MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 115

# Performance of an Elementary Data-Flow Processor*

by

David P. Misunas

*Edited transcription of a talk presented at the Second Annual Symposium on Computer Architecture, Houston, Texas, January 21-22; 1975.

February 1975

## Introduction

Efforts to develop a model of computation which can effectively express parallelism have yielded a new form of program representation, known as data flow [1, 2, 3, 4, 8, 9, 11]. The attractiveness of data flow lies in the fact that it is data-driven; that is, an instruction is enabled for execution only after each required operand has been provided by the execution of a predecessor instruction.

We have been conducting architectural studies to investigate the design of a processor which can efficiently execute data-flow programs by taking advantage of the parallelism inherent in the data-flow representation. The resulting architectures offer attractive solutions to some of the problems of parallel systems [5, 6]. The usual problems of processor switching and memory/processor interconnection are avoided by the use of interconnection networks which have a great deal of inherent parallelism. The structure of the processor allows a large number of instructions to be active simultaneously. These active instructions pass through the networks concurrently and form streams of instructions for each pipelined functional unit.

Initial investigations culminated in the development of an architecture for a processor that executes programs expressed in the elementary data-flow language [5]. The elementary language incorporates no fancy capabilities such as recursion, data structures, conditionals, or iteration. However, the language and its corresponding architecture are well-suited for the representation and performance of signal processing computations such as filtering, waveform generation, fast Fourier transforms, and so forth.

The next step involved developing the architecture described in the conference proceedings, the basic machine [6]. This machine and its corresponding language incorporate conditional and iterative mechanisms and a multi-level memory system in which the active memory is operated as a cache, and individual instructions are retrieved from the auxiliary memory as they become required for computation.

The design of the final machine in this series is nearing completion, expanding the architecture and language to incorporate procedures, recursive activation, and data structures of arbitrary size and shape. Also a forall construct is being implemented to permit parallel processing of the elements of a structure.

Today I want to say a few words about performance of the architectures. In order to do this, I am going to discuss the implementation on the elementary processor of a rather common computation, the fast Fourier transform.

## The Elementary Data-Flow Processor

The computational capability of the elementary data-flow processor is limited to programs expressed in the elementary data-flow language. A program in this language is constructed of two kinds of elements, called operators and links (Figure 1). Operators are represented as circles with a number of input arcs and one output arc. A link is designated by a small dot and receives results from an operator on its input arc and distributes them to other operators over its output arcs.

Tokens are represented by large solid dots and convey values over the arcs of the program. An operator with a token on each of its input arcs and no token on its output arc is enabled (Figure 2), and sometime later will fire, removing the tokens from its input arcs, computing a result using the values associated with the input tokens and associating that result with a token placed on its output arc. Similarly, a link is enabled when a token is present on its input arc and no token is present on any of its output arcs. It fires by removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs.

In Figure 3 we have a rather simple data-flow program. There is a value present on each input arc and thus links L1 and L2 are enabled. Either one can fire -- suppose L1 does. Then operator A2, which multiplies its input by the constant A, and link L2 are enabled. Once again, either can fire and in this manner tokens travel through the program until a token appears on the output conveying the value $Ax(x + y)$. Once operators A1 and A2 have fired, there are no tokens on the arcs emanating from L1 and L2, and the links can fire as soon as two new input values arrive. Thus, these elementary programs can easily represent pipelined computation.

The Memory of the processor holds a representation of the program to be executed. This Memory is a collection of Cells (Figure 4); one Cell must be associated with each operator of the program. Each Cell contains three registers, one to hold an instruction which encodes the type of operator and

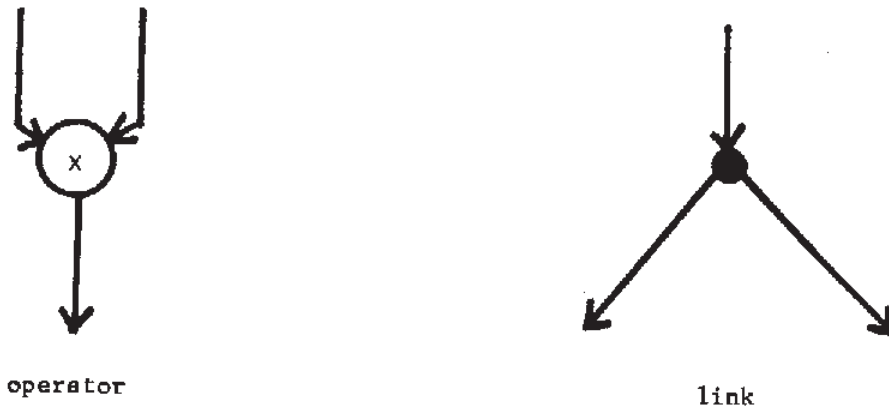


Figure 1. Operators and links of the elementary data-flow language.

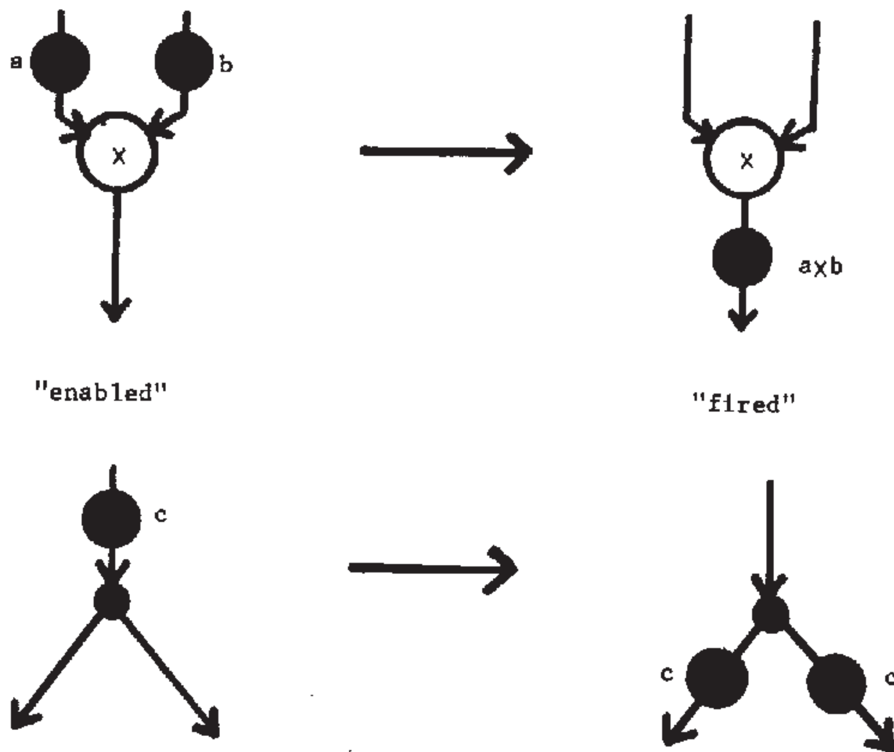


Figure 2. Firing rule for operators and links.

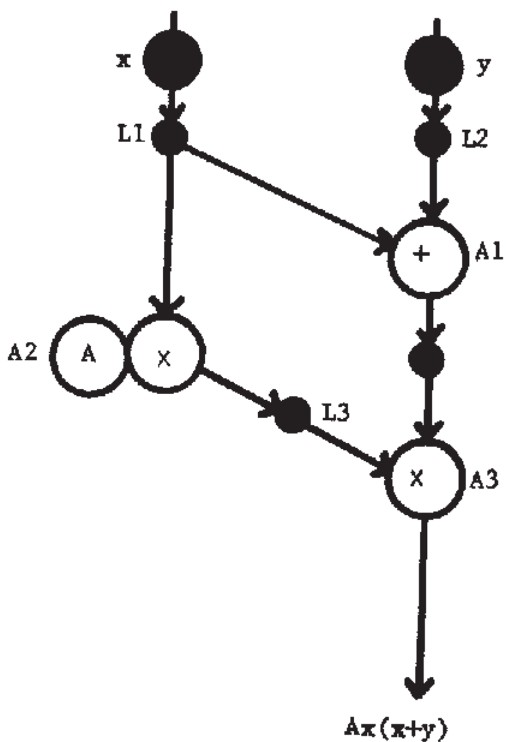


Figure 3. An elementary data-flow program.

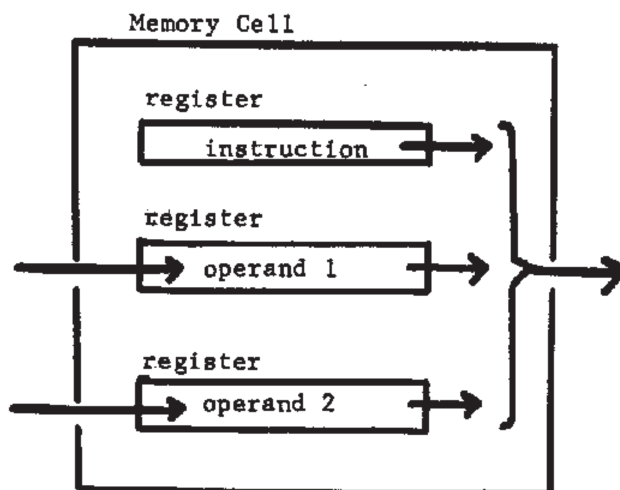


Figure 4. Structure of a Memory Cell.

its connections to other operators in the program and two registers that receive values for use in the next execution of the instruction. When all three registers of a Cell are full, the Cell is said to be enabled and signals that it is ready to have its contents operated upon by an appropriate Functional Unit.

The instruction format is shown in Figure 5. The first field contains the operation code, which specifies the type of Functional Unit to be used and the function it is to perform. The second and third fields hold the addresses of the Cells which are to receive copies of the result.

## The Fast Fourier Transform

The discrete Fourier transform of a sequence of N input samples is given by the equation:

$$F_k = \sum_{n=0}^{N-1} f_n W^{nk}, \quad \text{where} \quad W = e^{-j(2\pi/N)}$$

The value of each of the N elements of the output, $F_k$, is equal to the summation over the number of input samples of the sample, $f_n$, times some power of a weighting factor, W, requiring $N^2$ operations for the direct computation [7]. When the number of input samples is even, the transform can be computed by dividing the input sequence in half, performing two smaller transforms on the half-sized sequences, and combining the results with N multiplications and additions. When N is a power of two, this derivation can be successively repeated until the problem is reduced to a number of one-point transforms, which are null operations. The resulting fast Fourier transform (FFT) computation consists of $\log_2 N$ stages, each of size N.

The structure for a four-point FFT is shown in Figure 6 in its data-flow representation. This structure has identical geometry from stage to stage, allowing us to implement only one stage of the algorithm on the elementary processors and to cycle the data through that one stage $\log_2 N$ times. The four operators within the closed dotted line form one basic operational unit which is repeated throughout the program. Let us define these four operators as a basic operation and implement them as one Functional Unit, with two inputs and two

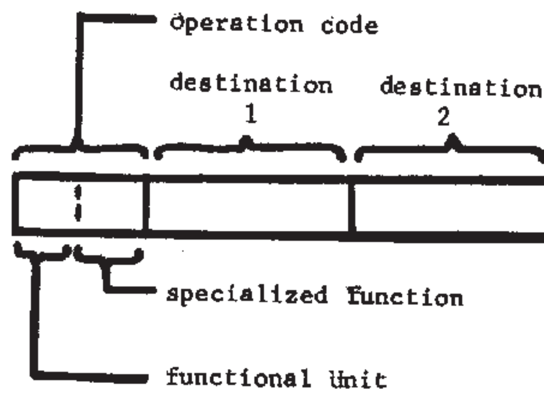


Figure 8. Instruction format.

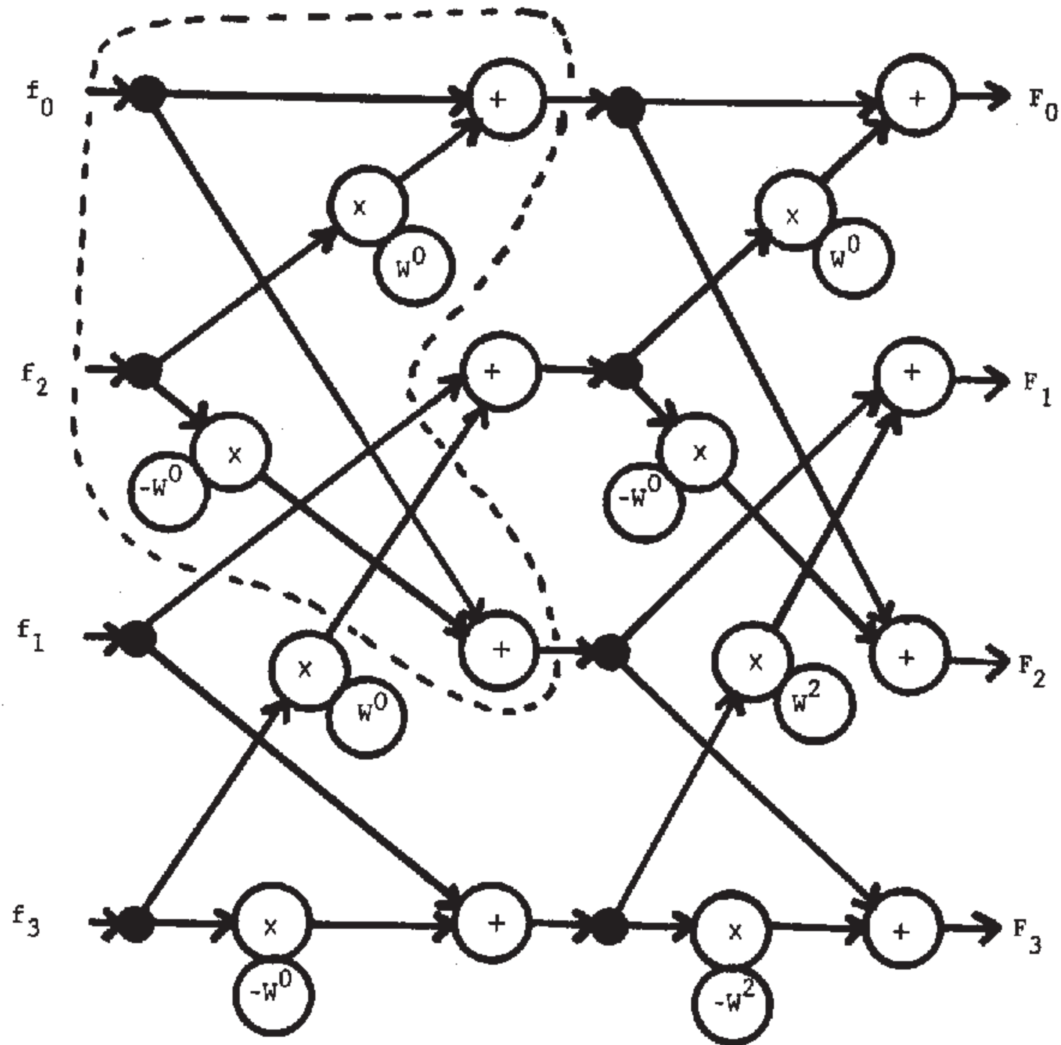


Figure 6. Data-flow representation of a four-point FFT.

outputs (Figure 7). The program is then considerably reduced, and an N point transform (assuming N some power of two) consists of $N/2 \cdot \log_2 N$ applications of this operation (Figure 8) and can be implemented on the elementary processor as N/2 Cells, each of which is executed $\log_2 N$ times in the computation.

Rather than examining the contents of each Cell for the FFT computation, let us look at a typical Cell, say Cell 0 (Figure 9), which, upon receiving two inputs, $f_0$ and $f_2$, becomes enabled, and passes its entire contents as an operation packet, consisting of the instruction and the two operands, to some FFT Functional Unit. The Functional Unit performs the required multiplication, addition, and subtraction and sends the two results as data packets, each consisting of a destination address and one result, to the appropriate registers specified in the destination address fields of the instruction, in this case register 1 of Cell 0 and register 1 of some Cell m. Cell 0, upon sending an instruction packet to a Functional Unit, once again has empty operand registers, and awaits the arrival of two new values. Upon their receipt, the Cell repeats the above process. The operand stored in the Cell is divided into two parts -- a value and a stage number which is used in the determination of the exponent of W.

## Network Structure and Performance

In a preceding section we described the structure and operation of a Memory containing the program. Now the question arises -- how do we interconnect that Memory with the Functional Units? Our answer to this classic problem is to construct a network, called the Arbitration Network, providing a path from each Memory Cell to each Functional Unit with the feature that many operation packets can pass through the network simultaneously, and each is directed into a queue for an appropriate Functional Unit (Figure 10). Since there are many inputs and only a small number of outputs, we must provide conflict resolution with arbitration units. An arbitration unit passes the first packet to arrive at one of its inputs to its output. If two packets arrive simultaneously, the unit chooses one to pass through on a round-robin basis. The switching of each packet to an appropriate Functional Unit based on its operation code is provided by switch units. A portion of the final part of the Arbitration Network is dedicated to each Functional Unit, creating a queue of instructions of a specific type.

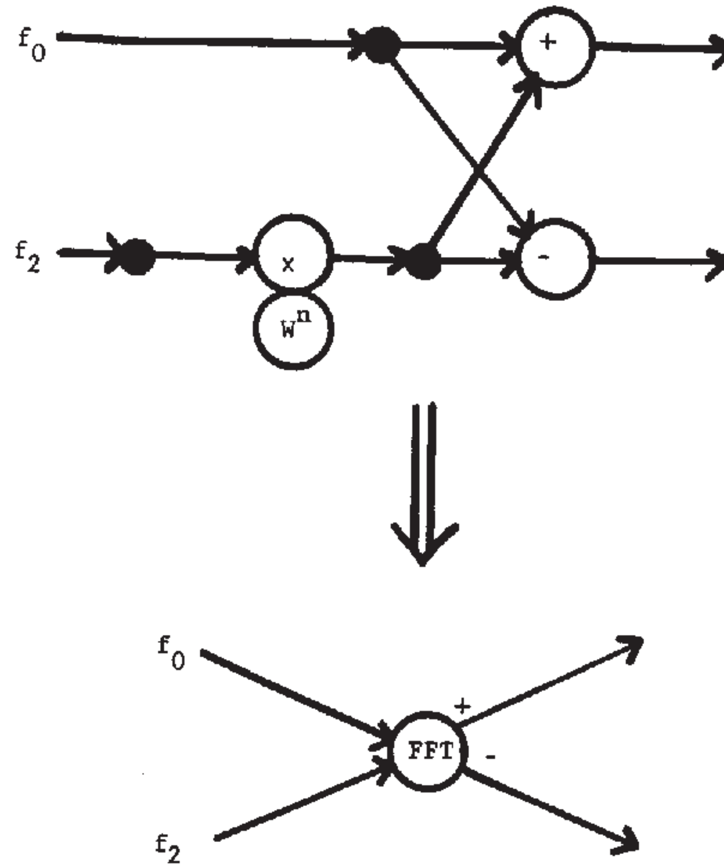


Figure 7. The basic FFT operator.

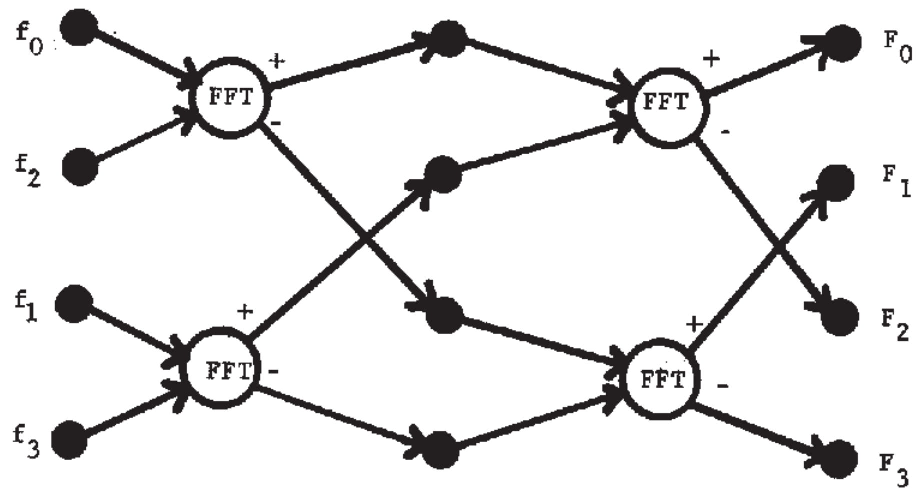


Figure 8. Revised representation of the four-point FFT.

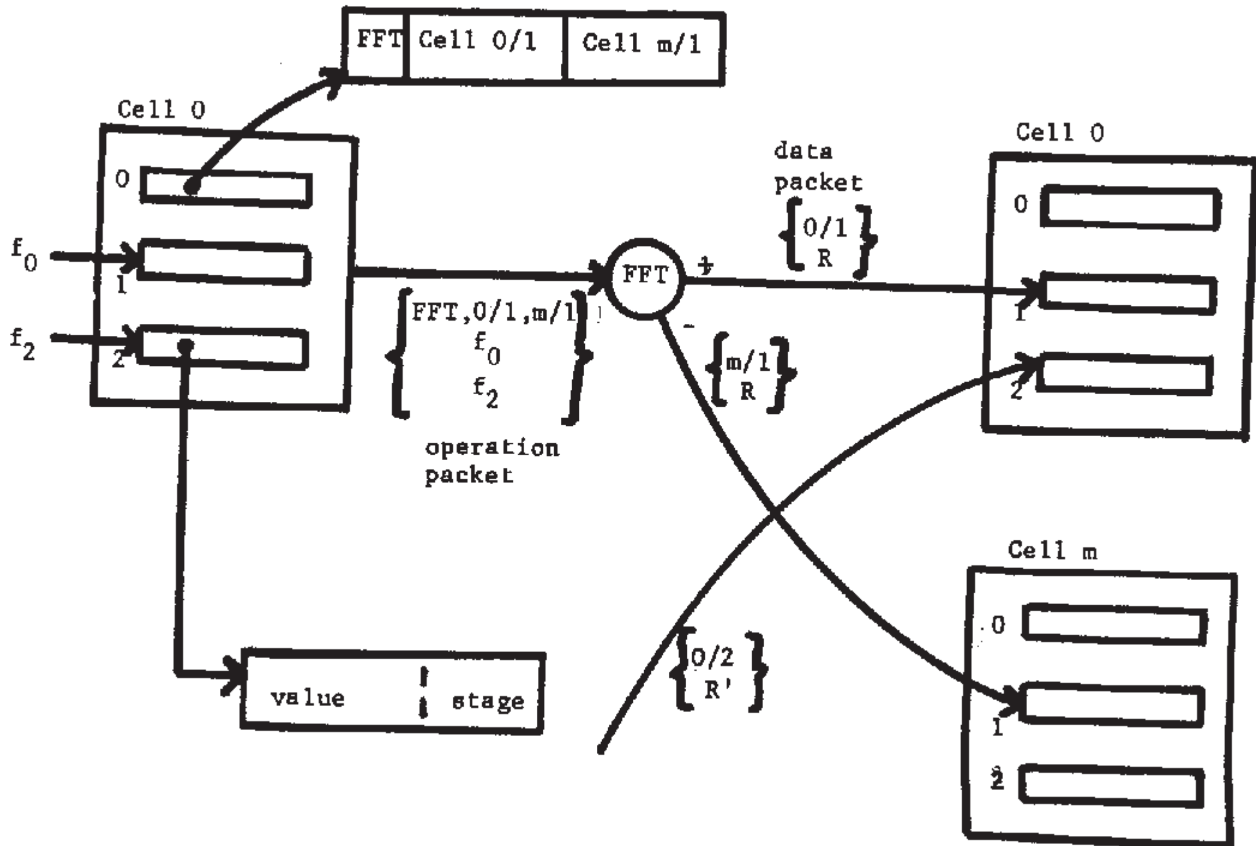


Figure 9. Execution of an FFT instruction.

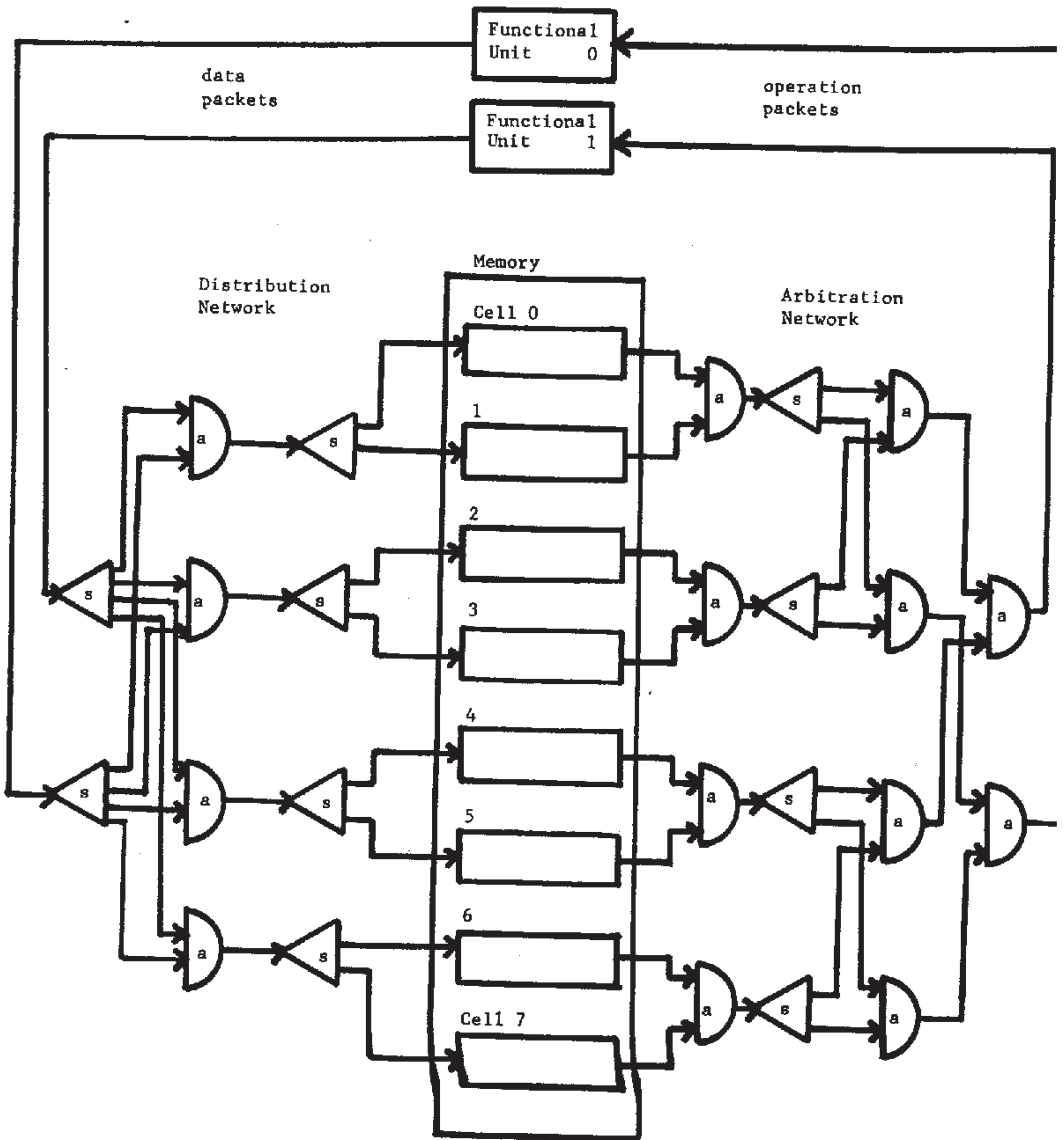


Figure 10. Structure of the elementary data-flow processor.

A Distribution Network is structured in a similar manner to provide a path from each Functional Unit to each Memory register. Switch units direct each data packet toward the appropriate register according to the destination address of the packet. A few arbitration units provide conflict resolution between the several inputs.

Due to the large number of inputs to the Arbitration Network, we wish to transfer data between the Memory Cells and the Arbitration Network in serial format to reduce the number of wires necessary. However, in order to maintain a high rate of packet flow at the output ports, we wish to transfer packets to the Functional Units in parallel format. For this reason, serial-to-parallel conversion is done gradually as a packet travels through the Arbitration Network. Parallel-to-serial conversion is performed in the Distribution Network for similar reasons.

The structure of the FFT algorithm provides for a large number of Cells to be enabled simultaneously, and hence, for a great deal of concurrent activity within the networks. In order to determine the processing time for a 1024 point FFT, we must examine the effect of this activity within the networks upon the processing time of an individual instruction, which is affected by the delay it encounters in passing through each network.

The minimum time necessary for a packet to travel through the Arbitration Network is found by considering its passage through the network without any conflict. This time is given by the summation over the stages it must travel through of the time required to transfer a packet through each stage:

$$\text{minimum delay} = \sum_{\text{stages}} (\text{no. bits serial} + 1)(\text{bit transfer time})$$

The transfer time for a stage is equal to the number of bits passing through the stage in serial plus one for a signal to indicate that the packet is ready to be transferred times the time necessary to transfer a bit. A similar equation applies to delay in the Distribution Network.

Let us examine the delay within a specific Arbitration Network (Figure 11). This network has three stages and seven arbitration units. Packets travel through Stage 0 in four-bit serial format and are gradually converted to a

| Stage Number | 0 | 1 | 2 |
|---|---|---|---|
| Serial Bits | 4 | 2 | 1 |
| Passage Time | 5t | 3t | 2t |

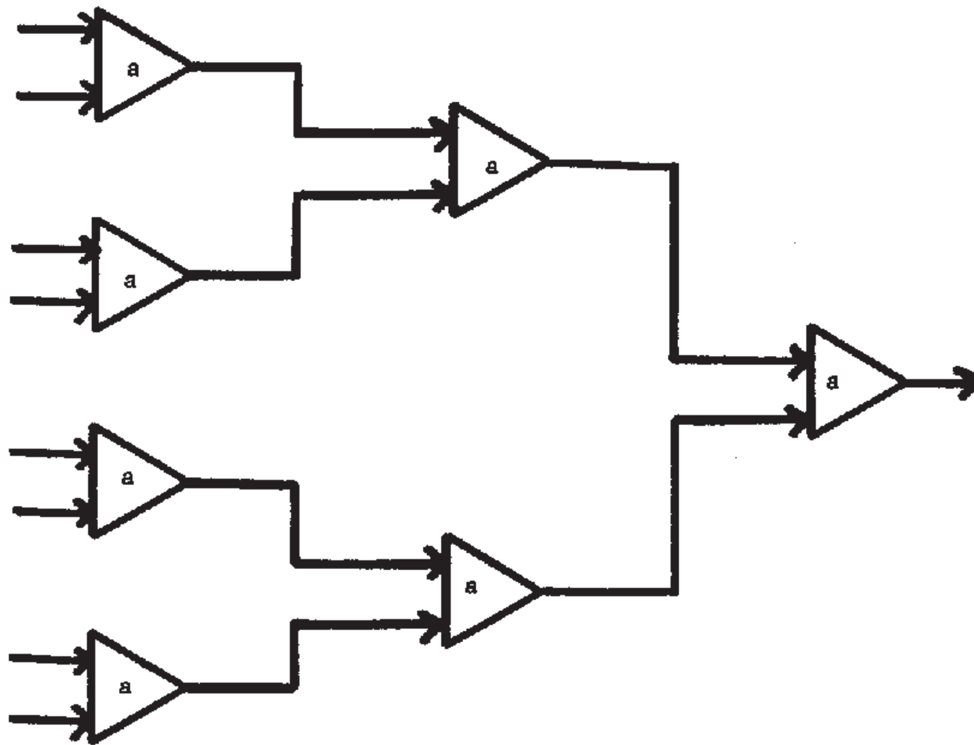


Figure 11. Structure of an elementary Arbitration Network.

more parallel format, passing through stage 1 in two-bit serial and stage 2 in one-bit serial format. As noted before, the passage time for a packet through each stage is equal to the number of serial bits plus one times the bit transfer time; let us call it t. For this structure, these figures are 5t, 3t, and 2t, respectively. The minimum delay through the network is equal to the summation of the stage delays, or 10t.

In order to find the maximum delay a packet can encounter in passing through the Arbitration Network, we must consider a network which has a packet present at every node in a machine in which every Memory Cell is enabled, placing a packet on each input to the Arbitration Network (Figure 12). If we consider the maximum delay which can be encountered by a packet, say the triangular one, we find that it arises only when all other packets in the network and at the inputs of the network pass through the output before our triangular friend does. In order for this to happen, not only must the triangular packet lose every conflict, but every packet on the path he will follow to the output must also lose every conflict. Thus, finding the maximum delay involves examining how many packets will flow through each stage before the triangular one.

For this network the worst case packet will be the 14th through stage 2, the 6th through stage 1, and the 2nd through stage 0. If we multiply the number of packets passing through each stage times the delay in that stage, we find that:

$$\begin{aligned} \text{maximum delay} &= 2(5t) + 6(3t) + 14(2t) \\ &= 56t \end{aligned}$$

Thus, there is a factor of six degradation in passage time between the minimum and maximum packet delays for this network.

However, when a delay close to the maximum occurs, it occurs because the network is being very highly utilized, and thus we are trading specific performance for overall performance of the machine.

Let us consider a 1000 Cell machine, 512 of which are being used for the computation of a 1024 point FFT. Also, let us assume that t is equal to 150 nsec. This is a rather conservative figure; one certainly will be able to complete a ready/acknowledge cycle in 15 TTL gate delays. Without getting into the structure of the networks and the tradeoffs involved, a reasonable set of delays is presented below:

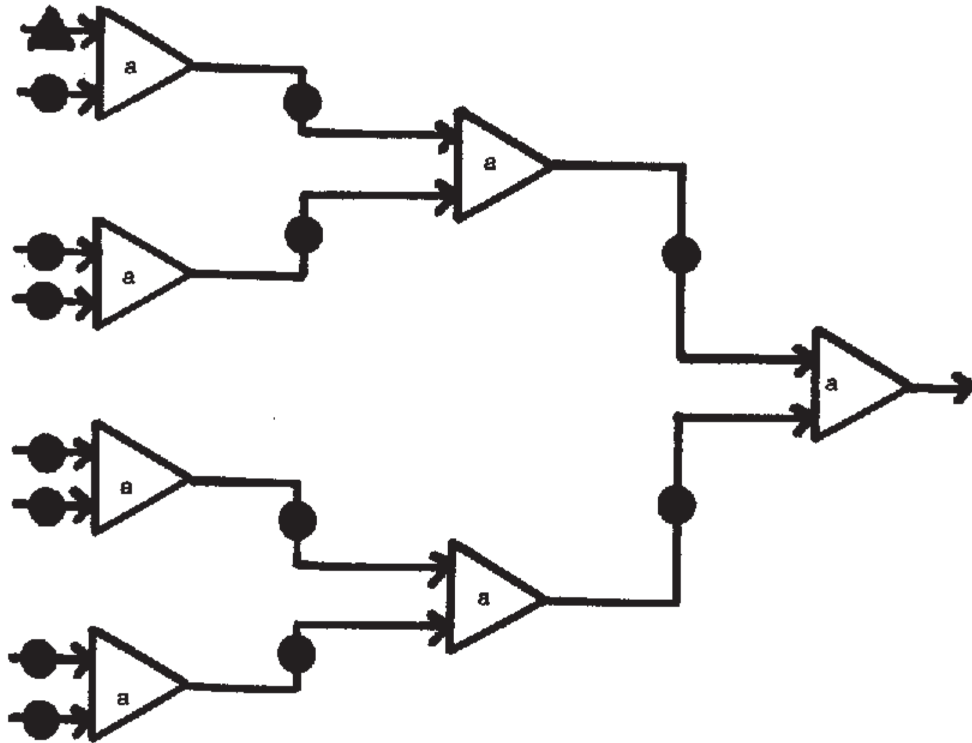


Figure 12. Example of a full Arbitration Network.

Best Case:

| | | |
|---|---|---|
| Delay in Arbitration Network | = 5 μsec. | (8 stages) |
| Delay in Distribution Network | = 5 μsec. | (6 stages) |
| Delay in Functional Unit | = 5 μsec. | |
| | 15 μsec. | |

However, we are not interested in the best case figures as much as the worst case ones, which turn out to be:

Worst Case:

| | |
|---|---|
| Delay in Arbitration Network | = 100 μsec. |
| Delay in Distribution Network | = 20 μsec. |
| Delay in Functional Unit | = 5 μsec. |
| | 125 μsec. |

There is an increased delay in the Distribution Network due to the presence of a few arbitration units to resolve conflicts between the various inputs.

125 μseconds seems like a long time to process one instruction. But if we think for a second, we remember that by the definition of the worst case delay, all other Memory Cells plus a large number of other packets have passed through the networks and Functional Units in this time. Thus, in 125 μsec. all 512 Cells of the FFT computation have been executed at least once. If we multiply this worst case delay by the number of stages ($\log_2 1024$ or 10), we get a worst case figure of 1.25 milliseconds for the computation of the entire FFT. This is a figure which is very highly competitive with the processing times of current FFT processors.

Examining the output ports of the Arbitration Network (Figure 13), we see that an operation packet can be sent to a Functional Unit every 300 nanoseconds, that is, at a rate of 3.3 million instructions per second (MIPS). Assuming that each stage of the pipelined Functional Unit operates in less than 300 nanoseconds, we can conclude that 3 FFT Functional Units are necessary to maintain the rate just developed.

Now, not only are we processing each FFT in a maximum time of 1.25 milliseconds, but there are 488 other Cells in the machine available to perform other computation. They might be occupied in doing some filtering, data manipulation, or another FFT. The use of these Cells does not affect the computation time necessary for the FFT since it was assumed that they were active for purposes of computing the worst case delay.

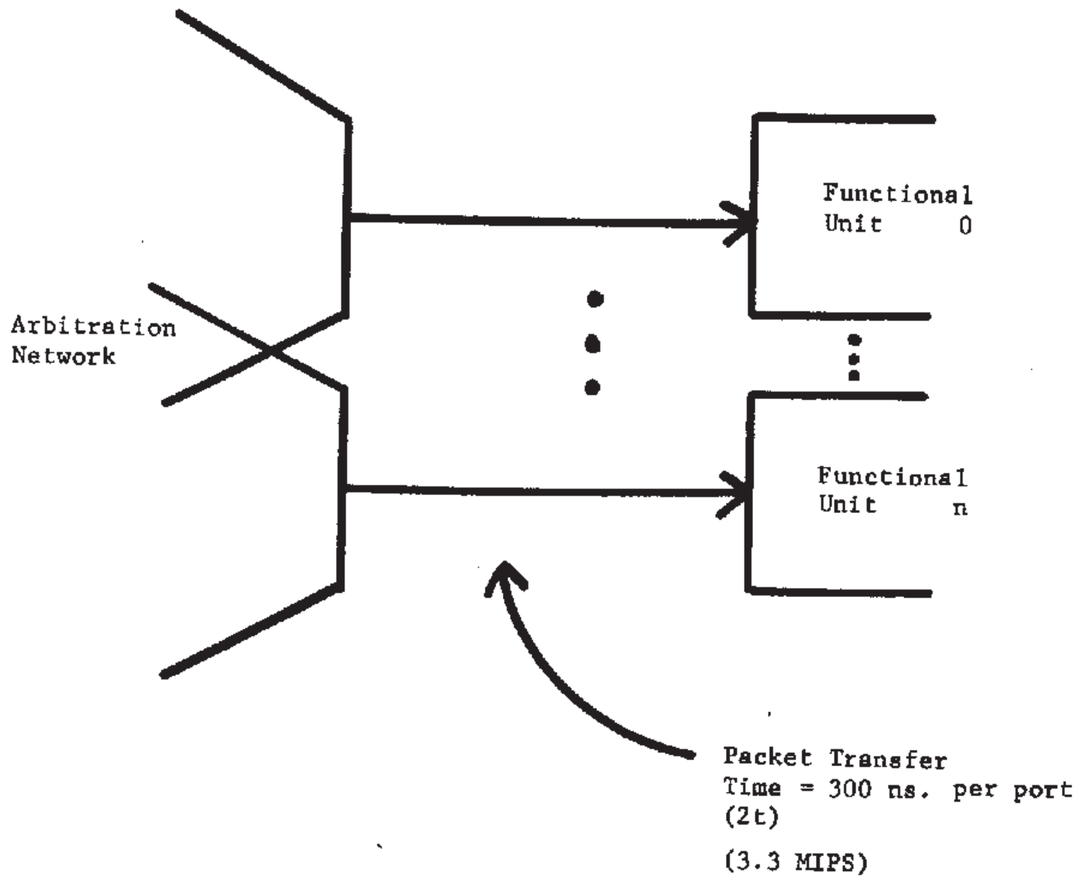


Figure 13. Packet transfer rate to Functional Units.

I want to emphasize that these worst case figures provide an extreme upper bound on the time necessary for the computation. It is difficult to imagine a case in which all Memory Cells are enabled and the networks are full. Thus, the actual computation time should be significantly lower than the worst case figure.

## Concluding Remarks

There will be those who will not be satisfied with this execution time and who will wish to perform their transforms even faster. There are several ways of increasing the performance of the architecture to accomplish this. First, another Functional Unit can be added. This addition requires that the packet format within the networks be changed so packets travel in more parallel versions. This change will then reduce the worst case delay time, decreasing the processing time for a stage of the transform.

Another way of speeding up execution involves changing technologies. The figures just presented were based on a TTL implementation. A change to ECL, for example, should allow approximately a five-fold decrease in execution time, yielding a maximum processing time of 250 microseconds for the 1024 point FFT.

It turns out that all is not quite as rosy as I have just depicted it. There is a deadlock problem caused by the fact that information flows only in one direction in the machine [10]. In the architecture just described, an instruction does not know whether its destination registers are free or not, so it just assumes they are, and it is possible for a number of packets destined for the same register to be in the Distribution Network simultaneously. These packets will occupy nodes of the network and block access to other registers of the Memory. The solution to this problem involves introducing control signals which are sent by an operator to immediately preceeding operators to notify them that its operand registers are empty. This change does not affect the time required for an FFT computation, and the figures I have just generated are still valid for a deadlock-free version.

Unfortunately, as yet, we have no cost figures for the machine. However, we are hopeful that the cost will be rather low due to the highly modular nature of the design and its apparent suitability for an LSI implementation.

References

1. Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, Calif., December 1968.

2. Bähr, A. Operation patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).

3. Dennis, J. B. First version of a data flow procedure language. Lecture Notes in Computer Science 19 (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York, 1974, 362-376.

4. Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. November 1973 (submitted for publication).

5. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974, 402-409.

6. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975, 126-132.

7. Gold, B., and C. M. Rader. Digital Processing of Signals. McGraw-Hill, New York, N. Y., 1969, 173-196.

8. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. of Appl. Math. 14 (November 1966), 1390-1411.

9. Kosinski, P. R. A data flow language for operating systems programming. Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September 1973), 89-94.

10. Misunas, D. P. Deadlock avoidance in a data-flow architecture. Proceedings of the Milwaukee Symposium on Automatic Computation and Control, IEEE, New York, April 1975.

11. Rodriguez, J. E. A Graph Model for Parallel Computation, Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.