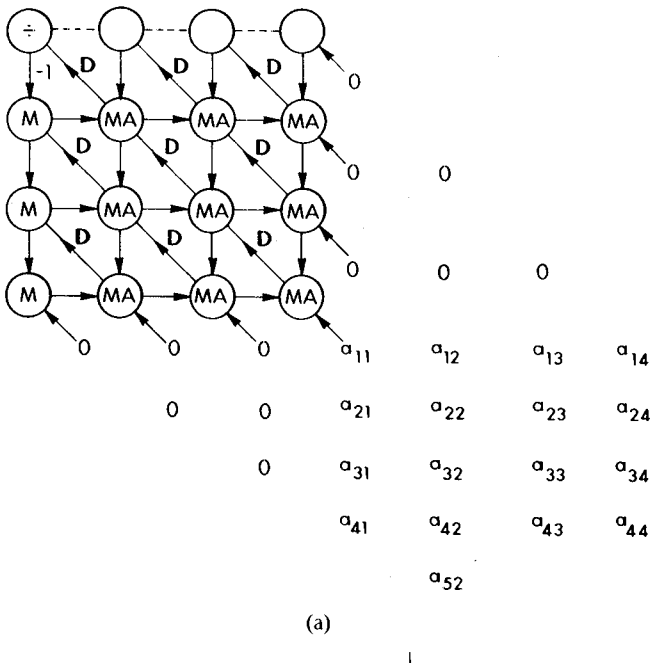


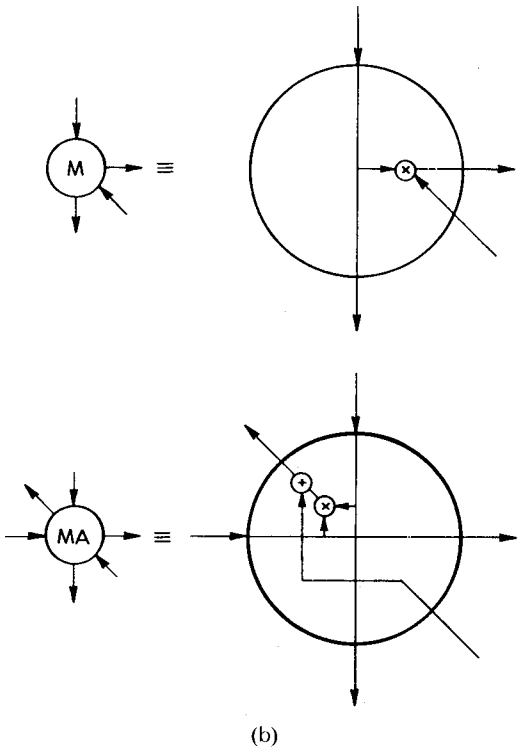
Fig. 10. (a) An SFG for multiplication of two banded matrices. (b) Systolic array for multiplication of two banded matrices.

rate is slower than that of the direct form design ($1.0T^{-1}$), because the data-transfer operation ($X \rightarrow X'$) alone consumes the same time as a multiply-and-add operation, an unnecessary delay. There are two solutions to this problem: one is to use a multirate systolic array and the other is to use a wavefront array based on asynchronous data-driven computing.

A multirate systolic array is a generalized systolic array, allowing different operations to consume different time units. As we have mentioned earlier, the finer the granularity in defining the basic module, the better the efficiency. For maximal efficiency, the granularity has to go all the way down to the bit level for a data-transfer operation, while the arithmetic operation may remain at the word level. For the ARMA filter design example, we can assign Δ as the time unit for a data transfer and T for a multiply-and-add. Consequently, in the circuit representation in Fig. 4(a), we replace D' on the feedforward path (for X) by Δ, and D' on



(a)



(b)

Fig. 11. (a) An SFG for LU decomposition. (b) The detailed diagram of the processing nodes.

the feedback paths (for Y and Z) by Δ. This means that the X data are pumped to the right with a delay of Δ, while the data Y and Z are transferred (to the left) with a (much longer) delay T. These modifications lead to a multirate systolic array as shown in Fig. 13. Since the original basic delay interval (D) is now replaced by $\Delta + T$; therefore, the input/output sequences $\{x_0, x_1, x_2, \cdots,\}$ and $\{y_0, y_1, y_2, \cdots,\}$ have to be pumped in and out by an interval $(T + \Delta)$, and attain a throughput rate of $1/(T + \Delta)$. In fact, a multirate systolic array is equivalent to a synchronized version of the wavefront array discussed in the next section.

2) Sharing Operation Modules—Improving Utilization Efficiency: It is possible to improve the processor utilization rate by as much as α times, where α is the time-scaling factor used in the localization process. The scheme is straightforward, noting that the interval between data will have to be α units apart, and therefore only one of α consecutive processor modules will be active at any instant. Therefore, a group of α consecutive PEs can *share a common arithmetic unit* without compromizing the throughput rates. Now let us use an example for a better illustration. Note that, according to the snapshots for the lattice systolic array (B) as depicted in Fig. 14, *only one upper MA module is active* in every three PEs at any time instant, and the same is true for the lower MA module. Therefore, as shown in Fig. 14, the three PEs can be combined into a (macro)PE and share the two common MA modules (one upper and one lower). A special-purpose ring register (with *period* = α) can be designed to handle the resource scheduling.

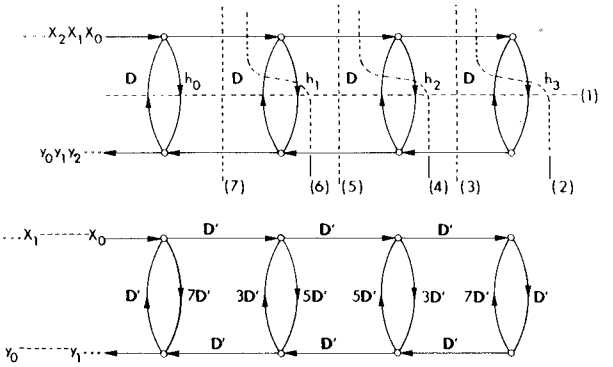


Fig. 12. (a) A nonsystolizable SFG example. (b) Localized but nonpipelinable SFG.

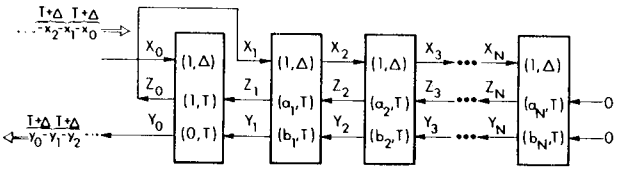


Fig. 13. A multirate systolic array for ARMA (IIR) filter.

V. Data-Flow Principle and Wavefront Array [17]

One problem associated with the systolic is that the data movements are controlled by global timing-reference "beats." In order to synchronize the activities in a systolic array, extra delays are often used to ensure correct timing. However, the price of this is an unnecessary slowdown in throughput rates. More critically, the burden of having to synchronize the entire computing network will eventually become intolerable for very- (or ultra-) large-scale arrays. The solution to the problem is to substitute the need for correct "timing" by correct "sequencing," as is used in data-flow computers and wavefront arrays. This leads to a completely different way of tackling the question "b) **when** to send data?" as posed in Section I-B. This time the answer lies in a data-driven, self-timed approach.

A. Data-Flow Multiprocessor

A data-flow multiprocessor [7] is an asynchronous, data-driven multiprocessor which runs programs expressed in

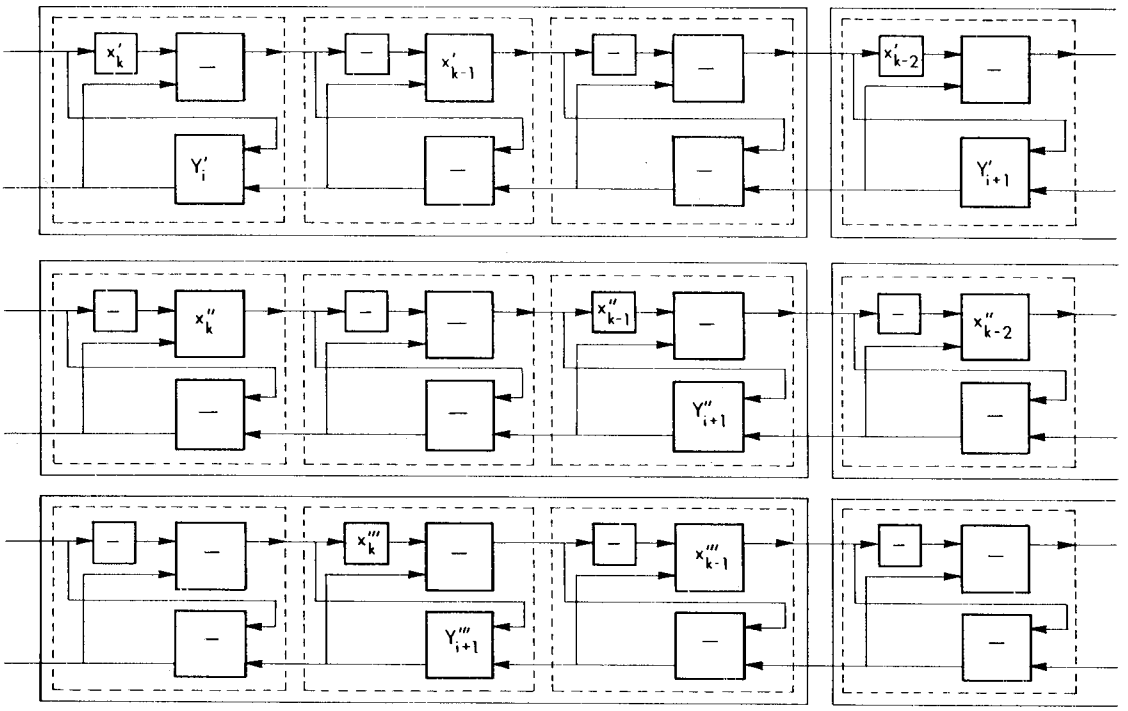


Fig. 14. Snapshots and time sharing of three PEs within a (macro)PE. (a) Snapshot at $t = i$. (b) Snapshot at $t = i + 1$. (c) Snapshot at $t = i + 2$.

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 over conventional multiprocessors are simple representation of concurrent activity, relative independence of individual PEs, greater use of pipelining, and reduced use of centralized control and global memory. However, for a general-purpose data-flow multiprocessor, the interconnection and memory conflict problems remain very critical. Such problems can be eliminated if the concepts of *modularity* and *locality* are imposed onto data-flow multiprocessors. This idea is the key motivation leading to concept of Wavefront Arrays.

B. Wavefront Arrays

Definition: Wavefront Array: A Wavefront Array is a computing network possessing the following features:

1) *Self-Timed, Data-Driven Computation:* No global clock is needed, since network is self-timed.

2) *Modularity and Local Interconnection:* Basically the same as in a systolic array. However, the wavefront array can be extended indefinitely without having to deal with the global synchronization problem.

3) *O(M) Speedup and Pipelinability:* (Similar to the systolic array.)

Note that the major difference distinguishing a wavefront array from a systolic array is the data-driven property. Consequently, the temporal locality condition (see 3 in the definition of Systolic Array) is no longer needed, since there is no explicit timing reference in the wavefront arrays. By relaxing the strict timing requirement, there are many advantages gained, such as speed and programming simplicity.

C. Incorporating Data-Flow Computing into Computing Networks

Our main goal here is to demonstrate that *all SFG computing networks can be converted into data-driven computing models.* Therefore, by properly incorporating the data-flow feature, every regular and modular SFG 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 delay operators D will be replaced by self-timed delays, i.e., handshaked "separator" registers.[9] In other words, the conversion of an SFG into a data-driven system involves substituting the delay D with implicit or explicit separators, and replacing the global clock by data handshaking. This process incorporates the data-flow principle into SFG's or systolic arrays.

Theorem: (Equivalence Transformation between SFG's and DFG's)

The computation of any SFG can be equivalently executed by a self-timed, data-driven machine with a data-flow graph (DFG) identical to the SFG, apart from substituting every time-delay operator D (controlled by a global clock) in the SFG with a *separator* ($\diamond$) that is locally controlled by handshaking.

Proof: What needs to be verified is that the global timing in the SFG can be (comfortably) replaced by the corresponding sequencing of the data tokens in the DFG.

[9]A handshaked separator is a device, symbolized by a diamond $\diamond$, which prevents any incoming data from directly passing through *until* the handshaking flag signals a "pass."

Note that the transfer of the data tokens is now "timed" by the processing node. This ensures that the relative "time" between data tokens received at the node is the same as it was in the SFG, as far as that individual node is concerned. By mathematical induction, this can be extended to show the correctness of the sequencing in the entire network.

For convenience, we shall term this trivial transformation the *SFG/DFG Equivalence Transformation.*

Example: Linear Phase Filter Design. As an example, let us now apply the rules to the SFG for a very popular linear phase filter, as shown in Fig. 15(a).[10] By the SFG/DFG

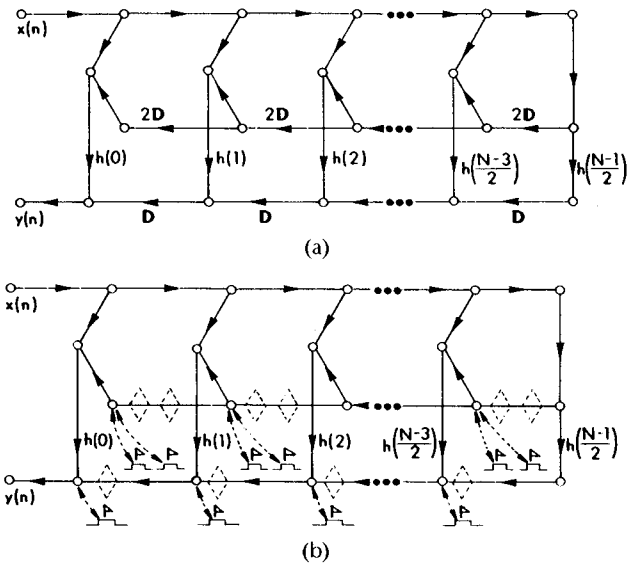


Fig. 15. (a) SFG for linear-phase filters. (b) Data-driven model (delta-flow graph) for the linear-phase filter.

equivalence transformation, the data-flow graph is derived as in Fig. 15(b). Now note that there are two separators inserted into every middle-level edge with handshaking (symbolized by "flags") to the branching node immediately succeeding them. In other words, the transfers occurring in the two separators are synchronized. In order to ensure the correct sequencing of data, the *W* data should propagate twice as slowly as the *Y* data do. Note that the separators play the role of ensuring such a correct sequencing.

Initial States: The general principle is that, all the separators are assigned initial values (regarded as data token), which are the same as those assigned to the delay registers in the corresponding SFG. We note that the initial state assignment under the SFG/DFG transformation is straightforward. This simplicity compares very favorably with the initial state reassignments in the systolization rule, where (because of the retiming involved in the systolization procedure) such reassignments may sometimes become rather complicated.

[10]Linear phase filters are very often used in one- and two-dimensional convolutions. They have two key features: one, that they have a symmetrical impulse response function, i.e., $h(n) = h(N-1-n)$, and two, they do not add phase distortion to the signal. Fig. 15(a) shows an SFG which takes advantage of the symmetry property, and reduces the amount of multiplier hardware by one half.

For a detailed illustration on the relationship between the initial states and the correctness of sequencing of data transfers in DFGs, let us again look into the linear phase filter example. Note that one initial zero is assigned to the separator of each *Y*-data edge; and two initial zeros are assigned to the two separators in each *W*-data edge. The "0" of the *Y*-data separator, when requested by the *Y*-summing node, will be passed to meet the *V* data arriving from the upper node. When the operation is done, a "data-used" flag will then be sent to the separator, clearing the way for sending the next *Y* data from the right-hand PE. The situation is similar for the *W* summing node, but only one "0" is "used" and the *W* data are still one separator away from meeting the "*X*" data in the summing node. It will have to wait until the *Y* data and the second "0" meet in the lower summing node. This explains why the propagation of *W* is slower than *Y*. (This is just what is needed to ensure a correct sequencing of data transfers.) Note also that there will be handshaking circuits needed for the nodes in the upper branches. Since there are no associated delays, there will be no initial data-tokens for the nodes.

1) Converting SFG's into Wavefront Arrays: The SFG/DFG equivalence transformation helps establish a theoretical footing for the wavefront array as well as provide more insights towards the programming techniques. The transformation implies that all regular SFGs can be easily converted into wavefront arrays, making modularly designed wavefront processing elements very attractive to use. Furthermore, because there is no concept of (global) time in a self-timed system, temporal locality is no longer an issue of concern. Therefore, the procedure of converting an SFG into a wavefront array is simpler than that of systolizing an SFG.

Another important feature is that data-flow graphs often provide useful clues for programming the data-driven wavefront arrays. An exemplificative program for the wavefront processing for linear phase filters, written in MDFL—Matrix Data Flow Language, will be discussed later.

VI. Wavefront Arrays and Computational Wavefronts

For further illustration, let us apply the SFG/DFG equivalence transformation to several one- and two-dimensional computing networks, e.g., ARMA and lattice filters, matrix multiplication, LU decomposition, etc.

A. One-Dimensional Wavefront Arrays

1) Wavefront Array for ARMA (IIR) Filter: Following the conversion strategy, an asynchronous wavefront model is derived as shown in Fig. 16. Therefore, at each node in Fig. 16, the operation is executed when and only when the required operands (data tokens) are available. An immediate advantage of this model is that a data transfer operation ($X \rightarrow X'$) uses only negligible time, Δ, compared with the time needed for an arithmetic operation. More precisely, the throughput rate achieved by the wavefront array is approximately $1.0(T + \Delta)^{-1}$, i.e., almost twice that of the pure systolic array in Fig. 4(b).

It is important to note that the pipelining in a wavefront array is different from the traditional idea of pipelining. Under the wavefront notion, $X^{(k)}$ is initiated at the leading PE ($n = 0$), and then propagated rightward across the processor array, activating the MA operations in all of the data-driven PEs. The updated data $\{Y_{n+1}^{(k)}, Z_{n+1}^{(k)}\}$ in the sum-

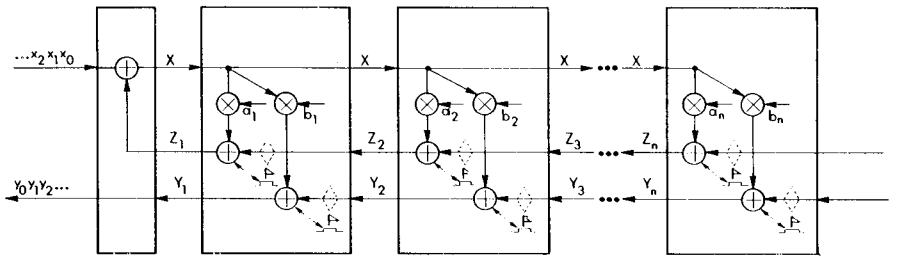


Fig. 16. Wavefront array for ARMA (IIR) filter. (Asynchronous, data-driven model, i.e., operations take place only on availability of appropriate data.)

ming nodes are fed back leftward, ready for the next wavefront. In this case, a reflection of the wave plays an interesting role. For convenience, we shall call such reflection a "ripple" wave. As illustrated in Fig. 17, a ripple from the kth wavefront in the $(n+1)$th PE will be needed (and expected) by the $(k+1)$th wavefront in the nth PE.

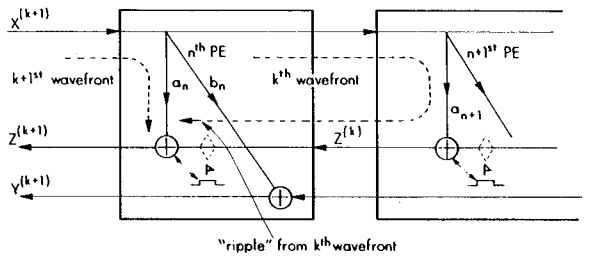


Fig. 17. A "ripple" wave from $(n+1)$th PE to nth PE.

2) Wavefront Array for Lattice Filter: In general, the data-driven model is not only faster (with maximized pipelining) but also has a simpler (self-timed) design, since a global timing reference is no longer required. For example, the (data-driven) wavefront model for the AR lattice as shown in Fig. 18 (compatible with the sytolic array Type-B) is considerably simpler than its systolic counterpart. Due to the data-driven nature of a wavefront array, it is guaranteed that the operation on the X data will have to wait until the Y data operation is done and the result transferred to the upper MA module. Therefore, the appropriate delays naturally fall into place, yielding the correct sequencing of the data. In contrast, in the (Type-B) systolic structure, the two kinds of signals, X (right-bound) and Y (left-bound), are propagating at different speeds, as shown in the snapshots in Fig. 14. Therefore, the (pure) systolic version is more complex than the wavefront solution, due to the timing of the complicated "ripple" effect.

B. Pipelining of Two-Dimensional Computational Wavefronts

From an algorithmic analysis perspective, the notion of computational wavefronts offers a very simple way to appreciate the wavefront computing. The separators are the handshaking device ensuring that the computational wavefronts are orderly following, instead of overtaking, their previous fronts.[11] We shall illustrate the wavefront concept and the related architecture and language designs with the matrix multiplication example. The computational wavefront for the first recursion in matrix multiplication will now be examined.

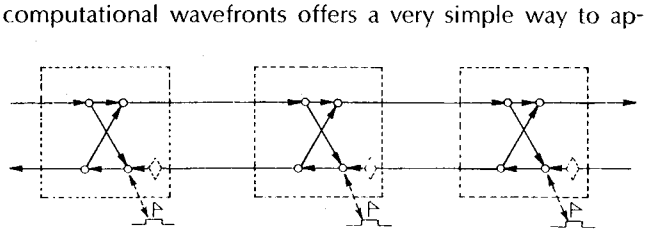


Fig. 18. Wavefront array for lattice filter.

The application of conversion rules to the (original) SFG is fairly straightforward. Basically, imposing handshaking upon all cut-sets will ensure correct sequencing. To see that this is true, a general configuration of computational wavefronts traveling down a processor array is illustrated in Fig. 19.

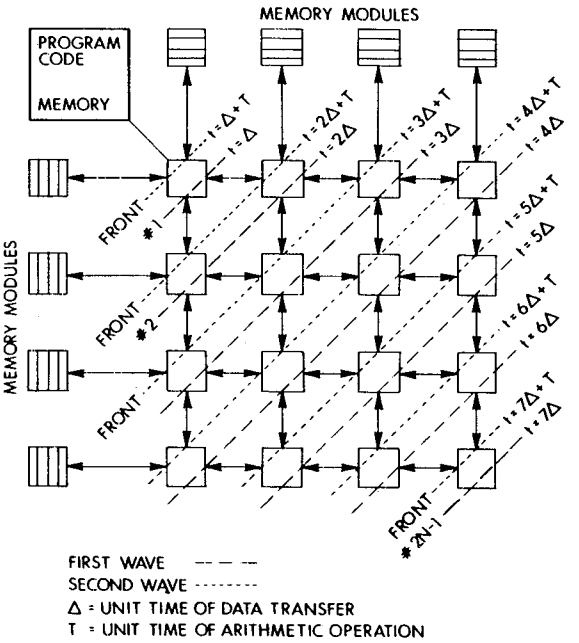


Fig. 19. Propagation of two-dimensional computational wavefronts

Suppose that the registers of all the processing elements (PEs) are initially set to zero

$$C_{ij}^{(0)} = 0, \qquad \text{for all } (i,j)$$

the entries of A are stored in the memory modules on the left (in columns), and those of B in the memory modules

[11] In fact, applying the data-flow concept along uniform cuts will lead to a self-timed, regular, and locally interconnected array—a wavefront array. As a matter of fact, the "wavefronts" will correspond to the cuts.

on the top (in rows). The process starts with PE (1, 1):

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

is computed. The computational activity then propagates to the neighboring PEs (1, 2) and (2, 1), which will executive in parallel

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

and

$$C_{21}^{(1)} = C_{21}^{(0)} + a_{21} * b_{11}.$$

The next front of activity will be at PEs (3, 1), (2, 2), and (1, 3), thus creating a computational wavefront traveling down the processor array. It may be noted that wave propagation implies localized data flow. Once the wavefront sweeps through all the cells, the first recursion is over (see Fig. 19).

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 (i, j) processor will execute

$$C_{ij}^{(2)} = c_{ij}^{(1)} + a_{i2} * b_{2j}$$

and so on.

1) Why the Name "Wavefront Array"?: The principle of wavefront processing is to successively pipeline the computational wavefronts as fast as resource and data availability allow, according to the concept of data-flow computing. As a justification for the name "wavefront array," we note that the computational wavefronts are similar to electromagnetic wavefronts (they both obey Huygens' principle) since each processor acts as a secondary source and is responsible for the propagation of the wavefront. In addition, wave-propagation implies localized data flow as well as localized control (handshaking). The pipelining is feasible because the wavefronts of two successive recursions will never intersect (Huygens' wavefront principle), thus avoiding any contention problems. (From the hardware perspective, the desired "separation" between two consecutive wavefronts is reaffirmed by the "separators" with proper handshaking.)

In other words, it is possible to have wavefront propagating in several different fashions. In the extreme case of nonuniform clocking, the wavefronts are actually crooked. However, what is important is that the order of task sequencing must be correctly followed. The correctness of the sequencing of the tasks is ensured by the wavefront principle [17].

2) Wavefront Array for LU Decomposition: By tracing backwards through the iterations in (3), we note that

$$C = c_{ij}^{(0)} = \sum_{k=1}^{N} a_i^{(k)} b_j^{(k)} = AB \qquad (4)$$

where $A = \{a_{mn}\} = \{a_m^{(n)}\}$, and $B = \{b_{mn}\} = \{b_n^{(m)}\}$ are the outputs of the array.

In comparison with (2) and (4), (3) is basically a reversal of the matrix multiplication recursions. Therefore, its wavefront processing should be similar to what is shown in Fig. 19. In fact, by converting the SFG as shown in Fig. 11 into a DFG, such a wavefront array can be directly obtained.

3) Least Square Error Solution and SVD: In many applications, we will be faced with solving a least square solution of an overdetermined linear system, as opposed to an exact solution of a nonsingular linear system. In this case, QR decomposition will prove to be much more useful than LU decomposition. The natural topology associated with QR decomposition is a square interconnect pattern. This can be shown by looking into the mathematical iterations and the corresponding SFGs. Similar systolic and wavefront arrays can be obtained by carrying out the systolization procedure or SFG/DFG equivalence transform. The details (omitted here) will be published in a later report.

The eigenvalue and SVD (singular value decomposition) problems are considerably more complicated. However, in [11], [21], it is shown that the notion of computational wavefronts can be employed to track down the activities in a square or linear array for computing eigenvalues or singular values.

VII. Wavefront Array Software/Hardware (WASH)

A. Programming Array Processors

The actual implementation of systolic or wavefront arrays can be either dedicated or programmable processors. Programmable arrays are preferred, due to the high cost of hardware implementation and the increasing varieties of application demands. Therefore, it is equally important to develop a complete set of software packages for most wavefront/systolic-type processing. For that, a formal algorithmic notation and programming language will be indispensable.

General guidelines for algorithmic notations for array processors are problem orientation, executability, and semantic simplicity. More importantly, an adequate language criteria must take into account the characteristics and the constraints of the arrays. Examples for appropriate array processing notations, which incorporate the language criteria for systolic/wavefront array processors, are CRYSTAL, [4] data space notation, [5] and the wavefront language (MDFL) [17].

B. Wavefront Language and Software Development

The effectiveness of programming in a processor array is directly related to the algorithm analysis technique. Our description of parallel algorithms hinges upon the notion of a computational wavefront. This leads to a special-purpose, wavefront-oriented language, termed Matrix Data Flow Language (MDFL) [17]. This denotation is in many ways very similar to the data space notation, which is based on the notion of applicative state transition systems described in [1], [5]. Among other commonalities shared by the two notations is, in particular, that they are both based on the data-flow principle [6].

The wavefront language is tailored towards the description of computational wavefronts and the corresponding data flow for the class of algorithms which exhibit the recursion and locality properties. Rather than requiring a program for each processor in the array, MDFL allows the programmer to address an entire front of processors. The wavefront idea can facilitate the description of parallel and pipelined algorithms and drastically reduce the complexity of parallel programming. To translate the global MDFL notation into microinstructions for the PEs, a preprocessor

is needed. For a wavefront array, the design of such a preprocessor is relatively easy, since we do not have to consider the timing problems associated with a synchronous systolic array.

As an example, let us now take a look at the computation of

$$C = A \times B.$$

The matrix multiplication can be carried out in N (outer product) recursions (see (1)). A general configuration of computational wavefronts traveling down a processor array is illustrated in Fig. 19. An example of an MDFL program for the corresponding array processing of the matrix multiplication is given in Fig. 20. (For the time being, please ignore the bracketed instructions.)

```
BEGIN
SET COUNT N;
REPEAT;
  WHILE WAVEFRONT IN ARRAY DO
    BEGIN
      {[FETCH C, DOWN;]}
        FETCH A, LEFT;
        FETCH B, UP;
        FLOW A, RIGHT;
        FLOW B, DOWN;
            (* NOW FORM C:= C + A × B)
        MULT A, B, D;
        ADD C, D, C;
      {[FLOW C, UP;]}
    END;
  DECREMENT COUNT;
UNTIL TERMINATED;
ENDPROGRAM.
```

Fig. 20. An MDFL program for matrix multiplication.

Note that, initially matrix A is stored (row by row) in the left Memory Module (MM). Matrix B is in the top MM and is stored column by column. The final result will be in the C registers of the PEs. This example illustrates the typical simplicity of the MDFL programming language.

1) Flexibilities with the Wavefront Programming: To demonstrate the flexibility of wavefront-type programmability, let us look at the multiplication of a banded-matrix A, $N \times N$, with bandwidth P, and a rectangular matrix B, $N \times Q$. Only a slight modification to the program in Fig. 20 is needed. First, the data storage in the memory modules will be the same as in Fig. 16(a). The major modification on the wavefront propagation 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.) Therefore, the program (see Fig. 20) remains almost the same, except for the two added bracketed instructions to shift partial sums upward.

Another major flexibility offered by the wavefront programming technique is that software reconfigurability can be used to map a linear or bilinear array onto a square array hardware. Therefore, a (hardwired) square array may be used for purpose of linear (or bilinear) wavefront array processing.

2) An Example on Linear Phase Filtering: In order to demonstrate the simplicity of programming based on the DFG representation, an MDFL program implementing the DFG for the linear phase filter (cf. Fig. 15(b)) is shown in

```
BEGIN
REPEAT;
  WHILE WAVEFRONT IN ARRAY DO
    BEGIN
        FETCH X, LEFT;
        FLOW X, RIGHT;
        TRANSFER W2 TO W1;
        FLOW W1, LEFT;
        FETCH W2, RIGHT;
        (! NOW COMPUTE V:= (W1 + X) + × H(K))
        ADD W1, X, U;
        MULT U, H(K), V;
        FETCH Y, RIGHT;
        (! NOW COMPUTE Y:= Y + V)
        ADD Y, V, Y;
        FLOW Y, LEFT;
    END;
  DECREMENT COUNT;
UNTIL TERMINATED;
ENDPROGRAM.
```

Fig. 21. An MDFL program for linear-phase filter.

Fig. 21.[12] The simple mapping between the DFG and the programming codes suggests a potentially significant impact of the SFG/DFG equivalence transformation to both the hardware and software developments of array processors.

The power and flexibility of the wavefront array and MDFL programming are best demonstrated by the broad range of the applicational algorithms suitable for the wavefront array [17]. Such algorithms can be roughly classified into three groups:

1) Basic Matrix Operations: such as a) Matrix Multiplication, b) Banded-Matrix Multiplication, c) Matrix–Vector Multiplication, d) LU Decomposition, e) LU Decomposition with Localized Pivoting, f) Givens Algorithm, g) Back Substitution, h) Null Space Solution, i) Matrix Inversion, j) Eigenvalue Decomposition, and k) Singular Value Decomposition.

2) Special Signal Processing Algorithms: a) Toeplitz System Solver, b) One- and Two-Dimensional Linear Convolution, c) Circular Convolution, d) ARMA and AR Recursive Filtering, e) Linear Phase Filtering, f) Lattice Filtering, g) DFT, and h) Two-Dimensional Correlation (image matching).

3) Other Algorithms: PDE (partial difference equation) solution.

Note that, if the communication constraint is relaxed, our technique for converting an SFG for a given application into a wavefront array will also work with other global-type algorithms, such as the FFT algorithm, the Householder transformation, the Kalman filter network, or other non-regularly interconnected arrays. The only additional requirement lies in routing the physical connections between PEs.

It is worth noting that the cut-set rules can be potentially very useful for designing fault-tolerant arrays. For systolic arrays without feedback, it has been shown in [32], [33] that a retiming along cut-sets allows a great degree of fault tolerance. The discussion in Section III-B should offer a theoretical basis for improving fault-tolerance of arrays with feedback via the cut-set retiming procedure. More interest-

[12]Note that the separators in the DFG are implemented simply by adding three lines of (internal register–transfer) code to the program, as opposed to adding a separate buffer register external to the PE.

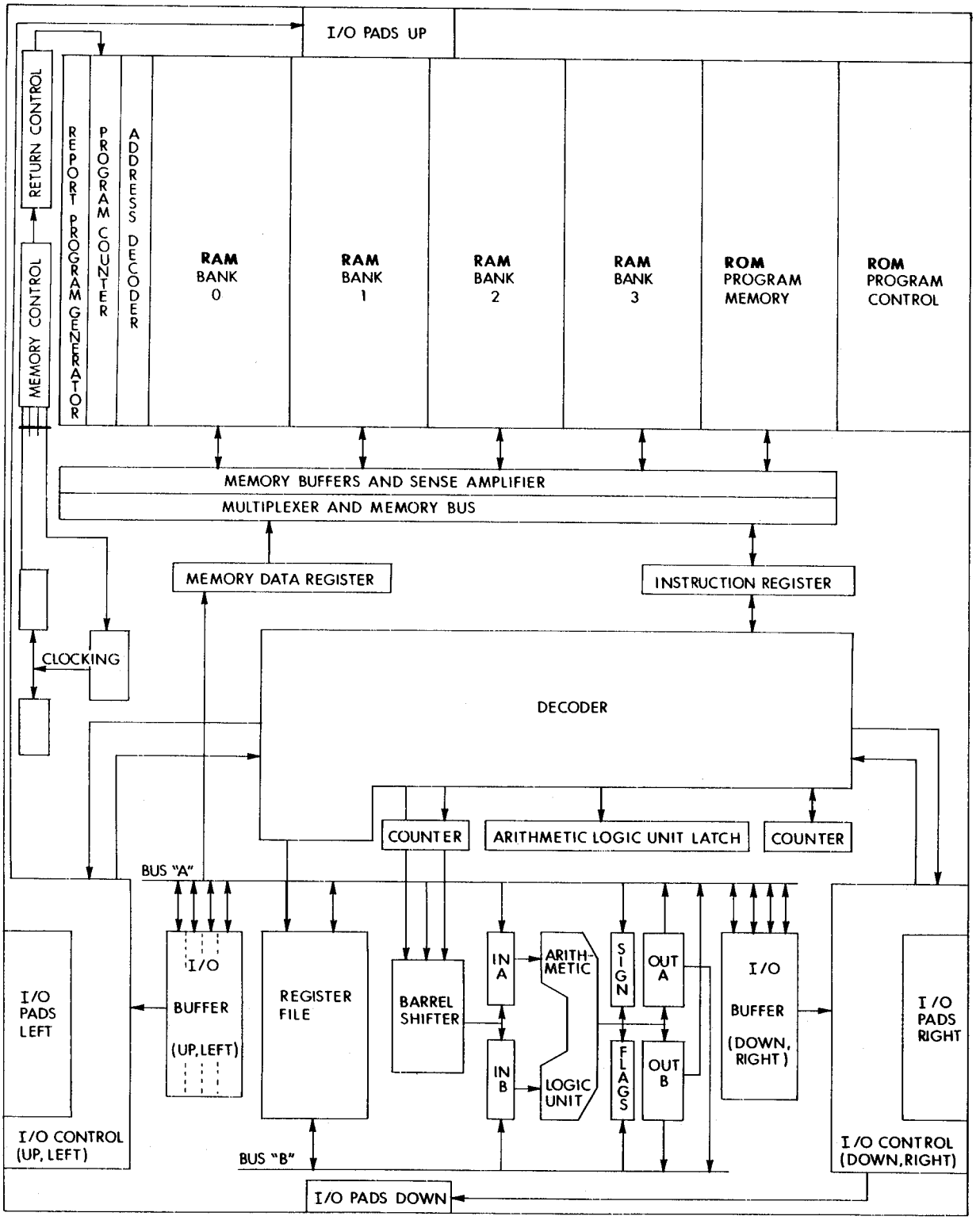


Fig. 22. Functional block diagram of wavefront PE.

ingly, with a slight modification, the self-timed feature of wavefront arrays offers a way of achieving the same fault-tolerance efficiency without any need of retiming.

In summary, in the first phase of the software development project, we 1) define the application/ algorithm domain, 2) develop a language tailored to the application, and 3) design a (language-based) wavefront architecture. In order to maximize the application algorithm domain (with minimal hardware overhead), the next phase is to develop a complete software library of all algorithms suitable for systolic/wavefront-type parallel processing. This software library, combined with design automation tools such as silicon compilers, will facilitate the construction of future VLSI systems design. The success of the project will demand joint and cohesive efforts from all related disciplines. For this end, we welcome any suggestions from interested readers and colleagues.

C. Hardware Design

In this subsection we will give an overview of the architecture of a PE, to be used as a basic module in a programmable array processor. A typical example will be the design of a PE to be used in a wavefront array. The basic wavefront array is either a square array of $N \times N$ PEs, a linear array of $1 \times N$ PEs, or a bilinear array of $2 \times N$ PEs. The PEs are orthogonally connected and are identical. The hardware of the PE is designed to support the features of the Matrix Data Flow Language (MDFL) introduced previously. Given the current state of the process technology, with a minimum feature size of 2 μm or less, we estimate the area of the chip taken by a PE to be 6×6 mm^2.

1) Architectural Outline: The PE that we have designed is a special-purpose microprocessor. The functional block diagram of the PE is shown in Fig. 22. The main functional blocks are datapath, program memory, I/O control units, and instruction decoder.

Our design objective is to limit the complexity of the datapath, preferring a regular and easy layout design. We have adopted a 32-bit-wide datapath for fixed-point computations. Moreover, the ALU in the PE is designed to support the operations that are of major importance for signal processing applications, such as multiplication and rotation. To speed up the throughput of the PE, we used a two-level pipelining scheme.

The PE can simultaneously perform data transfers in four directions. The transfer of data is controlled by an I/O controller, one for each of the four directions, which handles the two-way handshaking functions.

2) Instruction Set: The instruction set of the PE was selected to optimize the performance of the wavefront array as a whole. To reduce the complexity of the control unit, in a manner similar to that used in the RISC design [28], we wanted each instruction to take exactly one clock cycle. This implies that complex instructions should be decomposed into sequences of simpler (primitive) instructions. An example is the multiplication instruction, which is decomposed into three instructions: one for initializing the processor registers with the correct data, one that does the main multiplication step, and the last one which transfers the result back to the register file. The instruction set is divided into arithmetic instructions, register transfer instructions, conditional and unconditional jump instructions, and program loading instructions.

3) Design Specification and Verification: The PE described in this section is currently being specified and verified using the ISPS language. The ISPS language allows not only the specification of the design of a single PE, but also the simulation of an entire wavefront array. More importantly, it facilitates the verification of the correctness and suitability of the architecture of the PE before designing the lower (logic, circuit, layout) levels of the PE. It will also be an important tool for the development of the host interface, the memory units, etc.

VIII. Conclusions

The rapid advance in VLSI device technology and design techniques have encouraged the development of massively parallel-array processors. We have stressed the importance of modularity, communication, and system clocking in the design of VLSI arrays. For signal processing applications, a large number of algorithms possess the properties of recursiveness and locality. These properties naturally led to the systolic and wavefront arrays.

The two types of arrays share the important common feature of using a large number of modular and locally interconnected processors for massive pipelined and parallel processing. However, in several key aspects, the wavefront array is noticeably distinctive from the systolic array. First, it uses data-driven computing and thus gets around the burden of having to synchronize a (potentially ultra-) large-scale array. [18] Second, it maximizes the pipelining efficiency and offers a speed achievable only by multirate systolic arrays. Third, the data-flow principle allows a simpler language design facilitating a formal description of the activities. Finally, it can easily cope with the variations of communication delays in dynamically interconnected systems, such as reconfigurable waferscale integration designs.

In conclusion, we have shown that both the systolic and data-flow principles will play a major role in future supercomputing, especially for number crunching problems. Most computing networks described in signal-flow graphs (SFGs) can be systematically converted into systolic or wavefront arrays, following the procedures proposed. This should encourage more practitioners to develop advanced hardware and software for massively parallel-array processors. The impacts of the novel architectures upon future supercomputer designs cannot be overestimated.

Acknowledgment

The author wishes to thank his colleagues in the VLSI signal processing group at the University of Southern California, for their very valuable contributions to the Wavefront Array Software/Hardware (WASH) Project. He is also grateful to Dr. H. Lev-Ari of the Stanford University for many useful comments which were incorporated into this final draft.

References

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21 pp. 613–641, 1978.

[2] C. Caraiscos and B. Liu, "From digital filter flow-graphs to systolic arrays," submitted to *IEEE Trans. Acoust., Speech, Signal Processing*, 1984.

[3] H. J. Caulfield, W. T. Rhodes, M. J. Foster, and S. Horvitz, "Optical implementation of systolic array processing," *Opt. Commun.*, vol. 40, pp. 86–90, Dec. 1981.
[4] M. Chen and C. Mead, "Concurrent algorithms as space-time recursion equations," to appear in *VLSI and Modern Signal Processing*, S. Y. Kung *et al.*, Eds., Englewood Cliffs, NJ: Prentice Hall.
[5] A. B. Cremers and T. N. Hibbard, "The semantic definition of programming languages in terms of their data spaces," *Informatik-Fachberichte*, vol. 1, pp. 1–11 (Berlin, Springer-Verlag), 1976.
[6] A. B. Cremers and S. Y. Kung, "On programming VLSI concurrent array processors," in *Proc. IEEE Workshop on Languages for Automation* (Chicago, IL, 1983), pp. 205–210; also in *INTEGRATIONS* (The VLSI Journal), vol. 2, no. 1, Mar. 1984.
[7] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, vol. 13, pp. 48–56, Nov. 1980.
[8] P. Dewilde, personal communication, 1983.
[9] J. J. Dongarra *et al.*, *LINPACK USER'S GUIDE*. Philadelphia, PA: SIAM PUB., 1979.
[10] M. Franklin and D. Wann, "Asynchronous and clocked control structures for VLSI based interconnection networks," presented at the 9th Annual Symp. on Computer Architecture, Apr. 1982, Austin, TX.
[11] R. J. Gal-Ezer, "The wavefront array processor," Ph.D. dissertation, Dept. of Electrical Engineering, University of Southern California, Dec. 1982.
[12] J. W. Goodman, F. Leonberger, S. Y. Kung, and R. Athale, "Optical interconnections for VLSI systems," this issue, pp. 850–866; also prepared as an ARO Palantir Meeting Report.
[13] K. Hwang and F. Briggs, *Computer Architectures and Parallel Processing*. New York: McGraw-Hill, 1984.
[14] J. V. Jagadish, T. Kailath, G. G. Mathews, and J. A. Newkirk, "On pipelining systolic arrays," presented at the 17th Asilomar Conf. on Circuits, Systems, and Computers, Pacific Grove, CA, Nov. 1983, also "On hardware description from block diagrams," in *Proc. IEEE ICASSP* (San Diego, CA, Mar. 1984).
[15] H. T. Kung, "Why systolic architectures," *IEEE Computer*, vol. 15, no. 1, Jan. 1982.
[16] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Proc. Sparse Matrix Symp.* (SIAM), pp. 256–282, 1978.
[17] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront array processor: Language, architecture, and applications," *IEEE Trans. Comput.* (Special Issue on Parallel and Distributed Computers), vol. C-31, no. 11, pp. 1054–1066, Nov. 1982.
[18] S. Y. Kung and R. J. Gal-Ezer, "Synchronous vs. asynchronous computation in VLSI array processors," in *Proc. SPIE Conf.* (Arlington, VA), 1982.
[19] S. Y. Kung and J. Annevelink, "VLSI design for massively parallel signal processors," *Microprocessors and Microsystems* (Special Issue on Signal Processing Devices), vol. 7, no. 4, pp. 461–468, Dec. 1983.
[20] S. Y. Kung, "From transversal filter to VLSI wavefront array," in *Proc. Int. Conf. on VLSI 1983*, IFIP (Trondheim, Norway), 1983.
[21] S. Y. Kung and R. J. Gal-Ezer, "Eigenvalue, singular value and least square solvers via the wavefront array processor," in *Algorithmically Specialized Computer Organizations*, L. Snyder *et al.*, Eds. New York: Academic Press, 1983.
[22] C. E. Leiserson, "Area-efficient VLSI computation," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1981.
[23] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. Caltech VLSI Conf.* (Pasadena, CA), 1983.
[24] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," *J. VLSI Comput. Syst.*, vol. 1, no. 1, pp. 41–67, 1983.
[25] H. Lev-Ari, "Modular computing networks: A new methodology for analysis and design of parallel algorithms/architectures," Tech. Memo ISI-29, Integrated Systems Inc., Palo Alto, Ca., 1983.
[26] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[27] A. Oppenheim and R. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
[28] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, vol. 14, Sept. 1981.
[29] B. T. Smith *et al.*, *Matrix Eigensystem Routines, EISPACK Guide*, vol. 6 of *Lecture Notes in Computer Science*, 2nd ed. New York: Springer-Verlag, 1976.
[30] J. D. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Computer Science Press, 1984.
[31] A. Fettweis, "Realizability of digital filter networks," *AEU* (Electronics and Communication), vol. 30, pp. 90–96, 1976.
[32] H. T. Kung and M. S. Lam, "Fault-tolerant VLSI systolic arrays and two-level pipelining," in *Proc. SPIE Symp.*, vol. 431, pp. 143–158, Aug. 1983.
[33] ———, "Fault-tolerance and two-level pipelining in VLSI systolic arrays," in *Proc. Conf. on Advanced Research in VLSI* (MIT, Cambridge, MA, Jan. 1984), pp. 74–83.