MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

Computation Structures Group Memo 191

Building Blocks for Data Flow Prototypes

by

Jack B. Dennis
G. Andrew Boughton
Clement K. C. Leung

This paper will appear in the Proceedings of the 1980 Symposium on Computer Architecture, to be held at LaBaule, France, May 1980.

This research was supported by the University of California, Lawrence Livermore Laboratory under contract 8545403.

February 1980

Building Blocks for Data Flow Prototypes[a]

Jack B. Dennis
G. Andrew Boughton
Clement K. C. Leung
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A variety of proposed architectures for data flow computers have been advanced. Evaluation of the practical potential of these proposals is being studied through analysis and simulation, but these techniques cannot be used to study a machine design in sufficient detail to make accurate predictions of performance. As a basis for extrapolating cost/performance of these architectures, and for developing a methodology for data flow program preparation, the construction of prototype machines is needed. In this paper we present our plan for realizing experimental data flow machines as packet communication systems using two types of hardware elements: a microprogrammed processing element with provision for packet transmission and reception; and a router unit used to build networks to support packet communication among processing elements.

Introduction

A variety of physical structures for computers embodying the data flow concept have been proposed[10], and several experimental machines have been built or are in construction[4,5,16]. Which concepts of data flow computer organization are ripe for development into practically useful, widely applicable computer systems? Two large areas of concern are: (1) will a proposed design achieve performance attractive in terms of cost? (2) How difficult will it be to prepare and test programs for the proposed machines?

Since data driven computers are radically different from conventional machines, we cannot predict the performance that a data flow machine will achieve by analogy with or extrapolation from that of conventional computers. Rather, we must determine, for any proposed application, what program structure would be used and how fast the program would run on the hypothetical machine. Determining cost is also difficult, since in the absence of detailed designs of subunits the best assignment of function to subunits is not obvious, and neither is it clear which design choices can lead to the best use of LSI technology.

(a) This research was supported by the University of California, Lawrence Livermore Laboratory under contract no. 8545403.

Three approaches may be used to evaluate the performance offered by a proposed architecture: analysis, simulation; and construction. Analysis fails to give useful answers where a system or program is too complex or does not have a regular structure. Simulation can be used to study the speed or throughput of the subunits of which a machine is constructed, and small examples of hypothetical machine code may be run by programmed interpretation of the machine's instruction set. However, simulation of a practical machine at a sufficient level of detail to draw conclusions about its performance for a particular substantial computation is well beyond the current power of simulation techniques.

Construction of a prototype is particularly attractive for a data flow machine because most of the proposed architectures scale easily: A small machine can be built which can be made more powerful by increasing the number of units in the machine without any change in their design; or the technology used to construct the units can be changed so the whole system operates uniformly faster. In either case, the performance achieved by a prototype for a scaled-down application provides a practical basis for extrapolating the performance to be expected from a large-scale machine in the application.

In our data flow project at MIT, we are using all three approaches to determining the performance potential of data flow computers. The routing networks we have proposed for use in data flow systems have been studied through analysis[3] and simulation[13] to determine the relationship of throughput and performance. Programs having a regular structure such as the fast Fourier transform[8] and a hydrodynamics simulation problem[2] have been studied by analysis of program structure in relation to hypothetical computing systems.

The subject of the present paper is the approach we have adopted for building prototypes of proposed data flow computer architectures. In building prototype machines we wish to achieve the following objectives, which include answering questions of performance, but also include questions of construction methodology, programmability, reliability and cost:

1. Run data flow programs that have meaningful application.

2. Develop a basis for extrapolating performance to large scale data flow machines.

3. Develop and evaluate instruction set designs.

4. Develop algorithms for code generation from high level languages.

5. Determine functional specifications of units appropriate for LSI fabrication.

6. Evaluate applicability of asynchronous, self-timed logic design methodology.

7. Study questions of correctness and testing of programs and hardware designs.

8. Study requirements for and approaches to achieving fault tolerance.

Our first experimental prototype will be a realization of the data flow processor[8] whose operation is most readily visualized in the form shown in Fig. 1. This processor is an example of *packet communication architecture*[7] in which many hardware units are connected by links through which information is sent as packets using a self-timed ready/acknowledge protocol. Here the packets are *operation packets* of the form

<opcode, operands, destinations>

and *result packets* of the form

<value, destination>

The Activity Store of the machine consists of Instruction Cells which hold a stored representation of the data flow program graph[8] to be executed. Result packets containing operand values arrive at Instruction Cells from the Distribution Network. Each Instruction Cell sends an operation packet to the Operation Section when all operands have been received. The function of the Operation Section is to execute instructions and to forward

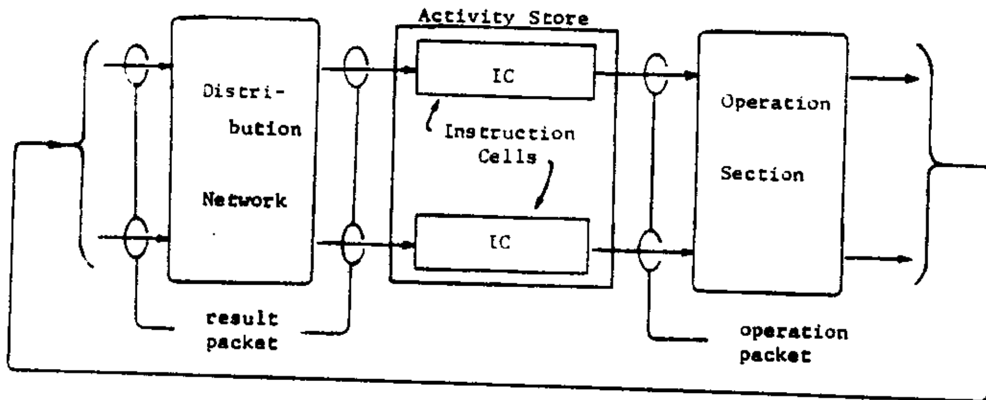


Figure 1. Data flow processor

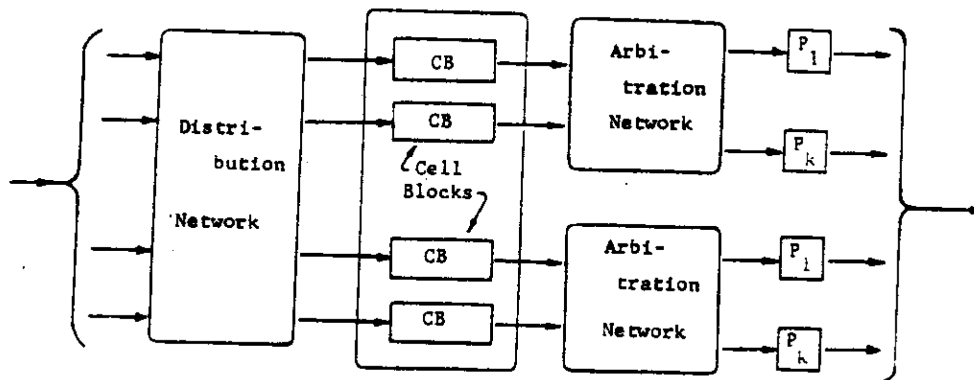


Figure 2. Practical form of the data flow processor

result packets to target instructions by way of the Distribution Network.

The data flow machine of Fig. 1 is impractical if the Instruction Cells are fabricated as individual physical units, since the number of devices and interconnections would be enormous in a machine having thousands of Instruction Cells. A more attractive structure is obtained if the Instruction Cells are grouped into blocks and each block realized as a single device. Such an Instruction Cell Block has a single input port for result packets, and a single output port for operation packets. Thus one Cell Block unit replaces many Instruction Cells together with the associated portion of the Distribution Network. Moreover, to further reduce the number of connections between Cell Blocks and other units, a byte-serial format for result and operation packets is used.

The resulting structure is shown in Fig. 2. Here, several Cell Blocks form a section of the machine served by a group of shared functional units $P_1, ..., P_k$. The Arbitration Network in each section of the machine passes each operation packet to the appropriate functional unit according to its operation code. The number of functional unit types in such a machine is likely to be small (four, for example), or just one universal functional unit type might be provided, in which case the arbitration network becomes trivial.

The Distribution and Arbitration Networks in Fig. 2 are examples of *packet routing networks*[3,12,14]. In general an N X N routing network accepts packets at N input ports and transmits each packet at the output port specified by a series of bits comprising a header or destination tag of the packet.

In a routing network for a data flow computer, the important properties are different from those of the cross-bar switch used in conventional multiprocessor systems. There, a prompt response by a memory unit to an access request from a processor is a necessity. In contrast, a packet routing network is one-way, and high throughput instead of small transit delay is the important desired property.

Prototype Construction Scheme

Many concepts for data flow computers are examples of packet communication architecture[5,9,10,16]. At MIT we are developing an approach to building prototypes of these systems.

The first prototype we plan to construct (Fig. 3) is a simplified form of the data flow machine shown in Fig. 2. Here a 4 X 4 routing network is realized by means of four 2 X 2 routing units, and each of the four Processing Elements (PE's) implements a Cell Block unit and a complete set of functional units for the supported machine language. Thus each PE holds a group of activity templates (instructions together with receivers for their operands),

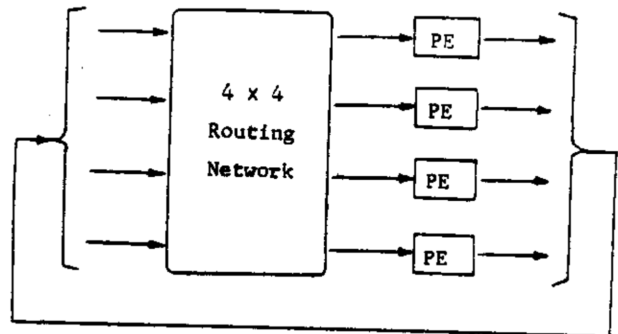


Figure 3. First data flow machine prototype.

receives result packets from the routing network, executes instructions as they become enabled, sending result packets to the routing network for distribution to specified PE's. This prototype may be expanded as desired by adding PE's and adding units to the routing network.

Since the purpose of the prototype is to serve as an *engineering model* to resolve design problems, to explore potential applications and to extrapolate performance prospects, flexibility is more important to us than speed of operation. In particular, we do not know a "good" design for the machine representation of data flow programs, nor do we know what algorithms will ultimately be used for code generation by a data flow compiler. For these reasons, we have chosen an approach to realization of the prototype that avoids making any commitment to a particular (data flow) machine language, or even to a specific division of the tasks of instruction execution among different hardware unit types. The 2 X 2 router unit performs such a basic function that a direct realization of its function will be used. The PE's are realized using a standard hardware unit containing a microprocessor together with program and data memory, which can be (micro) programmed to emulate any desired packet communication module. Our choice of designs for these two building blocks for prototype data flow machines are presented in the following sections.

The 2 X 2 Router and Routing Network Design

An N X N routing network supports packet communication between N source modules and N destination modules (which need not be distinct). The 2 X 2 router is a simple building block which can be used to construct classes of routing networks with different topologies and operational characteristics. We first discuss the design and implementation of the 2 X 2 router in more detail, and then illustrate its use in constructing two classes of routing networks.

A 2 X 2 router receives packets at its two input ports and delivers each received packet at one of two output ports according to a destination address carried by

the packet. Each packet is transmitted byte-serially between modules. Packet bytes are delivered and received using an asynchronous packet communication protocol. Each intermodule connection consists of a bundle of data wires and a pair of control wires (Fig. 4). Packet communication is synchronized by sending control signals over the control wires. Availability of a new packet at a connection is signaled by sending a *ready* signal over the ready wire, its receipt by returning an *acknowledge* signal over the acknowledge wire. To support variable length packets, packet boundaries in a byte stream are indicated by adding a *lastbyte* bit to each 8-bit byte. The *lastbyte* bit is on only for the last byte in each packet.

A 2 X 2 router is designed so that packets to be forwarded at different output ports can be processed concurrently (Fig. 5). Such concurrency is naturally supported by decomposing the router into two input modules (IM) and two output modules (OM) (Fig. 6). Each input module is a sequential machine which examines the destination address in each input packet, makes an output request to the specified output module and delivers the packet bytes when the request is granted. Each output module consists of a data multiplexer and an arbiter to resolve conflicts between output requests. A logic design of the 2 X 2 router module based on this modular structure has been completed. The asynchronous sequential circuits and arbiters used in this design are constructed using SSI

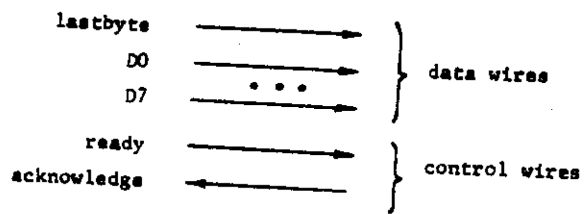


Figure 4. Hardware structure of a module connection.

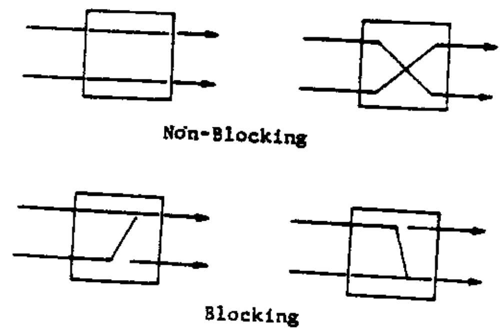


Figure 5. Parallel processing in a 2 x 2 router.

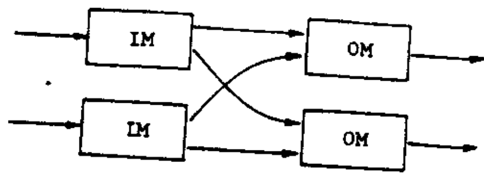


Figure 6. Structure of a router module.

components and discrete transistors. The data paths consist of MSI multiplexers and SSI logic gates.

We are also developing a custom LSI implementation of the 2 X 2 router since it is sufficiently well-specified, does not have an exuberant input/output pin requirement and will be used in quantities in network constructions. It is attractive to organize this implementation as an interconnection of self-timed hardware elements to improve the tolerance for variations in physical properties caused by non-uniformities in manufacturing processes and to simplify timing considerations in system integration and testing. An appropriate logic design has been derived methodically from a behavioral specification of the router to facilitate verification of functional correctness. The logic elements employed include the usual repertoire of logic gates, Muller C-elements, and 2-input/2-output arbiters.

Building a N X N *rectangular* routing network using 2 X 2 routers is done by the recursive construction illustrated in Fig. 7. Such a network has $\log_2 N$ stages each of which contains N/2 routers. The total number of routers employed is $(N/2) \log_2 N$. The path length between each pair of source and receiver modules grows uniformly as $\log_2 N$. All packets sent to a receiver module, independent of their sources, have identical destination tags. Routers in succeeding stages in the network examine successive bits in a destination tag to forward the packet along the proper path.

The N X N *triangular* routing network constructed as shown in Fig. 8 has rather different properties. The number of routers in this network grows as 2N-3. The destination address to be used in routing the packet through the network and the path delay incurred depend on both the source and the destination. Some processing elements are "closer" to each other in this network than to others and this proximity is preserved as the network grows.

We have included features in our 2 X 2 router designs to facilitate construction of these networks from identical units. First, note that if a destination address is no longer than eight bits, and if each router uses the first bit arriving on its D0 input to control packet switching, then correct routing by a *rectangular* network is accomplished by simply permuting the data wires D0 - D7 cyclically in the interconnection of successive router units. To

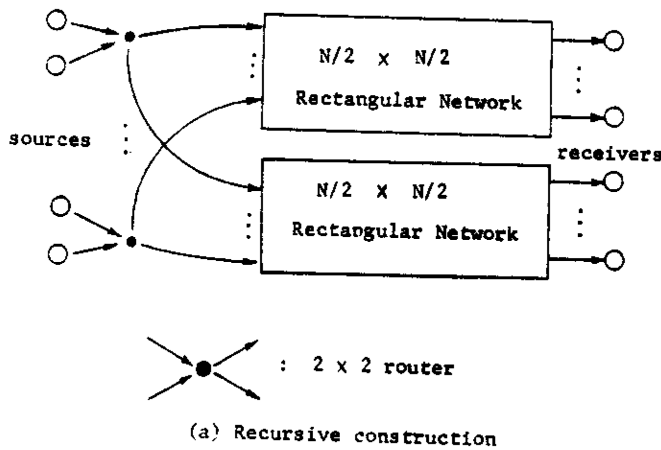
accommodate destination addresses more than eight bits in length, certain routers may be "programmed" by permanently asserting a control input which suppresses transmission of the first byte of each packet. In addition, we plan to provide some buffering of packets in our router chip since simulation has shown this to improve throughput, and the chip area required is nominal.

Path control in a routing network is distributed among the routers. There is no centralized control mechanism whose complexity must grow with network size and which may become a performance bottleneck. A rectangular or triangular network is capable of forwarding many packets concurrently to provide a high throughput. The average path length in both classes grows as $\log_2 N$. These characteristics are appropriate for supporting highly concurrent packet systems. The triangular network favors local communication within a subtree, but programs must be suitably structured to exploit this potentially higher bandwidth. A performance analysis for rectangular routing networks can be found in[3].

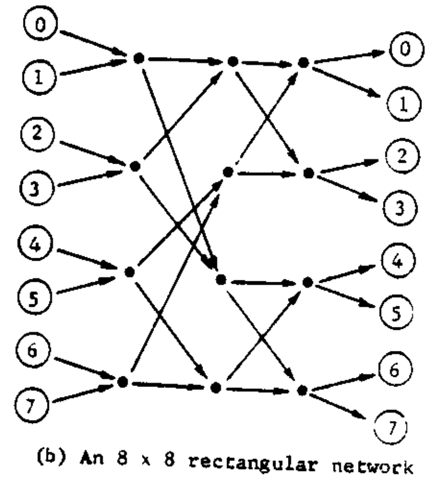
The Processing Element

The Processing Element (PE) is a hardware module which can be used to emulate any packet communication module. The PE will be used to emulate a Cell Block unit and a set of functional units in the first prototype configuration (Fig. 3), and will later be used to emulate a variety of packet communication modules required by more sophisticated data flow architectures. A typical prototype will consist of many PE's interconnected by one or more routing networks. To function in this manner, the PE must have two key characteristics; the flexibility to emulate a wide variety of packet communication modules and a wide variety of alternative behaviors for each module, and the capability to send and receive packets in the byte-serial format of the 2 X 2 router.

The PE should have the flexibility to perform a wide variety of tasks. Even in the first configuration the PE will perform both storage and arithmetic operations. Further, the exact specifications of the modules to be emulated by the

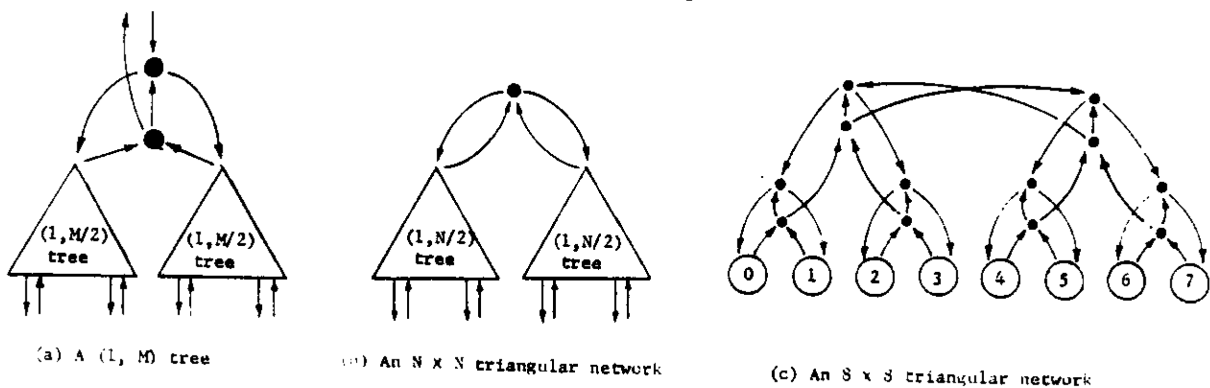


(a) Recursive construction



(b) An 8 x 8 rectangular network

Figure 7. Rectangular Routing Network



(a) A (1, M) tree

(b) An N x N triangular network

(c) An 8 x 8 triangular network

Figure 8. A triangular network constructed out of 2 x 2 routers.

PE have not yet been determined. Thus the PE should be a general purpose machine that can be programmed to emulate a particular packet communication module.

The PE should be capable of sending and receiving packets. In all applications, the PE will be emulating a packet communication module which processes packets in the byte-serial format adopted for the 2 X 2 router. For many of the modules, packet operations will represent a major portion of the work to be done. Thus the PE should be tailored to these packet processing operations.

The performance of the PE should be consistent with the intended use of the prototype facility to study the operation of various data flow machines. In particular the PE should be capable of a level of performance which allows the emulation of a data flow machine on the prototype to run much faster than a software simulation of that machine.

The control of malfunctions due to design errors and hardware faults in the PE is important. A problem in the prototype facility may be caused by errors in any of the following: the hardware, a module emulation program, the design of the data flow machine being studied, or the data flow program being executed. Thus it is important to be able to readily isolate and fix those problems caused by the hardware. In particular, the PE should be carefully designed for ease of construction and verification of its correct operation. In addition, the likelihood of a hardware fault during the operation of the PE should be minimized, and the PE should provide sufficient facility for easily diagnosing such a fault. To support these goals we feel it is important to keep the PE structurally simple.

We have completed a design for the PE based on the requirements outlined above. The data paths of that design are shown in Fig. 9. The design is basically a conventional microcomputer which has been extended to support the sending and receiving of packets. The overall approach of this design is based on the earlier work of Vishniac[15].

Bit-slice microprocessor chips were chosen for implementing the machine's data paths rather than a single chip microprocessor, and a microsequencer with a writable control store was chosen for the machine's control. This choice allowed the machine's data paths to be tailored to our particular packet processing requirements. In addition, this approach promised significantly better performance both by the speed of the chips in the data paths, and the capability of performing the emulation in microcode. In comparison to the large number of memory and interface chips required by the PE the number of chips required to implement the data paths from bit-slice components is small. Thus the additional board space and development cost required by the bit-slice components in comparison to a single chip microprocessor represented only a moderate increase in the overall system cost. The primary long term cost of the bit-slice design is the effort required to program in a low level micro language. However, since we intend to develop microcode for only a few modules at a time and

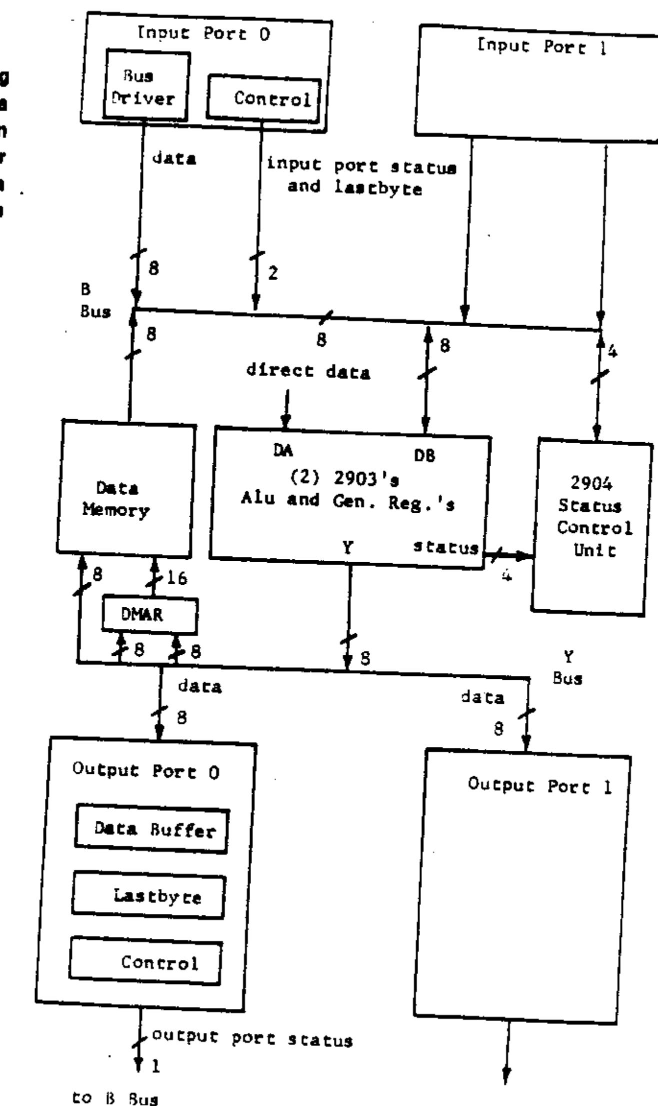


Figure 9. Data paths of the PE.

make extensive use of each micro emulation program, the additional effort necessary to program in microcode seems to be justified by the additional performance that will be gained. We are presently implementing a microassembler and simulator for the machine as programmer aids for writing and debugging microcode.

A horizontal micro instruction format was chosen to increase the flexibility of the machine. All important control lines in the processor were mapped into separate bits resulting in a rich instruction set. This large instruction set

allows the machine to achieve good performance on a wide variety of tasks. To further enhance the machine's flexibility we have chosen to use a 4K writable microstore which our studies indicate is sufficient for emulating even complex modules.

The PE has two sets of input and output ports capable of receiving and sending packets in the byte-serial format of the 2 X 2 router. We chose to make the packet ports very straightforward to achieve our desired goals of simplicity and reliability. Input and output operations on these ports are done under control of the microprocessor, and the microprocessor can determine which operation to perform by examining the various port status bits. It should be noted that the microprocessor is a synchronous system. The incoming asynchronous control signals, the *acknowledge* signal for each output port and the *ready* signal for each input port, must be synchronized to the microprocessor clock. To accomplish this each of these signals is strobed into a separate flip-flop at the beginning of each clock cycle. The outputs of these flip-flops are not used until the end of the clock cycle. The length of the clock cycle and the speed of the flip-flops are such that the chance that a flip-flop has not settled by the end of a clock cycle is much less that the chance of some other hardware fault such as a chip failure.

The input ports and output ports require a minimum of hardware. Each output port contains a data buffer, a *lastbyte* flip flop, and supporting control logic. The microprocessor can in one instruction load the data register and the *lastbyte* flip flop. The loading of these registers sets up the signals on the data lines of the corresponding output connection, and causes a *ready* signal to be generated. Once the *acknowledge* signal is received, the control logic of the port clears the *ready* signal and sets the appropriate port status bits. An input port contains a data bus driver and control circuitry. The microprocessor can read data from a particular input connection by enabling the corresponding port's bus driver. The reading of a port causes an *acknowledge* signal to be generated for the corresponding connection. The rest of the control cycle is handled by the control logic of the port in a manner similar to that described above for an output port. Other possible port designs which we examined included ports with the capability to interrupt the microprocessor and ports with the capability of doing direct memory access. While these other designs offered higher performance, they also required additional control circuitry at each port. Since simplicity and reliability were higher priorities than performance, we chose to use the simpler status-oriented port design.

The PE has been designed as a byte oriented machine in order to match the byte-serial format of the packets which it must process. For the majority of modules which the PE will emulate, byte oriented input and output operations predominate over multiple byte arithmetic operations. Thus the simplicity of the data paths and the micro instruction set obtained by supporting only byte operations was more important than the performance in arithmetic operations lost by not supporting multiple byte operations in hardware.

The PE has been designed to interface with the bus of an external supervisory minicomputer. The supervisor will be used to load programs and data into the PE, to control the operation of the PE, and to perform maintenance tests on the PE. The interface gives the supervisor access to the data and micro instruction memories of the PE. In addition, the interface gives the supervisor the capability to halt and single step the PE as well as directly access all the registers of the PE. The interface has been designed to only respond to bus commands with a particular bus address (programmed by switches). This allows many PE's to be controlled by a single supervisor machine.

The first proposed application of the PE, emulation of a Cell Block unit and a set of functional units for the configuration of Fig. 3, has been studied in some detail by Feridun[11]. The proposed emulation program consists of a supervisory routine and subordinate routines for the Cell Block and functional units. First versions of the supervisory and Cell Block routines have been written. The Cell Block program is capable of emulating 1000 instruction cells in 32K bytes of data memory. Further, there seems to be no difficulty in coding the additional routines required for this application, or in placing all of these routines in the 4K microinstruction store.

Remarks

In conclusion, a few remarks are in order. First, the reader may have observed that there will be a considerable performance mismatch between the throughput of the router and the packet processing rate of the PE. For the purpose of our engineering prototype we accept this disparity as a price for the flexibility we desire. Once unit designs, such as for a Cell Block module, have been checked by emulation using PE's, the way is open to investing in realizations (using PLA's or custom LSI, for example) having performance in better balance with the router module.

An immediate problem is the choice of instruction set to be emulated by the first prototype. We expect to implement a code generator that will produce machine code for the prototype from programs written in a suitable restriction of the VAL programming language[1].

Another form of data flow machine we expect to study is the data flow multiprocessor[10], which appears to be well suited to certain applications such as the Navier-Stokes models of airflow of interest to NASA. A more ambitious objective is to emulate a general purpose data flow machine[9] using routers, PE's and an additional module appropriate for emulating a *structure memory* to hold program and data base representations.

Acknowledgment

T. L. Tung worked on the logic design for the 2 X 2 router. W. B. Ackerman contributed to the design of the processing element. We are also grateful to Prof. C. L. Seitz of Caltech for the arbiter circuit used in the 2 X 2 router.

References

[1] W. B. Ackerman and J. B. Dennis, *VAL: A Value Oriented Algorithmic Language, Preliminary Reference Manual*, Laboratory for Computer Science, M.I.T., Technical Report TR-218 (June 1979), 80 pp.

[2] Arvind and R. E. Bryant, *Parallel Computers for Partial Differential Equation Simulation*, Laboratory for Computer Science, M.I.T., CSG Memo 178 (May 1979), 10 pp.

[3] G. A. Boughton, *Routing Networks in Packet Communication Architectures*, Dept. of Electrical Engineering and Computer Science, S.M. Thesis, M.I.T. (June 1978), 93 pp.

[4] M. Cornish, Private communication, Texas Instruments Company, Austin, Texas.

[5] A. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," *Proceedings of the ACM 1979 National Computer Conference* (June 1979), pp. 1079-1086.

[6] J. B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science, 19*, Springer-Verlag (1974), pp. 362-376.

[7] J. B. Dennis, "Packet Communication Architecture," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing* (August 1975), pp. 224-229.

[8] J. B. Dennis, C. K. C. Leung, and D. P. Misunas, *A Highly Parallel Processor Using a Data Flow Machine Language*, Laboratory for Computer Science, M.I.T., CSG Memo 134-1 (June 1979), 33 pp. Submitted to the *IEEE Transactions. on Computers*.

[9] J. B. Dennis, and K. Weng, "An Abstract Implementation for Concurrent Computation With Streams," *Proceedings of the 1979 International Conference on Parallel Processing* (August 1979), pp. 35-45.

[10] J. B. Dennis, "The Varieties of Data Flow Computers," *Proceedings of the First International Conference on Distributed Computing Systems* (October 1979), pp. 430-439.

[11] A. M. Feridun, *Data Flow Cell Block and Functional Unit Emulation*, Report in progress, Laboratory for Computer Science, M.I.T.

[12] A. Hopper and D. J. Wheller, "Binary Routing Networks," *IEEE Transactions. on Computers*, Vol. C-28, No. 10 (October 1979), pp. 699-703.

[13] P. Ressler, *Simulation of a Highly Parallel Processor*, Dept. of Electrical Engineering and Computer Science, S.B. Thesis, M.I.T. (January 1979), 40 pp.

[14] A. R. Tripathi and G. J. Lipovski, "Packet Switching in Bayan Networks," *Proceedings of the 6th Annual Symposium on Computer Architecture* (April 1979), pp. 160-167.

[15] E. Vishniac, *A Processor Module for Data Flow Computer Development*, Laboratory for Computer Science, M.I.T., CSG Memo 176 (May 1979), 55 pp.

[16] I. Watson and J. Gurd, "A Prototype Data Flow Computer With Token Labelling," *Proceedings of the ACM 1979 National Computer Conference* (June 1979), pp. 623-628.