Fourier Transforms in VLSI

CLARK D. THOMPSON

***Abstract*—This paper surveys nine designs for VLSI circuits that compute N-element Fourier transforms. The largest of the designs requires $O(N^2 \log N)$ units of silicon area; it can start a new Fourier transform every $O(\log N)$ time units. The smallest designs have about $1/N$th of this throughput, but they require only $1/N$th as much area.**

The designs exhibit an area–time tradeoff: the smaller ones are slower, for two reasons. First, they may have fewer arithmetic units and thus less parallelism. Second, their arithmetic units may be interconnected in a pattern that is less efficient but more compact.

The optimality of several of the designs is immediate, since they achieve the limiting area $*$ time^2 performance of $\Omega(N^2 \log^2 N)$.

***Index Terms*—Algorithms implemented in hardware, area–time complexity, computational complexity, FFT, Fourier transform, mesh-connected computers, parallel algorithms, shuffle-exchange network, VLSI.**

I. Introduction

ONE of the difficulties of VLSI design is the magnitude of the task. It is not easy to lay out one hundred thousand transistors, let alone ten million of them. Yet there is a sense in which the scale of VLSI is advantageous. The complexity of a VLSI chip is so great that asymptotic approximations can give insight into performance evaluation and design.

This paper shows how asymptotic analysis can aid in the design of Fourier transform circuits in VLSI. Approaching chip design in this way has three advantages. First of all, the analysis is simple: the calculations are easy to perform and thus easy to believe. Second, the analysis points out the bottlenecks in a design, indicating the portions that should be optimized. It is impossible to "miss the forest for the trees" when one is thinking of asymptotic performance.

A third advantage of the analytic approach is that it provides a simple framework for the evaluation and explanation of various designs. Limits on area–time performance have been proved for a number of important problems, including sorting, matrix multiplication, and integer multiplication [1]–[6]. In the case of the central problem of this paper, the N-element Fourier transform, it has been shown that no circuit can have a better area $*$ time^2 performance than $\Omega(N^2 \log^2 N)$[1] [7].

For the purpose of this paper, "time performance" is defined as the number of clock cycles in the interval between successive Fourier transformations by a possibly pipelined circuit. Note that it is a circuit's throughput, not its delay, that is measured by this definition of "time performance." However, in view of the importance of circuit delay in many applications, delay figures will be presented for all the Fourier transform-solving circuits in this paper. The optimality of any particular delay figure can be judged by how close the circuit comes to the limiting area $*$ delay^2 performance of $\Omega(N^2 \log^2 N)$ [5].

No matter how one defines a circuit's time performance, it is important to make explicit I/O assumptions when quoting or proving lower bound results. Vuillemin's implicit convention [7], followed here, is that successive problem inputs and outputs must enter and leave the circuit on the same pins. For example, pin 1 might be used to carry the most significant bit of input 1 for all problems. This is a natural assumption for a pipelined circuit; a more flexible form of parallelism would allow different problem instances to enter the circuit on different I/O lines. (The area $*$ delay^2 performance limit [5] quoted above relies on the assumption that all the bit of an input word enter the circuit through the same pin, or at least through a localized set of pins. However, Vuillemin's proof technique makes it clear that the delay bound also applies under the I/O assumptions of this paper.) Readers interested in a further discussion of the way I/O assumptions can affect lower and upper bounds on circuit performance are referred to [6].

The fact that there is a theoretical limit to area $*$ time^2 performance suggests that designs be evaluated in terms of how closely they approach this limit. Any design that achieves this limit must be optimal in some sense and thus deserves careful study. This paper presents a number of optimal and nearly optimal designs, corresponding to different tradeoffs of area for time. For example, in Section III, the "FFT network" takes only $O(\log N)$ time but $O(N^2)$ area to perform its Fourier transform. Thus it is a faster but larger circuit than, say, the "mesh implementation," which solves an N-element problem in $O(\sqrt{N})$ time and $O(N \log^2 N)$ area.

Section II of this paper develops a model for VLSI, laying the groundwork for the implementations and the analyses. The

Manuscript received December 8, 1980; revised September 17, 1982 and April 29, 1983. This work was supported in part by the National Science Foundation under Grant ECS-8110684, in part by the U.S. Army Research Office under Grant OAAGG29-78-G-0167, and in part by a Chevron U.S.A. Career Development grant.

The author is with the Division of Computer Science, University of California, Berkeley, CA 94720.

[1] The $\Omega(\,)$ notation means "grows at least as fast as": as N increases, the area $*$ time^2 product for these circuits is bounded from below by some constant times $N^2 \log^2 N$. In contrast, the more familiar $O(\,)$ notation is used exclusively for upper bounds, since it means "grows at most as fast as." For example, a circuit occupies area $A = O(N)$ if there is some constant c for which $A \leq cN$ for all but a finite number of problem sizes N. Finally, all logarithms in this paper are base two.

0018-9340/83/1100-1047$01.00 © 1983 IEEE

model is based on a small number of assumptions that are valid for any currently envisioned transistor-based technology. Thus the results apply equally well to the field-effect transistors of the MOS technologies (CMOS, HMOS, VMOS,...), to the bipolar transistors of I^2L, and to any GaAs process.

Section III describes nine implementations of Fourier transform-solving circuits in VLSI. Most of these circuits are highly parallel in nature. None of the circuits are original to the paper. However, only a few had been previously analyzed for their area-time performance.

Section IV concludes the paper with a summary of the performance figures of the designs.

II. The Model

Briefly, a VLSI circuit is modeled as a collection of nodes and wires. A node represents a transistor or a gate. A wire represents the conductor that carries signals from one node to another. In keeping with the planar nature of VLSI, restrictions are placed on the ways in which nodes and wires are allowed to overlap each other. Only one node and at most two wires can pass over any point in the plane.

The unit of time in the model is equal to the period of the system clock, if one exists. (Asynchronous circuits and "self-timed" circuits are discussed in the author's dissertation [5]; only the synchronous case is described here.) In particular, a wire can carry one bit of information in one unit of time. This bit is typically used to change the state of the node at the other end of the wire.

The unit of area in the model is determined by the "minimum feature width" of the processing technology. Wires have unit width and nodes occupy $O(1)$ area, that is, a node is some constant number of wire-widths on a side. The area of a node also includes an allowance for power and clock wires, which are not represented explicitly in the model.

Fan-out and wire-length restrictions are enforced by the model's timing and loading rules. These rules are predicated on the assumption that loads are capacitive in nature and proportional to wirelength and fan-out [8, p. 315]. Under this assumption, an $O(k)$-area circuit can drive a wire of length k with only $O(\log k)$ delay [8, p. 13]. Fig. 1 shows such a driver circuit for the case that $k = 3$. Each stage of the driver is twice the size of the previous one, so its output can drive twice the wirelength or twice the fanout. (The final stage of a driver is about 10 percent of the area of the circuit it drives, in a typical NMOS design.)

In Fig. 1, the size of a stage is indicated by the number of latches in that stage. This is somewhat contrary to current practice, in which each stage is a single device built of larger-than-normal transistors. Splitting stages into unit-sized pieces has two advantages, offsetting the obvious constant-factor disadvantages in size and speed. First, it results in a cleaner and simpler model. Second, and more importantly, it makes it clear that drivers can increase circuit area by at most a constant factor. The first 10 percent of the length of every long (but unit-width) wire can be replaced by its $O(1)$-width driver. Other wires may cross the "driver portion" of a long wire almost anywhere, in the gaps between its latches. If wires were

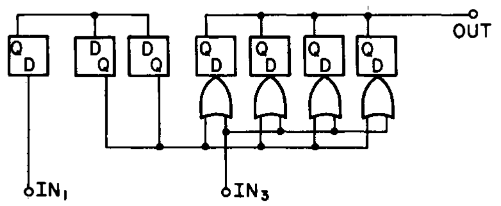

Fig. 1. A three-stage driver. The output of a k-stage driver at time t is given by the transmission function $\text{OUT}(t) = \text{IN}_1(t - k) \vee \text{IN}_k(t - 1)$. The high-power, low-delay input IN_k is used in the "mesh" construction of Section III-I.

not allowed to cross over drivers, quadratic increases in circuit area could be observed when long-wire drivers are included into an arbitrary layout [9].

This paper's assumption of capacitive loads and logarithmic wire delays seems adequate to model any of today's transistor-based technologies [10]. Technological "fixes" are available [11] to minimize the resistive effects and current limitations that degrade the performance of circuits with extremely long wires and/or large fan-outs [8, p. 230]. However, the model is invalid for any technology, such as IBM's superconducting Josephson junction logic [12], in which wire delays are influenced by speed-of-light considerations. Wire delay would then be linear in wire length, leading to entirely different asymptotic results [13].

The model is summarized in the list of assumptions below. A fuller explanation and defense of a similar model is contained in the author's Ph.D. dissertation [5]. A slightly different version of the model, with modified I/O assumptions, appears in [6].

Assumption 1—Embedding:

a) Wires are one unit wide.

b) At most two wires may cross over each other at any point in the plane.

c) Fan-out may be introduced at any point along a wire. In graph-theoretic terms, wires are thus hyperedges connecting two or more nodes.

d) A node occupies $O(1)$ area. Wires may not cross over nodes, nor can nodes cross over nodes.

e) A node has at most $O(1)$ input wires and $O(1)$ output wires. (In general, a node can implement any Boolean function that is computable by a constant number of TTL gates. Hence an "and gate" or a "j-k flip-flop" is represented by a single logic node: see Assumption 4.)

f) Unboundedly long wires and large fanouts are permissible under the following loading rule. A length-k wire may serve as the input wire for n gates only if it is connected to the outputs of at least $c_w k + c_g n$ identical gates with identical inputs. See, for example, the circuit in Fig. 1. The "loading constants" c_w, c_g are always less than one—otherwise it would be impossible to connect two gates together—but their precise values are technology-dependent.

Assumption 2—Total Area: The total area A of a collection of nodes and wires is the number of unit squares in the smallest enclosing rectangle. (A rectangular die is used to cut individual chips out of a silicon wafer: the larger the die, the fewer circuits per wafer. This paper's area measure is thus correlated with manufacturing costs.)

Assumption 3—Timing:

a) Wires have unit bandwidth. They carry a one-bit "signal" in each unit of time.

b) Nodes have $O(1)$ delay. (This assumption, while realistic, is theoretically redundant in view of Assumption 3a.)

Assumption 4—Transmission Functions:

a) The "state" of a node is a bit-vector that is updated every time unit according to some fixed function of the signals on its input wires. The signals appearing on the output wires of a node are some fixed function of its current "state." (With this definition, a node is seen to have the functionality of a finite state automation of the Moore variety.)

b) Nodes are limited to $O(1)$ bits of state.

Assumption 5—Problem Definition:

a) A problem instance is obtained by assigning one of M different values to each of N input variables. All M^N possible problem instances are equally likely: there is no correlation between the variables in a single problem instance, nor is there any correlation between the variables in different problem instances. (If successive problem instances were correlated, pipelined circuits might take advantage of this correlation, invalidating the lower bounds on achievable performance. It would of course be interesting to make a separate study of circuits that efficiently transform correlated data.)

b) N is an integral power of 2. This allows us to use the FFT algorithm in our circuits.

c) $\log M = \Theta(\log N)$: a word length of $\lceil \log M \rceil = c(\log_2 N)$ bits is necessary and sufficient to represent the value of an input variable, using any of the usual binary encoding schemes: one's complement, two's complement, etc. (This assumption allows us to suppress the parameter M in our upper and lower bounds. Binary encoding is required in the lower bound proof [7]; it should be possible to remove this restriction. The restriction on word length seems to be congruent with normal usage of the Fourier transform: in practice, c seems to be a small constant greater than 1.)

d) The output variables $\mathbf{y}$ are related to the input variables $\mathbf{x}$ by the equation $\mathbf{y} = C\mathbf{x}$. The (i, j)th entry of C has the value w^{ij}, where w is a principal Nth root of unity in the ring of multiplication and addition mod M. (This assumption defines a number-theoretic transform; results for the more common Fourier transform over the field of complex numbers are analogous.)

Assumption 6—Input Registers:

a) Each of the N input variables is associated with one input register formed of a chain of $\lceil \log M \rceil$ logic nodes. In other words, input register i corresponds to input variable x_i, $0 \le i \le N - 1$.

b) Each input register receives the value of its variable once, at the beginning of each computation; each input value is sent to exactly one input register. (This paper's model is thus "when- and where-oblivious" [3], since the times and locations of input events are not data-dependent. The model is also "semelective and unilocal" [14] since each input value is read at a single time and location. Less restricted I/O models are considered in [6].)

Assumption 7—Output Registers:

a) Each of the N output variables is associated with one output register formed of a chain of $\lceil \log M \rceil$ logic nodes.

b) A computation is complete when the correct value of each output variable is encoded in the state of the nodes in its output register. Presumably, some other circuit will make use of these output values at this time.

Assumption 8—Pipelined Time Performance: A collection of nodes and wires operates in "pipelined time T_p" if it completes computations at an average rate of one every T_p time units. The time bounds of this paper thus measure the period or throughput, and not the delay, of Fourier transform circuits. (However, delays are considered in Section IV's comparison of the nine designs.)

III. The Implementations

All of the Fourier transform circuits of this paper are built from a few basic building blocks: shift registers, multiply-add cells, random-access memories, and processors. These are described in the following.

A k-bit shift register can be built from a string of k logic nodes in $O(k)$ area. Each of the logic nodes stores one bit. Shift registers are used to store the values of variables and constants; these values may be accessed in bit-serial fashion, one bit per time unit.

Multiply-add cells are used to perform the arithmetic operations in a Fourier transform. Each cell has three bit-serial inputs w^k, x_0, and x_1. It produces two bit-serial outputs

$$y_0 = x_0 + w^k x_1 \text{ and } y_1 = x_0 - w^k x_1. \tag{1}$$

The inputs and the outputs are all $\lceil \log M \rceil = \Theta(\log N)$ bit integers.

It is fairly easy to see that a simple (if slow) multiply-add cell can be built from $O(\log N)$ logic gates [5]. The multiplication is performed by $O(\log N)$ steps of addition in a carry-save adder. Each addition takes $O(1)$ time, if carries are only propagated by one bit-position per addition. The result of the multiplication is available $O(\log N)$ time after the last partial product enters the adder, when the carry out of the least-significant bit position has had a chance to propagate to the most-significant position. The subsequent addition and subtraction can also be done in $O(\log N)$ time. Thus a complete multiply-add computation can be done in $O(\log N)$ time and $O(\log N)$ area.

The aspect ratios of the multiply-add cell and shift register may be adjusted at will. They should be designed as a rectangle of $O(1)$ width that can be folded into any rectangular shape.

An S-bit random-access memory with a cycle time of $O(\log S)$ can be built in $O(S)$ area, using the techniques of Mead and Rem [8, pp. 317–321]. (Their area and time analyses are essentially consistent with the model of this paper; see [5] for a comparative study of the two models.) The cycle time claimed above is the best possible, given the logarithmic wire delays implied by Assumption 1f, since most of the storage locations are at least $\sqrt{S}$ wire-widths from the output port of the memory. To achieve this optimal cycle time, the number of levels in Mead and Rem's hierarchical memory must grow proportionally with $\log S$.

Processors are used to generate control signals, whenever these become complex. Each processor is a simple von Neumann computer equipped with an $O(\log N)$-bit wide ALU, $O(\log N)$ registers, and a control store with $O(\log N)$ instructions. The cycle time of a processor is $O(\log N)$ time units. This is enough time to fetch and execute a register-to-register move, a conditional branch, an "add," or even a "multiply" instruction. It is also enough time to allow the processor's operands to come from an N-bit random-access memory.

At least $O(\log^2 N)$ units of area are required to implement a processor, since it has $O(\log N)$ words and thus $O(\log^2 N)$ bits of storage. A straightforward, if tedious, argument can be made to show that $O(\log^2 N)$ area is actually sufficient to build a processor [5]. Neither the ALU, the data paths, nor the instruction decoding circuitry will occupy more room (asymptotically) than the control store.

A. The Direct Fourier Transform on One Multiply-Add Cell

The naive or "direct" algorithm for computing the Fourier transform is to compute all terms in the matrix-vector product of Assumption 5d. Following this scheme, a total of N^2 multiplications are required when an N-element input vector $\mathbf{x}$ is multiplied by an $N \times N$ matrix of constants C, to yield an N-element output vector $\mathbf{y}$. Three degrees of parallelism immediately suggest themselves: the product may be calculated on one multiply-add cell, on N multiply-add cells, or on N^2 multiply-add cells. Each possibility is discussed separately in the discussion that follows.

A single multiply-add cell will take $O(N^2 \log N)$ time to perform all the calculations required in the direct Fourier transform algorithm. (Recall that a multiply-add calculation takes $O(\log N)$ time.) To this must be added the overhead of calculating the constants in the matrix C, since a prohibitively large amount of area would be required to store these explicitly. Fortunately, this calculation is quite simple. The constant required during the ijth multiply-add step (see statement 4 of Fig. 2) can generally be obtained by multiplying w^i by the constant used in the previous multiply-add step, $w^{i(j-1)}$. A single processor is capable of performing this calculation, supplying the necessary constants to the multiply-add cell as rapidly as they are needed. The time performance of the uniprocessor DFT design is thus $O(N^2 \log N)$.

The area required by the single multiply-add cell design is $O(\log N)$ for the multiply-add cell, $O(\log^2 N)$ for the processor supplying the constants, and $O(N \log N)$ for the random-access memory containing the input and output registers. This last contribution clearly dominates the others, giving the uniprocessor DFT design a total area of $O(N \log N)$. Its combined area $*$ time^2 performance is thus a dismal $O(N^5 \log^3 N)$. It has far too little parallelism for its area. The designs in the next two sections employ progressively more parallelism to achieve better performance figures.

B. The Direct Fourier Transform on N Cells

Kung and Leiserson [8, pp. 289–291] were apparently the first to suggest that the Fourier transform could be computed

```
1. FOR $i \leftarrow 0$ TO $N-1$ DO
2.    $y_i \leftarrow 0$;
3.    FOR $j \leftarrow 0$ TO $N-1$ DO
4.       $y_i \leftarrow y_i + w^{ij}x_j$;
5.    OD;
6. OD.
```

Fig. 2. The naive or "direct" Fourier transform algorithm.

by the "direct" algorithm on $2N - 1$ multiply-add cells connected in a linear array. These cells operate with a 50 percent duty cycle: the even-numbered cells and the odd-numbered cells alternately perform the computational step described below. An obvious optimization [8, p. 275] results in a circuit using only N multiply-add cells to accumulate the terms in the DFT.

The entire DFT calculation is complete in $4N - 3$ computational steps. During each step in which it is active, each even- (or odd-) numbered cell computes $y' \leftarrow y + cx$ using the value y provided by its right-hand neighbor (the leftmost cell always uses $y = 0$). The y' values eventually emerging from the leftmost cell are the outputs $\mathbf{y}$ in natural order. The inputs $\mathbf{x}$ to the circuit enter through the leftmost cell and are passed, unchanged, down the line of cells. Due to the 50 percent duty cycle of the cells, one y' value is produced (and one x value is consumed) every other computational step.

The only complicated part of the circuit has to do with computing the constant values c. A complete description of this computation is rather lengthy [8, pp. 290–291]; only a sketch is attempted here. Suffice it to say that each c value is obtained by a single multiplication from the c value previously used by the cell next closest to the center of the linear array. The only exception to this rule is that the constant-generating circuitry for the centermost cell must perform four multiplications to obtain the next c value. (Perhaps a fast multiplier might be provided for the centermost cell, to keep it from slowing down the whole array.) In any event, the constant-generating circuitry for each cell performs a fixed sequence of register–register operations, all of which can be completed in $O(\log N)$ time and $O(\log N)$ area.

The time performance of the N-cell DFT design is $O(N \log N)$, since each of the $4N - 3$ computational steps can be completed in $O(\log N)$ time. The total area of the N cells and their constant-generating circuitry is $O(N \log N)$.

Note that the total area of the N-cell DFT design is asymptotically identical to that of the 1-cell design. This is a reflection of the fact that a register takes the same amount of room (to within a constant factor) as a multiply-add cell. However, one can confidently expect that an actual implementation of the 1-cell design will be significantly smaller than an N-cell design due to this "constant factor difference." Section IV contains a further discussion of the significance of constant factors in the interpretation of the asymptotic results of this paper.

The area $*$ time^2 performance of the N-cell DFT design is $O(N^3 \log^3 N)$. This is far from optimal, but it is a great improvement on the 1-cell design. The next section describes an N^2-cell design that has a nearly optimal area $*$ time^2 perfor-

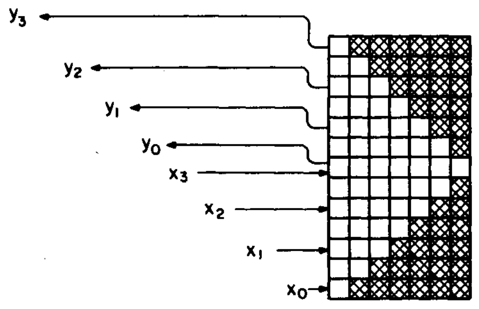

Fig. 3. Staggered I/O pattern for the N^2-cell DFT design.

mance figure.

C. The Direct Fourier Transform on N^2 Cells

One way of boosting the efficiency of the N-cell DFT design is to pipeline its computation. Instead of circulating intermediate values among one row of $2N - 1$ cells for $4N - 3$ steps, one can "unroll" the computation onto $4N - 3$ rows of $2N - 1$ cells. Now each problem instance spends just one computational step on each row of cells before moving on to the next row. (Note that there are actually about $8N^2$ cells in the "N^2-cell" design.)

All I/O occurs through the leftmost cell in the odd-numbered rows, in the staggered order shown in Fig. 3. This figure shows only the I/O for a single problem instance; inputs for successive problem instances may follow immediately behind the analogous inputs for the previous problem, after a delay of one computational step.

More precisely, the first input for each problem instance enters the leftmost cell of the first row. The second input enters the leftmost cell of the third row, two computational steps later (remember that each computational step, as defined in the previous section, involves only "even" or "odd" cells). The Nth input enters the leftmost cell of the $(2N - 1)$th row, $2N - 2$ computational steps after the first input entered the circuit. At the end of this step, the first output is available from this same cell. The second output comes from the leftmost cell of the $(2N + 1)$th row, after two more steps . . . and finally the Nth output emerges from the leftmost cell of the $(4N - 3)$th row, $(4N - 3)$ computational steps after the first input was injected into the circuit.

As noted above, the kth input for another problem instance can follow immediately behind the kth input for the previous problem, delayed by only one computational step. The circuit thus operates in pipelined time $T_p = O(\log N)$. The total area of the N^2-cell design is $A = O(N^2 \log N)$, since each cell occupies $O(\log N)$ area. The combined area $*$ time^2 performance of the design is only a factor of $O(\log N)$ from the optimal figure of $\Omega(N^2 \log^2 N)$. Thus it is pointless to look for a smaller circuit with a similar pipelined time performance. However, it is possible to make great improvements on this circuit's solution delay, as shown by the $(N \log N)$-cell FFT design presented later in this paper.

It is fairly easy to describe a few "constant factor" improvements to the N^2-cell DFT design. First of all, at least half of the cells on each row are idle, due to the 50 percent duty cycle inherent in the Kung–Leiserson approach. Secondly, the computations done in the hatched portion of Fig. 3 are irrelevant (the resulting y' values do not affect the circuit's outputs). Each of these considerations halves the number of required multiply-add cells, leaving fewer than $2N^2$ cells in an optimized design. Finally, the constant-generating circuitry described for the N-cell design need not be carried over to the N^2-cell design, for each cell uses the same a value every time it does a computational step. In other words, the constant matrix C can be "hard-wired" into the registers of the multiply-add cells.

An alternative design for a Fourier transform circuit with about N^2 cells may be derived from Kung and Leiserson's matrix–matrix multiplier [8, p. 277]. The inputs to the Kung–Leiserson multiplier are the constant matrix C and the matrix formed by concatenating N different problem input vectors $\mathbf{x}$; the outputs are N problem output vectors $\mathbf{y}$. The alternative design has the advantage of using about $4N^2$ multiply-add cells at a 33 percent duty cycle, that is, there are only $1.33N^2$ fully utilized cells in an optimized design. On the other hand, the alternative design has the disadvantage that it cannot continuously accept input problems: "gaps" of $2N - 1$ time units must intervene between "bursts" of N problem instances. Furthermore, additional wires are required in the alternative design to circulate the constant matrix C among the multiply-add cells.

D. The Fast Fourier Transform on One Processor

Up to now, all the circuits in this paper have computed the Fourier transform by the naive or direct algorithm. Great increases in efficiency are observed in conventional uniprocessors using the fast Fourier transform algorithm; it would be remarkable indeed if we could not take advantage of our knowledge of the FFT in the design of Fourier transform circuits.

There are a number of versions of the FFT in the literature, differing chiefly in the order in which they use inputs, outputs, and constants. Fig. 4 shows a "decimation in time" algorithm, taken from Fig. 5 of [15]. Fig. 5 shows a "decimation in frequency" algorithm, adapted from Fig. 10 of [15]. In both cases, the N problem inputs are stored in $\mathbf{x}$, the N problem outputs are $\mathbf{y}$, and w is a principal Nth root of unity.

Either Fig. 4 or 5 may be used as an algorithm for a uniprocessor that runs in $O(N \log N)$ computational steps. The total area of such a design is $O(N \log N)$, due mostly to input and output storage. (Recall that a single processor fits in $O(\log^2 N)$ area.) Total time for an N-element FFT is $O(N \log^2 N)$, since each computational step takes $O(\log N)$ time units. This is, as expected, a vast improvement over the uniprocessor DFT circuit. However, it is far from being area $*$ time^2 optimal, for its processor/memory ratio is too high. Adding more processors, as in the following design, increases the performance of an FFT circuit.

E. The Cascade Implementation of the Fast Fourier Transform

The cascade arrangement of $\log N$ multiply-add cells [16] is nicely suited for the computation of the Fourier transform

```
1.  FOR $b \leftarrow (\log N) - 1$ TO 0 BY $-1$ DO
2.      $p \leftarrow 2^b$; $q \leftarrow N/p$; /* note that $N = pq$ */;
3.      $z \leftarrow w^p$; /* $z$ is a principal $q$th root of unity */;
4.      FOR $i \leftarrow 0$ TO $N - 1$ DO
5.          $j \leftarrow i \bmod q$; $k \leftarrow reverse(i)$;
6.          IF ($k \bmod p$) = ($k \bmod 2p$) THEN
7.              $\langle x_k, x_{k+p} \rangle \leftarrow \langle x_k + z^j x_{k+p}, x_k - z^j x_{k+p} \rangle$;
8.          FI;
9.      OD;
10. OD;
11. FOR $i \leftarrow 0$ TO $N - 1$ DO /* unscramble outputs */;
12.     $y_{reverse(i)} \leftarrow x_i$;
13. OD.
```

Fig. 4. The FFT by "decimation in time." Note: *reverse*(i) interprets i as an unsigned ($\log N$)-bit binary integer, then outputs that integer with its bits reversed, i.e., with its most-significant bit in the least-significant position.

using the decimation in frequency algorithm. See Fig. 5 for the algorithm and Fig. 6 for a diagram of the cascade arrangement.

In a cascade, one of the outputs of each multiply-add cell is connected to the input of a shift register of an appropriate length. The shift register's output is connected to one of the multiply-add cell's inputs, forming a feedback loop. The remaining inputs and outputs of the multiply-add cells are used to connect them into a linear array. Problem inputs (values of $\mathbf{x}$) are fed into the leftmost cell; problem outputs (values of $\mathbf{y}$) emerge from the rightmost cell.

Each cell handles the computations associated with a single value of the loop index b in Fig. 5. The leftmost cell performs the loop for $b = \log N - 1$; the rightmost cell performs the loop computations for $b = 0$. The pairing of x values indicated in statement 7 of Fig. 5 is accomplished by the 2^b-word shift register associated with cell b.

The attentive reader will note that statement 7 is not exactly the same as the multiply-add step defined in (1). Statement 7 involves one constant value z^j, two variable values x_i and x_{i+n}, two additions, but two (instead of one) multiplications. Thus its computation will take about twice as much time or area as a "standard" multiply-add step.

The conditional test of statement 6 is implemented by having each cell monitor the bth bit of the count i of input elements that it has already processed. The condition of statement 6 is satisfied whenever that bit is 0. In this case, a cell performs the computation indicated in statement 7. It sends the new value for x_i to the right, and retains the new value for x_{i+p} in its shift register. Whenever the bth bit of i is 1, no multiply-add computations are performed. However, some data movement is necessary: the data appearing on the cell's lower input line should be copied into its shift register. Also, the values emerging from its shift register should be sent on to the next cell on its right.

```
1.  FOR $b \leftarrow (\log N) - 1$ TO 0 BY $-1$ DO
2.      $p \leftarrow 2^b$; $q \leftarrow N/p$;
3.      $z \leftarrow w^{q/2}$; /* $z$ is a principal $2p$th root of unity */;
4.      FOR $i \leftarrow 0$ TO $N - 1$ DO
5.          $j \leftarrow i \bmod p$;
6.          IF ($i \bmod 2p$) = $j$ THEN
7.              $\langle x_i, x_{i+p} \rangle \leftarrow \langle x_i + x_{i+p}, z^j x_i - z^j x_{i+p} \rangle$;
8.          FI;
9.      OD;
10. OD;
11. FOR $i \leftarrow 0$ TO $N - 1$ DO /* unscramble outputs */;
12.     $y_{reverse(i)} \leftarrow x_i$;
13. OD.
```

Fig. 5. The FFT by "decimation in frequency."

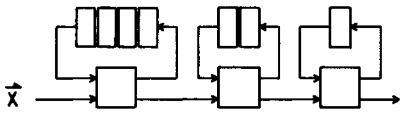

Fig. 6. The cascade arrangement of three multiply-add cells, for computing eight-element FFT's. The multiply-add cells are square; the rectangular boxes each represent one word of shift register storage.

One of the advantages of using the decimation in frequency algorithm on the cascade is the ease of computing the constants for its multiply-add steps. Only a few registers and a single multiplier are required to generate the constants required by each cell. Referring again to the program of Fig. 5, the constant z^j required in statement 7 may be obtained by multiplying the previously generated constant z^{j-1} by z. If this multiplication is performed whether or not statement 7 is executed, no conditional transfers are necessary in the constant-generating circuitry.[2]

As noted above, the constant-generating circuitry for each cell consists of a multiplier and a few registers. It is thus comparable in area and time complexity to the multiply-add cell itself. Thus the total area of the cascade design is obtained by multiplying the number of cells, $\log N$, by the area per cell $O(\log N)$. To this must be added the area of the shift registers. Unfortunately, there is a total of $N - 1$ words of storage in these registers, so the entire design occupies $O(N \log N)$ area. Thus the cascade, like the one-processor design, is almost all memory. An entire problem instance must be stored in the circuit while the Fourier transform is in progress.

The time performance of the cascade is somewhat improved over the one-processor design. Input values enter the leftmost processor at the rate of one per multiply-add step. An entire problem instance is thus loaded in $O(N \log N)$ time units. It is easy to see that the cascade can start processing a new problem instance as soon as the previous one has been completely loaded, so its pipelined time performance is $T_p = O(N \log N)$.

One awkward feature of the cascade is that it produces its output values in bit-reversed order. Formally, their order is derived from the natural left-to-right indexing (0 to $N - 1$) by reversing the bits in each index value, so that the least significant bit is interpreted as the most significant bit. The last few lines of Fig. 5 perform this bit-reversal, but they cannot be performed on the circuit described thus far. If natural ordering is desired, a processor should be attached to the output end of the cascade. If this processor has N words of RAM

[2] Note that $z^i = z^j$ whenever the bth bit of i is 0, since z is a $2p$th root of unity. Of course, exact equality obtains only when exact arithmetic is employed. This is easy to arrange in a number-theoretic transform. When roundoff errors cannot be avoided, for example in a complex-valued transform, it is probably best to use a conditional transfer to reset z^j to 1 whenever $j = 0$.

storage, a simple algorithm will allow it to reorder the outputs of the cascade as rapidly as they are produced.

F. The FFT Network

One of the most obvious ways of implementing the FFT in hardware is to provide one multiply-add cell for each execution of statement 7 in the algorithm of Fig. 4. (The algorithm of Fig. 5 might also be used, but, as noted in the previous section, its multiply-add computation is a little more complex.) Each cell is provided with a register holding its particular value of z^j. Since statement 7 is executed $N/2 \log N$ times, a total of $N/2 \log N$ multiply-add cells are required for this "full parallelization" of the FFT. Such a circuit is called an FFT network in this paper.

One possible layout for the cells in an FFT network is to have $\log N$ rows of $N/2$ cells each, as shown in Fig. 7. (This diagram was adapted from Fig. 5 of [15].) Each row of cells in the FFT network corresponds to an entire iteration of the "FOR b" loop of the algorithm of Fig. 4. The interconnections between the rows are defined by the way that the array $\mathbf{x}$ is accessed. The reader is invited to check that each multiply-add cell in Fig. 7 corresponds to an execution of statement 7 in Fig. 4 for the case $N = 8$.

Note that the inputs to the FFT network are in "bit-shuffled" order and its outputs are in "bit-reversed" order. This seems to minimize the amount of area required for interconnecting the rows. Additional wiring may of course be added to place inputs and outputs in their natural, left-to-right order.

The interconnections of Fig. 7 may be obtained from the following general scheme. Number the cells naturally: from 0 to $N/2 - 1$, from left to right. Then cell i in the first row is connected to two cells in the second row: cell i and cell $(i + N/4) \bmod N/2$. Cell i in the second row is connected to cells i and $\lfloor i/(N/4) \rfloor + ((i + N/8) \bmod N/4)$ in the third row. Cell i in the kth row (where $k = 1, 2, \cdots, \log N - 1$) is connected to two cells in the $(k + 1)$th row: cell i and cell $\lfloor i/(N/2^k) \rfloor + ((i + N/2^{k+1}) \bmod N/2^k)$. Another way of describing this "butterfly" interconnection pattern is to say that a cell on the kth row connects to the two cells on the next row whose indexes differ at most in their kth most significant bit. (The interconnections between rows in an FFT network can also be laid out in the "perfect shuffle" pattern described in the next section. However, this seems to lead to a larger layout, if only by a constant factor.)

A careful study of Fig. 7 and the preceding paragraph should convince the reader that $N/2$ horizontal tracks are necessary and sufficient for laying out the interconnections between the first two rows. Essentially, each cell in the first row has one "long" output wire that must cross the vertical midline of the diagram. This connection must be assigned a unique horizontal track to cross the midline. Once this is done, the rest of the wiring for that row is trivial, especially if the cells are "staggered" slightly as in Fig. 7.

The connections between the second and third rows occupy only $N/4$ horizontal tracks. No wires cross the vertical midline of the diagram, but each of the $N/4$ cells on either side of the midline have a fairly long connection that takes up to half of a horizontal track.

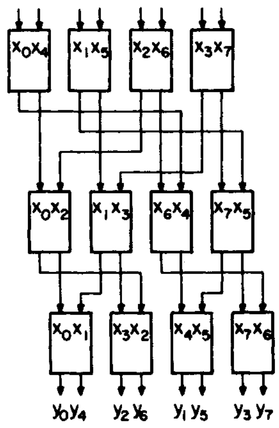

Fig. 7. The FFT network for $N = 8$.

In general, the connections emerging from the kth row ($k = 0, 1, \cdots, \log N - 1$) occupy $N/2^{k+1}$ tracks. Straight vertical wires are used to connect cell i in the kth row with cell i in the $(k + 1)$th row. The horizontal tracks are divided into 2^k equally sized pieces, then individually assigned to the "long" connection from each cell.

Following the scheme outlined above, a total of $N - 1$ horizontal tracks are required to lay out the interrow connections. An additional N horizontal tracks could be added above and below the FFT network to bring its inputs and outputs into natural order.

The number of vertical tracks in an FFT network depends strongly upon the width of the multiply-add cells. If these are set on end, so that each is $O(1)$ units tall and $O(\log N)$ units wide, then the entire network will fit into a rectangular region that is $O(N)$ units wide and $O(N)$ units tall. The height of the $\log N$ rows of multiply-add cells is asymptotically negligible.

The pipelined time performance of the FFT network is clearly $O(\log N)$ since a new problem instance can enter the network as soon as the previous one has left the first row of multiply-add cells. The delay imposed by each row's multiply-add computation and long-wire drivers is $O(\log N)$, and there are $O(\log N)$ rows, so the total delay of the network is $O(\log^2 N)$.

Note that this paper's layout of the FFT network must be optimal, for the circuit has an optimal area $*$ time^2 performance of $O(N^2 \log^2 N)$. Any asymptotic improvement in the layout area would amount to a disproof of Vuillemin's optimality result [7].

G. The Perfect-Shuffle Implementation of the FFT

Over a decade ago, Stone [17] noted that the "perfect shuffle" interconnection pattern of $N/2$ multiply-add cells is perfectly suited for an FFT computation by decimation in time. Fig. 8 shows the perfect shuffle network for the eight-element FFT, and Fig. 4 shows the appropriate version of the FFT algorithm.

Each multiply-add cell in a perfect shuffle network is associated with two input values, x_k and x_{k+1}. Here, k is an even

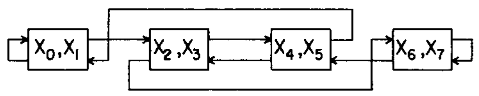

Fig. 8. The perfect shuffle interconnections for $N = 8$.

number in the range $0 \leq k < N - 1$. A connection is provided from one of the outputs of the cell containing x_k to one of the inputs of the cell containing x_j if and only if $j = 2k \bmod N - 1$. Note that this mapping of output indexes onto input indexes is one-to-one, and that it corresponds to an "end-around left shift" of the ($\log N$)-bit binary representation of k.

The computation of the FFT on the perfect shuffle network can now be described. First, the input values x_k are loaded into their respective multiply-add cells. Then a multiply-add step is performed: each cell ships its original x_k values out over its output lines, and computes new x_k values according to (1). It is not very obvious, but nonetheless it is true, that this corresponds to an entire iteration of the "FOR b" loop of Fig. 4. For example, the leftmost cell of Fig. 7 computes new values for x_0 and x_4, having received the original value of the former from its own output line and the original value of the latter from the third cell. This is the computation required by step 7 of Fig. 4 when $N = 8, b = 2, p = 4, q = 2, i = 0, j = 0$, and $k = 0$.

The FFT computation proceeds in this fashion for $\log N$ parallel multiply-add steps. In each step, the cell containing the (updated) version of x_k ships this value to the cell formerly containing the (updated) version of $x_{2k \bmod N-1}$. Each cell then performs a multiply-add computation, updating the two data values currently in its possession.

At the end of $\log N$ parallel multiply-add steps, each cell contains the final versions of its original data values. Unfortunately, the FFT computation of Fig. 4 is not complete. The outputs $\mathbf{y}$ are all available among the final $\mathbf{x}$ values, but they appear in "bit-reversed" order. Additional circuitry is required to bring them into natural order, following steps 11–13 of Fig. 4. The techniques of [18] could be employed in the design of reordering circuitry that would operate in $O(\log^2 N)$ time, without affecting the area performance of the perfect shuffle network. A detailed description of such circuitry is beyond the scope of this paper, since Assumption 7 does not require the circuit to produce its $\mathbf{y}$ values in any particular order.

The $\log N$ multiply-add steps require a total of $O(\log^2 N)$ time. The data movement involved in each multiply-add step does not require any additional time, at least in an asymptotic sense. As will be seen below, the "shuffle" connections between cells are implemented as single wires carrying bit-serial data. Each wire is less than $O(N)$ units long, and each work has $O(\log N)$ bits, so that the data transmission time per step is the same as the multiplication time, $O(\log N)$ time units.

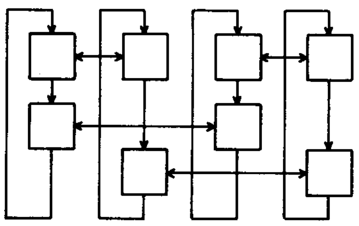

Fig. 9. The CCC network for $N = 4$.

The total area of the perfect shuffle implementation is a bit harder to estimate. There are $N/2$ multiply-add cells, each occupying $O(\log N)$ area. However, the best embedding known for the shuffle interconnections takes up $O(N^2/\log^2 N)$ area [19]. It is easy to see that no better embedding is possible, since otherwise the perfect shuffle circuit would have an impossibly good area $*$ time^2 performance.

H. The CCC Network

The cube-connected-cycles (CCC) interconnection for N cells is capable of performing an N-element FFT in $O(\log N)$ multiply-add steps [20]. Using the multiply-add cell of the previous constructions, the complete FFT takes $O(\log^2 N)$ time.

The CCC network is very closely related to the FFT network. In fact, a CCC network is just an FFT network with "end-around" connections between the first and last rows. For this reason, CCC networks do not exist for all N, only for those N of the form $(K/2) * (\log K)$ for some integer K. Fig 9 illustrates the CCC network for $N = 4$. It is derived from the four-element FFT network with "split cells": each cell handles one element of the input vector $\mathbf{x}$, instead of two as in the FFT network of Fig. 7. (The reader is invited to redraw Fig. 9, combining the cells linked by horizontal data paths. The resulting graph should be isomorphic to a "butterfly" whose outputs have been fed back into its inputs.)

The CCC network is somewhat smaller than the FFT network, since it uses only N cells to solve an N-element problem instead of the FFT network's $(N/2) * (\log N)$ cells. Furthermore, the CCC's interconnections can be embedded in only $O(N^2/\log^2 N)$ area [20]. This is an optimal embedding, for the combined area $*$ time^2 performance is within a constant factor of the limit, $\Omega(N^2 \log^2 N)$.

It is rather difficult to describe the data routing pattern during the computation of a Fourier transform on a CCC, although the basic approach is similar to that taken on the perfect shuffle network. Each of the $\log N$ multiply-add steps is preceded and followed by a routing step. These routing steps take $O(\log N)$ time each, for they move $O(1)$ words over each intercellular connection. Thus the total time spent in routing data does not dominate the time spent on multiply-add computations.

I. The Mesh Implementation

A square mesh of N processors is shown in Fig. 10. It consists of approximately $\sqrt{N}$ rows of $\sqrt{N}$ processors each, fitted with word-parallel interconnections. It is thus essentially the ILLIAC IV architecture, with the difference that each processor in the mesh is capable of running its own program. (A closer approximation to the ILLIAC IV would have N multiply-add cells, each deriving control signals from a central processor.)

The total area of the mesh is $O(N \log^2 N)$, since there are N processors each of $O(\log^2 N)$ area. The processors should each be laid out with a square aspect ratio, so that the $O(\log$

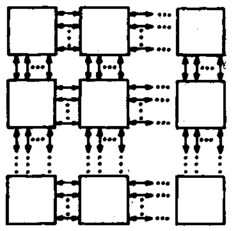
Fig. 10. The mesh of N processors; formed of $2^{\lfloor (\log N)/2 \rfloor}$ rows and $2^{\lceil (\log N)/2 \rceil}$ columns.

N) wires in each word-parallel data path do not add to the asymptotic area of the layout. Note that it takes $O(\log\log N)$ time to send a word of data from one processor to its neighbor, since analogous registers in adjacent processors are $O(\log N)$ units distant from each other.

Stevens [21] appears to have been the first to point out that the mesh can perform an N-element FFT in $\log N$ steps of computation. Each "step" consists of an entire iteration of the FOR b loop of Fig. 4. Each processor in the mesh performs the loop computation for one value of the index variable k. The total amount of data movement during the FFT can be minimized by making an appropriate assignment of index values k to individual mesh processors. It turns out that a fairly good choice is obtained from the natural row-major ordering (0 to $N - 1$) of the mesh. Processor k is then the "home" of the variable x_k.

(Another, more intuitive way of visualizing the computation of the FFT on the mesh is to view the latter as a time-multiplexed version of the FFT network. During each step, $N/2$ of the mesh's processors take on the role of the $N/2$ cells in one row of the FFT network. The wires connecting the rows of the FFT network are simulated by data movement among the processors of the mesh.)

An iteration of the FOR b loop of Fig. 4 can now be described. Each mesh processor examines the bth bit of *reverse*(k) to decide if it will perform the computation of statement 7. (For example, when $b = 0$ only the even-numbered processors will perform statement 7.) Next, each processor that will *not* perform statement 7 sends its current value of x_k to processor $k + 2^b$. (For example, when $b = 0$ each odd-numbered processor sends its x value to the processor on its left.) Statement 7 is then executed, and finally the updated x_k values are returned to their "home" processors.

When $b = \log N - 1$, the data movement required before statement 7 can be visualized by "sliding" all the x_k values in the bottom half of the mesh up to the top half of the mesh. In this way, processor 0 receives the current value of $x_{N/2}$, processor 1 receives the value of $x_{N/2+1}$, etc. This particular data movement will be called a "distance-$N/2$ route." In general, a distance-2^b route must be performed both before and after each execution of statement 7.

The time required by a distance-2^b route depends, of course, on the value of b. When $b = 0$ or $2^{\lceil (\log N)/2 \rceil}$, all data movement is between nearest neighbors (horizontal or vertical) in the mesh. As mentioned above, this takes only $O(\log\log N)$ time.

When $b = 2^{\lfloor (\log N)/2 \rfloor}$ or $\log N - 1$, it would seem that $O(\sqrt{N} \log\log N)$ time is required for a distance-2^b route. Each data element must ripple through about $\sqrt{N}/2$ processors. However, this result may be improved by using the "high-power" inputs on the long-wire drivers on the interprocessor data paths (see Fig. 1). Once the bits in a data element have been amplified enough to be sent to a neighboring processor, only one more stage of amplification is necessary to send these bits on to the next processor. Since the amplifier stages in a long-wire driver are individually clocked, all data elements in a routing operation may "slide" toward their destination simultaneously, moving by one processor-processor distance every time unit. The total time taken by a distance-2^b routing is thus easily seen to be $2^{b \bmod \lceil (\log N)/2 \rceil} + O(\log\log N)$.

The total time taken by all routings in a complete FFT computation is bounded by $O(\sqrt{N})$. Essentially, this is the sum of a geometric series whose largest term is the time taken by the longest routing operation, $O(\sqrt{N})$. The time performance of the mesh design is thus $O(\sqrt{N})$. At least asymptotically, the $O(\log^2 N)$ time required for the multiply-add computations is insignificant compared to the time required for the routing operations.

Three aspects of the mesh implementation deserve further attention. First of all, the individual processors are expected to come up with their own z^j values, as they execute statement 7 of Fig. 4. This is not difficult to arrange: each processor has $O(\log^2 N)$ bits of program storage, so it can easily perform a table look-up to obtain the required constants. One constant is needed for each processor, for each value of b.

Secondly, the algorithm described computes the y values in bit-reversed order (relative to the natural row-major ordering of the mesh). If the outputs are desired in natural order, another $O(\sqrt{N})$ routing operations are required [18], and the individual processors' programs become a bit more complicated.

One final note: the mesh implementation, as described, is area * time^2 optimal. A slightly less efficient, but possibly more practical design was suggested by one of the referees. Instead of using word-parallel buses between N processors in a mesh, one might provide bit-serial buses between N cells in a mesh. Now the best possible time performance is constrained by the bit-serial buses to be no better than $O(\sqrt{N} \log N)$. Similarly, the area could be reduced to as little as $O(N \log N)$. However, it will be a bit tricky to attain these performance figures. There is not enough area to store each cell's z^j values locally, so these values must be computed "on the fly" in (hopefully) only few extra multiplications. This seems to be impossible to accomplish directly. One solution to this difficulty is to have the cells exchange z^j values as well as x_k values. The bit-serial approach is thus inherently slower both in routing time and in the number of necessary multiplications. On the other hand, the word-parallel approach has wider buses and perhaps larger look-up tables, so that it takes up somewhat more area.

IV. Conclusion

The area and time performance of the nine implementations is summarized by Table I. Note that the last four designs are optimal in the area * time^2 sense. (Remember that $AT_p^2 = \Omega(N^2 \log^2 N)$ for the solution of the N-element Fourier

TABLE I
AREA-TIME PERFORMANCE OF THE FOURIER TRANSFORM-SOLVING CIRCUITS

	Design	Area	Time	Area$*$Time^2	Delay
III-A	1-cell DFT	$N \log N$	$N^2 \log N$	$N^5 \log^3 N$	$N^2 \log N$
III-B	N-cell DFT	$N \log N$	$N \log N$	$N^3 \log^3 N$	$N^2 \log N$
III-C	N^2-cell DFT	$N^2 \log N$	$\log N$	$N^2 \log^3 N$	$N^2 \log N$
III-D	1-proc FFT	$N \log N$	$N \log^2 N$	$N^3 \log^5 N$	$N \log^2 N$
III-E	Cascade	$N \log N$	$N \log N$	$N^3 \log^3 N$	$N \log^2 N$
III-F	FFT Network	N^2	$\log N$	$N^2 \log^2 N$	$\log^2 N$
III-G	Perfect Shuffle	$N^2/\log^2 N$	$\log^2 N$	$N^2 \log^2 N$	$\log^2 N$
III-H	CCC	$N^2/\log^2 N$	$\log^2 N$	$N^2 \log^2 N$	$\log^2 N$
III-I	Mesh	$N \log^2 N$	$\sqrt{N}$	$N^2 \log^2 N$	$\sqrt{N}$

transform.) In general, the problem with the nonoptimal designs is that they are processor-poor: the number of multiply-add cells does not grow quickly enough with problem size.

The mesh is the only design that is nearly optimal under any AT_p^{2x} metric for $0 \leq x \leq 1$. Here the limiting performance is $AT_p^{2x} = \Omega(N^{1+x} \log^{2x} N)$ [5]. None of the other designs with $O(N)$ or fewer multiply-add cells is fast enough, while the other designs are much too large.

When delay figures are taken into consideration, only the last three designs are seen to be optimal. The perfect shuffle, the CCC, and the mesh are the only designs that achieve the limiting area $*$ delay^2 product of $\Omega(N^2 \log^2 N)$ [5]. These designs keep all their multiply-add cells and wires busy solving Fourier transforms using the efficient FFT algorithm. All the others, save one, use too few processors or an inefficient algorithm. The FFT network is an interesting exception to this observation. Its delay inefficiency seems to be a result of its slow bit-serial multipliers. If fast parallel multipliers were employed, the delay in each stage of the FFT network might be as low as $O(\text{loglog } N)$. This would not increase its total area significantly, since its area is still dominated by its "butterfly" wiring. The improved FFT network could thus have a area $*$ time^2 product of as little as $O(N^2 \log^2 N \text{ loglog}^2 N)$.

Of course, asymptotic figures can hide significant differences among supposedly optimal designs due to "constant factors." The area and time estimates employed in this paper are not sensitive to the relative complexity of the various control circuits required in the designs. For example, the N^2-cell DFT, the cascade, the FFT network, and the perfect shuffle are especially attractive designs because they have no complicated routing steps. They are thus given a more detailed examination in the following.

As indicated in Table I, the N^2-cell DFT is nearly optimal in its area $*$ time^2 performance. However, it is by far the largest design considered in this paper since it uses more than N^2 multiply-add cells. (The others use $O(N \log N)$ or fewer cells.) Using current technology, one might place ten multiply-add cells on a chip [5]: this means that one hundred thousand chips would be needed for a thousand-element FFT! Thus the N^2-cell DFT design cannot be considered feasible until technology improves to the point that 100 or 1000 cells can be formed on a single wafer. Even then, the interconnections between chips will pose some difficulties, for there are 40 cells on the "edge" of a 100-cell chip.

The N-cell DFT is an attractive design at present, despite its nonoptimal area $*$ time^2 performance. It uses only $2N$ cells in a linear array, so that a 1000-element Fourier transform can be implemented with only 10^2 chips of 10 multiply-add cells each. This design is, of course, much slower than the N^2-cell DFT, since it produces only one element of a transform at a time rather than an entire transform.

The FFT network is also fairly attractive at present, for its $(N/2) * (\log N)$ cells can be formed on about the same number of chips as the N-cell DFT, yet its performance is equal to the N^2-cell DFT. The drawback of the FFT network is that the wiring on and between the chips is very area-consuming. It also has very long intercell wires, whereas the DFT designs use only nearest-neighbor connections.

The constant factor considerations of the perfect shuffle design are very similar to those of the FFT network discussed above. The perfect shuffle uses a factor of $\log N$ fewer cells than the FFT network, so it is a bit smaller and slower. However, it suffers from the same problem of long interchip wires and poor partitionability.

The cascade is another nonoptimal design, like the N-cell DFT, that deserves consideration because of its good "constant factors." It uses only $\log N$ multiply-add cells and N words of shift-register memory. These are arranged in a simple linear fashion. The cascade achieves the same performance as the N-cell DFT, producing one element of a Fourier transform during each multiply-add time. It is superior to the N-cell DFT in that it uses many fewer multiply-add cells.

It is interesting to speculate whether the cascade is the best way of producing one element of a Fourier transform at a time. A new metric and method of analysis is needed to answer this question, for such designs are necessarily nonoptimal. So much area is required to store the problem inputs and outputs that the optimal area $*$ time^2 figure can not be achieved.

Another interesting open problem is that of partitioning the perfect shuffle network. If 100 or 1000 multiply-add cells can be placed on a single chip, what sort of off-chip connections should be provided so that these chips can be composed into a large perfect shuffle design?

Finally, the implementations should be reevaluated under a speed-of-light (or transmission-line) model of wire delay. Under such a model, a length-k wire would have $O(k)$ delay, so all the delay figures in this paper must be reevaluated upwards. Since the wires could still have unit bandwidth, throughput figures would remain the same for all designs that could be sufficiently pipelined. However, some of the designs would be adversely affected. For example, the cycle time of the one-processor FFT design of Section III-D would increase to $O(\sqrt{N} \log N)$, since this is the distance to an arbitrary bit in its $O(N \log N)$-bit RAM.

ACKNOWLEDGMENT

The author gratefully acknowledges the insightful comments of two anonymous referees.

REFERENCES

[1] H. Abelson and P. Andreae, "Information transfer and area-time tradeoffs for VLSI multiplication," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 20–23, Jan. 1980.

[2] R. Brent and H. T. Kung, "The area-time complexity of binary multiplication," *J. Ass. Comput. Mach.*, vol. 28, pp. 521–534, July 1981.

[3] R. J. Lipton and R. Sedgewick, "Lower bounds for VLSI," in *Proc. 13th Annu. ACM Symp. on Theory of Computing*, May 1981, pp. 300–307.

[4] J. Savage, "Area-time tradeoffs for matrix multiplication and related problems in VLSI models," *J. Comput. Syst. Sci.*, vol. 22, pp. 230–242, Apr. 1981.

[5] C. D. Thompson, "A complexity theory for VLSI," Ph.D. dissertation, Carnegie-Mellon Univ., CMU-CS-80-140, Aug. 1980.

[6] ——, "The VLSI complexity of sorting," UCB/ERL M82/5, Feb. 1982.

[7] J. Vuillemin, "A combinatorial limit to the computing power of VLSI circuits," in *Proc. 21st Symp. Foundations of Comput. Sci.*, IEEE Comput. Soc., Oct. 1980, pp. 294–300.

[8] C. Mead and L. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 1980.

[9] V. Ramachandran, "On driving many long lines in a VLSI layout," in *Proc. 23rd Symp. Foundations of Comput. Sci.*, IEEE Comput. Soc., Oct. 1982, pp. 369–378.

[10] G. Bilardi, M. Pracchi, and F. P. Preparata, "A critique of network speed in VLSI models of computation," *IEEE J. Solid-State Circuits*, vol. SC-17, pp. 696–702, Aug. 1982.

[11] C. Mead and M. Rem, "Minimum propagation delays in VLSI," in *Proc. Caltech Conf. VLSI*, Caltech Dep. Comput. Sci., Jan. 1981, pp. 433–439.

[12] T. R. Gheewala, "Design of 2.5-micrometer Josephson current injection logic (CIL)," *IBM J. Res. Develop.*, vol. 24, pp. 130–142, Mar. 1980.

[13] B. Chazelle and L. Monier, "Towards more realistic models of computation for VLSI," in *Proc. 11th Annu. ACM Symp. Theory of Computing*, Apr. 1979, pp. 209–213.

[14] J. Savage, "Planar circuit complexity and the performance of VLSI algorithms," in *Proc. CMU Conf. VLSI*, IEEE Comput. Soc., Oct. 1981, pp. 61–68.

[15] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, Jr., D. Nelson, C. Rader, and P. Welch, "What is the fast Fourier transform?," *IEEE Trans. Audio Electroacoust.*, vol. AU-15, pp. 45–55, June 1967.

[16] A. Despain, "Very fast Fourier transform algorithms for hardware implementation," *IEEE Trans. Comput.*, vol. C-28, pp. 333–341, May 1979.

[17] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.

[18] C. D. Thompson, "Generalized connection networks for parallel processor intercommunication," *IEEE Trans. Comput.*, vol. C-27, pp. 1119–1125, Dec. 1978.

[19] F. T. Leighton, "Layouts for the shuffle-exchange graph and lower bound techniques for VLSI," Ph.D. dissertation, M.I.T., MIT/LCS/TR-724, June 1982.

[20] F. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," in *Proc. 20th Annu. Symp. Foundations of Comput. Sci.*, IEEE Comput. Soc., Oct. 1979, pp. 140–147.

[21] J. Stevens, "A fast Fourier transform subroutine for Illiac IV," Cen. Advanced Comput., Univ. Illinois, Tech. Rep., 1971.

Clark D. Thompson received the B.S. degree in chemistry and the M.S. degree in computer science/computer engineering from Stanford University, Stanford, CA, in 1975 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980.

He is presently on the faculty of the University of California, Berkeley, where he teaches courses on microprocessor interfacing and data structures. The goal of his current research is to apply theoretical insight to problems arising in VLSI design. He also maintains an interest in algorithms, models, and architectures for highly parallel computers.