Towards a Formal Treatment of VLSI Arrays

Lennart Johnsson
Computer Science
California Institute of Technology
Pasadena, CA 91125
and
Information Sciences Institute

Uri Weiser
Computer Science
University of Utah
Salt Lake City, UT 84112

Danny Cohen
Information Sciences Institute
Marina Del Rey, CA 90291
and
California Institute of Technology

Alan L. Davis
Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

This paper presents a formalism for describing the behavior of computational networks at the algorithmic level. It establishes a direct correspondence between the mathematical expressions defining a function and the computational networks which compute that function. By formally manipulating the symbolic expressions that define a function, it is possible to obtain different networks that compute the function. From this mathematical description of a network, one can directly determine certain important characteristics of computational networks, such as computational rate, performance and communication requirements. The use of this formalism for design and verification is demonstrated on computational networks for Finite Impulse Response (FIR) filters, matrix operations, and the Discrete Fourier Transform (DFT).

The progression of computations can often be modeled by wave fronts in an illuminating way. The formalism supports this model. A computational network can be viewed in an abstract form that can be represented as a graph. The duality between the graph representation and the mathematical expressions is briefly introduced.

Introduction

This paper addresses the problem of formally describing the behavior of computational networks at the algorithmic level. The focus is on the correspondence between equations defining a certain computation and networks performing that computation. In an equation there may be no concept of time. An equation expresses how a variable is related to a set of constants, other variables, and possibly itself. Time is, however, intrinsically associated with any computation performed by a physical device (Mead and Conway [10]).

VLSI technology promises to offer substantial computational power. With submicron technology, on the order of a million to ten million transistors can be placed on a single chip. The complexity of designing such a chip is orders of magnitude greater than that typical today. The need for proper abstractions at all levels of design is apparent. These abstractions have to be consistent so that a higher level description can be expanded in a hierarchical manner to levels where a direct mapping to silicon will generate circuitry that performs the function of a high-level description. One hierarchical approach, Rowson [12], not only brings the complexity of VLSI circuit design within reach of humans but also enhances correctness. The approach is also a good basis for advanced design tools such as silicon compilers (Ayres [1] and Johannsen [5]). Several silicon compilers exist today. The first silicon compiler, based on a special class of floorplans, has been followed by less restricted compilers. Currently available compilers accept an input at the register-transfer level.

Chen and Mead [2] have proposed a notation for designing concurrent systems. Their notation supports a hierarchical approach towards system design and enhances proof of liveness and safeness of concurrent systems. In mapping a behavioral description to silicon, it is also necessary to insure that the constructs used have a correspondence in circuits with an electrical behavior matching the description. Rem and Mead [11] have proposed a notation and composition rules that insure a correct correspondence between a syntactically correct behavioral description and the behavior of circuitry that can be generated from the description.

Early in the design of a computational network, a decision must be made about how the data required by the computation will enter the network. In a real-time signal-processing environment, a variable is typically observed (sampled) at discrete times, often at a constant frequency. In such applications it is natural to assume that the order of the data input to the network is the same as the order of the generation of the data. If the input data resides in a random access memory, then there are several ways for the data to be organized without requiring additional hardware or adversely affecting performance. The organization of the input data is, however, of prime importance for the size,

structure, and performance of the computational networks that can be designed to compute the desired function.

For simplicity, the derivations of the networks presented here are assumed to be synchronous. The results derived apply also to self-timed design (Seitz [13]). In the computation of most functions, not all computations can be performed concurrently. Sequence requirements implied by the algorithms used to compute a function have to be satisfied in order to obtain a correct result. It will be seen that the computational networks described here are composed of a collection of functional modules that take a certain collection of inputs and produce a collection of outputs. We define the notion of a time step to be the time it takes for a module of a network to compute its results from its inputs. In this manner, a time step can be viewed as the time quantum separating sequential sets of inputs to modules and outputs from modules.

The formalism we use to establish a correspondence between a mathematical expression and computational networks follows Cohen [3] by modeling a storage element (e.g., a flip-flop) with an operator that can be interpreted as a delay when acting on a series of conceptually sequenced data. A data sequence can be viewed such that successive elements of the sequence are separated by a single time step. In a self-timed system, the delay may vary in terms of absolute time, but the sequential order is always preserved. The operator is well defined mathematically and can be readily introduced in an expression to be evaluated by a network once the input data has been ordered in time. Hence, the mathematical expression can be transformed in a straightforward manner into a form that maps directly to a computational network. There are a number of mathematical expressions that are all functionally equivalent but whose direct hardware interpretation results in different networks. These equivalent forms can be obtained by formal manipulation of the equations. Correctness is assured since these transformations are function preserving.

Using this formalism it is possible to determine the essential properties of the networks directly from the equations defining them. Computational rate, performance, delay, modularity, and module count are all easily determined from the equations. The interconnection scheme and communication characteristics can also readily be found from the equations for networks with a high degree of regularity. We will show how this approach can be used to derive and characterize computational networks for Finite Impulse Response (FIR) filters, matrix-vector products, and the Discrete Fourier Transform (DFT). The details of these networks may be found in Cohen [3], Cohen and Tyree [4], Weiser and Davis [14] and Johnsson and Cohen [6] and [7]. Weiser and Davis [14] have also used this approach to treat networks for the multiplication of band matrices and the solution of triangular linear equations.

The mathematical approach pursued in this paper may also be used for verification. The modules used for the examples in this paper contain additions, multiplications, and delays. Their function can be described as a transfer function in the form of an operator that can be represented by a matrix. Since all of these networks are linear, the compositions of modules into arrays correspond in the functional domain to multiplication of matrices.

This mathematical approach is demonstrated in this paper only for one–dimensional arrays; however, this is not a limitation of the approach. See Weiser and Davis [14] for the application of this approach to two–dimensional arrays.

The progression of a computation can be modeled by the concept of a wave front. Wave fronts are an intuitively appealing way to illustrate how the computations proceed. Wave fronts can be defined either graphically, in terms of the networks, or mathematically, in terms of equations (see Johnsson and Cohen [6,7], and Weiser and Davis [14]). The use of wave fronts in the mathematical domain has the additional utility of simplifying the notational complexity. S. Y. Kung [9] has also used wave fronts in an informal way to describe the progression of computations in orthogonal arrays.

Notation

Space and time are fundamental characteristics of computational networks and their behavior. Computations may be distributed in time, in space, or both.

Let $X = \{x(k)\}$ be a sequence of variables. The index k is associated with time such that x(k) precedes x(k + 1) by one time step. We refer to such a sequence as a data stream.

Define the operator Z by $Zx(k) = x(k-1)$ and define $Z^j = ZZ^{j-1}$. Then $Z^j x(k) = x(k-j)$.

The elements of the sequence X may be observed in two fundamentally different ways. The first way is to view the elements over time as they pass by a particular point, which is fixed in space. The second way is to take a "snapshot" of the network, i.e., to view the elements of X as they are spread out in space at an instant of time. Such a snapshot is shown in Figure 1.

Hence, the operator Z may be considered as a delay in time when applied to a data stream at a certain position, or as a "shift over space" when considered for an instant of time.

Figure 1 shows the effect of Z^5 operating on a data stream.

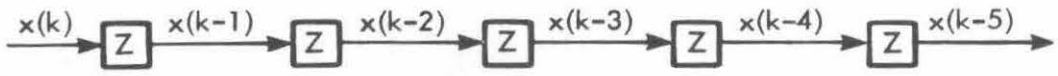

Figure 1: Z^5 operating on a data stream

From the definition of Z and Figure 1 it is natural to interpret Z as a delay when acting on a data stream.

Define Z^{-1}, the inverse of the Z operator, by $Z^{-1}x(k) = x(k+1)$.

Z^{-1} can be interpreted as a prediction when operating on data streams. It has the following properties

$$ZZ^{-1} = Z^{-1}Z = Z^0 = I, \text{ where I is the identity operator.}$$

The operator Z is commutative with respect to time–independent functions. However, when commuting the Z operator and the function, the Z operator must be distributed over the entire operand set of the function, e.g.,

$$ZF(X,Y) = F(ZX,ZY).$$

A graphical representation of this commutative–distributive property is shown in Figure 2.

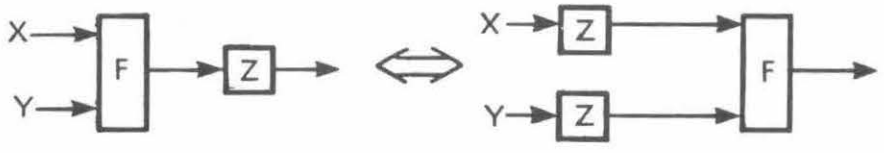

Figure 2: ZF(X,Y) = F(ZX,ZY)

Constants do not change over time and the Z operator does not affect constants. Therefore,

$$Z(CX) = (ZC)(ZX) = C(ZX).$$

Hence, as operators, Z and C commute: (ZC)X = (CZ)X .

Equations in which the Z operator is used to express sequencing can be given a direct interpretation in terms of computational networks. Different expressions correspond to different networks. In a few examples we will show not only the correspondence between expressions using the Z operator and computational networks but also how some properties of the computational network (such as modularity, computational rate, performance, and fault propagation) can be determined directly from the expressions.

Finite Impulse Response Filters

A Finite Impulse Response (FIR) filter can be defined as

$$y(k) = \sum_{i=0}^{N-1} a(i)x(k-i) \qquad (1)$$

where X is the input signal to the filter and a(i), i = 0,1,2,...,N–1 are the filter coefficients. The output of the filter is Y. The indices of X and Y are naturally associated with time in a real–time environment. The N multiplications required to compute each value of Y can be carried out in any order or concurrently, because equation (1) does not prescribe any order of evaluation. There exist several hardware realizations of equation (1). They may differ with respect to computational rate, performance, amount of hardware, reliability, etc.

For the FIR filter it is natural to assume that the data arrives sequentially and in order of increasing indices. Using the Z operator we will now discuss a few implementation alternatives.

Introducing the Z operator into equation (1) gives

$$y(k) = \sum_{i=0}^{N-1} a(i)Z^i x(k)$$

or

$$Y = \left(\sum_{i=0}^{N-1} a(i)Z^i \right) X \qquad (2)$$

A direct hardware interpretation of equation (2) would contain N(N–1)/2 delays, N multipliers, and N–1 adders. Having computed the products, which can be made concurrently, the N terms have to be added. If the N–1 additions are concurrent then the computational rate is limited by the carry propagation.

However, equation (2) can be rewritten as

$$y(k) = \Big(\big(\ldots((a(N-1)Z + a(N-2))Z + a(N-3))Z + \ldots\big)Z + a(0)\Big)\, x(k) \qquad (3)$$

From equation (3) it is obvious that N–1 delays suffice to implement the filter defined by equation (1). Equation (3) naturally corresponds to a linear array of modules, as indicated in Figure 3. The computational rate of an implementation corresponding to equation (3) is, however, still limited by N–1 concurrent additions.

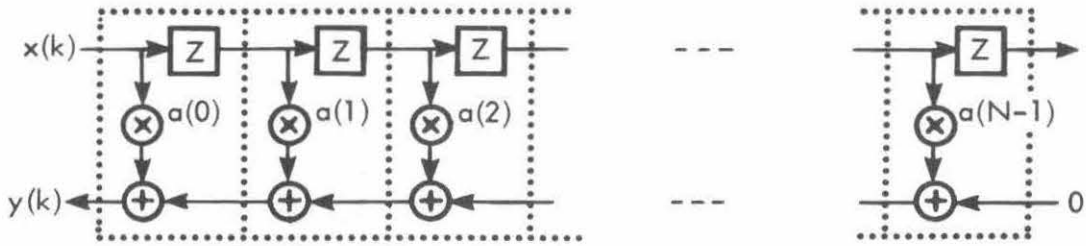

Figure 3: The implementation of the FIR filter

The coefficients a(i), i = 0,1,2,...,N–1, and the Z operator commute because the coefficients are constants. Note that the index k is associated with time and that the index i is associated with space ("module number"). Therefore, Z operates on the variable k, not on i. Using the property that a(i) and Z commute, equation (2) can be rewritten as

$$y(k) = (a(0) + Z(a(1) + Z(\ldots(a(N-2) + Z(a(N-1)))\ldots)))\, x(k)$$

or

$$Y = \Big(\sum_{i=0}^{N-1} Z^i a(i)\Big) X. \qquad (4)$$

Equation (4) corresponds to an implementation that has as many components as the implementation of equation (3), but the modules and the array have different properties. X is broadcasted to all modules. For large values of N, broadcasting ("fanout") is undesirable. Broadcasting implies long wires and large drivers; long wires are likely to be slow and may limit the computational rate. An implementation corresponding to equation (4) is not, however, limited by N–1 concurrent additions, as is the implementation of equation (3). The summation is in implementations corresponding to equation (4) performed in a pipelined fashion. Figure 4 shows modules and a linear array corresponding to equation (4).

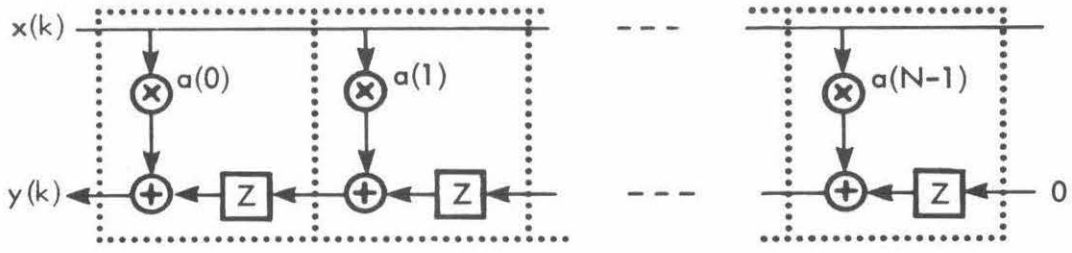

Figure 4: The implementation of the FIR filter

If the value of N is large enough for broadcasting to be a problem, then the computations of the FIR filter can be grouped so that broadcasting will not be a problem in each group. The groups are then connected via delays. The output of the FIR filter will be delayed a number of steps one less than the number of groups. The broadcasting may also be made in a tree–like manner. One possible solution to this fanout problem is shown in Figure 5.

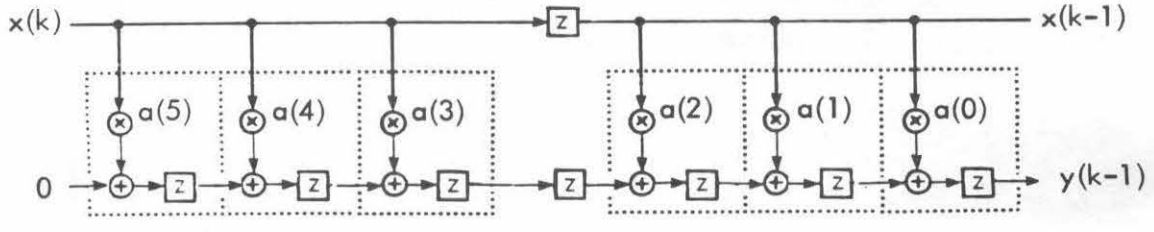

Figure 5: The implementation of the FIR filter

In the implementations discussed above, it has been assumed that N is small enough to allow for all modules required by the equations as given to be implemented. The input consisted of one data stream.

Matrix vector multiplication

The product Y of a matrix A by a vector X is defined by

$$y(m) = \sum_{i=1}^{N} a(m,i)x(i) \quad \text{for } m = 1,2, ..., M \tag{5}$$

Hence, each y(m) is the inner product of the mth row of A, {a(m,*)}, and the vector X.

The evaluation of each of these inner products is similar to the evaluation of the FIR filter shown above, with the differences that here there is a set of {a(i,*)} associated with the ith unit, not just one a(i) as before, and that the notation here starts at i = 1 whereas it starts at i = 0 in the FIR filter computation.

M*N multiplications are required for the evaluation of equation (5). In the following discussion they are distributed into M modules (distribution in space) each performing N multiplications (distribution in time).

Obviously, these M inner products are not independent because they share the same input vector, X.

We first introduce another implementation scheme for inner products; then we show several different ways to interconnect them in order to achieve the matrix–vector multiplication.

A straightforward use of the arrays discussed above for the FIR filter to compute a matrix–vector product would require one array for each component of the product, i.e., O(N*M) modules. Since this quantity may be prohibitively high, we use another approach that uses only M modules. This reduction in the number of modules is obviously reflected in the rate at which the output is computed, as seen below.

We follow approach B from Cohen and Tyree [4]. We pursue an implementation that is organized as M modules, each corresponding to a certain y(m). The matrix coefficients are given one column at a time such that each row, {a(m,*)}, is a data stream of coefficients given to the unit corresponding to that y(m). The vector X is also given as such a data stream to all the units.

With this organization, the operation of the operator Z on the data is

$$Zx(k) = x(k-1) \quad \text{and} \quad Za(m,k) = a(m,k-1) \tag{6}$$

Note that the above is a property of the data organization and not of the Z operator.

Define the partial sums involved in the computation of the products:

$$Y(m,k) = \sum_{i=1}^{k} a(m,i)x(i) \quad \text{for} \quad k = 1,2,3, \ldots, N$$

Obviously, $y(m) = Y(m,N)$. Also, $Y(m,0) = 0$.

The partial sums are recursively computed by $Y(m,k) = Y(m,k-1) + a(m,k)x(k)$.

Hence, $Y(m,k) = ZY(m,k) + a(m,k)x(k)$, which can be written as $(I-Z)Y(m,k) = a(m,k)x(k)$.

Multiply both sides by $(I-Z)^{-1}$ and get

$$Y(m,k) = (1-Z)^{-1}[a(m,k)x(k)] = \sum_{i=0}^{\infty} Z^i[a(m,k)x(k)] = \sum_{i=0}^{\infty} a(m,k-i)x(k-i) \tag{7}$$

A module for the implementation of equation (7) is shown in Figure 6.

However, it is apparent that a module corresponding to equation (7) has an infinite "response". The boundary conditions of the problem imply a need to bound this infinite response. This is done by using a modulo N counter to provide a reset mechanism, as in Cohen and Tyree [4].

With this additional control, designed for repetitive operation, the module can be redefined as

$$Y(m,k) = ZY(m,k) + a(m,k)x(k) \qquad \text{for } k \not\equiv 1 \pmod N$$

and (8)

$$Y(m,k) = 0 + a(m,k)x(k) \qquad \text{for } k = 1 \pmod N.$$

A module for the implementation of equation (8) is shown in Figure 7.

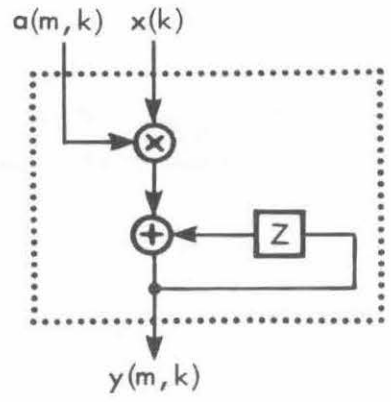

Figure 6: Infinite response module

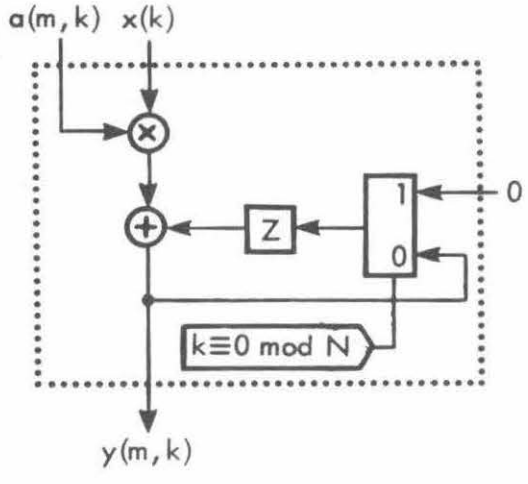

Figure 7: A finite response module

The reset occurs at the same time that the partial sum, Y(m,k), is equal to y(m). At this time y(m) is output, and the computation of the next y(m) begins.

Note that here there is a need for control signaling, which was not needed in the FIR case. This is because of the distribution of the FIR computation in space, where the "size" of the inner product (N) is determined by the actual size of the array, which implicitly defines the value of N. In the latter case

this computation is distributed in time, which necessitates the use of control signals for defining the value of N.

The discussion above focused on the single module for inner products. There are several ways for using M such devices in the design of a network for matrix–vector multiplication.

One way is to synchronize M such devices in parallel and to supply the same x(k) to all of them at the same time through any of many broadcasting techniques.[1] In such an arrangement a single reset control signal is broadcasted to every unit, and all the values of {y(m), m = 1,M} are available at the same step. Such a network is shown in Figure 8.

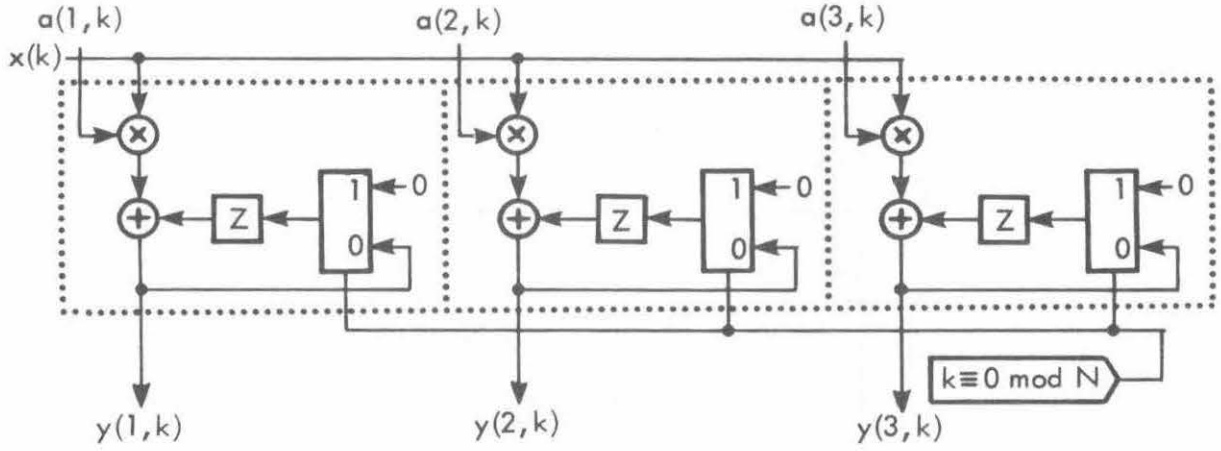

Figure 8: Synchronized matrix–vector multiplication

Another possible arrangement is based on the following relation:

$$Z^m y(m) = \sum_{i=1}^{N} Z^m[a(m,k)x(k)] = \sum_{i=1}^{N} [Z^m a(m,k)][Z^m x(k)] = \sum_{i=1}^{N} a'(m,k)[Z(m)x(k)],$$

where a'(m,k) = a(m,k–m), namely the same sequence "shifted" in time by m steps.

[1] It is assumed that the broadcasting delivers the same value to all of its recipients "at the same time".

This network has one delay in X between successive modules. It also changes the "phase" of each unit, such that the mth unit has to be reset when $k = m \pmod N$, unlike the condition $k = 0 \pmod N$ as above. Hence, each unit is reset at a different time (rather than all of them at once) and the elements of Y are available sequentially. In this implementation the input stream is continuous, one x(k) per cycle, as is the output stream, one y(m) per cycle.

The network for implementing this arrangement is shown in Figure 9.

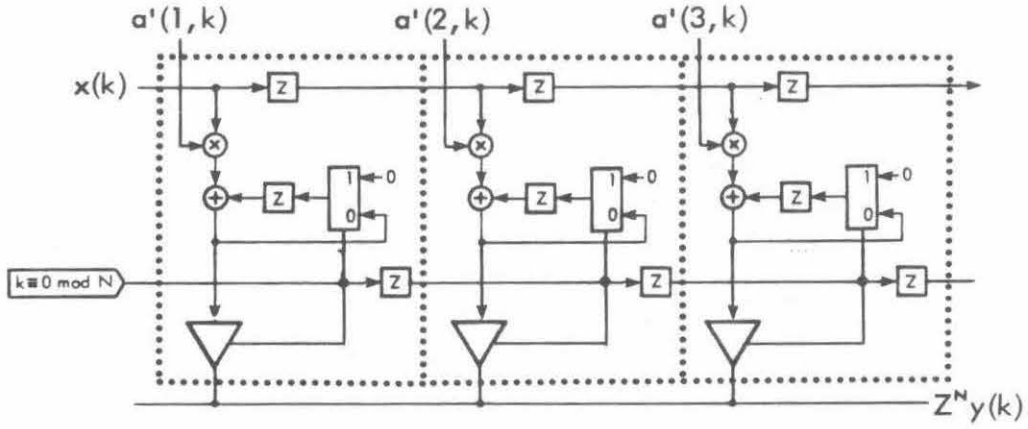

Figure 9: Pipelined matrix–vector multiplication

According to Figure 9, an array of M modules is capable of concurrently computing sequences of matrix–vector products of indefinite length, N.

Another related problem is the multiplication of a band matrix by a vector. A matrix A is defined to be a band matrix of width $r+s+1$ if $a(i,j) = 0$ for all $j>i+r$ and for all $i>j+s$.

A FIR filter can be considered as a multiplication of a band matrix by a vector. The filter coefficients are the diagonals of the matrix.

The structure of a band matrix suggests that the elements of the matrix are entered along its diagonals such that each diagonal is entered as a separate data stream.

In this arrangement the effect of Z on the matrix elements is $Za(m,k) = a(m-1,k-1)$.

The difference between this expression and equation (6) is due to the different order of the input data.

The concept of a wave front is useful because it simplifies the mathematical notation and it can be used as a conceptual tool in understanding how data progresses through a computational network. Wave fronts in computational networks are analogous to wave fronts in fluid dynamics. They are of particular importance in investigating laminar flow but are less useful in the study of turbulent flows.

Elements of a sequence of data values ordered in time can be viewed as a data stream. Elements of the same data stream are separated by a single time step and typically follow the same path through a computational network.

A wave front in fluid dynamics consists of a set of points in space which changes with time according to the propagation of the wave. Similarly, a wave front in a computational network consists of elements from different data streams. If we view a computational network abstractly as computing a result based on input operands, then it is possible to associate a set of data streams with a particular input operand. In the previous example of matrix–vector multiplication, the matrix operand consisted of a set of data streams, where each data stream corresponded to a row of the matrix. Using the wave front concept it is possible to change the view of an operand from a set of data streams to a set of wave fronts. These wave fronts are essentially a series of parallel cross sections of the set of data streams. More precisely, such a wave front contains exactly one element from each data stream. We are particularly interested in wave front sequences that contain all of the data elements present in the set of data streams comprising the operand.

In the implementation of the matrix–vector multiplication, as shown in Figure 8, all of the multiplications corresponding to a column are performed concurrently, and the matrix elements enter the array column by column. The input data set applied at time k may be defined as a wave front WF(k). Applying the Z operator to every data stream results in the wave front WF(k–1).

In general, $ZWF(k) = WF(k-1)$. By this definition a wave front corresponds to a column of the matrix A. Figure 10 shows these wave fronts.

However, in the implementation shown in Figure 9, the wave fronts are skewed because of the relative delay between successive modules. These wave fronts are shown in Figure 11.

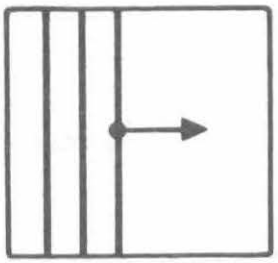

Figure 10: Vertical wave fronts

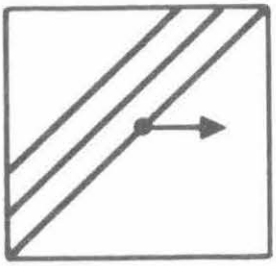

Figure 11: Skewed wave fronts

It is possible to perform several types of transformations on wave fronts. For example, it is possible to transform a wave front representing a row in a band matrix to a wave front representing a column in that matrix, and vice versa. This is done by applying Z^i to each element of the initial wave front, where the value of i corresponds to the data element position in the initial wave front. This is, in effect, a rotation of the initial wave front that results from applying the variable delay to its elements.

The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is defined by

$$y(k) = \sum_{m=0}^{N-1} w^{mk} x(m) \quad \text{for} \quad k = 0,1,2, \ldots, N-1$$

where $w = e^{-2\pi i/N}$

The DFT can be considered as a special case of a matrix-vector product. The approach corresponding to equation (8) is directly applicable. The elements in a row or a column can be easily generated because the ratio between consecutive elements is a constant. The computational

networks we derive here explore the fact that the ratio between consecutive elements in a column is a constant.

Define $Y(m,k) = w^{mk}x(m)$ such that each $Y(m,*)$ is a data stream. Therefore, $ZY(m,k) = Y(m,k-1)$.

The sequence $\{Y(m,k), m = 0,1,2...,N-1\}$ may be generated by

$$Y(m,k) = w^{m}Y(m,k-1) = w^{m}ZY(m,k) = w^{mk}Y(m,0)$$

and $Y(m,0) = x(m)$.

Obviously,

$$y(k) = \sum_{m=0}^{N-1} Y(m,k) \quad \text{for} \quad k = 0,1,2, \ldots, N-1. \tag{9}$$

A module generating the variables $Y(m,*)$ is shown in Figure 12.

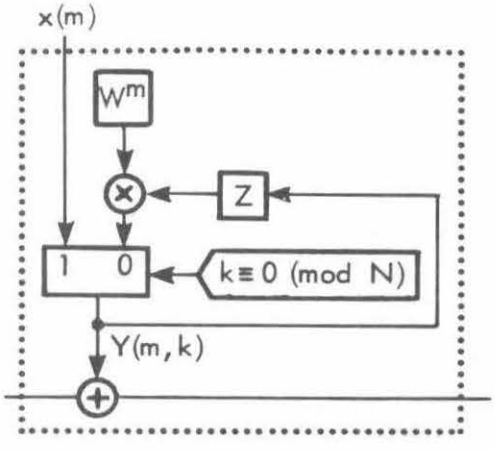

Figure 12: The Y(m,*) module

An array corresponding to equation (9) suffers from the need to perform N–1 additions in one step. The modules in the array are initiated with different values, $\{x(m)\}$, which become available at successive steps. The input values therefore have to be stored for the initialization of the modules.

The implementation corresponding to equation (9) can be improved by using pipelined addition.

Let the $\{Y(m,*)\}$ modules be interconnected into an array in a pipelined manner as shown in Figure 4.

Let s(k) be the output of this array at time k; then

$$s(k) = Y(N-1,k) + Z(Y(N-2,k) + Z(...Z(Y(0,k))...)). \tag{10}$$

The modules are initialized so that

$$\begin{aligned} &Y(m,k) = x(k) && \text{for } k = m \pmod N,\ m = 0,1,2,..,N-1 \\ &\text{and} \\ &Y(m,k) = w^m Y(m,k-1) && \text{otherwise.} \end{aligned} \tag{11}$$

Equation (11) expresses a sampling mechanism. The values of the data stream X are multiplexed into the N modules in a cyclic manner. The output S is well defined from time N–1 and on. To study the output S we rewrite equation (10) as

$$s(k) = Y(N-1,k) + Y(N-2,k-1) + Y(N-3,k-2) + ... + Y(0,k-(N-1)).$$

Hence $s(N-1) = x(N-1) + x(N-2) + ... + x(0) = y(0)$

$$s(N) = w^{(N-1)}x(N-1) + w^{(N-2)}x(N-2) + ...+ w^{0}x(0) = y(1)$$

...

$$s(2(N-1)) = w^{(N-1)(N-1)}x(N-1) + w^{(N-2)(N-1)}x(N-2) + ... + w^{0(N-1)}x(0) = y(N-1)$$

$$s(2N-1) = x(2N-1) + x(2N-2) + ... + x(N) = y(N)$$

...

An array corresponding to equations (10) and (11) is shown in Figure 13.

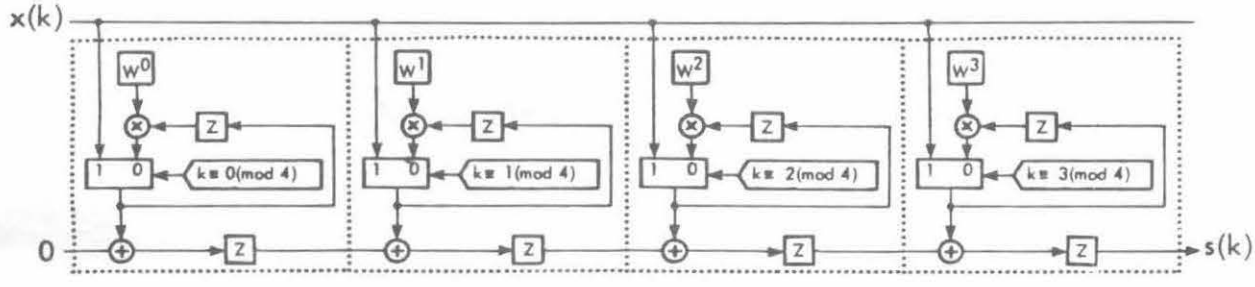

Figure 13: DFT array of size 4

The set of multiplications being performed during a step of the computations lies on lines parallel to the diagonal from the lower left-hand corner to the upper right-hand corner of the matrix defining the DFT. These lines can be defined as wave fronts. The computations start at the upper left-hand corner and proceed downward and to the right. The wave fronts are illustrated in Figure 14. The multiple wave fronts shown in Figure 14 correspond to the case where a sequence of DFTs are computed by the array. There is one wave front for each DFT. Hence, computations belonging to two DFTs are typically performed concurrently.

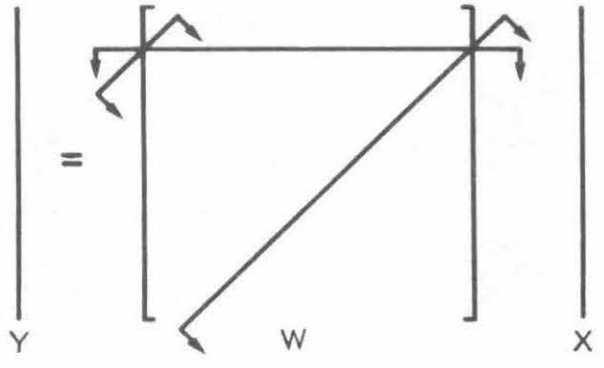

Figure 14: Wave fronts for DFT

Multiplexing the elements of the stream into a set of N modules can also be used in the general matrix-vector multiplication case. The matrix elements cannot in general be generated within a module but have to be supplied to the modules. Each module should be supplied with the matrix elements in a column in row order. The data stream associated with a column should be delayed one time step with respect to the stream associated with the preceding column. The loop around the multiplier is replaced with a storage element into which the elements of the stream X are multiplexed. The output of the multipliers should be added in a pipelined manner. Figure 15 illustrates the data organization schematically with the matrix rows indicated by dashed lines. Wave fronts can be associated with the set of matrix elements that enters the array at any given time. The wave fronts defined in this way are indicated by solid lines in Figure 15 and correspond to the diagonals in Figure 14.

The implementation corresponding to equations (10) and (11) contains O(N) modules and computes the DFT in O(N) time. Following the FFT logic it is possible to reduce the number of modules to $O(\log_2 N)$, by further exploration of the properties of the coefficients.

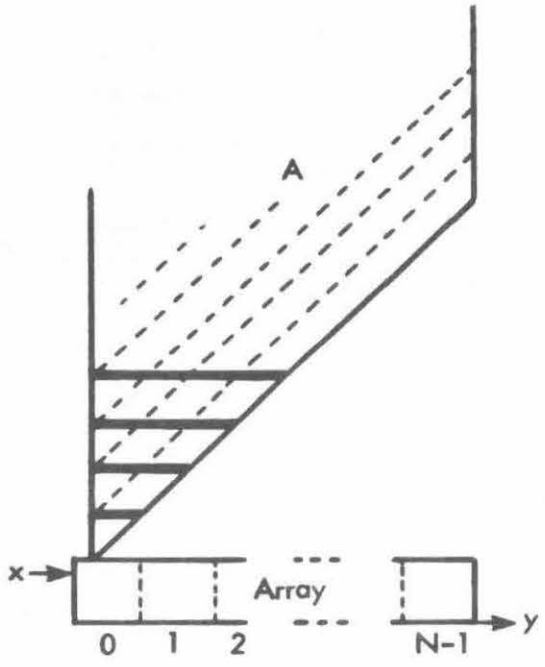

Figure 15: Wave fronts for DFT, v2

We assume $N = 2^n$.

The coefficient $\{w^{jk}\}$ has the property

$$w^{(j+N/2)k} = \begin{cases} = w^{jk}, \text{ k even, } j = 0,1,2,\ldots,N/2-1 \\ = -w^{jk}, \text{ k odd, } j = 0,1,2,\ldots,N/2-1. \end{cases}$$

Hence,

$$y(2j) = \sum_{k=0}^{\frac{N}{2}-1} w^{2jk}(x(k) + x(k + \tfrac{N}{2})) \text{ for } j = 0,1,2,\ldots,\tfrac{N}{2}-1$$

and

$$y(2j+1) = \sum_{k=0}^{\frac{N}{2}-1} w^{k(2j+1)}(x(k) - x(k + \tfrac{N}{2})) \text{ for } j = 0,1,2,\ldots,\tfrac{N}{2}-1$$

or

$$y(2j+1) = \sum_{k=0}^{\frac{N}{2}-1} w^{2jk}(w^{k}(x(k) - x(k + \tfrac{N}{2})) \text{ for } j = 0,1,2,\ldots,\tfrac{N}{2}-1,$$

Thus, the even and odd components of the DFT can be obtained as matrix–vector products by using the same N/2 by N/2 matrix operating on different vectors.

Define a new data stream V obtained from the stream X as follows:

Let $v(k) = (1 + Z^{-N/2})x(k)$ for $k = 0,1,2...N/2$

and $v(k) = w^k(1-Z^{N/2})x(k)$ for $k = N/2, N/2+1, ..., N-1$.

The former definition requires a negative power of Z, a prediction that cannot be implemented in general. However, by multiplying both sides of this definition by $Z^{N/2}$ no prediction is needed. This means that instead of computing v(k), the network actually computes $Z^{N/2}v(k)$, which is the desired value delayed by N/2 steps. Since the first half of {v(k)} has to be delayed by N/2 steps we also delay the second half by the same amount.

A module computing the sequence V is shown in Figure 16.

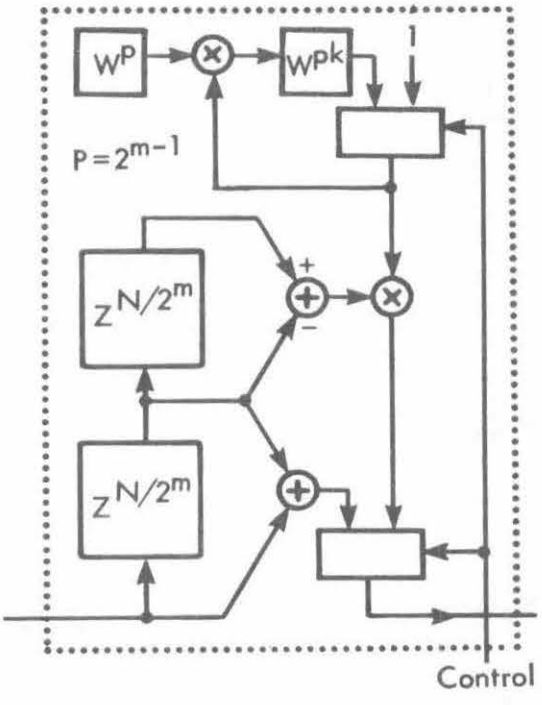

Figure 16: A modified butterfly module

Then

$$y(2j) = \sum_{k=0}^{\frac{N}{2}-1} w^{2jk}(v(k) \quad \text{for } j = 0,1,2,\ldots,\tfrac{N}{2}-1$$

and (12)

$$y(2j+1) = \sum_{k=\frac{N}{2}}^{N-1} w^{2jk}v(k) \quad \text{for } j = 0,1,2,\ldots,\tfrac{N}{2}-1$$

or in matrix form

$$\begin{bmatrix} y(0) \\ y(2) \\ \cdots \\ \cdots \\ y(N-2) \\ y(1) \\ y(3) \\ \cdots \\ \cdots \\ y(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & w^2 & w^4 & \cdots & w^{N-2} & 0 & 0 & 0 & \cdots & 0 \\ 1 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 1 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 1 & w & w & \cdots & w & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & w^2 & w^4 & \cdots & w^{N-2} \\ 0 & 0 & 0 & \cdots & 0 & 1 & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & 1 & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & 1 & w & w & \cdots & w \end{bmatrix} \begin{bmatrix} v(0) \\ v(1) \\ \cdots \\ \cdots \\ \cdots \\ v(N/2) \\ \cdots \\ \cdots \\ \cdots \\ v(N-1) \end{bmatrix}$$

The original matrix has been transformed into a block diagonal matrix with identical diagonal blocks. Obviously, the observation used to obtain equation (12) can be applied to each of the diagonal blocks recursively. Eventually, the block diagonal matrix becomes a diagonal matrix, and the computation of the DFT is completed. Figure 17 shows an array of $\log_2 N$ modules computing the DFT.

The input has to be in normal order. The output appears in bit-reversed order. There is no broadcasting, and no module has to contain any long wires. The computational rate is limited by $\log_2 N$ adders in series. The computational rate can be improved by introducing delays in a straightforward manner. The first component of the DFT will appear at the output of the array at the time the last data item, x(N–1), arrives, and the last component will appear at the output N–1 steps later.

Please note that this implementation performs FFT by using decimation in frequency.

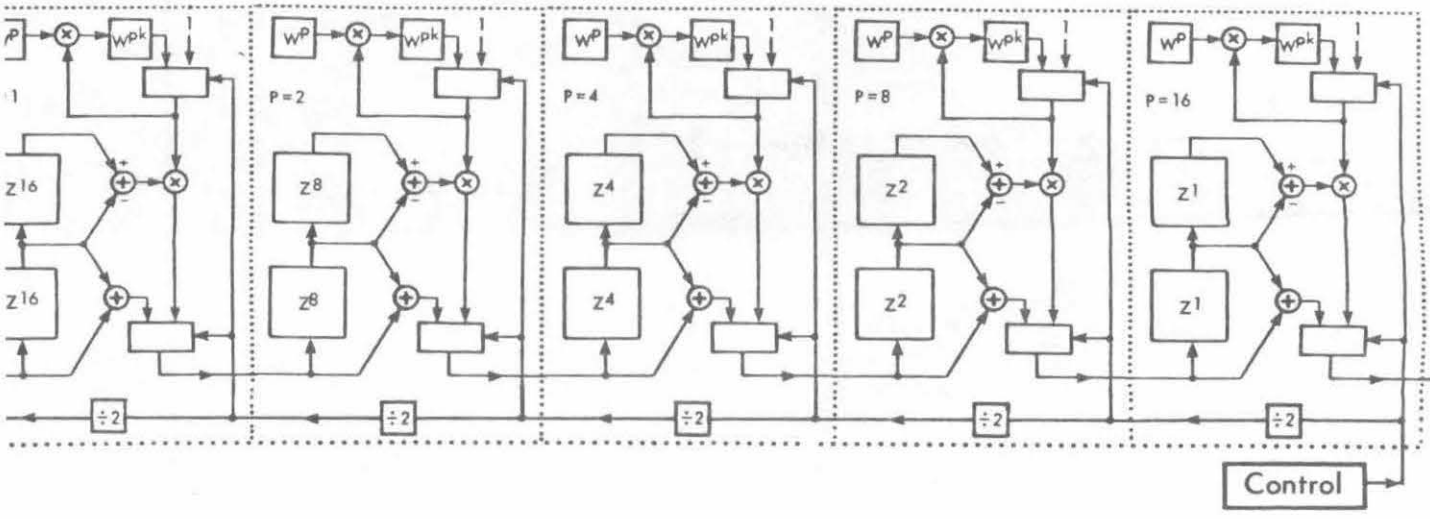

Figure 17: An FFT array

In the FIR filter, the matrix–vector multiplication cases, and the DFT network of N modules, the data streams can be considered as laminar. The wave fronts can be considered as corresponding to points of constant phase on waves. The network considered for the computation of the DFT by $\log_2 N$ modules does not preserve the laminar flow. The concept of wave fronts is not attractive in this case as in the case for turbulent flows.

Conclusions

The simple model of storage used in this paper is given a precise mathematical meaning. Once the organization of the input data is determined, the Z operator (i.e., the mathematical model of a storage element) can be used to model the ordering of input data streams directly. The mathematical equation that defines a function to be computed can be transformed from a form that contains no concept of time to a form that contains information about time as well as space. This new form is typically an expression containing the Z operator.

Equations containing information about the time and space required to compute a function can be manipulated formally, since the properties of the Z operator are well defined. It is possible to give expressions containing the Z operator a direct hardware interpretation. Properties such as computational rate, performance, delay, modularity, communication structure, and fault tolerance can be determined directly from an expression using the Z operator.

The methodology suggested in this paper is useful in synthesis as well as analysis and verification of computational networks. It is possible to iterate between formal manipulations of expressions in the Z

operator and graphs describing computational networks. Modeling the behavior of a network using the Z operator to describe a storage element makes it conceptually straightforward to verify whether or not the network actually computes the function it is supposed to compute.

A formal approach as outlined in this paper is particularly useful in complex problems where designs based entirely on intuition may be incorrect or may have a performance lower than necessary for a given amount of hardware. For instance, using the methodology suggested here, Weiser and Davis [14] have discovered designs of systolic arrays with two to three times higher performance than the corresponding arrays by H. T. Kung and Leiserson [8]. Furthermore, the designs are proven to be correct.

Explicit control can be modeled within the formalism. The expressions become more complex, but the formalism allows for the treatment of space–time tradeoffs. If at the first iteration of the design cycle the spatial requirements of a computational network for a function are too large, the hardware requirements can be reduced by mapping the computations to a network of reasonable size. In doing so, it is necessary to model the control explicitly. Eventually the control will become fairly complex, but it can still be included in the formalism.

The formalism allows for a precise definition of wave fronts. The concept of wave fronts is useful both in designing computational networks and in finding suitable organizations of the input data. Wave fronts are particularly useful in problems where the data flow can be considered laminar. The formalism proposed is not, however, limited to networks with laminar flow as the example describing arrays for the FFT shows.

Acknowledgments

The authors gratefully acknowledge the support for this research provided generously by the Defense Advanced Research Projects Agency, under contracts MDA–80–C–0523 with the USC/Information Sciences Institute and N00014–79–C–0597 with the California Institute of Technology, and by the Burroughs Corporation for the Data Driven Research Project at the University of Utah.

Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, any person or agency connected with them, nor of the Burroughs Corporation.

References

1. Ayres, R., "IC design language," *Proceedings of the 16th Design Automation Conference*, June 25-27, 1979, pp. 307-309. IEEE No. 79CH1427-4, Library of Congress Card No. 76-150348.

2. Chen, M., and C. Mead, "A notation for designing concurrent systems," *Internal Document (3927)*, Caltech, Computer Science Department, August 1980.

3. Cohen, D., "Mathematical approach to iterative computational networks," *Proceedings of the Fourth Symposium on Computer Arithmetic*, pp. 226–238, October 1978, also published as USC/Information Sciences Institute RR–78–73, November 1978.

4. Cohen, D., and V. C. Tyree, "VLSI system for Synthetic Aperture Radar (SAR) processing," *Proceedings of the Society of Photo–Optical Instrumentation Engineers (SPIE)*, Vol. 186, pp. 166–177, 1979.

5. Johannsen, D., "Bristle Blocks: A silicon compiler," *Proceedings of the Caltech Conference on VLSI*, January 1979.

6. Johnsson, L., and D. Cohen, "Computational arrays for the Discrete Fourier Transform," *COMPCON*, February 1981.

7. Johnsson, L., and D. Cohen, *A Mathematical Approach to Computational Networks for the Discrete Fourier Transform*, USC/Information Science Institute, RR–81–90, 1981 (forthcoming).

8. Kung, H. T., and C. E. Leiserson, "Algorithms for VLSI processor arrays," Section 8.3 in [10].

9. Kung, S. Y., "VLSI matrix computation array processor," *The MIT Conference on Advanced Research in Integrated Circuits*, February 1980.

10. Mead, C., and L. Conway, *Introduction to VLSI Systems*, Addison–Wesley, 1980.

11. Rem, M., and C. Mead, "A notation for designing restoring logic circuitry in CMOS," *Second Caltech Conference on VLSI*, January 1981.

12. Rowson, J., *Understanding Hierarchical Design*, Caltech, Computer Science Department, Report 3710, April 1980.

13. Seitz, C., "System timing," Chapter 7 in [10].

14. Weiser, U., and A. Davis, *Mathematical Representation for VLSI Arrays*, University of Utah, Computer Science Department, Report UUCS–80–111, September 1980.