MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 108

# A Computer Architecture for Highly Parallel Signal Processing

by

Jack B. Dennis
David P. Misunas

A paper to be presented at the ACM National Conference,
November 1974

This work was supported by the National Science Foundation
under research grants GJ-432 and GJ-34671.

August 1974

# A Computer Architecture for Highly Parallel Signal Processing

by

Jack B. Dennis and David P. Misunas

Abstract: A computer of unusual architecture is described that achieves highly parallel operation through use of a data-flow program representation. The machine is especially suited for signal processing computations such as waveform generation, modulation, and filtering, in which a group of operations must be performed once for each sample of the signals being processed. The difficulties of processor switching and memory/processor interconnection arising in attempts to adapt Von Neuman type computers for parallel operation are avoided by an organization in which sections of the machine communicate through transmission of information packets, and delays in packet transmission do not compromise effective utilization of the hardware. The design concept is especially suited to implementation using asynchronous logic and large-scale integrated circuits. Application of the concepts to generalized data-flow program languages is under study.

## Introduction

Highly parallel computers such as the Illiac IV and the CDC Star achieve their processing speed by imposing constraints on the structure of the data being processed. Both of these machines are organized to perform very well for data represented as vectors. To realize its potential, computation using the Illiac IV must be organized to exploit the machine's ability to execute the same operation simultaneously for many sets of operands. In the case of the CDC Star, the maximum processing rate of the pipelined processing unit is approached only if the computation is organized to make effective use of streaming operations on very long vectors. In both machines the programmer is forced to use unusual and intricate data representations if highly parallel execution is to be achieved. Thus these machines are developments contrary to

what is generally seen as one of the most important issues in contemporary computer practice -- the difficulty of developing correct programs. Even such an important notion as the use of subroutines is inadequately supported in these machines.

Other approaches to parallel processing in practical computer systems have not proved to be successful for highly parallel program execution: The classical multiprocessor computer system is limited in capability by the growth in complexity of the memory-processor switch as system size increases. Also, the problem of compiling user programs into concurrently executable parts of sufficient size that processor switching cost is acceptable is very difficult in the absence of unnatural constraints on the user language. Large processors like the CDC 6600 and the IBM 360/91 achieve parallel instruction execution by discovering absence of data dependency in instructions of a sequential program. The degree of parallelism achievable in this way has proved to be small.

We have been studying concepts of computer organization that can yield highly parallel program execution but with no sacrifice in the generality and ease of programming in the language supported. The ultimate goal of this work is a computer architecture able to achieve highly parallel execution of programs expressed in a user language such as CLU [11] which embodies linguistic features designed to support the development of well structured programs. An outline of the concepts expected to be employed in such a computer has been given in [3]. A central idea is the use of a machine language that permits a simple mechanism for identifying all instructions available for execution.

We have found that a program representation based on the concept of data flow is well suited for highly parallel program execution. In a data-flow representation, an instruction is enabled (that is, made available for execution) just when each required operand has been supplied by execution of predecessor instructions. Completion of instruction execution produces a result which is forwarded to each specified successor instruction for use as an operand. Data-flow representations for programs have been described by Karp and Miller [8], Rodriguez [16], Adams [1], Dennis and Fosseen [7], Bährs [2], Kosinski [9, 10], and Dennis [6].

In the present paper, we describe a machine capable of achieving highly parallel program execution for a special class of data-flow programs that correspond to the model of Karp and Miller [8]. These data-flow programs are well suited to representing signal processing computations such as waveform generation, modulation, and filtering, in which a group of operations is to be performed once for each sample (in time) of the signals being processed.

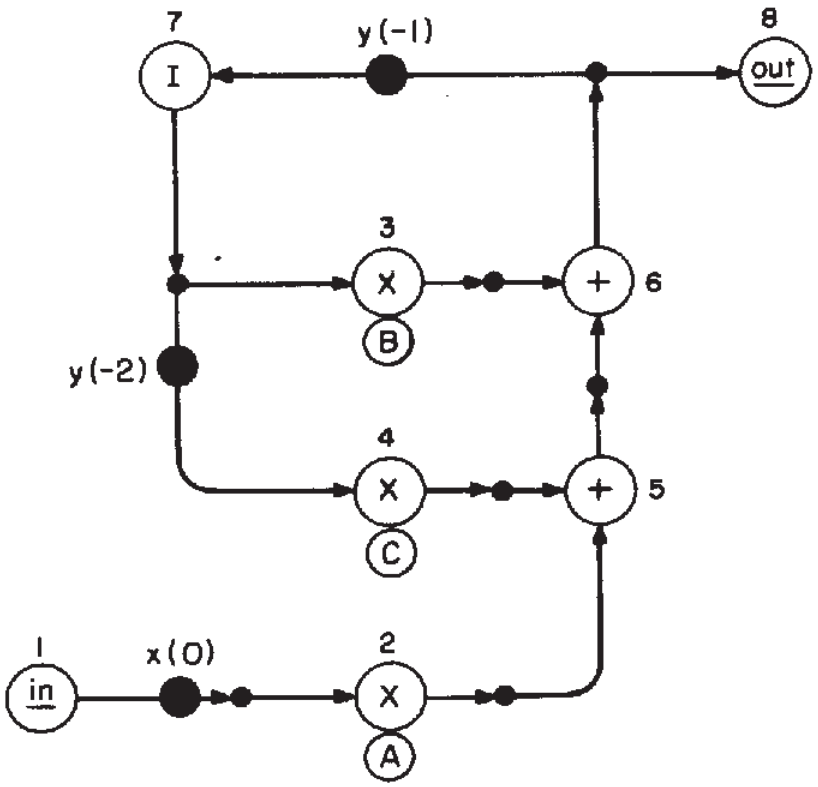
Our machine avoids the problems of processor switching and processor/memory interconnection present in attempts to adapt convention Von Neuman type machines for parallel computation. In our design, processors in the usual sense do not exist. Sections of the machine communicate by the transmission of fixed size information packets, and the machine is organized so that the sections can tolerate delays in packet transmission without compromising effective utilization of the hardware. In future work we expect to further develop these ideas and investigate the feasibility of their application to the design of highly parallel computers using a generalized data-flow language such as described by Dennis [6], Kosinski [9, 10] and Bährs [2].

## General Description

To illustrate the basic concepts of operation of the proposed processor, consider the data-flow program shown in Figure 1. This program represents the computation required for a second-order recursive digital filter

$$y(t) = A\,x(t) + B\,y(t-1) + C\,y(t-2)$$

where $x(t)$ and $y(t)$ denote input and output samples for time $t$. In this diagram, operators 2, 3 and 4 are unary operators that multiply by the fixed parameters A, B and C; operators 5 and 6 are binary operators that perform addition; and operator 7 is an identity operator that transmits its input values unchanged. Each small solid dot is a link that receives results from an operator and distributes them to other operators for use as operands. Input operator 1 represents a port through which an external stream of values that represent the input signal $x(t)$ is presented to the program. Similarly, output operator 8 represents an output port at which the sequence of values representing $y(t)$ is delivered during program execution.




1. A Data Flow Program.

The large solid dots (tokens) show the presence of values at certain input arcs of operators and define the initial configuration for program execution. An operator with tokens on each of its input arcs and no token on its output arc is enabled, and may fire by removing the tokens from its input arcs, computing a result using the values associated with the tokens, and associating the result with a token placed on the output arc of the operator. A link is enabled when a token is present on its input arc and no token is present on any of its output arcs. It fires by placing tokens on each of its output arcs and removing the token from its input arc. The new tokens distribute copies of the value associated with the input token over each output arc of the link.

The processor has seven major sections, organized as shown in Figure 2:

Memory Section
Arbitration Network
Functional Units
Distribution Network
Controller
Command Network
Control Network

In addition, the block labeled "Host" represents a source of operating commands to the machine and could be either a manual console or a separate computer.

The design is conceived as using asynchronous communication of information packets between sections of the machine. Each connection between sections is independently coordinated using an acknowledge signal for each packet of information transmitted over the connection, and each connection is represented in Figure 2 by a line with an arrow indicating direction of packet flow.

The information units transmitted through the Arbitration Network from the Memory Section to the Functional Units are instruction packets; each instruction packet specifies one unit of work for the Functional Unit to which it is directed. The information units sent through the Distribution Network from Functional Units to the Memory Section are result packets; each result packet delivers a newly computed value to a specific register in the Memory Section.

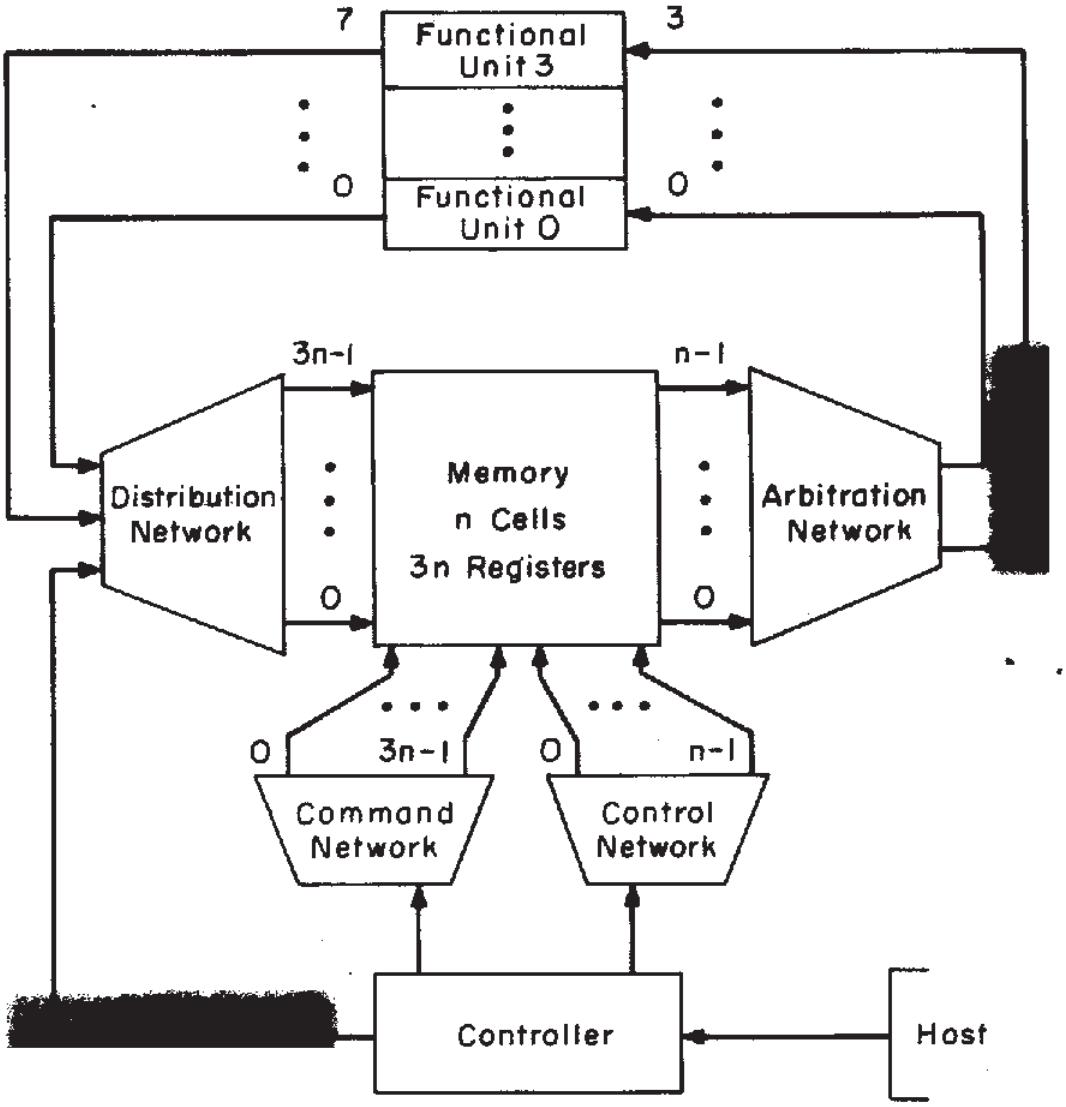



Figure 2. General Organization of the Processor.

The Memory Section of the processor holds a representation of the program to be executed and holds computed values awaiting use. The Memory is a collection of Cells; one Cell must be associated with each operator of the program. Each Cell (Figure 3) contains three Registers -- one Register to hold an instruction which encodes the type of operator and its connections to other operators of the program, and two Registers that receive operand values for use in the next execution of the instruction. Each Register may be set to behave as a constant or as a variable. If set to act as a variable, a register expects to receive a result packet containing an operand value through the Distribution Network; if set to act as a constant, a Register retains the value delivered to it when the program was loaded into the Memory. The instruction Register of a Cell is normally set to act as a constant. When all three Registers of a Cell are full, the Cell is said to be enabled and the contents of the three Registers (instruction and two operand values) form an instruction packet which is presented to the Arbitration Network.

Figure 4 shows the instruction format. Assuming four Functional Units, the operand code has a field of two bits which specifies the Functional Unit required and a field that indicates which specialized capability of the Functional Unit is to be used. Each destination field contains the address of a Memory Register which is to receive one copy of each result generated by execution of the instruction. The initial contents of Memory Cells for the digital filter example is shown in Figure 5. Empty parentheses indicate an operand register waiting to receive a value.

In Figure 5 cell 2 is enabled and presents the instruction packet

$$\begin{Bmatrix} \text{mult}, & 13, & - \\ x(0) & & \\ A & & \end{Bmatrix}$$

to the Arbitration Network. Some Functional Unit will compute

$$z = A \times x(0)$$

and send the result packet

$$\begin{Bmatrix} 13 \\ z \end{Bmatrix}$$

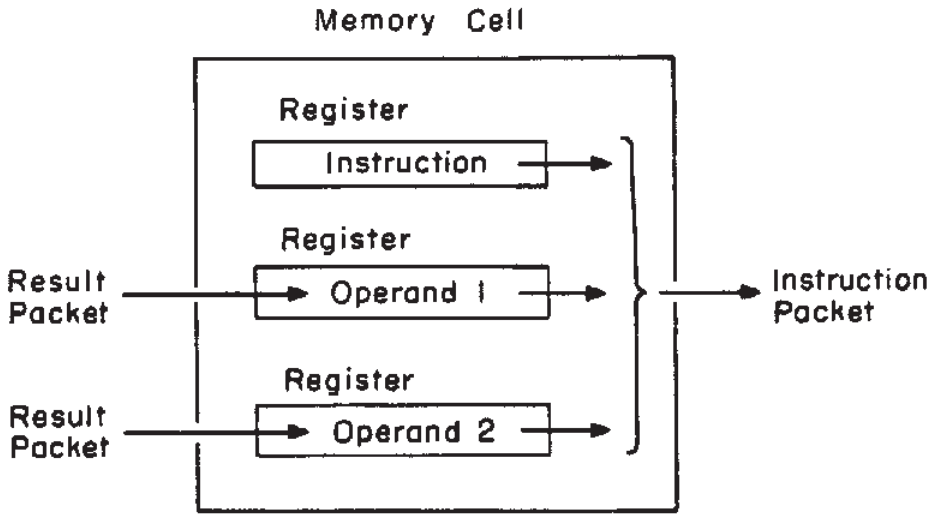



Figure 3. Operation of a Memory Cell.

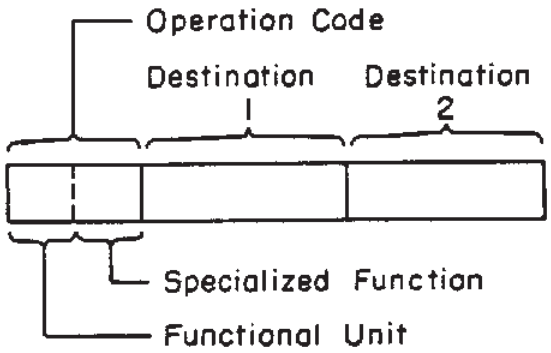



Figure 4. Instruction Format.

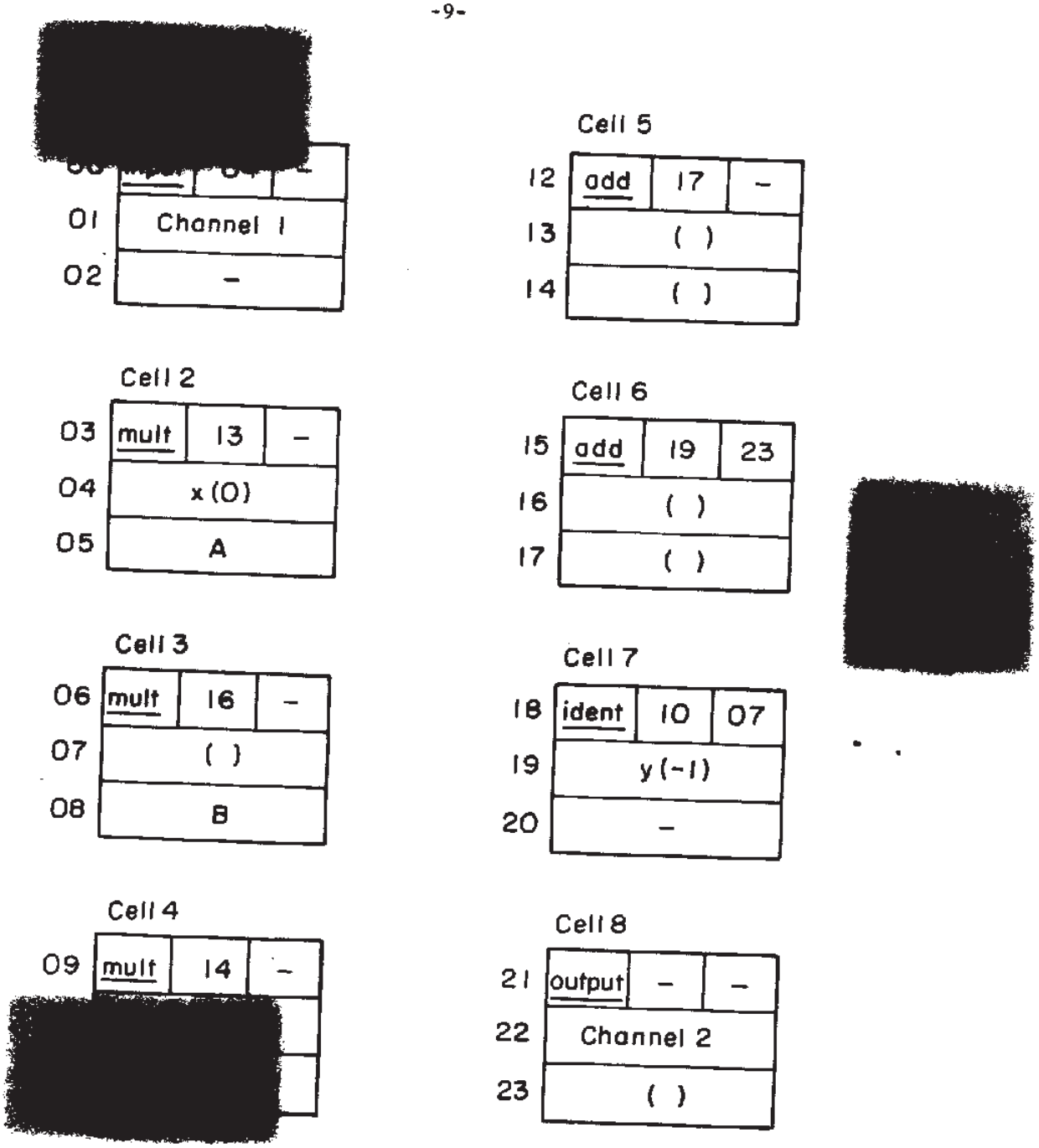



Figure 5. Initialization of Memory Cells for the Digital Filter Computation.

through the Distribution Network, and operand Register 13 in cell 5 will receive the value z.

As illustrated in Figure 6, each Functional Unit receives from the Arbitration Network all instruction packets directed to it by their operation codes, and in general, delivers two result packets to the Distribution Network. To realize maximum throughput, each Functional Unit is constructed as three pipelines; one pipeline performs the computation of the result value $z = x \otimes y$ where x and y are the operands from the instruction packet. The second and third pipelines carry the destination addresses d1 and d2 so these may be associated with the result z when it emerges from the computational pipeline.

Since the data-flow form of a program exposes many possibilities for concurrent execution of instructions, we can expect that many Cells in the Memory Section of the processor will be enabled at once. As the Functional Units have high potential throughput, we must show how the Arbitration and Distribution Networks can be organized to handle many packets concurrently so all sections of the processor are effectively utilized. The Arbitration Network is designed so many instruction packets may flow into it concurrently from Cells of the Memory Section and merge into four streams of packets -- one for each Functional Unit. The network is built of the four types of units shown in Figure 7.

The Arbitration Unit passes packets arriving at input ports A and B, one-at-a-time to output port C using a round-robin discipline to resolve any ambiguity about which packet should be sent next. The Switch Unit assigns packets arriving at port A to ports B or C according to some property of the packet. In the Arbitration Network, Switch Units separate instruction packets into four categories, one for each Functional Unit, by testing the operation codes of the instructions they contain.

Figure 8 shows how Arbitration Units and Switch Units might be arranged into an Arbitration Network. This network contains a unique path for instruction packets from each Memory Cell to each Functional Unit.

Since the Arbitration Network has many input ports and only four output ports, the rate of packet flow will be much greater at the output ports. Thus a serial representation of packets is appropriate at the input ports to

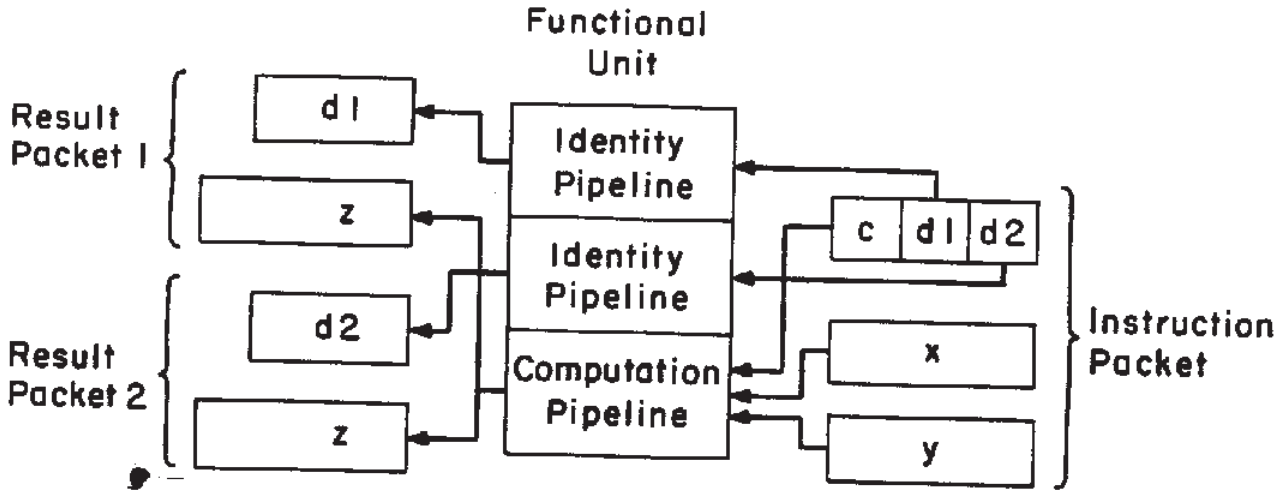



Figure 6. Pipeline Operation of a Functional Unit.

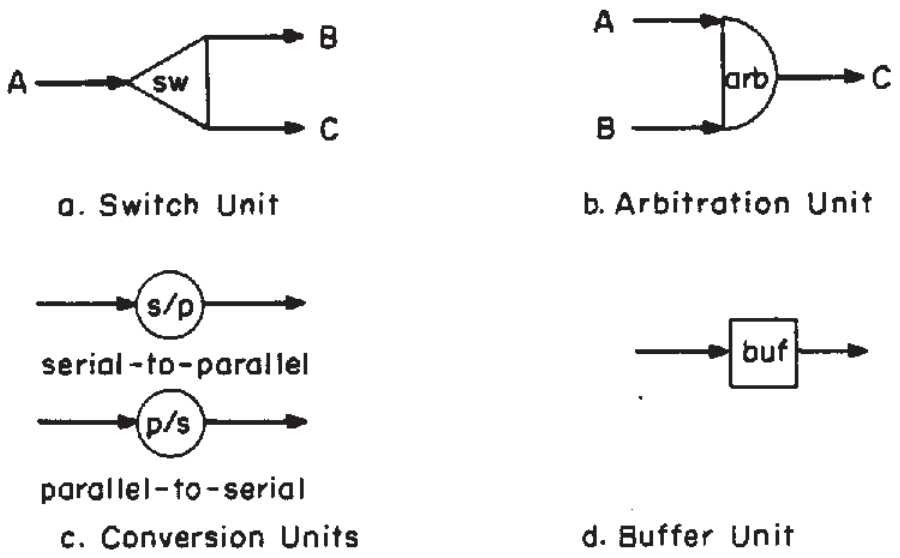



Figure 7. Units for the Arbitration Network and the Distribution Network.

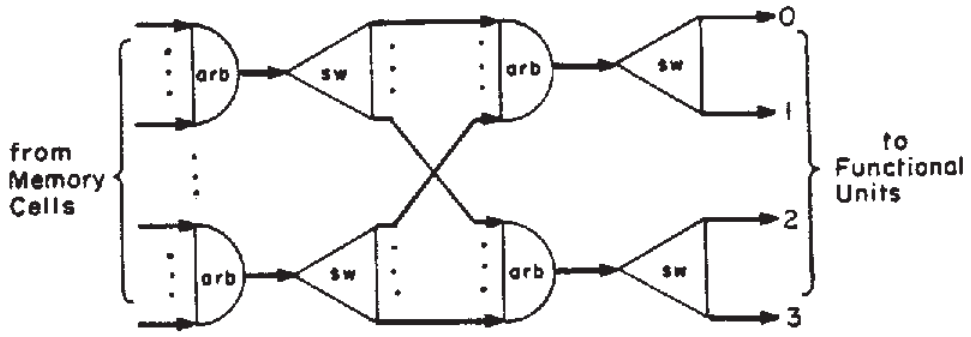



Figure 8. Primitive Arbitration Network.

minimize the number of connections to the Memory Section, but a more parallel representation is required at the output ports so a high throughput may be achieved. Evidently, serial-to-parallel conversion is required within the Arbitration Network, and Conversion Units (Figure 7c) must be included. In addition, a packet emerging from a Conversion Unit must be prevented from engaging a subsequent Arbitration Unit until the serial packet has been completely absorbed by the Conversion Unit. Thus Buffer Units are needed at the output of each converter. Figure 9 shows an improved Arbitration Network including Conversion Units and Buffer Units.

The Distribution Network is similarly organized. As shown in Figure 10, many Switch Units route result packets to the Memory Registers specified by their destination addresses. A few Arbitration Units are required so result packets from each Functional Unit can have access through the Distribution Network to each Register of the Memory Section.

## Detailed Specification of Machine Units

The concepts of computer organization presented above are very attractive for application of our work [4, 5, 13, 15] on speed independent logic design. We have developed complete logic designs for each section of the machine in the form of a speed independent interconnection of a small set of basic asynchronous module types. Space does not permit presenting detailed designs for all units of the machine, so we will limit discussion to the most interesting part, the Cells of the Memory Section.

A system is speed independent if the correctness of its operation is unaffected by the presence of delay. Two kinds of delay are considered: delay in interconnecting wires; and internal delays in the components that are interconnected. An interconnection of components is called a type 1 speed independent system if arbitrary delays on the interconnecting wires do not affect correctness of system operation. A system is type 2 speed independent if its correct operation is not affected by arbitrary delays inserted at the output terminals of system components.

From an engineering viewpoint, it is attractive to construct systems as type 1 interconnections of components, for timing considerations may be ignored in designing the physical placement of the components. Hence we have chosen a small set of basic module types sufficient to describe the

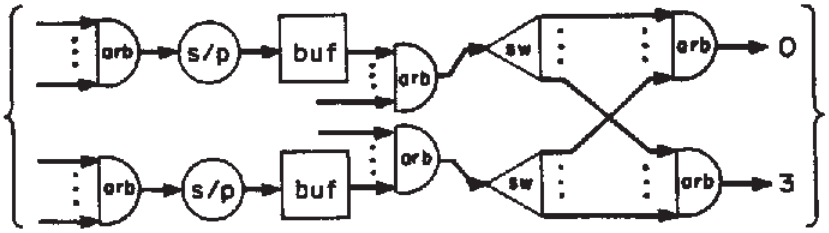



Figure 9. Improved Arbitration Network.

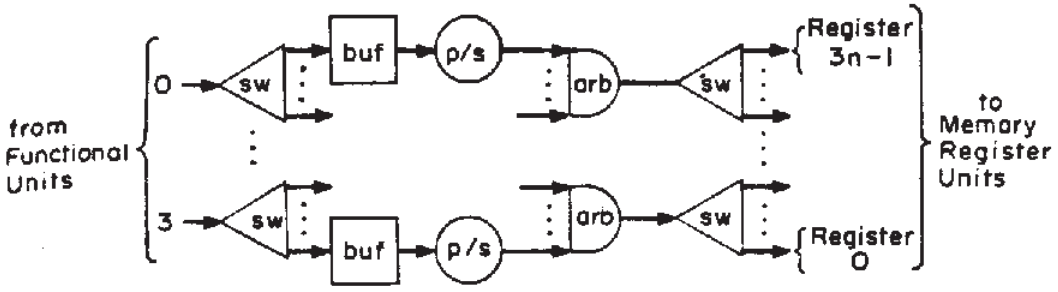



Figure 10. Distribution Network.

proposed machine as a type 1 speed independent interconnection of modules.

In the use of basic modules to form units and sections of the processor, a specific communication discipline known as reset signalling [14] is used. A 0-to-1 transition on a wire represents a meaningful event; we say that the wire has sent a signal from the module that drives the wire to the module to which the wire is connected. Before the wire can transmit a subsequent signal, a 1-to-0 transition must occur; this transition is called a reset of the wire. During the interval between a signal event and the following reset event, the wire is said to be active.

Each section of the processor is a collection of interconnected units, which in turn are collections of interconnected basic modules. Each set of wires that connect one unit to another is called a link, and consists of one or more groups of wires. Within each group of wires a strict signalling discipline is observed: The wires of a group are divided into enable wires and acknowledge wires. In the quiescent condition of the processor, all wires are reset. During operation, the signal and reset events that occur on the wires of a group follow a repeated cycle of four steps:

1. Signals are sent over at least one enable wire.
2. Signals are returned over at least one acknowledge wire.
3. The enable wires used in step (1) are reset.
4. The acknowledge wires used in step (2) are reset.

All events of one step must occur before any event of the following step.

For the description of the Memory Cell, five basic module types are used; these are illustrated in Figures 11 through 14. The OR module and the NOT module (Figure 11) are the conventional OR gate and inverter. It is convenient to draw the OR module as a line on which any lines representing input wires terminate in arrowheads. Only one input wire of an OR module may be active at a time. The OR module permits signals from several sources to go to a common destination. The NOT modules change reset events into signal events, and vice versa, and are the starting points for all action in the system.

The C-module in Figure 12 is an important switching element in speed independent systems. The output of a C-module becomes 1 when both inputs

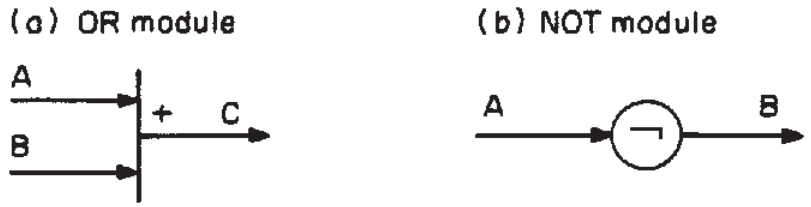



Figure 11. The OR and NOT modules.

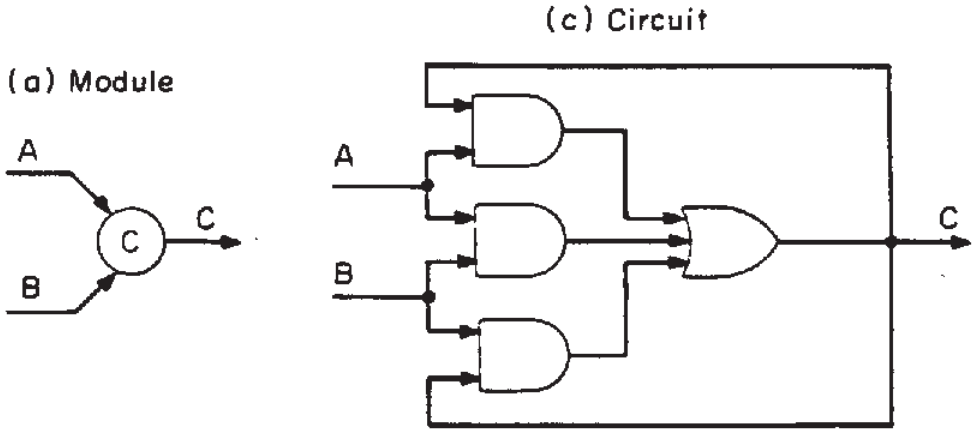



Figure 12. The C-module.

become 1, and the output will return to 0 only when both inputs have become 0. The realization of the C-module shown in Figure 12 is not a speed independent connection of switching elements. Yet its behavior will be correct if delays within the circuit are properly controlled.

The use of the C-module as a synchronization element and as a buffer storage element is demonstrated in the design of the data switch module (Figure 13a). This module waits for signals to arrive on enable wire E and one of input wires A1 and B1. It then sends a signal on the output wire (A2 or B2) corresponding to the active input wire. The output wire resets only when both the enable wire and the active input wire have reset. For speed independent operation, it is necessary to use the data switch module only where signals cannot arrive on both input wires A1 and B1 without an intervening reset.

The circuit for the data switch given in Figure 9 is a type 2 interconnection of switching elements because of the connection from the enable input to the two C-modules. Delay in the enable connection to the unused C-module may keep the C-module enable input from resetting in time to prevent a false output when an input signal arrives at that C-module. Nevertheless, if there is negligible delay in the wires, the circuit will exhibit the correct behavior.

A basic kind of memory element for speed independent systems is the bit pipeline module shown in Figure 14. Signals on wires E1 or E0 cause a 1 or 0 to be entered into the pipeline, after which an acknowledge signal is returned on wire A. Up to n bits may be entered into a pipeline module BP[2n] having 2n sections. When the pipeline is not empty, and an enable signal is sent on wire E, the pipeline emits its longest held bit by a signal on R1 or R0.

The pipeline module is constructed of a cascade of data switch modules with feedback connections using OR and NOT modules arranged so that bits advance through the data switches until all alternate stages hold information. As in the case of a data switch module, inputs E1 and E0 must not be simultaneously active.

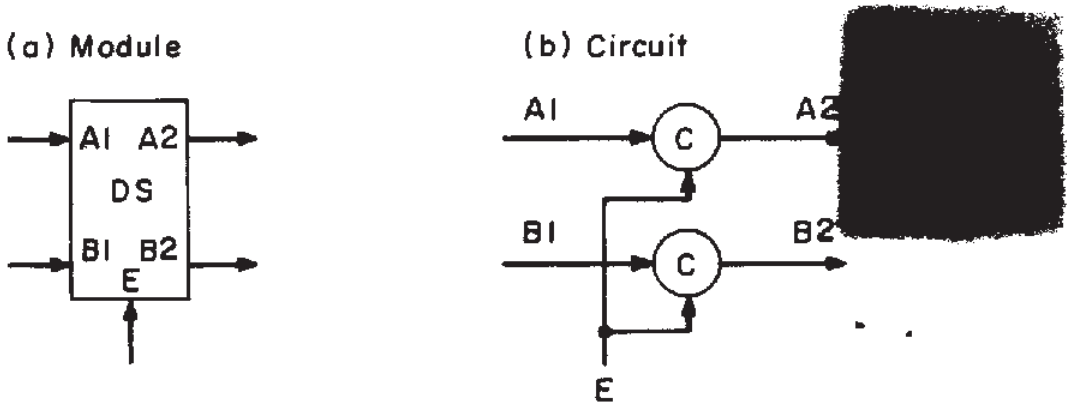



Figure 13. The Data Switch Module.

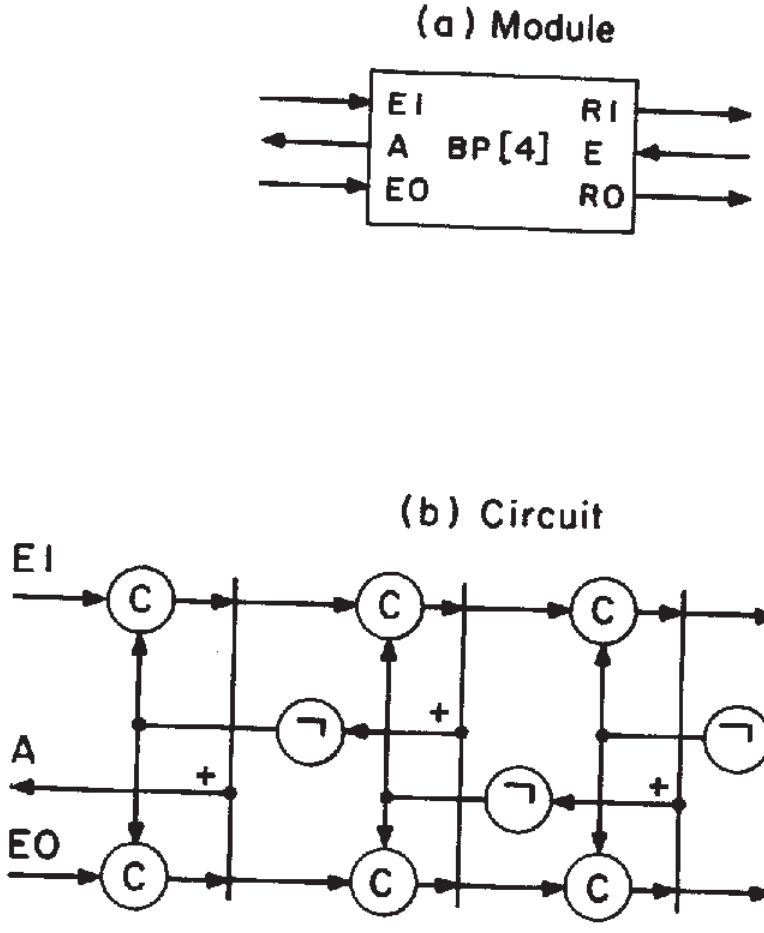



Figure 14. The Bit Pipeline Module.

Description of a Memory Cell

The Memory Section of the processor consists of a number of Memory Registers organized into Cells, each Cell having the structure shown in Figure 15. Each Cell contains three Register Units having consecutive addresses, and each Register Unit consists of a Register that holds an m-bit instruction or operand and a Control. In Figure 15, the sets of wires making up links between sections of the processor are indicated by curly brackets, and the division of each link into groups is indicated by square brackets.

Result packets are transferred from the Distribution Network to each Register Unit through input ports IA, IB, and IC. An instruction or operand is transferred into a Register by m data transactions on the wires of group 1. The Distribution Network informs the Control that all data transactions required to deliver the packet are complete by means of a transaction on wires S and AS of group 2.

Execution of a program is controlled from the Control Network through input port ID. One cycle of instruction execution is completed for each transaction on wires R and AR. A signal on wire R requests each Control to send an enable signal on wire E when its associated Register is filled with an instruction or operand. The conjunction of an enable signal from each Control is detected by C-modules, and a signal is sent over the enable wire E of output port OA to inform the Arbitration Network that this Cell has an instruction packet ready for transmission. When it is able, the Arbitration Network receives the packet by m transactions on the enable and acknowledge wires of group 1. When packet transmission is complete, an acknowledge signal is returned to each Control over wire AE of group 2. When each Register Unit completes its action for one cycle of instruction execution, an acknowledge signal is sent on wire AR of the Control. The conjunction of all three acknowledge signals produces an acknowledge signal to the Control Network on wire AR indicating that this Memory Cell has completed the requested instruction execution cycle.

Before execution of a program, each Register Unit is set to one of four modes. The Register Unit may be set to idle, in which case it is not utilized in the computation, or it may be set to hold a constant (con) or a

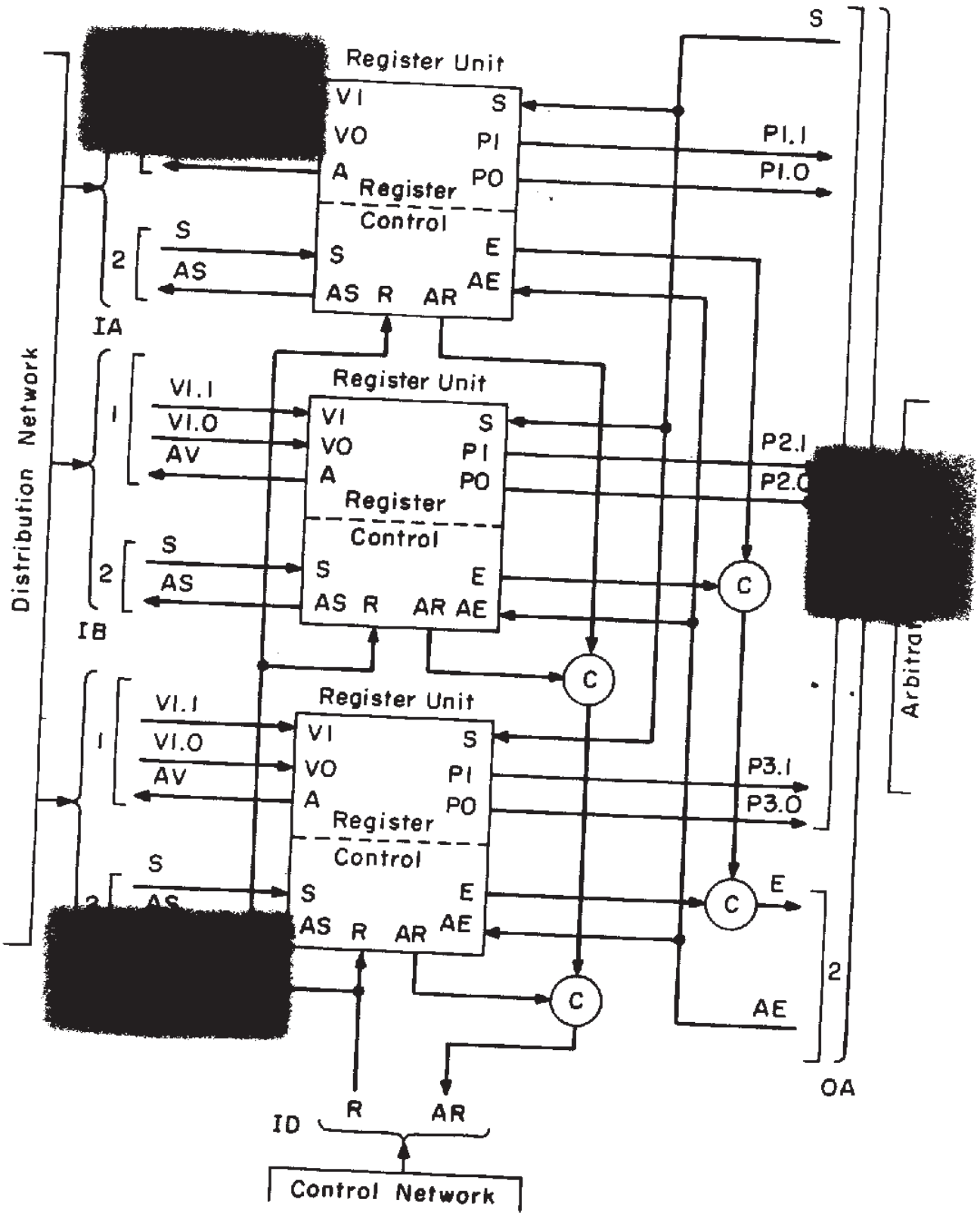



Figure 15. Specification of a Memory Cell.

variable (var). In the case of a constant, the Register is loaded with an initial value before program execution and will retain this value while transmitting it as part of an instruction packet. If the Register contains a variable, it may be either empty or full. A full Register holds an operand value in the initial configuration of the Memory, and must receive a new value through the Distribution Network after each packet transmission to complete one cycle of instruction execution. An empty register must receive a value in each cycle of instruction execution before packet transmission may begin.

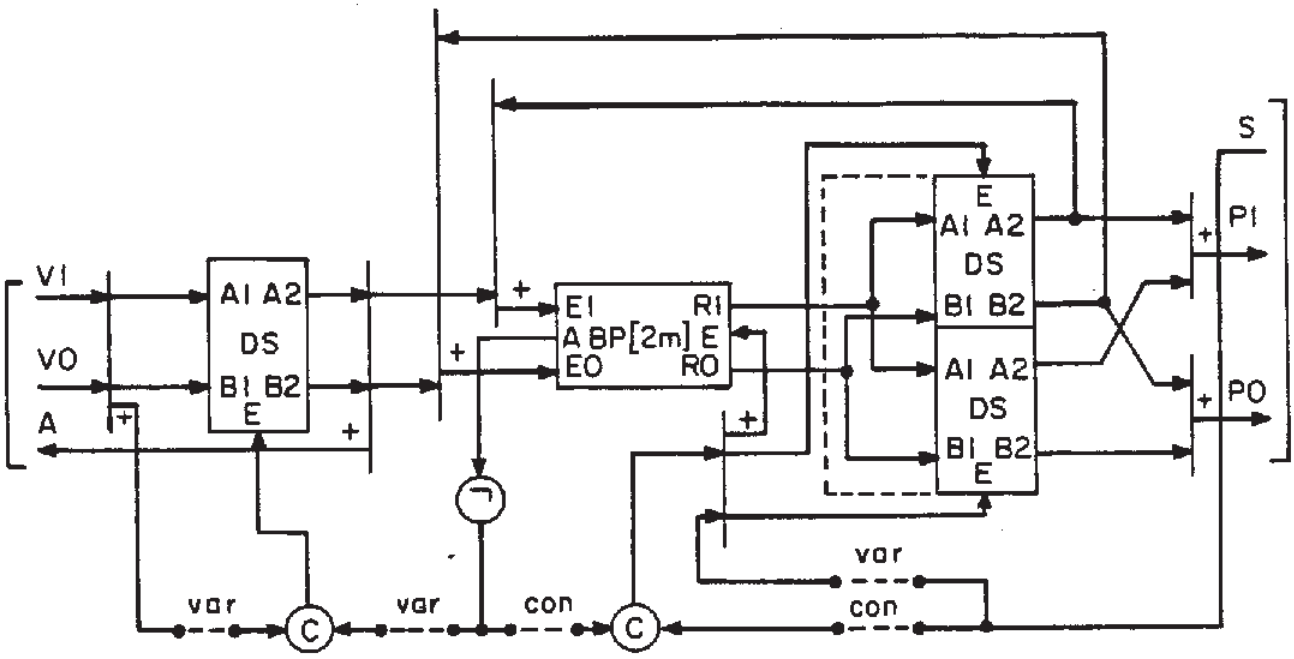
In the diagram of the Register Unit (Figure 16), the mechanism for setting the mode of the register has been omitted for simplicity; the signal paths required for each mode are indicated by labelled gaps in the drawings. In the complete design, the mode of each Register is set by transactions through the Command Network initiated by the Controller in response to commands from the Host.

The Register portion of a Register Unit is specified in terms of the basic speed independent modules in Figure 16a. Data presented at the input port enters the bit pipeline module through the data switch module if the first stage of the bit pipeline is empty and the Register is in variable mode. Each bit of data is acknowledged by a signal from the data switch.

Data is requested from the pipeline by a signal from the Arbitration Network on the S wire. If the Register contains a variable, the output of the bit pipeline passes through the lower of the pair of data switch modules and exits on the P1 or P0 output wires. If the register holds a constant, the upper output data switch passes the data to the output and also returns it to the pipeline input. The P0 and P1 output wires are reset when wire S resets indicating that the Arbitration Network has absorbed the data, and, if the Register is in constant mode, that the data has been reentered in the bit pipeline.

The Control part of a Register Unit is detailed in Figure 16b. The response of the unit to the arrival of a signal on the R wire is determined by the mode of the unit. If the Register is idle, an acknowledge signal is immediately returned to the Control Network. If the Register contains a constant, an enable signal is immediately sent to the Arbitration Network on wire E. When an acknowledge signal returns on wire AE indicating that an

(a) Register




(b) Control

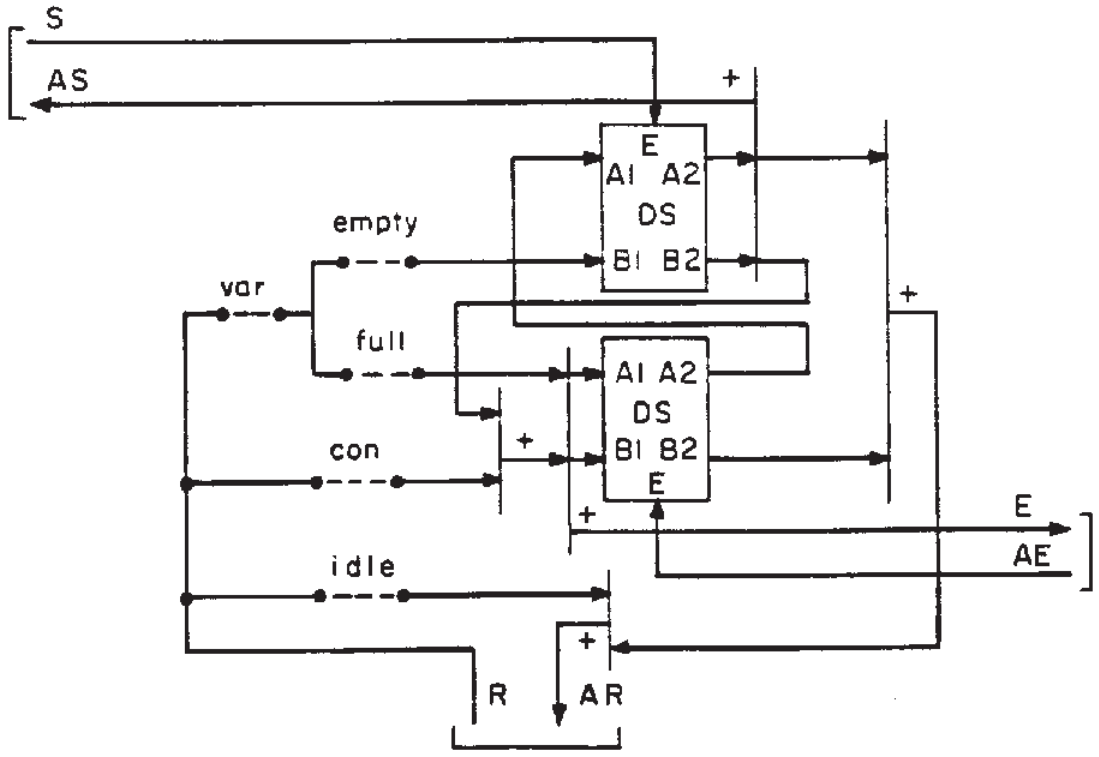



Figure 16. Specification of a Register Unit.

instruction packet has been completely transmitted to the Arbitration Network, an acknowledge signal is sent to the Control Network.

If the Register is in variable mode, the action depends on whether the Register is empty or full. If it is full, the Control behaves initially as in constant mode -- it sends an enable signal on wire E. However, the acknowledge signal on wire AE causes the lower data switch to send a signal to the upper data switch module, which waits until a space signal arrives from the Distribution Network indicating that the Register has been reloaded. Then acknowledge signals are sent to the Distribution Network on wire AS and to the Control Network on wire AR.

If the Register holds a variable, but is initially empty, an enable signal waits at the upper data switch for the Register to be filled from the Distribution Network. Action then completes as if the Register were in constant mode.

The reader should keep in mind that in describing the Memory Cell as a speed independent interconnection of basic modules we have simply given a precise specification of the intended behavior of the unit. In the physical realization of Memory Cells, the detailed design will depend heavily on the device technology employed in their fabrication, and may be quite different from the circuits implicit in our drawings. The requirement is that the physical Memory Cell have behavior equivalent to the specified behavior when used as a component in a type 1 speed independent system.

## Conclusions

The idea of organizing a computer so execution of instructions is triggered by the presence of their operands has been discussed by Seeber and Lindquist [17], Patil [14], Dennis [3], Shaptro, Saint and Presberg [18], and Miller and Cocke [12]. However, none of these authors has suggested a detailed and efficient scheme for communicating enabled instructions and operands to functional units for processing. We are hopeful that the architecture proposed here offers an attractive solution to this problem -- a solution that can be extended to the design of processors that support programming languages suitable for general purpose computation.

References


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

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

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

4. 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.

5. Dennis, J. B., and S. S. Patil. Speed independent asynchronous circuits. Proceedings of the Fourth Hawaii International Conference on System Sciences, Western Periodicals Co., North Hollywood, Calif., 1971, 55-58.

6. Dennis, J. B. First version of a data flow procedure language. Symposium on Programming, Institut de Programmation, University of Paris, Paris, France, April 1974, 241-271.

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

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

9. Kosinski, P. R. A Data Flow Programming Language. Report RC 4264, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., March 1973.

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

11. Liskov, B. H., and S. N. Zilles. Programming with abstract data types. Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50-59.

12. Miller, R. E., and J. Cocke. Configurable Computers: A New Class of General Purpose Machines. Report RC 3897, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., June 1972.

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

14. Patil, S. S. An Abstract Parallel Processing System. S.M. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., June 1967.

15. Patil, S. S., and J. B. Dennis. The description and realization of digital systems. Proceedings of the Sixth Annual IEEE Computer Society International Conference, IEEE, New York, N.Y. 1972, 223-226.

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

17. Seeber, R. R., and A. B. Lindquist. Associative logic for highly parallel systems. AFIPS Conference Proceedings 24, 1963, 489-493.

18. Shapiro, R. M., H. Saint, and D. L. Presberg. Representation of Algorithms as Cyclic Partial Orderings. Report CA-7112-2711, Vol. 1, Applied Data Research, Wakefield, Mass., December 1971.