MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 104

Data-Flow Computer Architecture

by

Jack B. Dennis
David P. Misunas
P. S. Thiagarajan

(Proposal Submitted to NSF)

August 1974

A. Introduction

Many concepts of great importance to progress in softward technology -- modularity, data structures, multiprocessing, hierarchies of abstractions -- were unknown at the conception of the stored program computer, and it should not be surprising that the architecture of conventional machines bears little relation to modern advanced programming concepts. The result has been programming languages that make the undesirable compromise of foregoing clean semantics to achieve efficient execution on conventional machines, and languages in which it is difficult or impossible to divide program design into a hierarchy of independent decisions about representing the abstractions of the problem domain. (See the work of Parnas [33], Liskov[30, 31] and Dennis [10].)

Our research has had the goal of understanding the influence of architectural concepts on the design and implementation of programming languages and systems. This work has led to the discovery of radically new concepts of computer organization that could lead to computer systems much better suited to the needs of programming --- machines that directly implement the fundamental semantic constructs of advanced programming languages.

Computers designed according to our concepts will achieve highly-parallel program execution by identifying and exploiting the concurrency of program parts. We see this as essential in computers intended to support advanced programming concepts and still operate efficiently. These computer systems must implement a uniform mechanism -- in the spirit of Multics -- for accessing information regardless of its location in the memory system. However, the performance of contemporary systems is severely limited by the memory management problem -- a conventional proccessor must have direct access to enormous main memories to ensure that enough active processes are available to keep the processor busy.

Our idea is to use a very small "page size" so only immediately useful data is brought into higher memory levels. By not bringing in information unlikely to be referenced, this should yield a large decrease in the amount of fast memory required to support a given computational load. By itself, this idea would yield very little work for processors if programs were sequential -- there would be lots of waiting for data. Our answer is to find lots of actions to perform within programs so the processors can be kept busy anyway. We believe we have discovered a basis for hardware structures able to achieve this objective.

Our architectural concepts are based on data-flow representations for programs which expose the concurrency of program parts. Data-flow models are closely related to the "applicative" programming languages, and have been developed to equal the expressive power of high-level languages.

We propose to develop and evaluate computer systems using data-flow machine languages. We will pursue this project in stages starting with a processor for a simple data-flow language that nevertheless has an interesting and important domain of application in stream-oriented signal processing. The design of a processor for the elementary data-flow language has already been completed and we propose to construct a prototype machine of modest capability. The second stage will be a highly parallel processor for a Fortran-level data-flow language. We have solved some of the major problems in extending our architectural concepts to this level of expressive power, but much further work is needed before construction can be proposed.

We envision a third stage of the project in which data-flow concepts are used in the realization of a general purpose computer system. This is attractive to us but admittedly speculative as there is much difference of opinion regarding the amount of exploitable parallelism in general purpose computer applications. Nevertheless, we are encouraged by the work of Kuck [29] who has shown that a surprisingly large degree of parallelism can be found from analysis of ordinary programs.

In the following sections we briefly review developments in computer architecture related to our work, discuss the fundamental concepts of data-flow models for program execution, and present our work so far on the design of computers based on data-flow machine languages. The final section of the proposal presents the program of research we wish to pursue: Further development of architectural concepts; studies relating to the feasibility and expected performance of data-flow computers; and the construction of a prototype processor of modest size.

In summary, we propose an orderly step-by-step approach to the trial and evaluation of a radically different concept of computer organization. We will start at a level of language in which we are confident of success, and work toward systems of increasing generality. In each step the architecture developed will implement a specified level of user language, so there can be no question regarding the class of programs the machine will serve -- the principal software problems will be solved in the specification of the machine.

B. Background

Two characteristics of our studies in computer architecture are:

1. Each architecture developed must implement a precisely specified base language.
2. The architecture developed must be capable of achieving efficient highly-parallel program execution.

Although there have been a number of projects and studies having one of these objectives, we are not aware of any project in which computer organizations having both characteristics were sought. In the following paragraphs we review previous work in computer architecture having either of the two goals given above, and discuss its relation to our proposed research.

B1. High-Level Language Machines

The Burroughs B 5000 computer is the earliest significant example of a language-oriented advance in computer architecture. The built-in stack mechanism is designed to match the convention for accessing variables in a procedural language such as Algol 60. More recently, the idea of designing a computer to directly execute programs represented in a high-level language has inspired many projects. For example, machines based on Fortran [5], Algol 60 [20], APL [19] and Lisp [15] have been proposed. These projects are significant in that the practical feasibility of translating the run-time facilities of conventional language systems into wired logic and/or microprograms has been demonstrated.

The Symbol Project [23, 36, 37] is closest in direction to our proposed effort. Symbol is both a language and a machine. The Symbol language was designed without imposing the restrictions usually made to allow efficient implementation on machines of conventional architecture -- the basic data objects of Symbol are tree structures which may be created and altered in shape and content during program execution. The Symbol machine was designed to directly implement the Symbol language; thus it has built-in algorithms for creating and accessing representations of trees.

B2. Parallel Machines

The chief motivation for the design and construction of highly parallel machines has been to achieve a high processing rate by exploiting the complexity of processor organization made feasible by advances in technology. Unfortunately, the computer organizations chosen for attaining highly parallel operation have been matched to the characteristics of particular classes of data structures (vectors, matrices), and have led to machines that are substantially more difficult to program than conventional machines.

The Illiac IV [4] is the most ambitious and familiar example of an array processor. A single instruction decoding station drives many processing elements simultaneously, where each processing element has its own memory and hence can operate on its own data. Algorithms must be organized to make maximum use of array and vector operations if the power of an array processor is to be realized.

The Texas Instruments Advanced Scientific Computer [43] and the Control Data Star-100 [22] are two examples of machines using pipelined arithmetic elements to achieve highly parallel computation. Yet a pipelined arithmetic unit can achieve high performance only if it is fed a continuous stream of instructions and operands. In these machines streaming operation is established only by use of special instructions, and highly parallel operation is obtained only when the computation can be represented in terms of primitive operations on long data streams or vectors.

Array and pipelined processors, in spite of their high potential throughput, have the serious drawback that the programmer is forced to use unusual and intricate representations for his data if their performance potential is to be realized. Thus these architectures have contributed little toward the goal of reducing the cost of creating correct and easily understood programs.

Efforts to use advanced technology to increase the performance of processors employing conventional machine languages include the CDC 6600 [40] and the IBM 360/91 [2, 41]. These machines "look ahead" in the instruction stream to detect instructions that may be executed concurrently, and have several functional units that may perform independent instructions simultaneously. Nevertheless, these designs have proven to be limited in their ability to exploit parallelism and effectively utilize their functional units.

The idea of a multiprocessor computer [16] -- several monosequence processors sharing access to a number of memory units -- was first realized in the Burroughs B 5000 system. Although such multiprocessor computer systems are able to effectively multiplex their processor and memory resources among many concurrently executing programs, they have not proven suitable for exploiting the internal parallism of programs. This is because a large overhead cost is associated with switching a processor from one activity to another, and the partitioning of programs into many long sequential segments is a difficult, if not impossible, problem. Moreover, the number of processors in such systems is limited by the complexity of the processor/memory switch.

The Carnegie-Mellon C.mmp project [44] is an interesting current activity in multiprocessor computer systems. The C.mmp system embodies an attractive near-term approach to distributing a computational task over an unusually large number of processor and memory units. Making effective use of the processing potential of the C.mmp requires careful partitioning of the workload among the processors to minimize resource use for interprocessor communication, and careful design of each part to fully utilize a processor.

C. Data-Flow Languages and Processors

A base language founded on the notion of data flow is attractive to us both as a semantic model for programs expressed in high-level languages and as the specification for the functional behavior of computer systems. In a data-flow representation of a program, the application of a function or a test is free to proceed as soon as the values required for its application are available. Moreover, the result of a function evaluation or the outcome of a decision is made available to precisely those functions and predicates in the program that depend on it.

The concepts of data-flow representation have developed through the work of Rodriguez [38], and Dennis and Fosseen [12] at MIT, and the work of Adams [1], Karp and Miller [26], Bähre [3] and Kosinski [28]. Our most recent and complete formulation of a data-flow language is in the paper by Dennis [11], which is attached to this research proposal as Appendix A. This language is a complete semantic base for source programs expressed in command-oriented languages such as Algol 60 and applicative languages such as PAL. Since the basic procedure construct in the data-flow language is functional (that is, contains no free variables and does not produce side effects), this data-flow language satisfies important criteria [10] for serving as the base language of computer systems that support the modular construction of programs.

In our work, a data-flow program is a directed bipartite graph in which the two types of nodes are called actors and links. Actors perform the basic steps in the process of computation -- the application of operators and the testing of predicates. Links pass values from each actor to the actors that require its results. The values generated and consumed by actors are represented by tokens that are placed on the arcs of a data-flow program and convey values between actors.

The expressive power of a data-flow language is determined by the variety of actors allowed. We have identified three levels of data-flow language for which corresponding computer architectures have attractive application. Thus, our ultimate objective of developing a general purpose computer system based on data-flow semantics can be approached by pursuing several less ambitious projects to develop data-flow processors of increasing generality starting with a very simple language.

The simplest language level is called the <u>elementary data-flow language</u>. Programs in this language have actors of just one type called <u>operators</u>. An example of an elementary data-flow program is shown in Figure 1, and represents the following simple computation:

```
Input   a, b, c
   y := (a + b)/c
   x := (a * (a + b)) + c

Output  y, x
```

The rectangular boxes in Figure 1 are operators and each arithmetic operation in the above computation is reflected in a corresponding operator. The small dots are links. The large dots represent tokens; the configuration shown represents the initial condition for execution of the data-flow program, in which the tokens carry the initial data values.

Execution of a data-flow program is described by a sequence of configurations, each consisting of the program graph and a distribution of tokens. Program execution advances from one configuration to the next through the <u>firing</u> of some node of the program. The firing rules for elementary data-flow programs are as follows: An operator or link is <u>enabled</u> if a token is present on each of its input arcs and there is no token on any of its output arcs. The enabling of an actor indicates availability of the values required for application of the corresponding

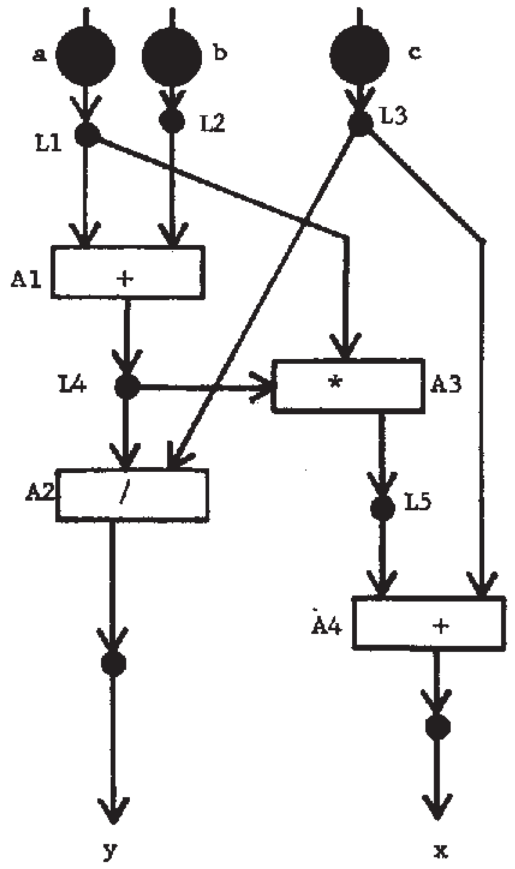


Figure 1. Elementary data-flow program.

primitive function. A new token distribution is defined by the firing of any enabled node. An operator fires by removing the tokens from its input arcs, applying the corresponding function to the values carried by the input tokens and placing a single token carrying the computed value on its output arc. The firing rule for a link is identical except that the firing of a link merely results in the replication of the value carried by the token on its inpur arc.

For example, in the program shown in Figure 1, links L1, L2 and L3 are initially the only enabled nodes. The firing of L1 makes copies of the value a available to operators A1 and A3; firing L2 presents the value b to operator A1 alone. Once both L1 and L2 have fired, (in any order), operator A1 is enabled since it will have a token on each of its input arcs. After A1 has fired, completing the computation of a + b, link L4 will be enabled. The firings of L3 and L4 will enable the concurrent firing of operators A2 and A3, and so on.

Although the elementary data-flow language is very primitive, its expressive power is not as limited as the example in Figure 1 might indicate. The graphs of elementary data-flow programs may contain cycles, allowing for repeated application of a group of operators to streams of data. By designing a processor to execute an elementary data-flow program for a specified number of repetitions of each operator, highly parallel execution of a sizable class of stream-oriented computations becomes feasible. For example, the computation required for a second-order digital filter

$$y(t) = Ax(t) + By(t - 1) + Cy(t - 2)$$

can be represented as an elementary data-flow program as shown in Figure 2. By executing this program for n firings of each node, the values of y(t) for t = 0, 1 ..., n - 1 can be generated.

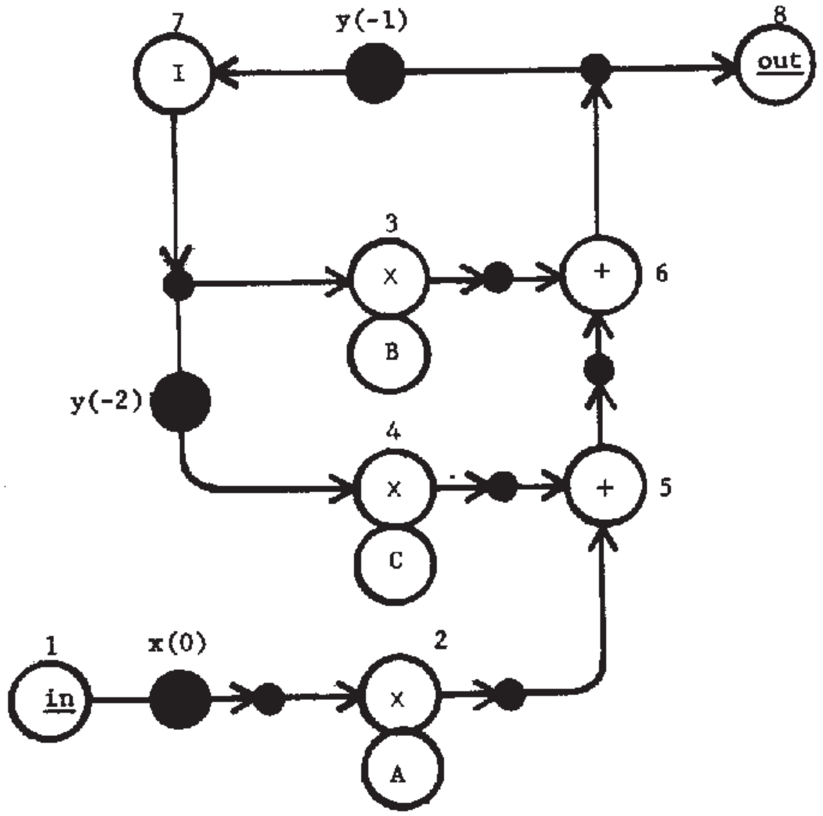


Figure 2. Data-flow program for the second order recursive digital filter computation.

C.1 The Elementary Data-Flow Processor

We have succeeded in designing the architecture of a processor, called the elementary processor, capable of highly parallel execution of programs expressed in the elementary data-flow language [13]. This processor is viewed as a special-purpose computer particularly suited to stream-oriented signal processing, and which would be operated under the control of a conventional host computer.

The organization of the elementary data-flow processor is shown in Figure 3. The data-flow program to be executed is represented in the Memory Cells of the processor. Each Memory Cell corresponds to an operator of the data-flow program and consists of three registers. The first register holds an instruction which specifies the operation to be performed and the address(es) of the register(s) to which the result of the operation is to be directed. The second and third registers hold the operands for use in execution of the instruction. Figure 4 shows the contents of the Memory Cells of the elementary processor for the initial configuration of the program shown in Figure 2.

When a Cell contains an instruction and the necessary operands, it is <u>enabled</u> and signals the Arbitration Network that it is ready to transmit its contents as an <u>instruction packet</u> to a Functional Unit which can perform the desired operation. The instruction packet flows through the Arbitration Network, which directs it to an appropriate Functional Unit by decoding the instruction portion of the packet.

The result of Functional Unit operation is one or more <u>result packets</u> each consisting of a computed value and the address of a register in the Memory. The result packets are presented to the Distribution Network which, by means of the destination address, routes the data value to the correct register of the Memory. The Cell containing that register will become enabled if all operands are now present in the Cell.

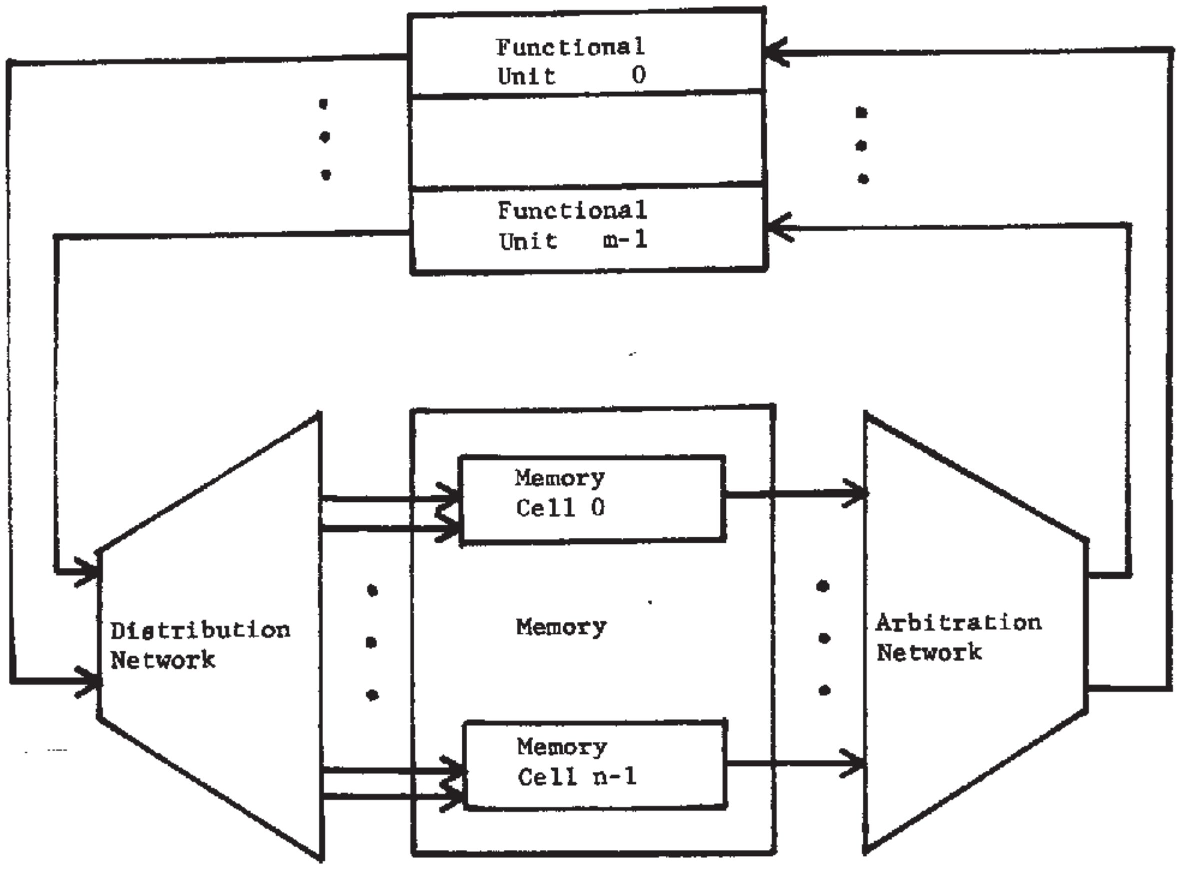


Figure 3. Organization of the elementary data-flow processor.

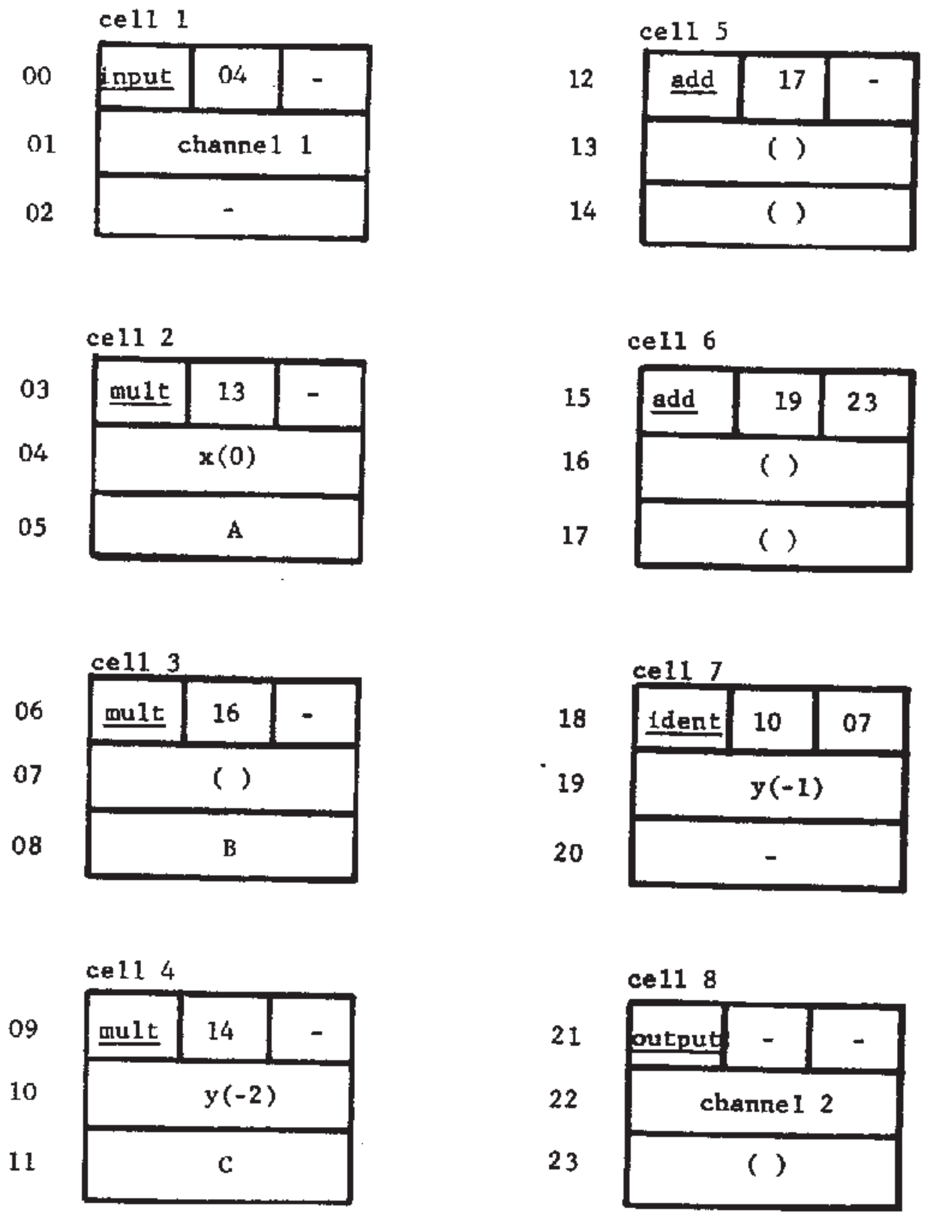


Figure 4. Initialization of Memory Cells for the digital filter computation.

Many Memory Cells may be enabled simultaneously and it is the function of the Arbitration Network to efficiently deliver instruction packets to the Functional Units and to queue instruction packets waiting for each Functional Unit. Similarly, the Distribution Network may also have many packets traveling through it simultaneously. In addition, the Functional Units are organized in a pipeline fashion. Thus, all major sections of the processor are organized to operate with a high level of concurrency. Even through a single instruction packet may experience a significant delay in passing through the Arbitration Network, the Functional Units and the Distribution Network, the processor can maintain a high computation rate.

One major problem in achieving highly parallel computation is the memory/processor interconnection problem in multiprocessor computer systems [16]. The idea of an active Memory connected to the Functional Units through the Arbitration and Distribution Networks offers an interesting and, in our opinion, a very appealing solution to this problem. Since there is considerable flexibility in the structure of the Arbitration Network and Distribution Network, a structure can be selected that yields balanced utilization of all sections of the processor.

The architecture of the elementary data-flow processor is naturally implemented as a speed-independent interconnection of separate units. In fact, we have worked out specifications for all major sections of the processor in terms of a small set of asynchronous module types that communicate with each other in a speed-independent fashion. This design is presented in the attached Appendix B. The methodology of specification used in this work derives from our previous work on speed-independent logic design and Petri Nets [7, 32, 34].

We have been attracted to the use of asynchronous logic since we feel that it is only through the use of asynchronous logic that a "natural" hardware realization of a data-flow language with all its attendant concurrency can be obtained. Traditionally, logic designers have been reluctant to employ asynchronous logic due to fear of timing hazards. However, these problems are avoided in our work by modular design using uniform speed-independent communication conventions throughout the machine. Moreover, we believe the task of ensuring correctness of an implementation will prove far easier when units are precisely specified using the methodology we have developed for the elementary processor.

C2. The Basic Data-Flow Processor

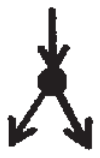
The second level of data-flow language for which development of a corresponding architecture is attractive incorporates semantic constructs sufficient to represent computations expressed in a Fortran-level source language. Such a data-flow language must be able to represent conditionals and iteration, and its domain of values must include arrays. Moreover, achieving highly parallel computation in Fortran-level programs requires that the parallelism inherent in array operations be exploited, and that multiple concurrent activations of subprograms be supported. While most of these constructs are covered by the data-flow language of Appendix A, we have not yet made a definite choice of data-flow language for Fortran-level programs.

Nevertheless, we have recently been successful in extending the architectural concepts of the elementary data-flow processor to a data-flow language that is a substantial step toward a Fortran-level capability. This language is called the <u>basic data-flow language</u>, and extends the elementary language with constructs for representing conditionals and iteration.

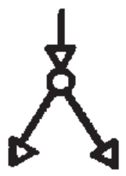
The links and actors of basic data-flow programs are shown in Figure 5. There are now two classes of values -- data values, and the control values {<u>true</u>, <u>false</u>} -- and corresponding types of link nodes. <u>Deciders</u> apply predicates to data values and yield control values; the <u>Boolean operators</u> perform Boolean operations on control values. The firing rules for these nodes are the same as for the operators and data links of elementary data-flow programs.

The T-gate, F-gate and merge actors provide the means for controlling the flow of data tokens according to the outcomes of decisions. For example,

A. Links

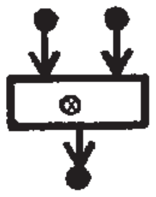


(a) data link

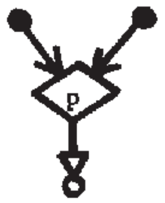


(b) control link

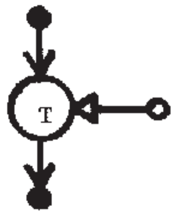
B. Actors



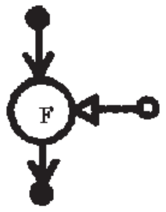
(a) operator



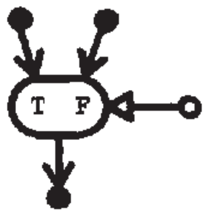
(b) decider



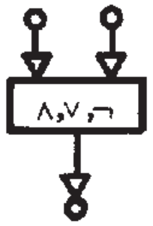
(c) T-gate



(d) F-gate



(e) merge



(f) Boolean operator

Figure 5. Links and Actors of the basic data-flow language.

a T-gate will pass the data token on its input arc to its output arc when it receives a control token carrying the value true at its control input. It will absorb the data token on its input arc and place nothing on its output arc if a false-valued control token is received. The merge node has two data input arcs labeled T and F, and also a control input arc. It will pass to its output arc a data token from the input arc corresponding to the value of the control token received. Any token present on the other input arc will not be affected. A more detailed description of these actors and the firing rules associated with them may be found in Appendix A.

Using the actors and links of the basic data-flow language, conditionals and iteration can be easily represented. In illustration, Figure 6 gives a basic data-flow program for the following computation:

```
 input y, x
    n: = 0
 while y < x do
    y: = y + x
    n: = n + 1
          end
output y, n
```

The control input arcs of the three merge nodes carry false-valued tokens in the initial configuration so the input values of x and y, and the constant 0 are admitted as initial values for the iteration. Once these values have been received, the predicate y < x is tested. If it is true, the value of x and the new value for y are cycled back into the body of the iteration through two T-gates and two merge nodes. Concurrently, the remaining T-gate and merge node return an incremented value of the iteration count n. When the outcome of the decider is false, the current values of y and n are delivered through the two F-gates, and the initial configuration is restored.

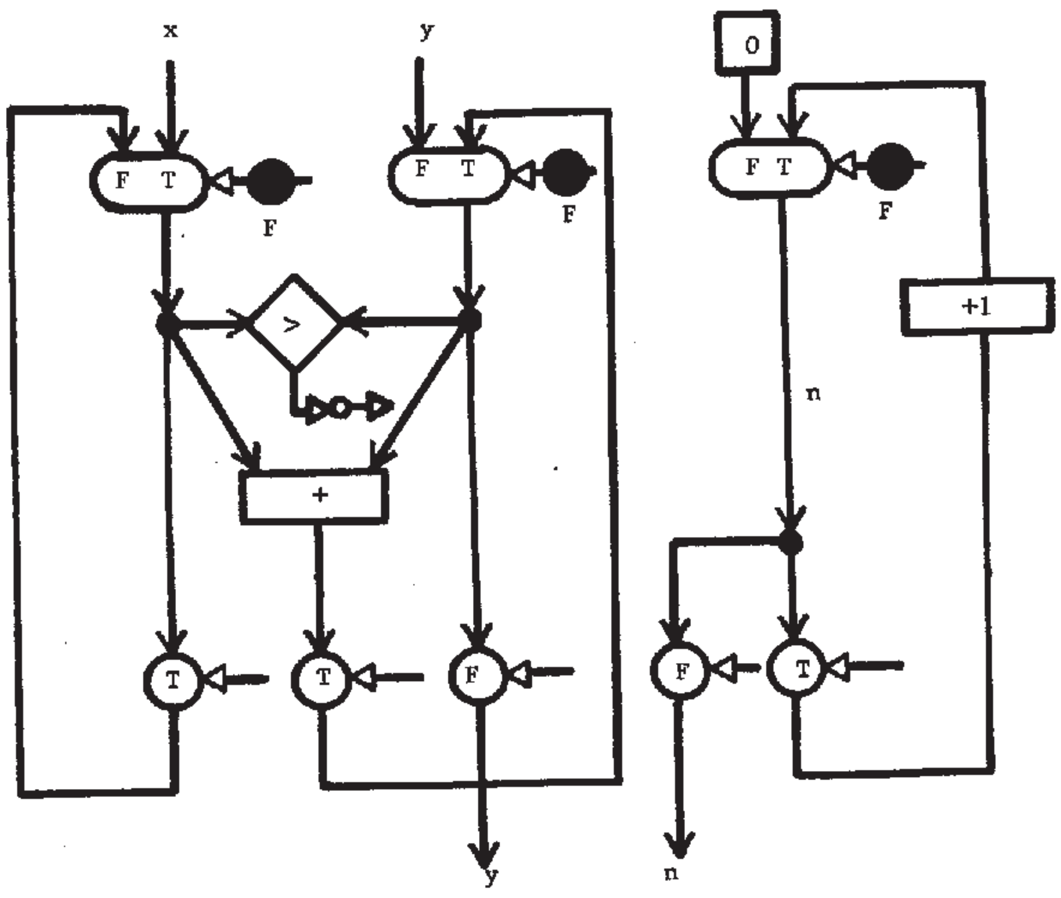


Figure 6. Basic data-flow program.

We have developed an outline for the architecture of a processor, called the basic processor, that directly executes basic data-flow programs. Two problems required solution in the design of this processor: One problem is the addition of mechanisms to implement the new actors of basic data-flow programs -- deciders, gates, merge nodes, and Boolean operators. The other problem is the inclusion in the processor of a multi-level memory system so only the active portions of a data-flow program will occupy the Cells of the processor.

The organization of a processor that extends the capability of the elementary processor to basic data-flow programs is shown in Figure 7. One Memory Cell of this machine is assigned to hold an instruction corresponding to each operator, decider, or Boolean operator of the basic data-flow program. Instructions corresponding to operators combine with their operands to form instruction packets that are sent to the Functional Units as in the elementary processor. Instructions corresponding to deciders and Boolean operators generate instruction packets sent to the Decision Units. The Decision Units produce <u>control packets</u> containing control values. These packets are directed to destination registers through a Control Network according to destination addresses included in the instructions.

The arrival of a control packet at a destination register either provides an operand value to a Boolean operator, or performs a gating function for one operand of an operator or decider. The gating function requires a second field in the operand registers of Cells containing operator or decider instructions. The new field contains a <u>gating code</u> having three possible values with the following meanings:

<u>gating code</u>	<u>meaning</u>
<u>no</u>	no gating function occurs at the operand register.
<u>true</u>	the operand register implements a T-gate actor.
<u>false</u>	the operand register implements an F-gate actor.

If the gating code of an operand register is <u>no</u>, then the arrival of a control packet at the register is an error -- the register is enabled

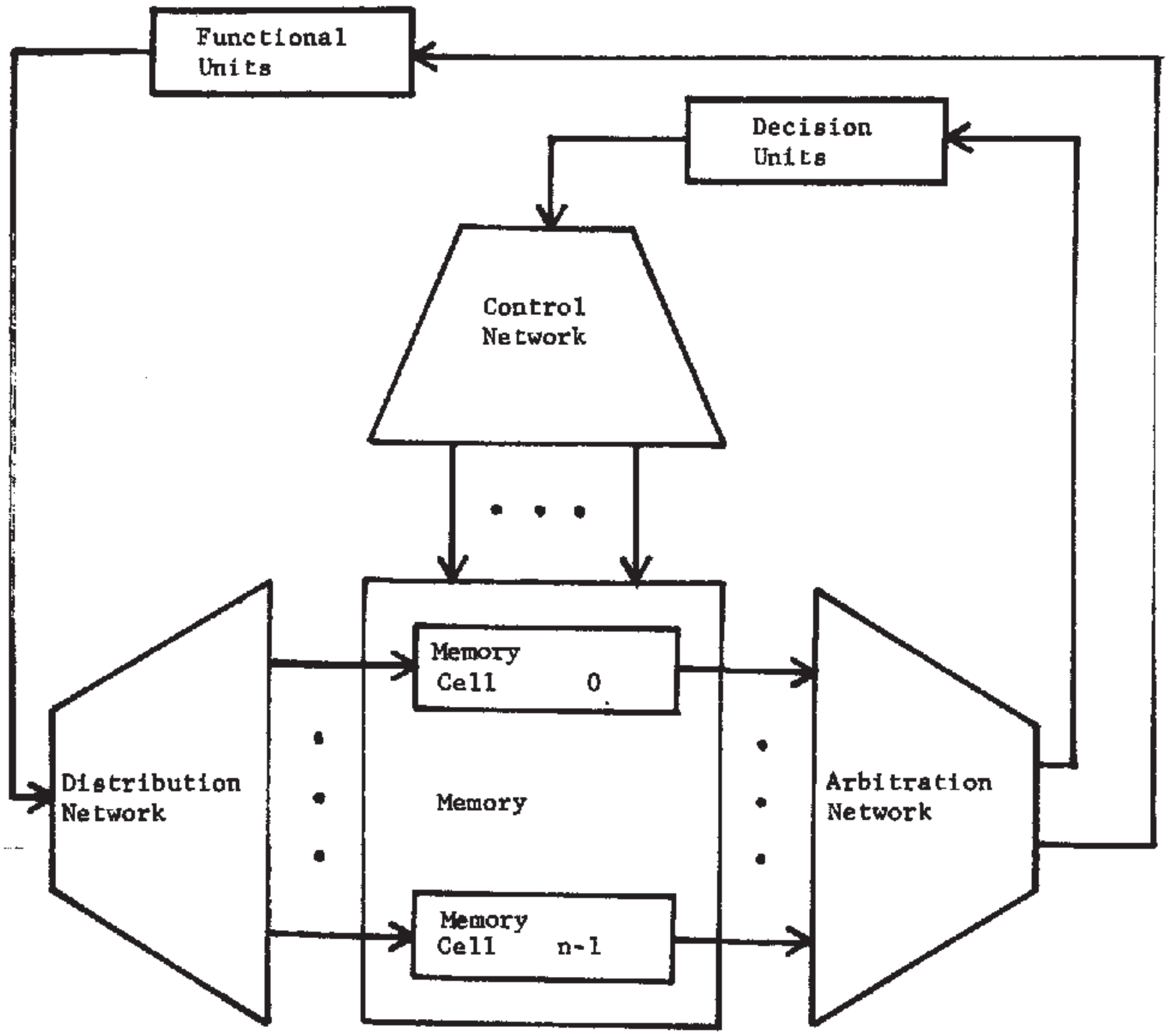


Figure 7. Organization of the basic data-flow processor without two-level memory.

by arrival of a result packet as in the elementary processor. If the gating code is <u>true</u>, then the operand register is enabled only if a result packet, and a control packet containing <u>true</u> are received. Arrival of a control packet containing <u>false</u> causes the associated result packet to be discarded and resets the operand register to its initial condition. The complementary behavior occurs if the gating code is <u>false</u>. In this way, the function of T-gates or F-gates at the inputs of operators or deciders is realized. The function of a merge actor is realized by simply directing result packets from both input sources to the same destination address; the merging of token flow occurs within the Distribution Network.

In Figure 7 the Control Network is shown separate from the Distribution Network because it is in fact a much simpler structure. The values transmitted consist of just one bit, so the parallel-to-serial conversion included in the Distribution Network is not needed. This simplicity permits quick propagation of decisions and increases exploitation of concurrency by the processor.

The architectural concept for a basic data-flow processor just described has the weakness that each instruction of a program must occupy a Memory Cell throughout the course of program execution. For elementary data-flow programs, this weakness is of no consequence since all nodes of an elementary program are continuously and equally active. With the inclusion of constructs for conditionals and iteration, the various parts of a data-flow program will be active for different phases of a computation, and some parts may not be executed at all. To avoid providing a Memory Cell for each instruction of a basic data-flow program, we have developed an organization (Figure 8) in which the Memory Cells (now designated Instruction Cells) act as a cache memory for instructions held in a second-level Instruction Memory. Instructions are requested from the Instruction Memory as result packets or control packets required for instruction execution arrive at the Instruction Cells. Instructions

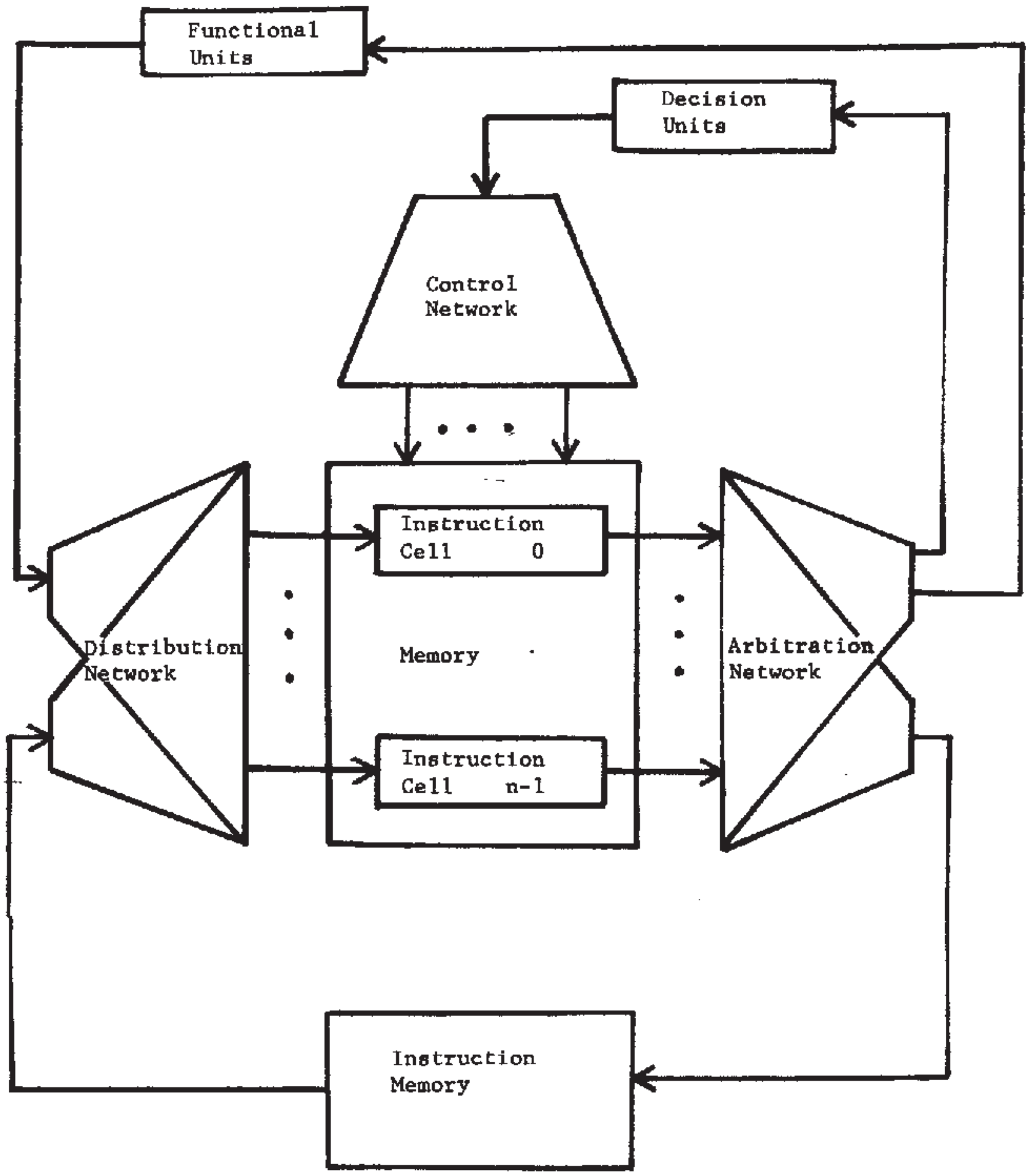


Figure 8. Organization of the basic data-flow processor with Instruction Memory.

(which may be partially enabled) are displaced from Instruction Cells to the Instruction Memory as Instruction Cells are preempted for newly active instructions.

The design of the basic data-flow processor has been developed to the level of behavioral specifications for each major unit shown in Figure 8. This design is the subject of a paper [14] that has been submitted for conference presentation, and is included as Appendix C of this proposal.

The basic data-flow processor lacks some essential semantic constructs needed for highly-parallel execution of Fortran-level programs -- specifically, subprograms and arrays. The extension of our architectural concepts to include these and further generalizations of expressive power is one subject of our proposed research.

D. Proposed Research

We propose to begin a long-range program to design and evaluate a series of computers capable of direct, highly parallel execution of programs represented in data-flow form. For the period covered by this initial proposal, our work will include: 1. Further development of our architectural concepts to extend to direct execution of Fortran-level and general-purpose data-flow programs; 2. Studies to evaluate the feasibility, performance and practicality of the elementary and basic data-flow processors; and 3. The construction of a prototype elementary processor.

D1. Architectural Concepts

We plan to extend the ideas embodied in the elementary processor in two stages -- first a processor for Fortran-level data-flow programs, and then a computer system for general data-flow programs. The Fortran-level processor is intended to be a highly parallel machine for execution of numerical computations expressed in a Fortran-like language and should prove attractive for numerical problems calling for very high processing rates.

The problems to be solved to obtain a viable design for a Fortran-level data-flow processor fall into two categories -- extending the set of linguistic constructs, and solving the problems introduced by using several levels of physical storage media. The constructs that must be added to the basic data-flow language include procedures and array operations that expose their parallelism for exploitation by a data-flow architecture. Both of these extensions introduce the need for dynamic execution structures, and require a more sophisticated memory structure than we have devised for the basic processor.

The second stage of architecture development is the further extension of our ideas to a general purpose computer system using a data-flow representation for programs. The direction this work will take is difficult to predict. Yet, since it is intended to lead to a design for a complete general purpose computer system, its design will have to encompass not only the complete language of Appendix A, but also such issues as program creation, execution monitoring, ownership of procedures and data, and access control. Our study of these aspects of a general data-flow computer will be coordinated with the development of the CLU programming language and system by Professor Liskov [30], for which separate funding is being sought.

D2. <u>Feasibility and Evaluation Studies</u>

The merit of our data-flow processors depends on their cost, their performance, and their ease of application to important problem areas. We propose to study these topics as follows:

<u>Feasibility of LSI Implementation</u> -- The elementary processor is a highly repetitive structure using many copies of only a few kinds of units. Hence, it is attractive to consider using LSI technology even for the construction of only a few complete systems. A large portion of the elementary processor could be fabricated using only a few chip types of moderate complexity.

The feasibility of using LSI technology to construct a small number of systems depends on the number of different device types and the investment required to put a device into production.

We believe our methods of design specification in terms of speed-independent interconnection of basic asynchronous modules can lead to fewer design errors and greater confidence in correct translation of specifications into masks for chip manufacture. This should lead to a significant reduction in the number of design iterations required to perfect a device, and a lower investment for the manufacture of custom devices.

We propose a detailed study of the expected cost of translating our logic specifications into masks for LSI device fabrication. We wish to understand the practical tradeoffs between tooling cost and logic speed as applied to our architectural concepts, and we wish to choose a level of formal description for communicating device specifications to the semiconductor manufacturer.

It seems likely that the layout of masks should be done by copying and interconnecting standard cells that are universal building blocks for asynchronous logic. We propose to determine a suitable set of standard cells, perhaps in correspondence with the basic modules introduced in Appendix B.

Analysis of Performance -- An attractive aspect of the elementary data-flow processor is the possibility of configuring the Arbitration and Distribution Networks, and designing the instruction set so that all parts of the machine are equally well utilized. The computation rate of the elementary processor is determined by its physical structure and by the constraints on instruction sequencing imposed by data-flow programs. We propose to identify useful measures of the sequencing constraints imposed by a program, and investigate methods for calculating bounds on computation rate in terms of these measures. In this work the methods developed by Ramchandani [35] in our research group should prove useful.

Similar problems of performance analysis will be formulated and analyzed for the basic data-flow processor. In addition, we must be certain there is no possibility that our data-flow processors will deadlock -- cease all activity -- when computation remains to be performed. Previous work on liveness of Petri Nets [18] and asynchronous systems [25] should be useful here.

Programming -- Application of the elementary data-flow processor requires a suitable textual language for stream-oriented computation. Several such languages have been developed [27, 21], and it will not be difficult to design a suitable user language. However, the Fortran-level data-flow processor poses an interesting language design project. The challenge arises because basic data-flow programs will be able to represent program structure by use of procedures and by use of streams. To our knowledge, no satisfactory language has been proposed that is able to satisfactorily represent both forms of computation. We will attempt to develop the semantic basis for such a language. We hope that the language will permit natural expression of such algorithms as the fast Fourier transform for highly parallel execution. This work will be founded on our studies of data flow schemas [12] and parallel computation [8].

D3. Construction of a Prototype Elementary Processor

We propose to build a prototype elementary data-flow processor of modest capability using available commercial integrated circuit logic elements. We view this project as serving three roles: It will demonstrate the validity of our architectural concepts, which represent a radical departure from past approaches to highly parallel computation; we will gain experience in the technical problems of realizing moderately large systems using speed-independent principles of operation; and there are interesting application areas -- especially stream-oriented signal processing -- for which the prototype processor will have valuable application.

Two promising applications are represented by current projects at M.I.T. One of these is the work of Professor Barry Vercoe in the M.I.T. Experimental Music Center on developing sound synthesis facilities for use by serious composers of music. The signal processing computations described by works expressed in the Music 360 language [42] are natural computations for the elementary data-flow processor. The other project is in the area of speech analysis and synthesis -- the signal processing computation required for terminal analog speech synthesis is an ideal application for the proposed prototype processor.

The choice of processor size for prototype construction is a compromise among cost of construction, physical size and power consumption, and the difficulty of construction, checking and testing. From careful study of alternatives, we have concluded that the prototype machine should be a 64-cell machine having the following characteristics:

memory cells	64
registers	192
word size	32 bits
functional units	4
maximum computation rate	5 million instructions per second (mips)
physical size *	3 standard racks
power consumption *	10 Kw.

* The large size and power consumption arise from our use of active memory of unique design. Both figures would be greatly reduced in an LSI realization made possible by the repetitive nature of the architecture. In spite of this defect, the prototype machine will achieve high and balanced utilization of all components in applications well matched to its capabilities.

A 64-cell machine is sufficiently large to support interesting applications by the research groups mentioned earlier, yet small enough that we are confident of success in its construction. The computation rate of 5 mips is derived from complete preliminary logic designs for all units of the processor using commercial TTL devices, and represents the maximum rate at which the Arbitration Network can pass instruction packets to the four Functional Units. Structures for the Arbitration Network and Distribution Network in the prototype have been chosen to support at least this rate. Failure of the processor to achieve the 5 mips rate can only result from sequencing constraints in the data-flow program, or the absence of sufficient operators in the program to fully utilize the capacity of the processor. Note that the computation rate is equivalent to a much higher instruction rate for a conventional machine due to the control and addressing instructions required to implement stream-oriented signal processing on a conventional machine.

The cost of constructing the prototype processor is as follows:

semi-conductor components	14,100
circuit board layout [1]	10,200
circuit board fabrication [1]	15,700
power supply and distribution	6,000
racks and hardware	8,800
backplane wiring [1]	6,000
functional units	15,000
test equipment	6,000
TOTAL	81,800

[1] to be contracted.

In addition to these purchased materials and services, direct labor is required for design verification, assembly, checkout and drafting of diagrams. These costs are itemized in the Budget.

For testing the prototype processor and to implement simple user and command languages, a host computer is required. This machine should be a mini-computer to facilitate easy connection to the prototype processor and other experimental equipment. For compatibility reasons (the projects that are potential users of the prototype have private PDP-11 computers), the host computer should be a PDP-11. We propose the following configurations of Digital Equipment Corporation Units:

PDP-11 E 10 processor	21,000
16k Memory	
RK11-D DEC Pack Disk	
TA11 Dual Cassette	
LA30 Decwriter Terminal	
Basic Operating Software	1,700
Interfacing Hardware	2,500
TOTAL	$25,200

This machine will also be used to test printed circuit boards for the data-flow processor and will be used in the development of automated design aids for subsequent projects.

References

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

2. Anderson, D. N., F. J. Sparacio and R. M. Tomasulo, The IBM System/360 model 91: Machine philosophy and instruction-handling. IBM Journal of Research and Development, 11, 1 (January 1967), 8-24.

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

4. Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. C. Slotnick and R. A. Stokes, The Illiac IV computer. IEEE Trans. on Computers, C-17, 8 (August 1968), 746-757.

5. Bashkow, T. R., A. Sasson and A. Kronfeld, System design of a Fortran machine. IEEE Trans. on Computers, EC-16, 4 (August 1967), 485-499.

6. Dennis, J. B., Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., Amsterdam 1969, 484-492.

7. Dennis, J. B., Modular asynchronous control structures for a high performance processor. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, 55-80.

8. Dennis, J. B., Coroutines and parallel computation. Proceedings of the Fifth Annual Princeton Conference on Information Science and Systems, March 1971.

9. Dennis, J. B., On the design and specification of a common base language. Proceedings of the Symposium on Computers and Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn 1971, 47-74.

10. Dennis, J. B., Modularity. Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, 1973, 128-182.

11. Dennis, J. B., First version of a data flow procedure language. Proceedings of the Symposium on Programming, University of Paris, April 1974.

12. Dennis, J. B., and J. B. Fosseen, Introduction to Data Flow Schemas. Submitted for publication, November 1973.

13. Dennis, J. B., and D. P. Misunas, A computer architecture for highly parallel signal processing. Paper accepted for presentation at the ACM National Conference, San Diego, Calif., November 1974.

14. Dennis, J. B., and D. P. Misunas, A preliminary architecture for a basic data flow processor. Paper submitted for presentation at the Second Annual Symposium on Computer Architecture, January 1975.

15. Deutsch, L. P., A Lisp machine with very compact programs. Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, Calif., August 1973.

16. Flynn, M. J., Some computer organizations and their effectiveness. IEEE Trans. on Computers, C-21 (September 1972), 948-960.

17. Gertz, J. L., Hierarchial Associative Memories for Parallel Computation. Report MAC-TR-69, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., June 1970.

18. Hack, M. H., Analysis of Production Schemata by Petri Nets. Report MAC-TR-94, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., February 1972.

19. Hassit, A., J. N. Lageschulte and L. E. Lyon, Implementation of a high level language machine. Comm. of the ACM, 16, 4 (April 1973), 199-212.

20. Haynes, L. S., Structure of a polish string language for an Algol-60 language processor. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, 8, 11 (November 1973), 131-137.

21. Henke, W. L., MISTYN: A Graphical Notation and System for the Interactive Computer-Aided Specification and Realization of Digital Signal Processing. Quarterly Progress Report 105, M.I.T. Research Laboratory of Electronics, Cambridge, Mass., April 1972, 123-133.

22. Hentz, R. G., and D. P. Tate, Control Data Star-100 processor design. Proceedings of the Sixth Annual IEEE Computer Society International Conference, 1972, 1-4.

23. Hutchison, P. C., and K. Ethington, Program execution in the SYMBOL-2R computer. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, 8, 11 (November 1973), 20-26.

24. Isaman, D. L., Memory Systems for Parallel Processing of Dynamic Data Structures. Doctoral research in progress.

25. Jump, J. R., and P. S. Thiagarajan, On the interconnection of asynchronous control structures. To be published in the J. of the ACM.

26. Karp, R. M., and R. E. Miller, Properties of a model for parallel computations: determinacy, termination and queueing. SIAM J. of Applied Math., 14, 6 (November 1966), 1390-1411.

27. Kelly, J. L., C. Lochbaum and V. A. Vyssotsky. A block diagram compiler. Bell System Technical Journal, 40, 3 (May 1961), 669-676.

28. Kosinski, P., A data flow language for operating system programming. SIGPLAN Notices, 8, 9 (September 1973), 89-94.

29. Kuck, D. J., Y. Muraoka, and S.-C. Chen, On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. IEEE Trans. on Computers, C-21, 12 (December 1972), 1293-1310.

30. Liskov, B. H., A design methodology for reliable software systems. AFIPS Conference Proceedings, 41, Part I (December 1972), 191-199.

31. Liskov, B. H., and S. Zilles, Programming with abstract data types. SIGPLAN Notices, 9, 4 (April 1974), 50-59.

32. Misunas, D. P., Petri nets and speed independent design. Comm. of the ACM, 16, 8 (August 1973), 474-481.

33. Parnas, D. L., On the criteria to be used in decomposing systems into modules. Comm. of the ACM, 15, 12 (December 1972), 1053-1058.

34. Patil, S. S., and J. B. Dennis, The description and realization of digital systems. Proceedings of the Sixth Annual IEEE Computer Society International Conference, September 1972, 223-226.

35. Ramchandani, C., Analysis of Asynchronous Concurrent Systems by Petri Nets. Report MAC-TR-120, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., February 1974.

36. Richards, H., Jr., and C. Wright, Jr., Introduction to the SYMBOL-2R programming language. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, 8, 11 (November 1973), 27-33.

37. Richards, H., Jr., and R. J. Zingg, The logical structure of the memory resource in the SYMBOL-2R computer. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, 8, 11 (November 1973), 1-10.

38. Rodriguez, J. E., A Graph Model for Parallel Computations. Report MAC-TR-64, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., September 1969.

39. Rumbaugh, J., Parallel Distributed Machine Architecture for a Data Flow Base Language. Doctoral research in progress.

40. Thornton, J. E., Parallel operation in Control Data 6600. AFIPS Conference Proceedings, 26, Part II (May 1964), 33-41.

41. Tomasulo, R. M., An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development, 11, 1 (January 1967), 25-33.

42. Vercoe, B. L., The Music 360 language for sound synthesis. Proceedings of the Sixth Annual Conference of the American Society of University Composers. University of Houston, April 1971.

43. Watson, W. J., The Texas Instruments advanced scientific computer. Proceedings of the Sixth Annual IEEE Computer Society International Conference, 1972, 291-293.

44. Wulf, W. A., and C. G. Bell, C.mmp -- a multi-mini-processor. AFIPS Conference Proceedings, 41, Part II (December 1972), 765-777.