Bit-Serial Inner Product Processors in VLSI

Misha R. Buric

Bell Laboratories
Murray Hill, New Jersey 07974

Carver A. Mead

California Institute of Technology
Pasadena, California 91125

1. Introduction

Many problems in signal and image processing, pattern recognition, and feedback systems involve models with vector variables. Besides vector addition and multiplication by a scalar, an inner product of vectors is a basic arithmetic operation in these models. It is computationally most demanding, so that there is considerable interest in finding ways to speed up its implementation. Array configurations of simple processors for performing vector and matrix operations have been extensively reported. A number of ideas can be found in [1], [2], [7], [8], and their references.

In this paper we describe a bit-serial pipelined implementation of an inner product processor, and related interconnections of a number of such processors on a single chip. We argue that bit-serial computational models are particularly suited for VLSI, because of relatively inexpensive communication links and arithmetic processing elements, in terms of the area occupied on silicon. Sixteen inner product processors, described here, may be easily placed on a single 40-pin chip in today's NMOS technology with a 2 micron lambda. Similar arguments for bit-serial arithmetic were used in [3], in a description of a design of a general purpose massively parallel processor.

Generally, all multiprocessor schemes can be divided into two classes with respect to the interconnection patterns among processing elements. Static communication links characterize those array configurations in which a fixed algorithm is executed repeatedly and synchronously with the input data flow. These structures are especially useful if the input information is being continuously provided by sampling some real world variables, and the purpose of the processing is to provide a compressed version of the data, or to transform it into another representation. Many examples can be found in speech and image processing. This structure and its variations is examined in this paper. Another, more general class of multiprocessor schemes involves flexible communication interconnections among the processors through switched networks, [7].

2. Basic Processors and their Interconnections

An inner product of two n-dimensional vectors, x and y, is defined by

$$z = x^T y = \sum_{i=1}^{n} x_i y_i \tag{1}$$

where

$$x^T = [\, x_1\ x_2 \ldots x_n \,] \text{ and } y^T = [\, y_1\ y_2 \ldots y_n \,]$$

We use a convention that all vectors are column vectors, and that T denotes a transpose operation. A transposed column vector is a row vector. It is assumed here that all the vector elements are integers. The equation (1) can be rewritten in an iterative form as follows:

$$z_i = x_i y_i + z_{i-1} \ , \ i=1,2, \ldots n \qquad (2)$$

$$z_0 = 0 \ , \ z = z_n$$

There are a number of ways to implement this equation, ranging from a single multiplier-accumulator combination, to an array of n processors.

In the first case, which may be called an iteration in time, the 2n operands are fetched two at the time, they are multiplied together, and added to the previously accumulated value. This requires n steps, assuming that the pipeline registers are used to enable overlapping of the operations.

On the other side, an iteration in space is characterized by a pipelined array configuration of n processors that operates on 2n operands simultaneously, and provides a new result every cycle. The i-th processor is assigned to the i-th elements of the input vectors, and to the (i-1)-st partial sum, and it produces the i-th partial sum. Due to pipelining, a processor may receive new operands every cycle.

Iteration in space will be considered first, because it provides the necessary throughput for large vector sizes. In a VLSI implementation of this scheme, the cost of arithmetic elements, and the communication cost of providing 2n operands and interconnecting individual processors, would be prohibitive if we were to use word-parallel arithmetic.

Bit-serial arithmetic and communication, however, offers a viable alternative for two reasons. First, the arithmetic components and the communication lines occupy much smaller area on silicon. Therefore, a larger number of basic processors may be integrated on a single chip. Second, the nature of inner product computation requires more bits of precision for larger vector sizes. A fixed-word parallel arithmetic is not suitable for a flexible precision control, since the overflow conditions may occur for larger vectors, unless a sufficiently large adders are provided in advance. On the other side, bit-serial addition does not suffer from this problem, since the precision may be maintained arbitrarily high with the same hardware.

In this text we use b^{-1} as a bit-delay operator, analogously to z^{-1} which we reserve for signal processing applications. This notation is convenient for treating bit-serial systems, because there may be various delays in a complex array of serial elements, so that a systematic treatment of path delays is important.

Our implementation of a single inner product processing element is shown in Figure 1.

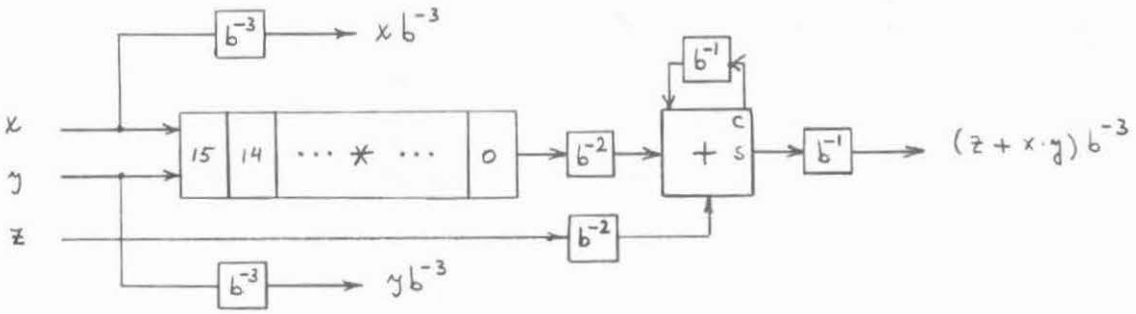

Fig. 1

It consists of a modular bit-serial multiplier, (two's complement, 16-bit), and a single-bit carry-save adder. The input variables x, y, and z enter the processor with the least significant bit first, and the result starts appearing three bit times later. The computation is synchronized by a control bit, that is applied simultaneously with the LS bits of the operands. The processor provides 31 bits of the result. An additional 0 bit is inserted between the result and the LS bit of the next product. The input operands are padded with 16 zeros to the left of the most significant bit, so that new operands may be applied every 32 clock cycles.

Clocking is two-phase, such that phase one controls all data transfers, and all logic functions are evaluated in phase two. The delays b^{-1} imply shift-register pipeline stages.

There are some variations of the circuit in Figure 1 that are application dependent. The delayed input variables do not have to be provided at the output in some array configurations. Also, we will show later in the text that the adder may be connected to a pair of multipliers instead as shown in Figure 1.

The size of a single inner product processor was 2720 by 155 lambda with a linear layout, and 680 by 620 lambda in a rectangular configuration. There was no attempt made to minimize the size of the basic multiplier cell. Despite this, it is feasible to place sixteen such processors on a 40-pin chip, but the limitation in this and future implementations is not so much the area, as much as the number of external connections.

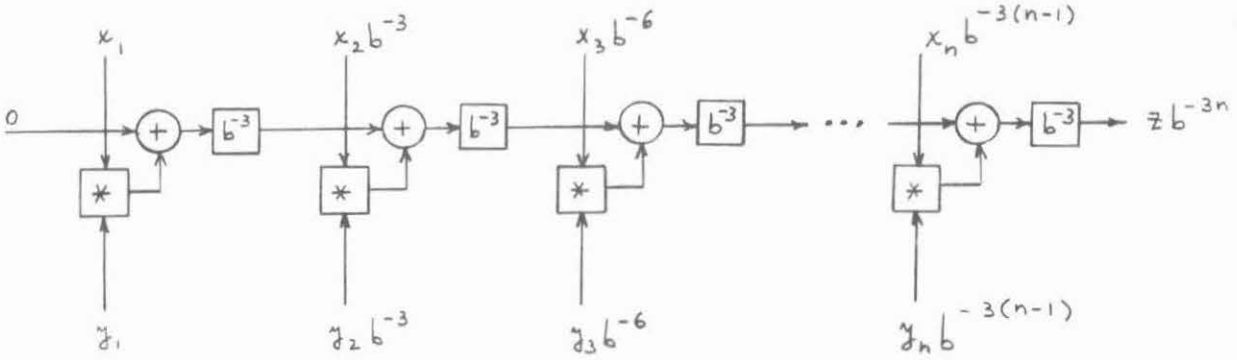

Fig. 2

This brings up the main question: what is the best interconnection among processors on a chip, which makes further combinations of such chips most flexible?

A linear array, as in Figure 2, has many desirable properties for a number of applications. The number of pins for a 16 processor combination is less than 40, and the array may grow arbitrarily large. The expression for an inner product of two large vectors has the same iterative form in terms of inner products of their smaller parts, as in equation (2). That is, if x and y are two vectors of size M, we can partition them into K groups of smaller vectors of size n, so that the inner product $x^T y$ may be evaluated as follows:

$$x^T = [\, c_1^T \;\; c_2^T \;\; \cdots \;\; c_K^T \,], \quad y^T = [\, d_1^T \;\; d_2^T \;\; \cdots \;\; d_K^T \,]$$

$$z = x^T y = c_1^T d_1 + c_1^T d_2 + \; \ldots \; + c_K^T d_K$$

$$z_i = c_i^T d_i + z_{i-1}, \quad i = 1,2, \; \cdots \; , K \, ,$$

$$z_0 = 0 \;\; , z = z_K$$

If each partial inner product of size n is computed in a separate chip, then the complete result is obtained by a linear connection of the K chips. The throughput rate remains the same, one inner product per 32 clock cycles regardless of the vector sizes. Of course, extra cycles may be inserted between two products if there is a need for the overflow control in the adder stages. In Figure 2 the variables x_i , y_i are integer elements of vectors x and y, which enter the processor bit serially. The expression $x_i b^{-j}$ means that the integer x_i is delayed by j bit cycles. Notice that each element of the vectors is delayed by b^{-3}, with respect to the previous element. This skewing does not pose any conceptual difficulties, but for practical reasons it would be easier to apply all elements simultaneously, without any bit delays among the operands. A possible solution is to include an appropriate shift register delay at the input of each section, but this would increase the area of the chip.

A better form of a linear array of basic inner product processors, suggested in [8] for word-parallel arithmetic, is shown in Figure 3. This configuration does not require any delays

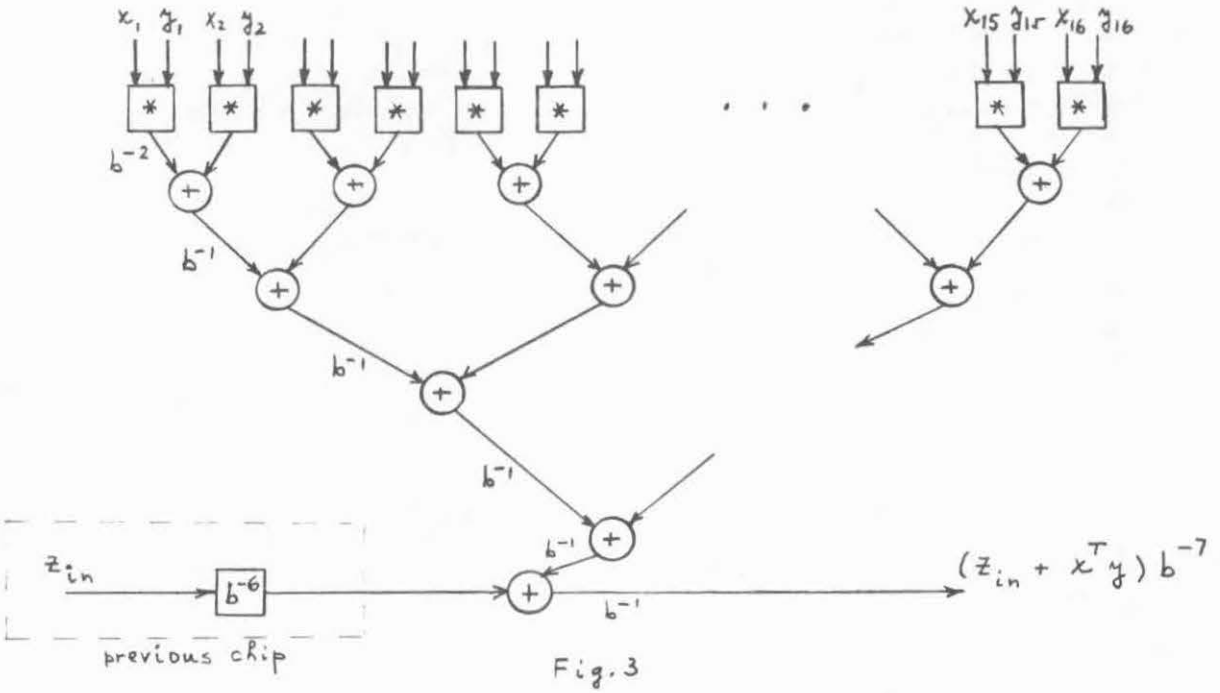

Fig. 3

among the vector elements, and produces the initial product with the delay of only b^{-6} in a sixteen processor chip. Here, the adders are organized in a tree structure, and an additional carry-save adder is added to the chip, with no internal connections. The purpose of this adder is to allow interconnections of an arbitrary number of chips for larger inner products. The pipeline registers are assumed to be included in each adder, and the corresponding delays are shown externally.

A connection of individual 16 element chips for operating on larger vectors is shown in Figure 4. Summation of the partial inner products is again done by a tree structure of the adders, which are obtained from a pool of free adders.

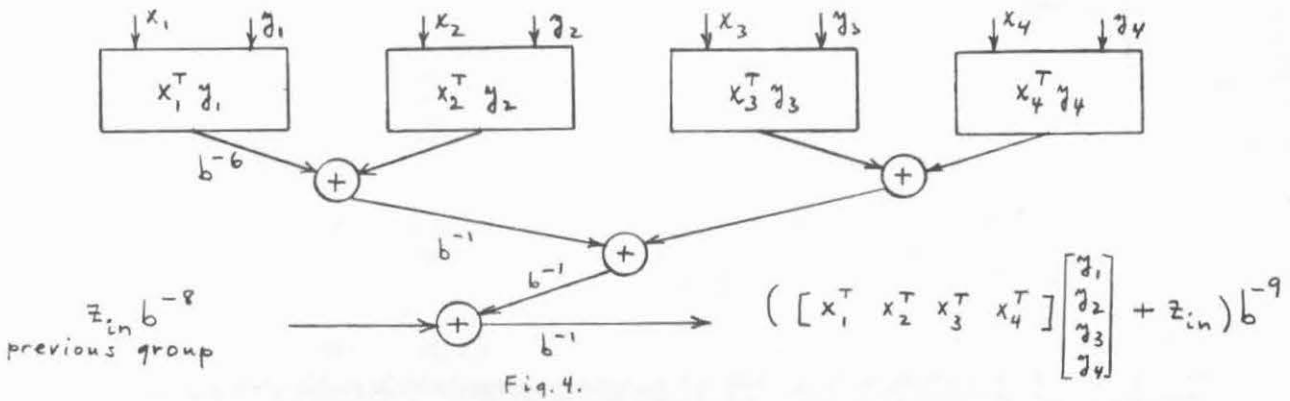

Fig. 4.

In Figure 4 each x_i and y_i are 16 dimensional vectors.

This will be the main scheme in further discussion. It is sufficient for applications such as FIR and IIR filtering, matrix multiplications, vector convolutions and others.

Alternatively, the inner product processors may be connected together in a hexagonal array suggested in [1] and [2] for matrix operations. In this case, every processor has three

inputs and three outputs, but a chip with sixteen processors, shown in Figure 5, has two pairs of four inputs for the vector variables, two pairs of four delayed outputs of the same variables, and seven inputs and outputs for the result variables. In this case the input variables would be provided at the outputs delayed by b^{-3}, as in Figure 1. Larger hexagonal arrays may easily be constructed out of these sixteen processor chips, if each is viewed as a new basic element in a hierarchy. Even though this approach addresses a very important issue of the data flow through the network simultaneously with the computation, it requires more complex synchronization of the input operands.

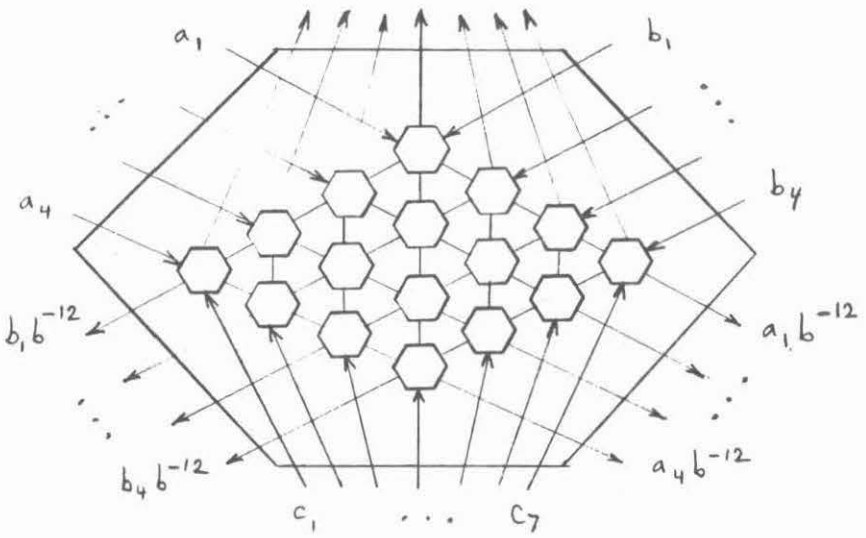

Fig. 5

3. Data Flow Control

It was assumed so far that an inner product of two vectors was computed by iterating n processors in space. The 2n operands were supplied simultaneously bit-serially, and due to pipelining there was no delay between consecutive results of a series of inner products. Therefore, the computational throughput was adjusted to match the input data rate at the expense of more processors. This approach can be extended to matrix products, because they consist of a number of inner products. First, multiplication of an n-dimensional vector by a matrix, (m by n), can be accomplished by m arrays of inner product processors, each consisting of n basic elements. One operand to all arrays is the vector, and the second operand is one row of the matrix for each array. Each array computes one component of the result simultaneously with others. Hence the throughput still remains the same as in the case of a single inner product. Similar structure can be used for a product of two matrices, (m by n) and (n by k), where mk linear arrays of dimension n compute mk results simultaneously.

In applications of inner product arrays outlined above there is a need for a data flow network that connects the source of operands with the computational structure, and provides for internal data flow during the operation. This is also important for purpose of matching the input data rate with the processing throughput. For example, in an FIR filter application a new data sample may be provided every T microseconds, and a filter has to perform one inner product of length N on two vectors. One vector consists of N previous samples, and the other is composed of filter coefficients. The result of the inner product is output once per period T. Let x_k be the input sample, and z_k the output value at time k. Then:

$$a^T = [a_0 \ \cdots \ a_{N-1}] \ , \ x^T = [x_k \ x_{k-1} \ x_{k-2} \ \cdots \ x_{k-N+1}]$$

$$z_k = a^T x = \sum_{i=0}^{N-1} a_i x_{k-i}$$

Therefore, both input and output are single integers, but the processor array operates on two vectors. In addition, the x vector has to be updated every sample time, in such a way that all the components change their position by one place:

$$x_{k+i+1} = x_{k+i} \ , \ i = 0,1, \ \cdots \ ,N-2$$

with the new sample becoming x_k. This is a shifting operation, suggesting a set of shift registers for this application. Now, if our inner product processor generates a result in Δt microseconds, Δt being much smaller than sampling period T, we can use a smaller processor, with only $N\frac{\Delta t}{T}$ basic processors, but there has to be a mechanism for accumulating partial results within one inner product and cycling through all partial vectors of this smaller size.

This simple example is an illustration of a problem which contains a combination of communicational and computational complexity. A standard measure of computational complexity in this case would indicate that the problem is solvable in $O(N)$ time, and since we can use an N processor array it becomes an $O(1)$ problem. However, there are additional communication costs of providing 2N operands, and performing N data exchanges. Also, in a VLSI implementation of this example it would be only sensible to provide all data movements within the same chip that contains the inner product processor, so that there is a single external connection for input and output.

This is typical for many algorithms with vector variables. Each operand interacts with a number of other operands before the computation is completed. Another example is a convolution of two vectors of size N, which requires 2N inner products of the same vectors, but one of them is shifted each time. If there is a reasonable restriction that the data be brought into a VLSI vector processor only once, which minimizes the number of interactions with the external world, then there has to be some data storage on the chip with a flexible data exchange scheme. This issue prompted a hexagonal array approach in [1] and [2], in which the data storage and flow takes place in each basic processing element, and the topology of a network is tailored for the problem.

Here, we examine another alternative that seems to be well suited for bit-serial vector processing. Consider a shift-register element that has two inputs, horizontal and vertical, and one output, Figure 6. It can shift the data either horizontally or vertically, as determined by a shift control signal. Next consider a standard shift-register of length N, say 16, which consists of N-1 standard cells and one two-input cell, shown in the same figure.

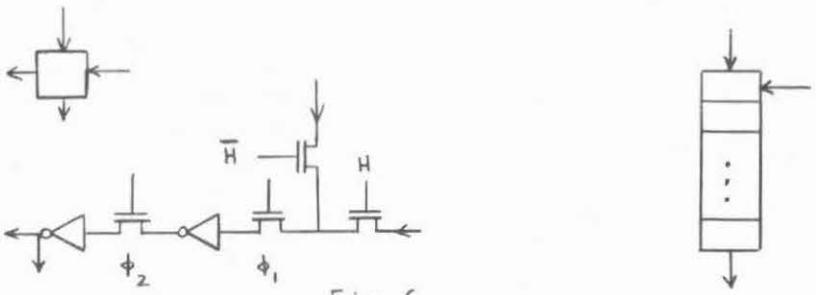

Fig. 6

A set of such shift registers can now be connected in a storage array with a two-dimensional shift capability. To demonstrate an application of such an array with an inner product processor let us consider an FIR filter implementation.

Suppose the filter is specified at 512 points, and the sampling period is 100 microseconds. A conservative estimate of the inner product performance is 3 microseconds per product. Therefore, a 16 processor array is sufficient for computing inner products of 512-dimensional vectors by iterating in time 32 times. A diagram of this configuration is shown in Figure 7, for

a smaller array. In order to iterate in time, a bit-serial accumulator is provided. There are two sets of register arrays, data and filter registers. The data registers are connected vertically in 16 circular groups, and horizontally into a linear array. In a case of a constant filter the filter registers may be connected in the same way as the data registers, even though there are applications in adaptive filtering where a different connection would be used. Each register group has a single one-bit input and output, the output being used for expansion purposes.

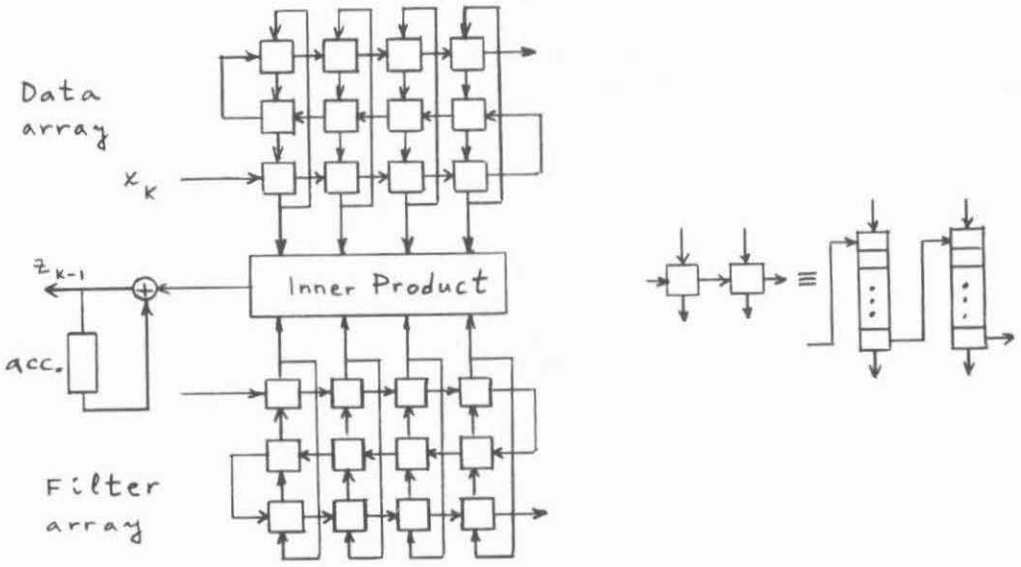

Fig. 7

The filter coefficients are loaded into the registers one at the time by using horizontal shifting. During each sample period the computation consists of 32 partial product cycles, and one memory shift cycle. Each partial product cycle results in a 16 component inner product, during which the registers are being shifted vertically. This provides bit-serial operand streams to the inner product processor, and simultaneously prepares new operands for the next cycle. At the same time the accumulator adds previous partial product to the one being computed. The last partial product cycle produces the result that is then shifted outside. In the next step a memory shift is done by shifting registers horizontally. The first register receives a new sample from the external source while it is transferring its content to the next neighbor to the right. At the same time the accumulator is cleared for the next round of partial product cycles. This sequence of steps is repeated for each new input sample.

This example is indicative of tradeoffs that have to be made in a practical design of array schemes for vector processing. The goal is to minimize the silicon area, while matching the processing speed with the available input/output data rates. Here, the area of two-dimensional shift register arrays was much smaller than an array of 512 inner product cells that could be used for the same computation. However, if the input sampling rate was on the order of 3 microseconds and the filter was specified with the same number of points, then a large processor array would be used. In addition, if the input rate was even larger, two arrays would have to be used, each operating on alternate samples.

A two-dimensional shift register array may be used in a similar way for other vector and matrix operations. An alternative to this approach is to use standard memory arrays with a specialized memory access facilities. An example is given in [9].

4. The Multiplier

There are a number of reported bit-serial multipliers in literature [4], [5], [6]. Most of them preserve only N most significant bits of the result. We have devised yet another configuration, which preserves all 2N-1 bits. This is important for inner product computations, where a large number of individual products are accumulated.

If two integers are given in a binary representation, then they can be viewed as vectors whose components are in the set [0,1]. Then, their product is a vector whose elements are obtained by a convolution of the two operand vectors. Alternatively, a polynomial representation with a delay variable b^{-1}, can be used for representing integers for bit-serial arithmetic. A convolution of two vectors is equivalent to a polynomial multiplication, if the vector elements are equated with the polynomial coefficients. Let x and y be two N-bit integers, represented as

$$x = \sum_{i=0}^{N-1} x_i b^{-i} \quad , \quad y = \sum_{i=0}^{N-1} y_i b^{-i}$$

The product polynomial is given by

$$z = \sum_{k=0}^{2N-2} z_k b^{-k} \quad , \quad z_k = \sum_{i=0}^{k} x_i y_{k-i}$$

It is interesting to note that each z_k is an inner product of two binary vectors, so that a multiplier design becomes an exercise in configuring a regular array structure for inner product computation. In order to derive such a structure, we rewrite the result polynomial as:

$$z = \sum_{k=0}^{2N-2} (z_k' + z_k'') b^{-k}$$

$$z_k' = \sum_{i=0}^{[\frac{k}{2}]} x_i y_{k-i} \quad , \quad z_k'' = \sum_{i=0}^{[\frac{k+1}{2}]-1} y_i x_{k-i}$$

These two expressions can be written in an iterative way, such that they map into a linear pipelined array of N sections. The j-th section computes the following:

$$z_j' = x_j y_k + z_{j+1}' b^{-1} \quad , \quad j \leqslant k$$

$$z_j'' = y_j x_k + z_{j+1}'' b^{-1}, \quad j < k$$

$$k = 0,1, \cdots N-1 \quad , \quad j = 0,1, \cdots, N-1 \quad , \quad z_j = z_j' + z_j''$$

The section 0 provides the product polynomial, with $z_0 b^{-1}$ being the least significant bit. Notice that the additions in the above expressions are arithmetic, with a carry bit.

A diagram of a single section of the multiplier, and the connection of the sections, are shown in Figure 8. The operands are applied on two single bit buses, x and y, one bit per cycle. The control bit is provided simultaneously with the LS bits, and it advances from the 0-th section to the remaining stages synchronously with the bit rate. Its purpose is to enable x and y latches in each section. In the i-th cycle, it deposits i-th x and y bits in the i-th latches. Each section computes two partial sums, z_i' and z_i''. Carry-save adders add together three values, a product of two bit values, previous carry bit, and the delayed partial sum from the next section. The 0-th section also contains an adder for forming the final result. Finally, two's complement multiplication is obtained by applying a special "subtract" signal, simultaneously with the most significant bits of the operands, x_{N-1} and y_{N-1}. This has the effect of converting *all* adders to borrow-save subtractors at this time, (except in the last section). In this implementation, it takes 2N-1 steps to perform a multiplication of two N bit numbers.

The floor-plan of a single multiplier section looks very much like the diagram in Figure 8. All signals were chosen to run horizontally, so that a multiplier with an arbitrary number of bits can be constructed in many ways, by abutting sections on two edges only.

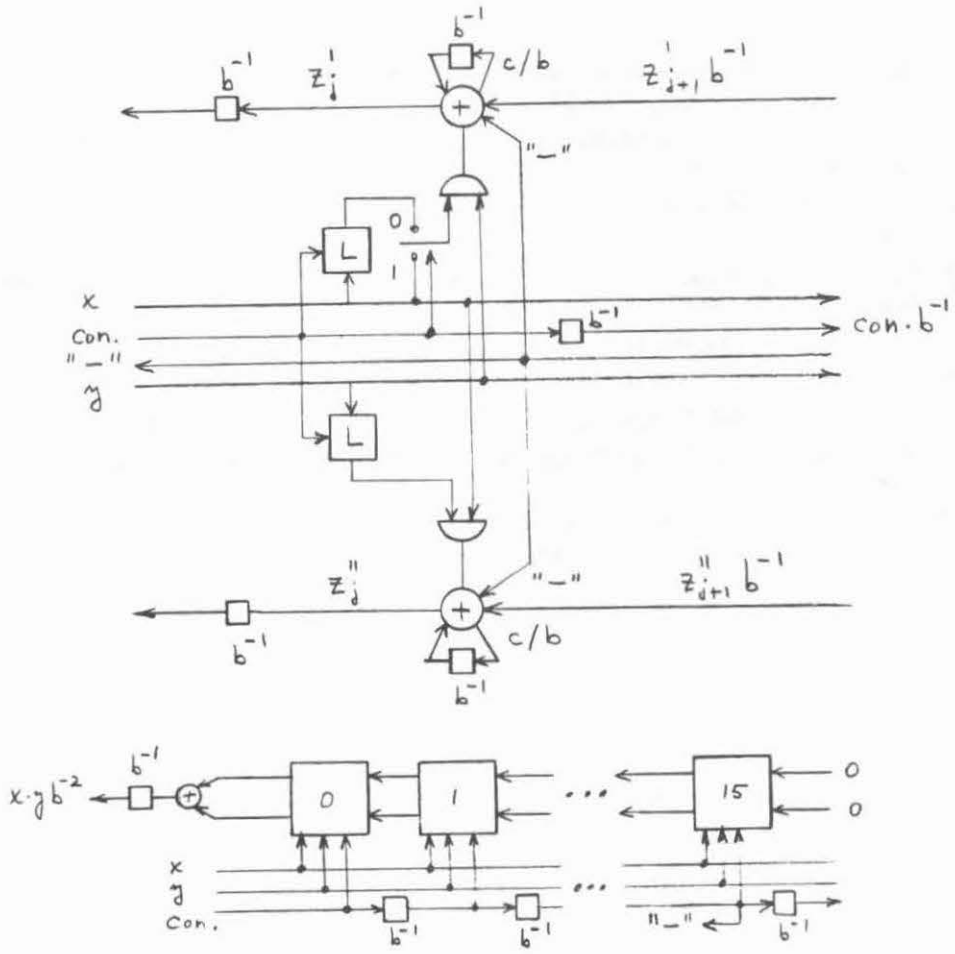

Fig. 8

5. Conclusion

A bit-serial approach to computations of vector inner products offers many advantages over word arithmetic. The size of basic processing elements and communication links is much smaller, and the array configurations are easy to implement. The slower rate of operation of a single multiplier-adder combination is offset by a much higher throughput rate of a large number of processors. A single element has been designed and tested, and a sixteen processor combination with a tree of adders is under way. The approach is especially useful for real-time signal processing tasks.

6. Acknowledgements

We are grateful to David Hagelbarger for suggesting the structure of the multiplier. Also, we would like to thank Sandy Fraser and Mike Maul for making the chip production possible.

7. References

1. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems,* Addison-Wesley, 1980, 263-330.

2. H. T. Kung, *The Structure of Parallel Algorithms,* Carnegie-Mellon University, August 1979.
3. K. E. Batcher, "Design of a Massively Parallel Processor", *IEEE Trans. Comput.,* Vol. C-29, pp. 836-840, Sept. 1980.
4. E. K. Cheng and C. A. Mead, "A Two's Complement Pipeline Multiplier", *Proc. ICASSP,* Apr. 1976.
5. R. F. Lyon, " Two's Complement Pipeline Multipliers" *IEEE Trans. Commun.,* 418-425, Apr. 1976.
6. L. B. Jackson et al., "An Approach to the Implementation of Digital Filters", *IEEE Trans. Audio Electroacoust.,* vol. AU-16, pp. 413-421, Sept. 1968.
7. Special Issue on Parallel Processing, *IEEE Trans. Comput.,* vol. C-29, Sept. 1980.
8. E. E. Swartzlander et al., "Inner Product Computers", *IEEE Trans. Comput.,* vol. C-27, pp. 21-31, Jan. 1978.
9. K. E. Batcher, "The Multidimensional Access Memory in STARAN", *IEEE Trans. Comput.,* vol. C-26, pp. 174-177, Feb. 1977.