MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
CAMBRIDGE, MASSACHUSETTS 02139

COMPUTATION STRUCTURES GROUP MEMO 222

THE MIT DATA FLOW ENGINEERING MODEL

by

JACK B. DENNIS
WILLIE Y-P. LIM
WILLIAM B. ACKERMAN

(An abbreviated version of this paper was published in the Proceedings of the IFIP 9th World Computer Congress, Paris, France, September, 1983).

This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and the National Science Foundation under grant number MCS-7915255.

NOVEMBER 1982

The MIT Data Flow Engineering Model

Jack B. Dennis
Willie Y-P. Lim
William B. Ackerman

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139.
November 1982

Abstract

Data flow computers differ sufficiently from conventional computers that new concepts of program structure, and new programming languages and methodology are needed for their effective utilization. At the MIT Laboratory for Computer Science, we have constructed an engineering model consisting of eight processing units coupled by a packet communication network built of two-by-two routers. This facility is being used to study issues and evaluate alternatives in the design of high performance data flow computing systems. Principal aspects being addressed are: (1.) development of good machine code structures for computations basic to numerical physics problems, especially structures capable of handling massive data bases composed of large arrays of numerical data; (2.) the design of instruction sets suitable for data flow processing elements; (3.) evaluation of packet-switched interconnection networks; (4.) the methodologies to be used for designing the custom devices required in a prototype machine and providing the level of fault tolerance to achieve reliable system operation.

1. Introduction

It is generally agreed that any substantial increase in the computational capacity of high performance computers will require the exploitation of parallelism using to advantage the large density of circuits possible with advancing integrated circuit technology. Conventional approaches using overlapped instruction execution, vector processing architecture, or large coordinated arrays of processing elements have reached the

---

1. This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and the National Science Foundation under grant number MCS-7915255.

limits of their potential and have failed to provide resolution of the problems of programmability. We have advanced data flow concepts [3, 6] of computer architecture to address the challenge of producing the next generation of high performance machines.

In a data flow computer, instructions of the stored program are activated by arrival of the data values on which the instruction is to act. The machine's memory contains instruction cells, each containing an instruction and space to receive operand values. There is no program counter in a data flow computer; since instructions are activated by data, many instructions may be available for execution at once, allowing for a high degree of concurrency in program operation. The limit on the level of parallelism that can be achieved is that available in computations to be run. We have found that many important problems offer the parallelism of computation needed to fully utilize a high performance data flow computer.

Because data flow computers are a radical departure from conventional computer architectures, new tools of programming and new design methodologies are required. Major issues being addressed are devising strategies or algorithms for translating data flow programs expressed in an applicative programming language into efficient machine level program structures; and developing practical hardware designs embodying the principles of packet communication architecture. The question of what features for fault tolerance should be built into the architecture to achieve the desired degree of reliability must also be addressed. The MIT Data Flow Engineering Model has been constructed to support the study of these and other issues.

We begin in Section 2 with a brief review of the architecture we have proposed for high performance data flow computers. The engineering model constructed to study and evaluate various software and hardware aspects of data flow computers is described in Section 3. The software systems we have developed to control and monitor operation of the engineering model, and to translate and link data flow source programs written in the applicative language VAL are outlined in Section 4. The engineering model has been useful for studying a number of implementation issues. Three of these areas, discussed in Sections 5 through 7, are: the design and generation of run-time structures for data flow programs; evaluation of design methodologies for asynchronous systems; and development of strategies for testing and fault diagnosis.

2. Data Flow Overview

In a data flow computer, operations of a stored program are coded in instruction cells containing an opcode, fields for storing operands, destination fields, and control information. When an instruction cell is executed or "fired", one or more result packets are sent to target instruction cells specified by the destination fields. Since each cell holds only one set of operand values, the stored program must be structured so that an instruction cell can fire only when its target instruction cells have space for storing the result values. For this

purpose, each instruction cell may be fired only when it has received an acknowledge signal from each instruction cell that received its previous result values. A count of acknowledge signals received and the value to which this count must be reset are part of the control information in each instruction cell.

A data flow program, represented as a collection of instruction cells, is essentially a directed graph with instruction cells as the nodes, and an arc for each destination field of an instruction cell. Such a graph is termed a data flow graph. Figure 1 shows an example of a data flow graph for a very simple data flow program computing the sum of the two products $A*B$ and $C*D$. In the firing of instruction cells, the result values being sent can be viewed as "tokens" or packets passed along the arcs. In the figure it can be seen that after the firing of the two multiplier instruction cells, tokens are presented to the addition instruction cell which then fires to produce the desired sum.

Many forms of data flow machines have been proposed and several experimental models have been

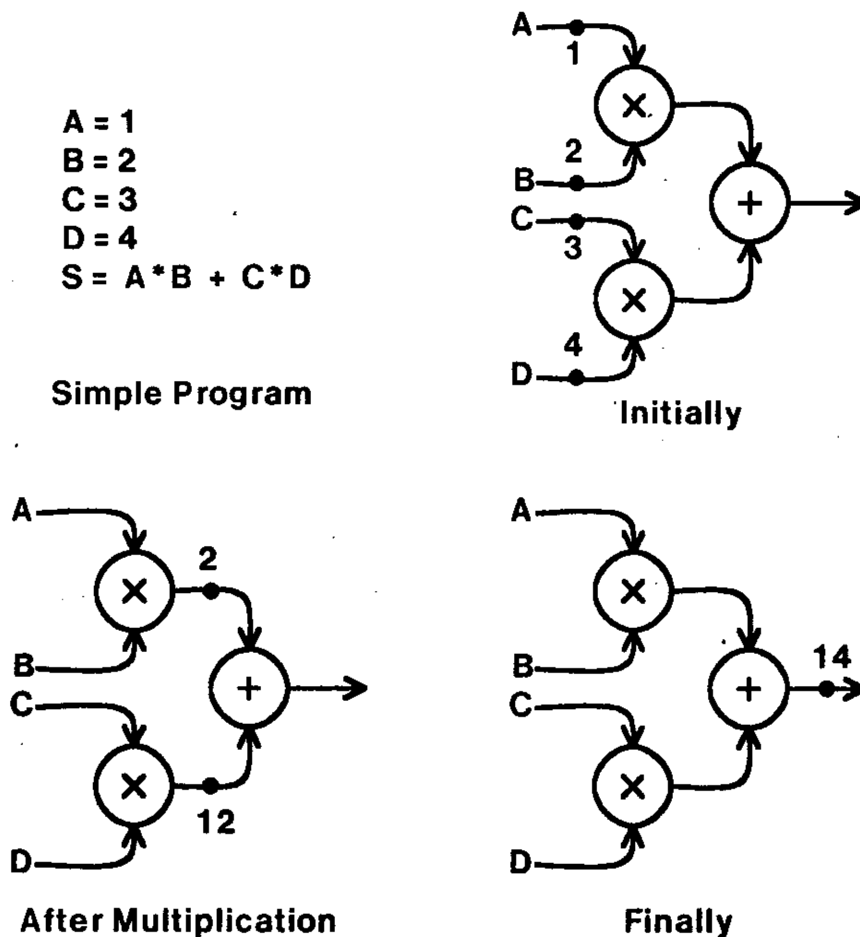


Figure 1. An Example of a Data Flow Graph

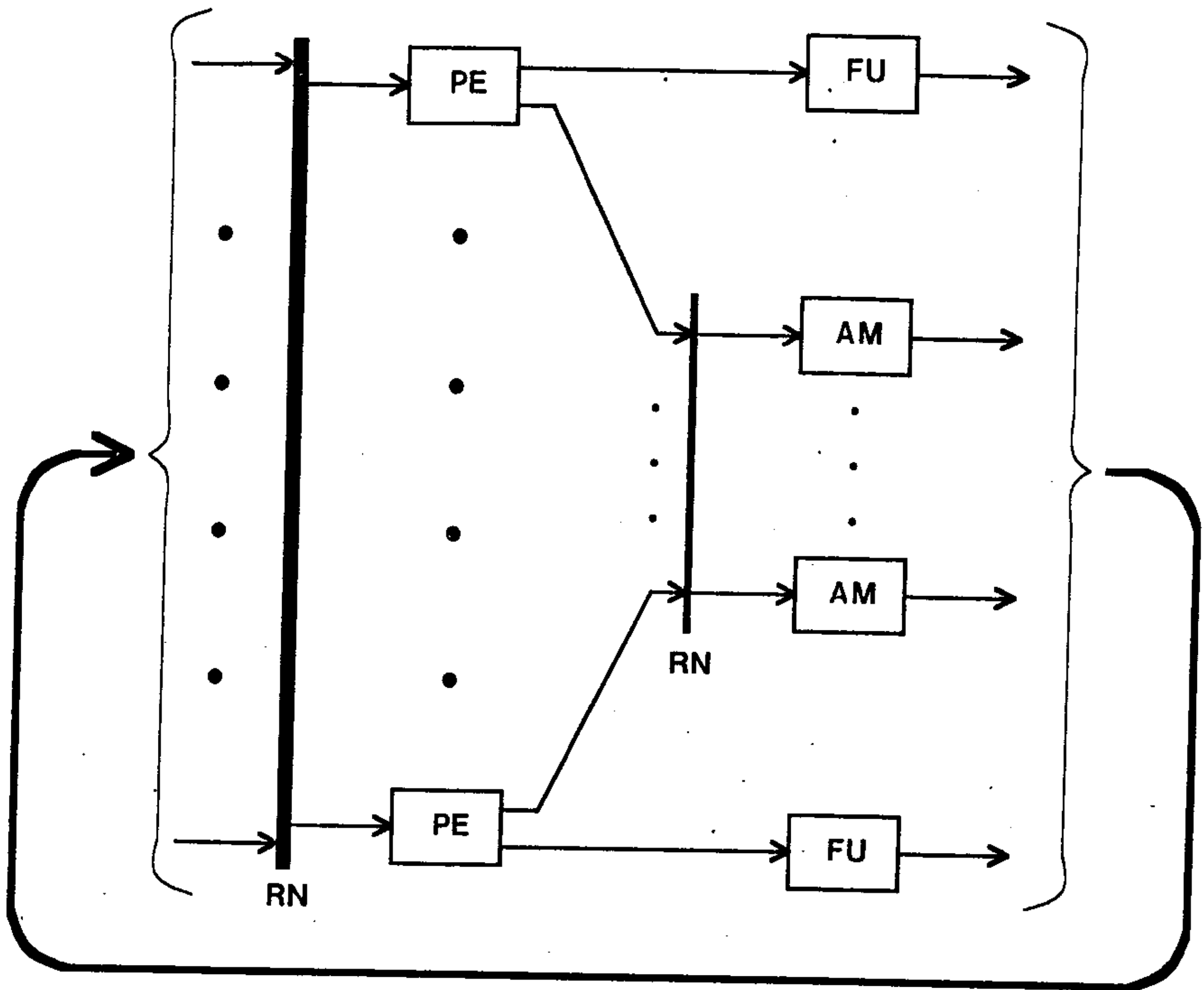


Figure 2. A Large Scale Data Flow Machine

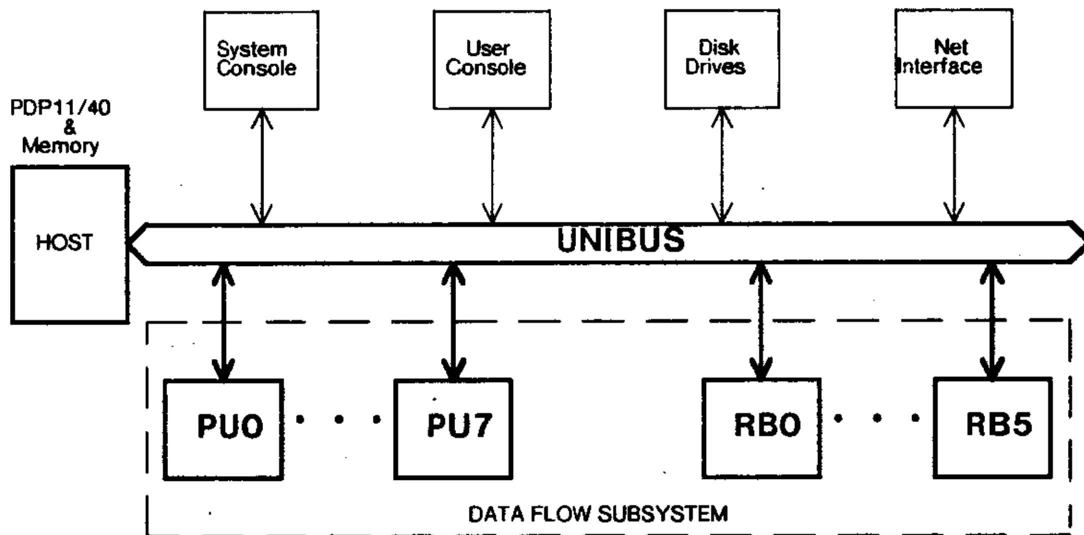
built [5]. The form of data flow computer we have advocated for high performance computation is shown in Figure 2. In this physical structure, there are four basic kinds of components – processing elements, functional units, array memories, and interconnection networks. The processing elements hold instruction cells, determine when cells are ready for execution, and perform the simplest instructions. The functional

units perform the complex scalar operations including floating point operations. The efficient handling of large arrays of data is supported by the array memories. These components are joined by interconnection networks that forward information packets from unit to unit by a "store and forward" protocol. The data flow engineering model is designed for emulating this and other similar architecture.

3. The Engineering Model

The engineering model has the configuration shown in Figure 3. A PDP 11/40 mini-computer serves as "host" to a data flow subsystem consisting of eight identical processing units (PUs) connected by an 8-input, 8-output (i.e. 8 × 8) packet routing network. All components of the data flow subsystem, shown in Figure 4 are on the host's UNIBUS. The communication among PUs and routers is asynchronous and packet oriented while that between the host and the data flow subsystem is synchronous and bus oriented. The design of the PU, routers, and routing network for the engineering model has been discussed in [7].

The host performs a number of supervisory functions. It loads the PUs with programs and data. The states of the PUs can be controlled by the host for debugging and normal operations. Once programs are loaded, the host may start the PUs and continuously monitor them until they indicate they are done. The host



PU0, . . . , PU7 : Data Flow Processing Units

RB0, . . . , RB5 : Router Boards (2 Routers per Board)

Figure 3. Architecture of the MIT Data Flow Engineering Model

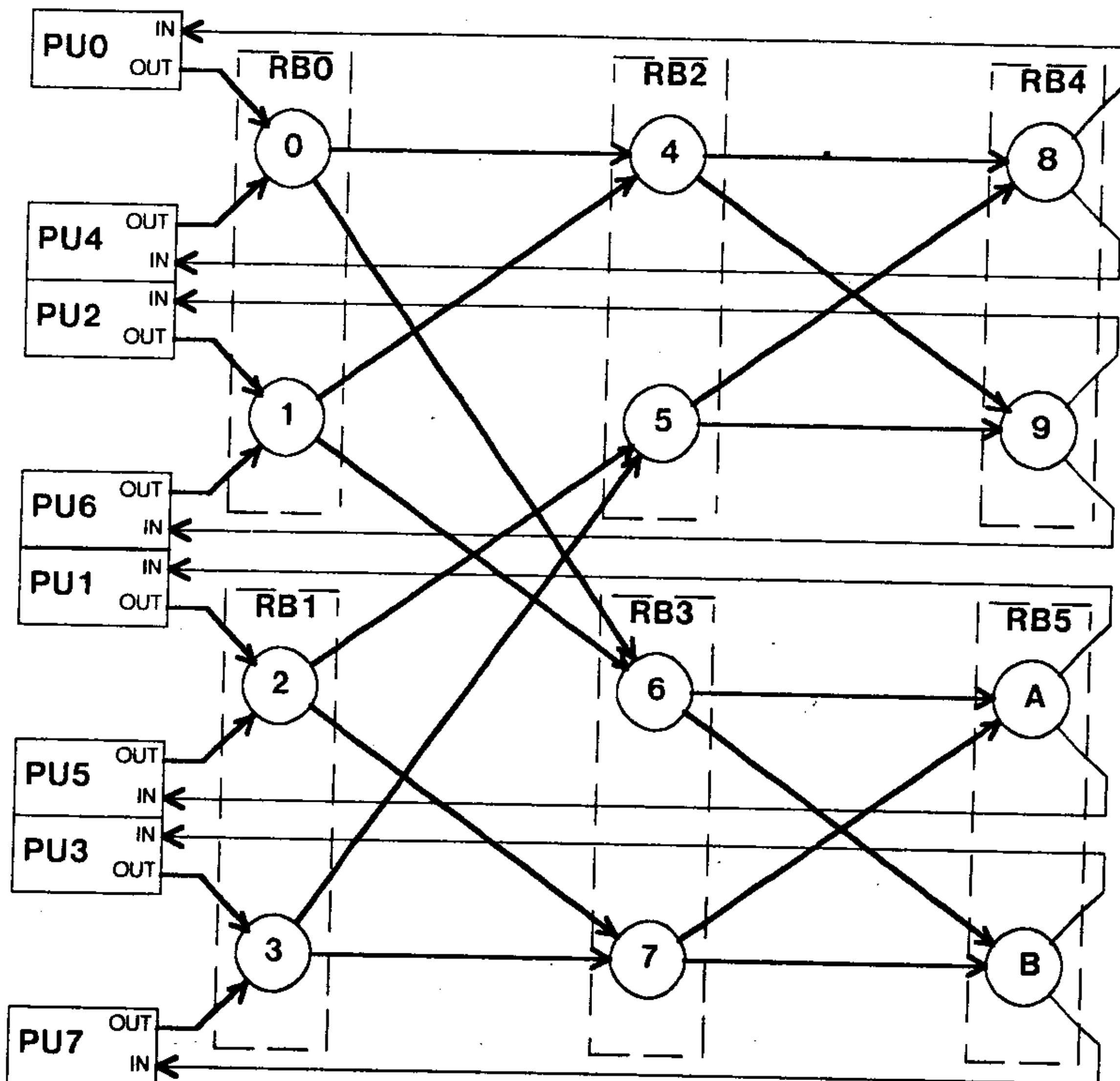


Figure 4. Interconnection Structure of the Data Flow Processors

can read data from PUs and this is how the results of data flow computation are returned to the user. The host can also set the states of the routers for testing purposes. These capabilities are used by the host during the running of diagnostic programs for the PUs and the routing network.

3.1 Processing Units

The PUs are general-purpose microprocessor-based computers capable of performing the usual manipulations of data in their scratchpad registers, the data memory, and the communication ports. Each PU is composed of a microprocessor (built from AM 2903, 2904 and 2910 chips) with 16 8-bit scratchpad registers (forming the processor proper), 4K of 40-bit words of program memory, 64Kbytes of data memory, two pairs of 8-bit wide communication ports, and a UNIBUS interface. Figure 5 shows a simplified architecture of the PU. The program memory holds the microcodes for the PU. The data memory is used for storing the information used by the emulated data flow processing element. The communications ports are used for simulating byte serial packet transmission.

The flexibility of the processing units allows for quite a variety of emulations. By writing suitable microcode, one could emulate data flow computers (or other systems) in any way one desires.

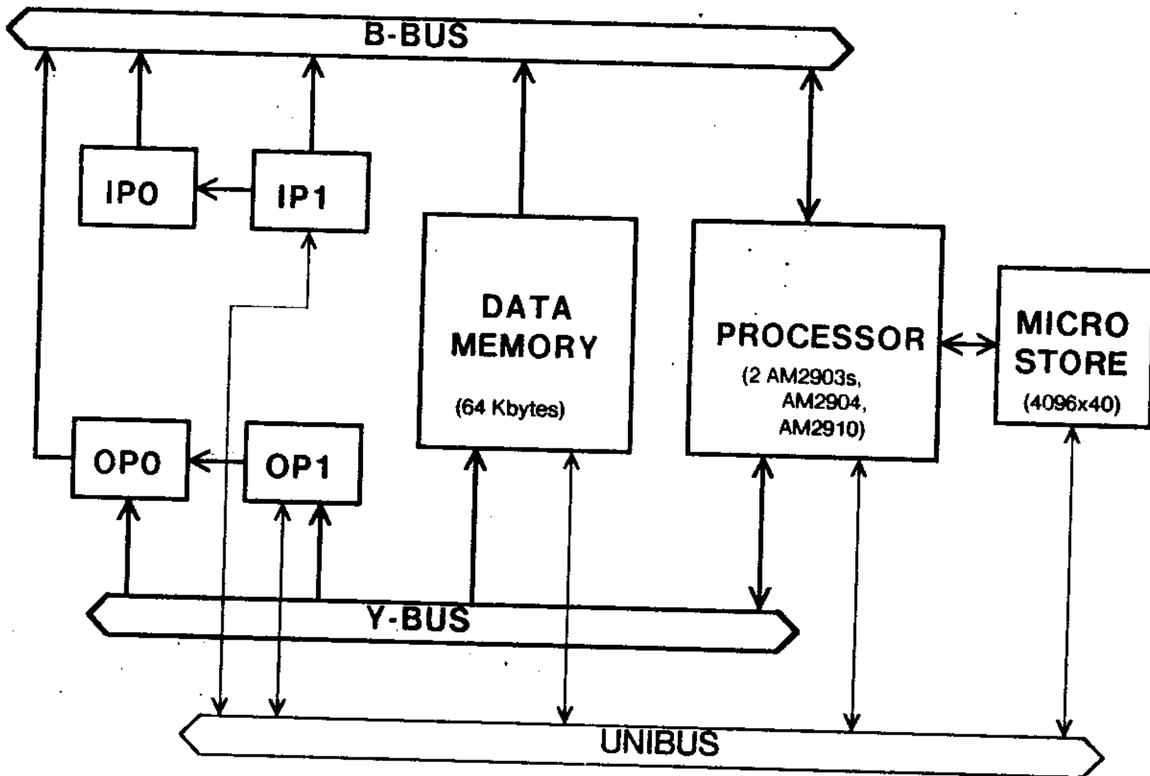


Figure 5. Structure of the Data Flow Processing Unit

3.2 Routing Network

To connect the PUs, we use an interconnection of 2-input 2-output (i.e. 2 × 2) switching elements or routers. Transmission of packets through the routers is byte serial. Each byte of the packet is tagged with an extra bit indicating whether the byte is the last byte or not. We refer to the 8 data bits and the last byte bit as a *packet byte*. The communication protocol implemented uses two control signals: a ready signal is sent by the source to the destination to indicate arrival of the byte; and an acknowledge signal is sent by the destination to

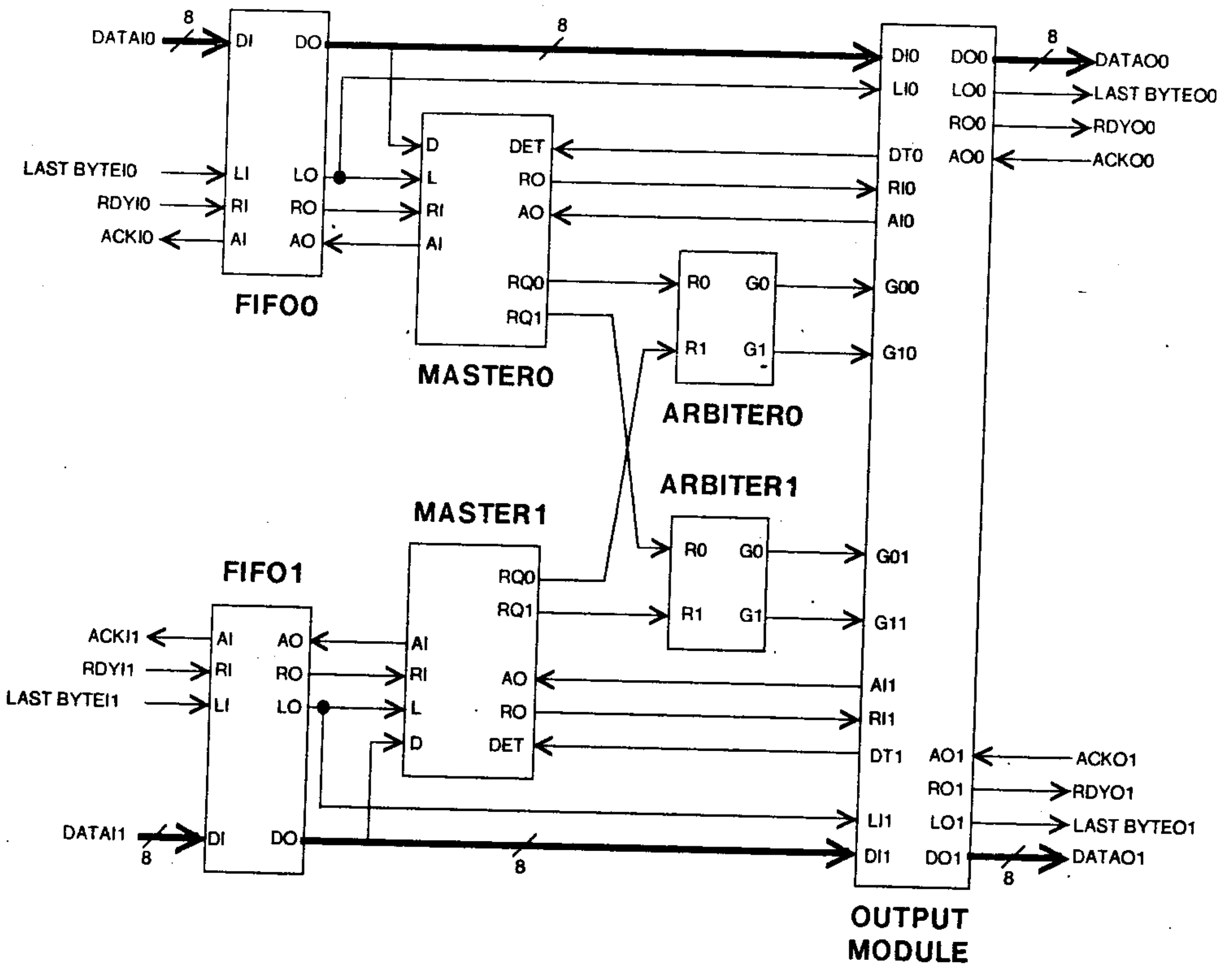


Figure 6. Structure of the 2 × 2 Router

the source to acknowledge the receipt of the byte. Hence the data paths between ports of the routers are comprised of 11 wires. The protocol works as follows. The ready and acknowledge lines are initially low. The packet byte is then asserted on the data wires and the ready line is set high to indicate that a packet byte is available. Once the destination has acted on the data byte, it sets the acknowledge line high. In response the source sets the ready line low. The destination then indicates readiness to receive the next packet byte by resetting the acknowledge line.

Each router is an asynchronous circuit that routes packets, transmitted byte serially, from an input port to an output port. Figure 6 shows the structure of the router. The router establishes a link for packet transmission between one of its input ports and one of its output ports. Since packets may differ in length, the link must remain established until all bytes of the packet have been transmitted. Each input port has a buffer (the FIFO module) for holding 64 incoming packet bytes. Also associated with the input port is a asynchronous state machine (the MASTER module in Figure 6) with six states: the initial state where the input port is not connected to any of the output port; two states for each output port (giving a total of four states) for the cases where the packet byte being sent through is or is not the last one for the packet; and a sixth state which is used when the packet length is greater than one. A request to establish a link is sent on receipt of the first packet byte (containing the destination address) by the MASTER module to the appropriate output port ARBITER module. The output port selected is determined by the value of the least significant bit of the first packet byte. In the case where there are two competing requests for the same output port, only one of the input port is granted access to the output port. Meanwhile packet transmission through the other input port is delayed until the output port is available. Based on the grant signal generated by the ARBITER module, the appropriate input and output ports are connected via the OUTPUT module. Note that if there is no contention for the same output port, packet transmission can proceed concurrently for the two pairs of input and output ports.

The routing network used has the structure of a baseline network [20] as shown in Figure 4. For an 8×8 network, there is a total of 12 routers organized as 3 columns of 4 routers each. To completely address all the 8 output ports, 3 bits are needed. Hence only the least significant 3 bits of the destination address byte of the packet is used. At the i-th stage, for $i=0$, 1 and 2, the least significant i-th bit of the destination address byte is used. Since each router selects an output port based on the value of the least significant bit of the destination address byte, rotation of the byte must be performed from one stage to the next. Furthermore the address byte must be rotated back to its original state before it leaves the network. To accomplish both forms of bit rotation, the OUTPUT module of each router can be set to rotate all bytes through it by a fixed amount. Hence for stages 0 and 1 the bits are rotated to the right by 1 while at stage 2, the last stage of the network, the bits are rotated back (i.e. to the left) by 2 to compensate for the bit rotations by the earlier stages. This simple

scheme will work if the total number of bytes used does not exceed 8, limiting the network size to be 256 × 256. Extension to larger networks can be accomplished by using a larger byte, by dropping address bytes once used up, or by designing the 2 × 2 router to allow testing of the second byte.

4. Software System

The PUs and routers are all I/O devices on the PDP-11/40 computer, which uses the UNIX [15] operating system. All operations involving the PUs and routers are supervised by programs running on the PDP-11 host. These programs include extensive hardware diagnostics for the PUs and routing network, and the program *APPLY* that supervises a data flow emulation.

The majority of the software involved in data flow emulation, however, runs on a DECsystem-20 with the TOPS-20 operating system. This software, all written in CLU [13], includes the micro-assembler for the processing units [1] and a variety of translation, optimization, simulation, and diagnostic programs for manipulating VAL programs. Files produced on the DECsystem-20 are sent to the PDP-11 via a local communication network.

Several projects are studying the automatic translation of VAL programs into instruction cell templates, and some studies of hand generated code for certain benchmark applications are being carried out. The supporting software tools are shown in Figure 7. The program *VAL* performs syntax and data type analysis of the source program and produces a parse tree file. The *VALSYS* program can directly interpret this, if desired, to check functional correctness of the source program. The graph generator (actually several programs) converts the parse tree file into instruction cells and codes the result into text in standard format for transmission to the PDP-11 host machine. In addition to these programs, all running on the DECsystem-20, there are also the microcode assembler *ASM* and a simulator *SIM* for the PUs. All microcode for the PUs is assembled on the DECsystem-20. After translation into a standard textual form, they are transmitted to the PDP-11.

On the PDP-11, the utility programs *ULOAD* and *APPLY* load the files into microstore and data memory, and monitor processing unit operation. Once a processing unit is loaded and started, it emulates the cells in its data memory, listening for incoming packets containing data or acknowledge tokens. These packets can come either from the routing network (that is, from another PU) or from the PDP-11. The latter path is for the initial arguments to the program.

APPLY prompts the operator for the arguments to the VAL program, using information contained in the instruction cell file, formats those arguments into packets, and sends the packets to the appropriate PUs. The tokens then flow through the data flow graph under control of the PUs. When the final results have been

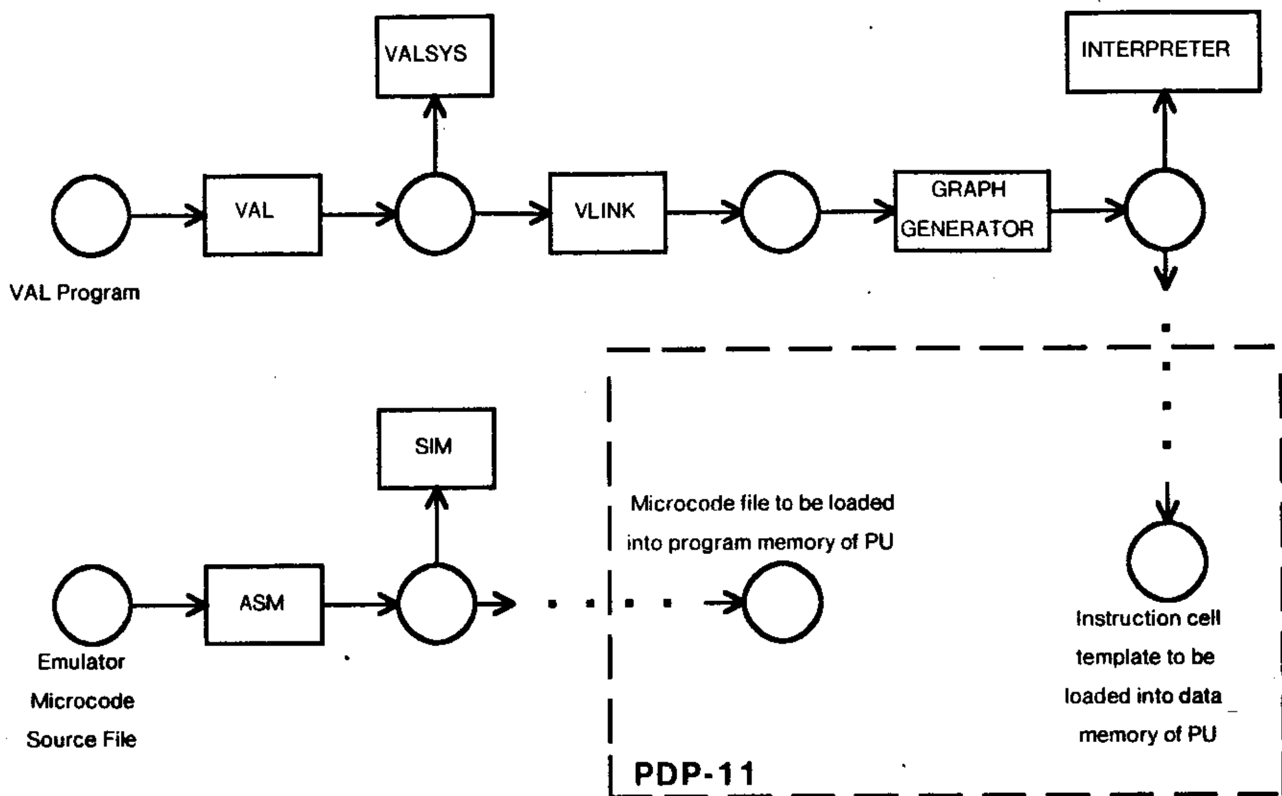


Figure 7. Software System

computed, the PUs send packets to the PDP-11. *APPLY* decodes these packets and displays the results, along with statistics that it reads from the PUs' data memory.

5. Run-time Structures for Data Flow Programs

The engineering model is being used to explore several techniques for structuring data flow programs to exploit concurrency and to process large collections of numerical data. So far these projects use common data formats (described below) for instruction cells, and use the same rules for cell firing and the execution of non-array instructions.

5.1 Instruction Cells

In the standard emulation, each instruction cell is represented by a 32 byte block of data memory, containing a part (the "template") that does not change once loaded, and a part that is manipulated by the PU. The former includes the operation code, information indicating presence and values of constant operands, the destination addresses of cells to which the result of a firing should be sent, and the number of acknowledgments that must be received before the cell is ready to fire again. The latter includes the values of the operands that have been received (4 bytes, or 32 bits each), the presence of received operands, and the number of acknowledgments still being awaited. A typical instruction cell is shown in Figure 8.

Cells become enabled as a consequence of receiving operand or acknowledgment tokens, either from the routing network or from other cells in the same PU. When this happens, the cell is added to the end of the queue of enabled cells. The microcode services this queue, executing the instruction in the cell at the head of the queue. It then sends the results to the indicated destinations, either through the routing network for cells in other PUs, or directly to the cells for "local" destinations.

5.2 Implementing Arrays in Data Flow Processors

Several techniques for handling arrays in data flow programs are being explored using the engineering model. These involve dedicating one or more PUs to act as "array memory servers" loaded with special microcode, and possibly altering the structure of the routing network. These projects will involve specialized instruction cell operation codes, possibly with unusual firing rules, and special translation techniques for producing the necessary instruction cells from the source program.

One such project [8] employs transmission of arrays from one part of the program (the "producer") to

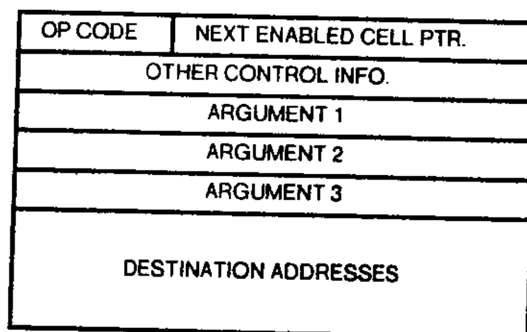


Figure 8. A Typical Instruction Cell

another part (the "consumer") as sequences of values transmitted serially. This is possible when the array elements are produced and consumed in regular sequential order, as is often the case. This approach to array computation is motivated by the analysis of benchmark programs in hydrodynamics and weather forecasting [18]. The type of array storage actually required in such a case is a "FIFO buffer pipeline", and operation codes have been developed to simulate such a pipeline by sending tokens to the memory server for later retrieval.

Another project to be implemented in the future involves array operations which, at the level of the emulated data flow computer, closely resemble the high-level array operations of the VAL language. The array "tokens" in this case will be pointers to arrays allocated dynamically by the server. Memory management will be performed by using reference counts on all arrays. The efficiency of this scheme will be enhanced by interleaving the arrays, and appropriate transformations will be made to the VAL program to handle the interleaved "slices" efficiently [2].

6. Design Methodology for Asynchronous Systems

In the engineering model, the data flow subsystem is composed of 8 independently clocked data flow processors. As the number of processors increases, it is infeasible to implement a data flow system running on a single clock. The large distance between processors will severely limit the speed of the clock and hence the system performance if the synchronous approach is taken. If the system components are independently clocked, then every data transmission between these components have to be synchronized with respect to the clock of the receiving component. This can lead to synchronizer failure [4] where the receiving party can at times fail to detect all the arrivals of such externally generated data. The higher the data or clock rate the more likely it is for synchronizer failure to occur. On the other hand if the system is totally asynchronous, there is no need for such synchronization as no clocks are used. The only problem of a similar nature that can occur in asynchronous system is the arbitration of signals competing for the same resource – e.g. the arbitration performed by the ARBITER module of the 2×2 router described in Section 3.2. The time it takes to perform the arbitration is unbounded though the mean arbitration time is small. However this problem only arises when there is contention of requests. For our form of data flow system, arbitration is required in the routing network. Furthermore arbitration occurs less frequently than synchronization for multi-clock systems. This is because, for independently generated data, the data rate at each input port of a router is much higher than the rate of occurrence of contentions for an output port in the router. The above reasons have led us to explore asynchronous systems for implementing data flow machines – for example multi-clock systems, asynchronous systems, self-timed schemes [16] or "stoppable" clock schemes.

The design of asynchronous systems on the scale of large data flow computers requires new approaches and methodologies. By restricting the set of primitive building blocks to be a small, well understood and correctly implemented collection of elements, a methodology [10, 17] can be developed for the design of large, correct asynchronous systems built from these primitive blocks. One methodology we are developing starts with a high level description written in the language PADL[12] which is designed for the structural and functional specification of packet communication systems. PADL has been used for specifying asynchronous versions of the 2 × 2 router and parts of the processing unit. We have implemented the routers of our engineering model as asynchronous systems built from TTL parts, and an exploratory router chip has been designed for NMOS fabrication.

Schemes using stoppable clocks have been investigated by a number of people [14, 16, 19]. Such an approach is a compromise between the familiar and cost-effective synchronous approach and the radically different, asynchronous approach. Stoppable clocked systems seem to have the attractive features of both approaches. Using stoppable clocks, system components can be designed as synchronous finite state machines and hence can benefit from the vast experience that has been accumulated in the field. Finally, synchronization schemes are possible using stoppable clocks that avoid the possibility of synchronizer failure that is always present in multi-clocked system.

7. Test Strategies and Fault Diagnosis

The use of asynchronous circuits requires new approaches in test strategies. Restricting the class of faults to single stuck-at faults, a test strategy has been developed for testing the router [11]. Such faults can cause errors in the packet data value, and failures in the communication protocol. Errors in data value can be detected by providing redundancy in the data transmitted such that the redundant data can be used for checking the integrity of the data sent. Failures in the communication protocol are due to either the source or destination failing to send the proper signal (ready or acknowledge signal), or either of them failing to detect signals sent by the other party. Such failures are detected using a time-out mechanism. A fault in the last byte bit may have both kinds of effects. The fault can be viewed as occurring in the packet byte but it can cause either the fragmentation of a packet into smaller packets or a packet with no last packet byte. All these have undesirable effects on the operation of the routing network. A fault diagnosis strategy has been developed and implemented for the network of the engineering model.

The original design of the router does not provide facilities for testing and fault diagnosis. Such facilities have been added and implemented in the engineering model. All the facilities are available to the host. They include capabilities for router initialization which is important for the proper initialization of the

whole routing network. Since the routers are asynchronous, those at the preceding stages of the network must be initialized before those at the current stage. Otherwise the uninitialized outputs of the earlier stages will cause the routers being initialized to be set in some unknown and very often erroneous states. To aid in the debugging of the router, each of its modules can be isolated for testing purposes. The test strategy of the network very often calls for the establishment of privileged paths. For example if a router is suspected of being faulty, privileged paths are used to send test packets through it and to receive its output packets, if there are any. Very often in checking the integrity of established paths, packets are sent to those output ports of the routers that have already been connected to the input ports. The arrival of the new packets should not cause the established paths to be broken. To do this test, each router can be set to a special mode where the flow of packets through either input ports can be interrupted. Subsequent packet flow through the interrupted path can be triggered by the host. The host can also read status information from the routers.

8. Conclusion

The engineering model is a useful tool in our research on data flow architecture. Its flexibility has provided us with a powerful means for exploring trade-offs and issues in the implementation of a data flow machine. The experience gained from the engineering model will be useful in the design of practical full-scale data flow systems. For example the design of a VLSI version of the data flow processors or routers will be influenced by the experience gained from the run-time structures and the performance of the routers and routing network of the engineering model. Also the implementation of array memories in VLSI will be guided by the success of the array implementation schemes being explored. In addition, the engineering model will support studies of new schemes for achieving fault tolerance in computer systems employing packet architecture [9].

The "hands on" experience provided by the ability to construct and run data flow graphs on an actual hardware system has improved our general understanding of data flow machines and increased our confidence in the importance of these concepts for future computer systems.

9. Acknowledgements

The successful implementation of the engineering model is due to the efforts of many people. Clement Leung supervised development of software support for the engineering model with substantial help from Dean Brock. Andy Boughton designed the processing units based on earlier work of Ephraim Vishniac, and directed their construction and test. The logic design for the router was done by Thye-Lai Tung; the design has been verified and enhanced with testing facilities. The installation of the processors and the tailoring of

the host operating system were done by Ed Shaw. The design of the instruction cells has been influenced by the work of Ken Todd who together with Nena Bauman has contributed to the implementation of the data flow emulation software on the engineering model. The original version of microcode to emulate a data flow processing element was written by Arif Feridun. Work on schemes for array implementation has been done by Guang-Rong Gao. Others who have contributed to the project in one way or another are Tam-Anh Chu, Narinder Singh, Jason DeSilva, Cindy Gilbert, Chung Ho, Douglas Klunder, Victor Kwong, Joel Lilienkamp, Matthew Stern, Willie Tsang, Richard Tucker, and Inal Uygur.

10. References

[1] W.B. Ackerman, "Packet Communication Microprocessor Programming Manual," *Computation Structures Group Memo 192-1*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1980.

[2] W.B. Ackerman, "Efficient Implementation of Applicative Languages," Ph.D. Thesis in preparation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1982.

[3] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," *Technical Report 114a*, Department of Information and Computer Science, University of California, Irvine, California, December 1978.

[4] T.J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, vol. C-22, no. 4, April 1973, 421-422.

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

[6] J.B. Dennis, "Data Flow Supercomputers," *Computer*, IEEE, vol 13, no. 11, November 1980, 46-54.

[7] J.B. Dennis, G.A. Boughton, and C.K.C. Leung, "Building Blocks for Data Flow Prototypes," *Proceedings of 1980 Symposium on Computer Architecture*, LaBaule, France, May 1980, 1-8.

[8] G.R. Gao, "An Implementation Scheme for Array Operations in Static Data Flow Computer," Master thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1982.

[9] C.K.C. Leung, "Fault Tolerance in Packet Communication Computer Architectures," *Technical Report MIT/LCS/TR-250*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1980.

[10] C.K.C. Leung, "On a Top-down Design Methodology for Packet System," In *Computer Hardware Description Languages and their Applications*, M. Breuer and R. Hartenstein (Eds.), North-Holland Publishing Company, Amsterdam, Holland, 1981, 171-184.

[11] W.Y-P. Lim, "A Test Strategy for Packet Switching Networks," *Proceedings of the 1982 International Conference on Parallel Processing*, Bellaire, Michigan, August 1982, 96-98.

[12] W.Y-P. Lim and C.K.C. Leung, "PADL – A Packet Architecture Description Language," *Computation Structures Group Memo 221*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1982.

[13] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, B., and Snyder, A., *CLU reference manual*, Technical Report MIT/LCS/TR-225, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, October 1979.

[14] M. Pechoucek, "Anomalous Response Times of Input Synchronizers," *IEEE Transactions on Computers*, vol. C-25, no. 2, February 1976, 133-139.

[15] D.M. Ritchie and K.L. Thompson, "The UNIX Time-sharing System," *Communications of the ACM*, vol. 17, no. 7, July 1974, 365-375.

[16] C. Seitz, "System Timing", Chapter 7 of Introduction to VLSI Systems by C. Mead and L. Conway, Addison-Wesley, Reading, MA, 1980.

[17] N.P. Singh, "A Design Methodology for Self-timed Systems", *Technical Report MIT/LCS/TR-258*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1981.

[18] R.C.J. Somerville, P.H. Stone, M. Halem, J.E. Hansen, J.S. Hogan, L.M. Druyan, G. Russell, A.A. Lacis, W.J. Quirk, and J. Tenenbaum, "The GISS Model of the Global Atmosphere," *Journal of the Atmospheric Sciences*, vol. 31, January 1974, 84-117.

[19] M.J. Stucki and J.R. Cox, Jr., "Synchronization Strategies," *Proceedings of the the Caltech Conference on VLSI*, California Institute of Technology, Pasadena, California, January 1979, 375-393.

[20] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-29, no. 8, August 1980, 694-702.