A Preliminary Architecture for a Basic Data-Flow Processor*

Jack B. Dennis and David P. Misunas
Project MAC
Massachusetts Institute of Technology

Abstract: A processor is described which can achieve highly parallel execution of programs represented in data-flow form. The language implemented incorporates conditional and iteration mechanisms, and the processor is a step toward a practical data-flow processor for a Fortran-level data-flow language. The processor has a unique architecture which avoids the problems of processor switching and memory/processor interconnecion that usually limit the degree of realizable concurrent processing. The architecture offers an unusual solution to the problem of structuring and managing a two-level memory system.

Introduction

Studies of concurrent operation within a computer system and of the representation of parallelism in a programming language have yielded a new form of program representation, known as data flow. Execution of a data-flow program is data-driven; that is, each instruction is enabled for execution just when each required operand has been supplied by the execution of a predecessor instruction. Data-flow representations for programs have been described by Karp and Miller [8], Rodriguez [11], Adams [1], Dennis and Fosseen [5], Bährs [2], Kosinski [9, 10], and Dennis [4].

We have developed an attractive architecture for a processor that executes elementary data-flow programs [6, 7]. The class of programs implemented by this processor corresponds 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. This elementary data-flow processor avoids the problems of processor switching and processor/memory interconnection present in attempts to adapt conventional Von Neuman type machines for parallel computation. 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.

We wish to expand the capabilities of the data-flow architecture, with the ultimate goal of developing a general purpose processor using a generalized data-flow language such as described by Dennis [4], Kosinski [9, 10] and Bährs [2]. As an intermediate step, we have developed a preliminary design for a basic data-flow processor that executes programs expressed in a more powerful language than the elementary machine, but still not achieving a generalized capability. The language of the basic machine is that described by Dennis and Fosseen [5], and includes constructs for expressing conditional and iterative execution of program parts.

In this paper we present solutions to the major problems faced in the development of the basic machine. A straightforward solution to the incorporation of decision capabilities in the machine is described. In addition, the growth in program size and complexity with the addition of the decision capability requires utilization of a two-level memory system. A design is presented in which only active instructions are in the operational memory of the processor, and each instruction is brought to that memory only when necessary for program execution, and remains there only as long as it is being utilized.

*The work reported here was supported by the National Science Foundation under research grant GJ-34671.

The Elementary Processor

The Elementary Processor is designed to utilize the elementary data-flow language as its base language. A program in the elementary data-flow language is a directed graph in which the nodes are operators or links. These nodes are connected by arcs along which values (conveyed by tokens) may travel. An operator of the schema is enabled when tokens are present on all input arcs. The enabled operator may fire at any time, removing the tokens on its input arcs, computing a value from the operands associated with the input tokens, and associating that value with a result token placed on its output arc. A result may be sent to more than one destination by means of a link which removes a token on its input arc and places tokens on its output arcs bearing copies of the input value. An operator or a link cannot fire unless there is no token present on any output arc of that operator or link.

An example of a program in the elementary data-flow language is shown in Figure 1 and represents the following simple computation:

```
input a, b
   y := (a+b)/x
   x := (a*(a+b))+b
output y, x
```

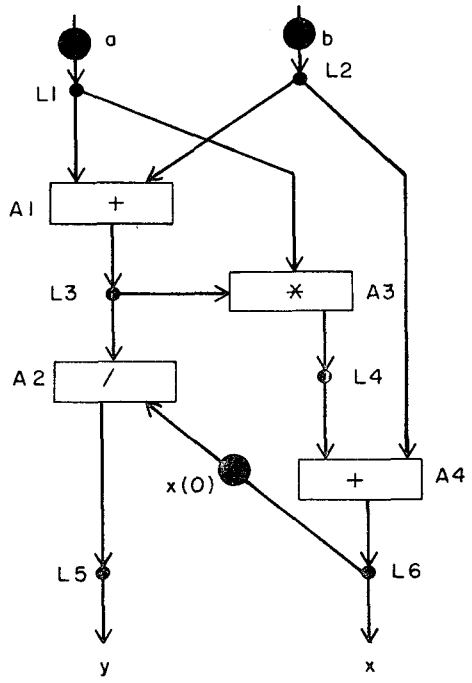


Figure 1. An elementary data-flow program.

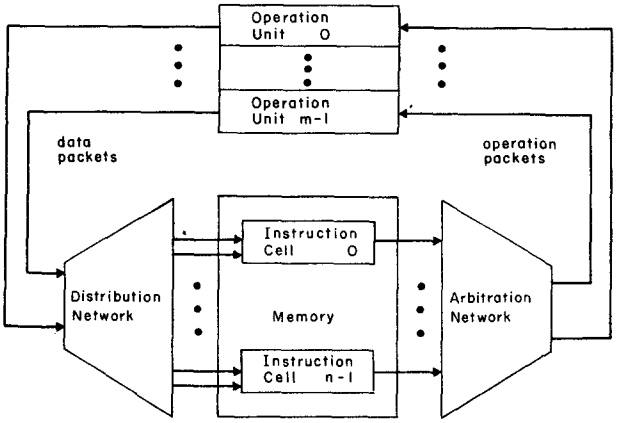


Figure 2. Organization of the elementary data-flow processor.

The rectangular boxes in Figure 1 are operators, and each arithmetic operator in the above computation is reflected in a corresponding operator in the program. The small dots are links. The large dots represent tokens holding values for the initial configuration of the program.

In the program of Figure 1, links L1 and L2 are initially enabled. The firing of L1 makes copies of the value a available to operators A1 and A3; firing L2 presents the value b to operators A1 and A4. Once L1 and L2 have fired (in any order), operator A1 is enabled since it will have a token on each of its input arcs. After A1 has fired (completing the computation of a+b), link L3 will become enabled. The firing of L3 will enable the concurrent firing of operators A2 and A3, and so on.

The computations represented by an elementary program are performed in a data-driven manner; the enabling of an operator is determined only by the arrival of values on all input links, and no separate control signals are utilized. Such a scheme prompted the design of a processor organized as in Figure 2.

A data-flow schema to be executed is stored in the Memory of the processor. The Memory is organized into Instruction Cells, each Cell corresponding to an operator of the data-flow program. Each Instruction Cell (Figure 3) is composed of three registers. The first register holds an instruction (Figure 4) which specifies the operation to be performed and the address(es) of the register(s) to which the result of the operation is to be directed. The second and third registers hold the operands for use in execution of the instruction.

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

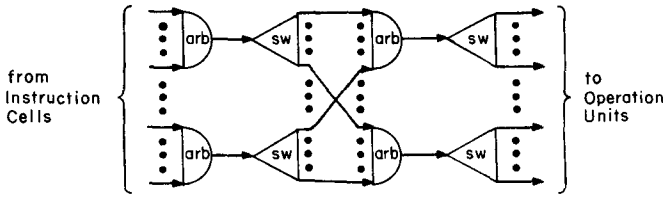


Figure 5. Structure of the Arbitration Network.

The result of an operation leaves an Operation Unit as one or more data packets, consisting of the computed value and the address of a register in the Memory to which the value is to be delivered. The Distribution Network accepts data packets from the Operation Units and utilizes the address of each to direct the data item through the network to the correct register in the Memory. The Instruction Cell containing that register may then be enabled if an instruction and all operands are present in the Cell.

Many Instruction Cells may be enabled simultaneously, and it is the task of the Arbitration Network to efficiently deliver operation packets to Operation Units and to queue operation packets waiting for each Operation Unit. A structure for the Arbitration Network providing a path for operation packets from each Instruction Cell to each Operation Unit is presented in Figure 5. Each Arbitration Unit passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any ambiguity about which packets should be sent next. A Switch Unit assigns a packet at its input to one of its output ports, according to some property of the packet, in this case the operation code.

The Distribution Network is similarly organized using Switch Units to route data packets from the Operation Units to the Memory Registers specified by the destination addresses. A few Arbitration Units are required so data packets from different Operation Units can enter the network simultaneously.

Since the Arbitration Network has many input ports and only a few 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 minimize the number of connections to the Memory, but a more parallel representation is required at the output ports so a high throughput may be achieved. Hence, serial-to-parallel conversion is performed in stages within the Arbitration Network. Similarly, parallel-to-serial conversion of the value portion of each result packet occurs within the Distribution Network.

The Operation Units of the processor are pipelined in

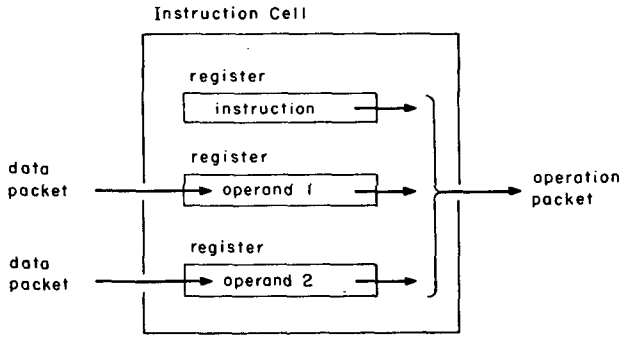


Figure 3. Operation of an Instruction Cell.

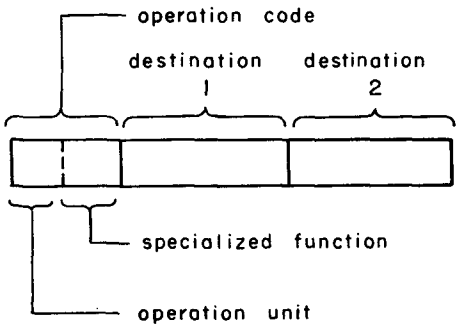


Figure 4. Instruction format.

Figure 6. Links of the basic data-flow language.

order to allow maximum throughput. The destination address(es) of an instruction are entered into identity pipelines of the Operation Units and are utilized to form data packets with the result when it appears.

A more detailed explanation of the elementary processor and its operation is given in [6]. We have completed designs for all units of the elementary processor in the form of speed-independent interconnections of a small set of basic asynchronous module types. These designs are presented in [7].

The Basic Data-Flow Language

Our success in the architecture of the elementary data-flow processor led us to consider applying the concepts to the architecture of machines for more complete data-flow languages. For the first step in generalization, we have chosen a class of data-flow programs that correspond to a formal data-flow model studied by Dennis and Fosseen [5].

The representation of conditionals and iteration in data-flow form requires additional types of links and actors. The types of links and actors for the basic data-flow language are shown in Figures 6 and 7.

Data values pass through data links in the manner presented previously. The tokens transmitted by control links are known as control tokens, and each conveys a value of either true or false. A control token is generated at a decider which, upon receiving values from its input arcs, applies its associated predicate, and produces either a true or false control token at its output arc.

The control token produced at a decider can be combined with other control tokens by means of a Boolean operator (Figure 7f), allowing a decision to be built up from simpler decisions.

Control tokens direct the flow of data tokens by means of T-gates, F-gates, or merge actors (Figure 7c, d, e). A T-gate passes the data token on its input arc to its output arc when it receives a control token conveying

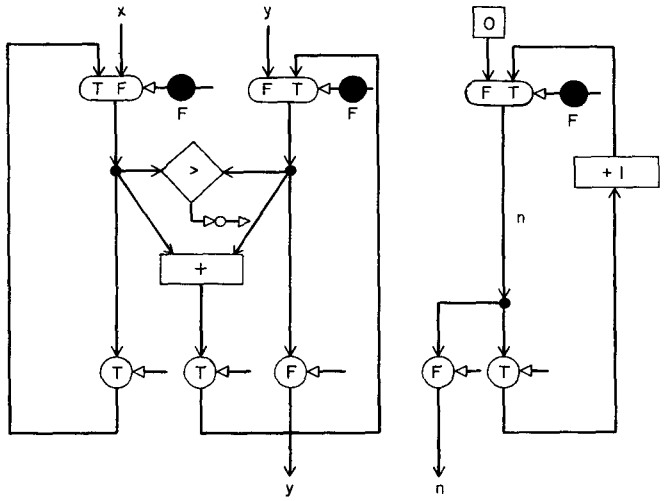


Figure 8. Data-flow representation of the basic program.

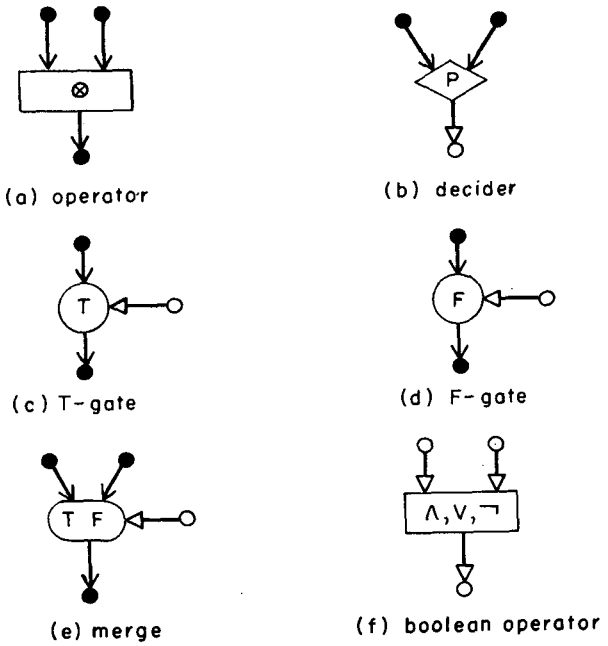


Figure 7. Actors of the basic data-flow language.

the value true at its control input. It will absorb the data token on its input arc and place nothing on its output arc if a false-valued control token is received. Similarly, the F-gate will pass its input data token to its output arc only on receipt of a false-valued token on the control input. Upon receipt of a true-valued token, it will absorb the data token.

A merge actor has a true input, a false input, and a control input. It passes to its output arc a data token from the input arc corresponding to the value of the control token received. Any tokens on the other input are not affected.

As with the elementary schemas, a link or actor is not enabled to fire unless there is no token on any of its output arcs.

Using the actors and links of the basic data-flow language, conditionals and iteration can be easily represented. In illustration, Figure 8 gives a basic data-flow program for the following computation:

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

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

The Basic Data-Flow Processor

Two problems must be faced in adapting the design of the

elementary data-flow processor for basic data-flow programs. The first task is to expand the architecture of the elementary machine to incorporate decision capability by implementing deciders, gates and merges. A fairly straightforward solution to this problem will be presented.

However, in contrast to elementary data-flow programs, the nodes of a basic data-flow program do not fire equally often during execution. As computation proceeds, different parts of the program become active or quiescent as iterations are initiated and completed, and as decisions lead to selection of alternate parts of a program for activation. Thus it would be wasteful to assign a Cell to each instruction for the duration of program execution. The basic data-flow processor must have a multi-level memory system such that only the active instructions of a program occupy the Instruction Cells of the processor. In the following sections we first show how decision capability may be realized by augmenting the elementary processor; then we show how an auxiliary memory system may be added so the Instruction Cells act as a cache for the most active instructions.

Decision Capability

The organization of a basic data-flow processor without the two-level memory is shown in Fig. 9. As in the elementary processor, each Instruction Cell consists of three Registers and holds one instruction together with spaces for receiving its operands. Each instruction corresponds to an operator, a decider, or a Boolean operator of a basic data-flow program. The gate and merge actors of the data-flow program are not represented by separate instructions; rather, the function of the gates is incorporated into the instructions associated with operators and deciders in a manner that will be described shortly, and the function of the merge actors is implemented for free by the nature of the Distribution Network.

Instructions that represent operators are interpreted by the Operation Units to yield data packets as in the elementary processor. Instructions that represent deciders or Boolean operators are interpreted by the Decision Units to yield control packets having one of the two forms

$$\left\{\underline{\text{gate}},\ \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},\ \langle\text{address}\rangle\right\}$$

$$\left\{\underline{\text{value}},\ \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},\ \langle\text{address}\rangle\right\}$$

A gate-type control packet performs a gating function at the addressed operand register. A value-type control packet provides a Boolean operand value to an Instruction Cell that represents a Boolean operator.

The six formats for the contents of Instruction Cells in the basic processor are given in Figure 10. The use of each Register is specified in its leftmost field:

- I instruction register
- D operand register for data values
- B operand register for Boolean values

Only Registers specified to be operand registers of consistent type may be addressed by instructions of a valid program.

The remaining fields in the Instruction Cell formats are: an instruction code, op, pr or bo, that identifies the class and variation of the instruction in the Cell; from one to three destination addresses d1, d2, d3 that specify target operand registers for the packets generated by instruction execution; in the case of deciders and Boolean operators, a result tag t1, t2, t3 for each destination that specifies whether the control packet is of gate-type (tag = gate) or of value type (tag = value); and, for each operand register, a gating code g1, g2 and either a data receiver v1, v2 or a control receiver c1, c2.

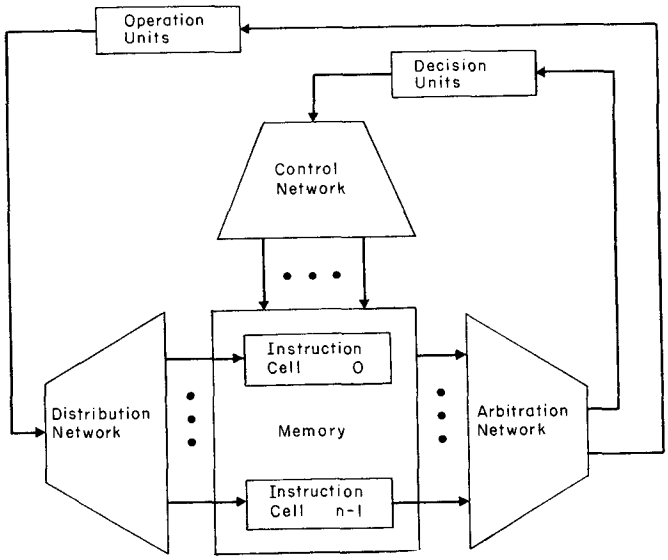


Figure 9. Organization of a basic data-flow processor without two-level memory.

The gating codes permit representation of gate actors that control the reception of operand values by the operator or decider represented by the Instruction Cell. The meanings of the code values are as follows:

code value	meaning
no	the associated operand is not gated.
true	an operand value is accepted by arrival of a true gate packet; discarded by arrival of a false gate packet.
false	an operand value is accepted by arrival of a false gate packet; discarded by arrival of a true gate packet.
cons	the operand is a constant value.

The structure of a data or control receiver (Fig. 11) provides space to receive a data or Boolean value, and two flag fields in which the arrival of data and control packets is recorded. The gate flag is changed from off to true or false by a true or false gate-type control packet; the value flag is changed from off to on by a data packet or value type control packet according to the type of receiver.

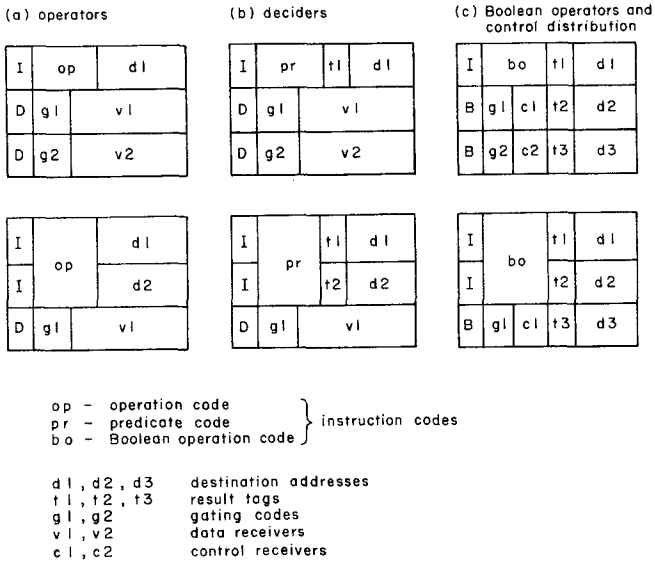


Figure 10. Instruction Cell formats for the basic processor.

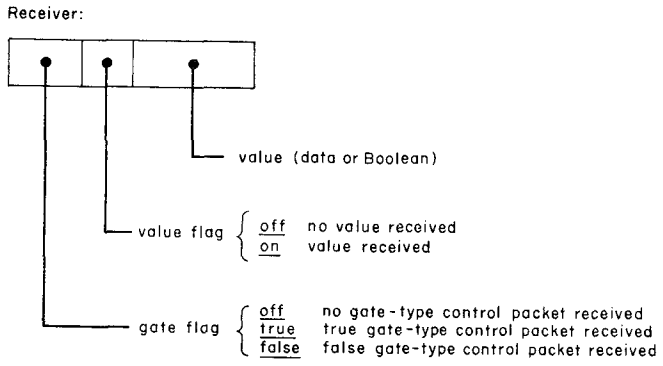
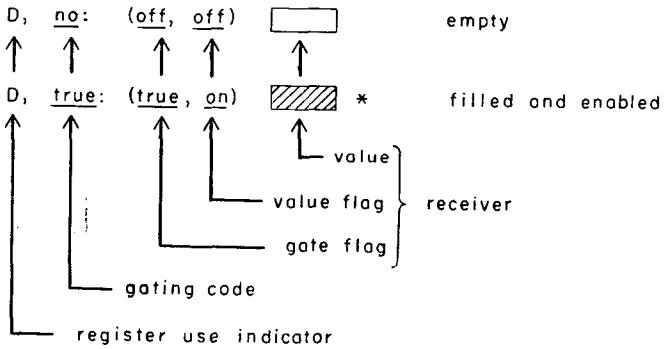


Figure 11. Structure and states of receivers.

Instruction Cell Operation

The function of each Instruction Cell is to receive data and control packets, and, when the Cell becomes enabled, to transmit an operation or decision packet through the Arbitration Network and reset the Instruction Cell to its initial status. An Instruction Cell becomes enabled just when all three of its registers are enabled. A register specified to act as an instruction register is always enabled. Registers specified to act as operand registers change state with the arrival of packets directed to them. The state transitions and enabling rules for data operand registers are defined in Fig. 12.

In Fig. 12 the contents of an operand register are represented as follows:



The asterisk indicates that the Register is enabled. Events denoting arrival of data and control packets are labelled thus:

- d data packet
- t true gate-type control packet
- f false gate-type control packet

With this explanation of notation, the state changes and enabling rules given in Fig. 12 should be clear. Similar rules apply to the state changes and enabling of Boolean operand registers. Note that arrival of a gate-type control packet that does not match the gating code of the Register causes the associated data packet to be discarded, and resets the Register to its starting condition.

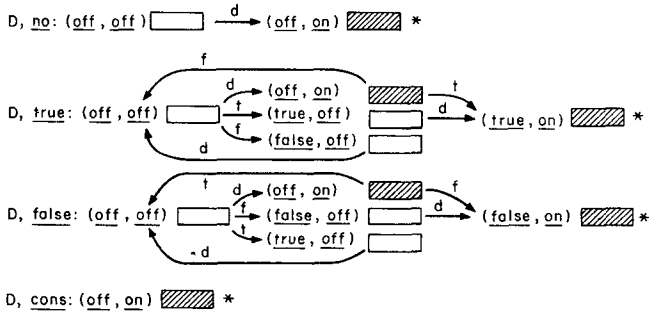


Figure 12. State transition and enabling rules for data operand registers.

The operation packets sent to Operation Units and decision packets sent to Decision Units consist of the entire contents of the Instruction Cell except for the gating codes and receiver status fields. Thus the packets sent through the Arbitration Network have the following formats:

To the Operation Units:

```
op, v1, v2, d1
op, v1, d1, d2
```

To the Decision Units:

```
pr, v1, v2, t1, d1
pr, v1, t1, d1, t2, d2
bo, c1, c2, t1, d1, t2, d2, t3, d3
bo, c1, t1, d1, t2, d2, t3, d3
```

An initial configuration of Instruction Cells corresponding to the basic data-flow program of Fig. 8 is given in Fig. 13. For simplicity, Cells containing control distribution and data forwarding instructions are not shown. Instead, we have taken the liberty of writing any number of addresses in the destination fields of instructions.

The initial values of x and y are placed in Registers 2 and 5. Cells 1 and 2, containing these values, are then enabled and present to the Arbitration Network the operation packets

{ident; 8, 11, 14 / x}

and

{ident; 7, 13, 20 / y}

These packets are directed to an identity Operation Unit which merely creates the desired data packets with the values of x and y and delivers the packets to the Distribution Network.

Upon receipt by the Memory of the data packets directed to Registers 7 and 8, cell 3 will be enabled and will transmit its decision packet to a Decision Unit to perform the less than function. The result of the decision will be returned through the Control Network as five control packets. If the result is true, Cells 4, 5 and 6 will be enabled and will send their contents through the

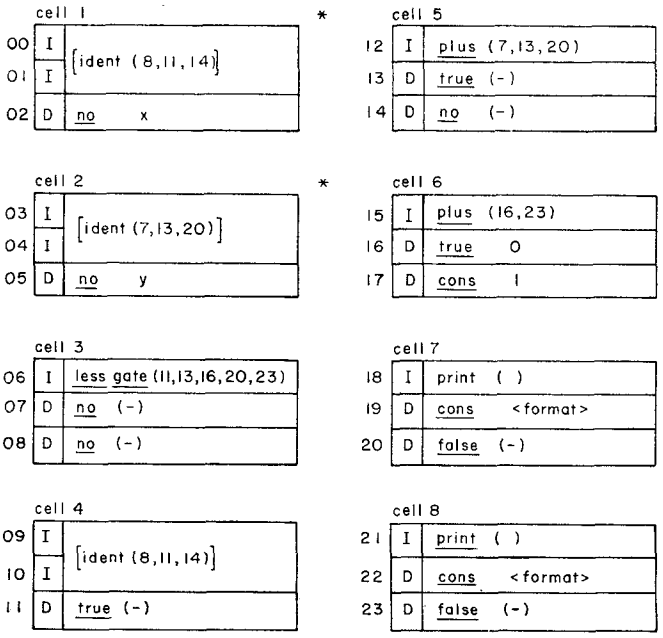


Figure 13. Instruction Cell initialization for the basic data-flow program in Figure 8.

Arbitration Network to Operation Units capable of performing the identity and addition operations. If the result of the decision is false, output cells 7 and 8 will be enabled, and cells 4, 5, and 6 will have their gated operands deleted.

Two-Level Memory Hierarchy

The high level of parallel activity achievable in data-flow processors makes a unique form of memory hierarchy feasible: the Instruction Cells are arranged to act as a cache for the most active instructions of the data-flow program. Individual instructions are retrieved from auxiliary memory (the Instruction Memory) as they become required by the progress of computation, and instructions are returned to the Instruction Memory when the Instruction Cells holding them are required for more active parts of the program.

The organization of a basic data-flow processor with Instruction Memory is given in Fig. 14.

Instruction Memory

The Instruction Memory has a storage location for each possible register address of the basic processor. These storage locations are organized into groups of three locations identified by the address of the first location of the group. Each group can hold the contents of one Instruction Cell in the formats already given in Fig. 10.

A memory command packet {a, retr} presented to the command port of the Instruction Memory, requests retrieval of an instruction packet {a, x} in which x is the Cell contents stored in the group of locations specified by address a. The instruction packet is delivered at the retrieve port of the Instruction Memory.

An instruction packet {a, x} presented at the store port of the Instruction Memory requests storage of Cell contents x in the three-location group specified by address a. However, the storage is not effective until a memory command packet {a, store} is received by the Instruction Memory at its command port, and any prior retrieval request has been honored. Similarly, retrieval requests are not honored until prior storage requests for the group have taken effect.

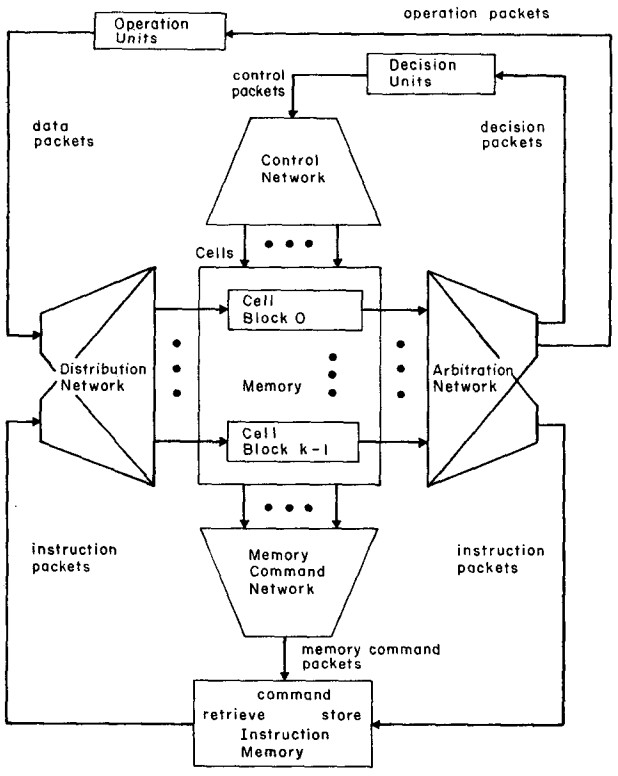


Figure 14. Organization of the basic data-flow processor with auxiliary memory.

We envision that the Instruction Memory would be designed to handle large numbers of storage and retrieval requests concurrently, much as the input/output facilities of contemporary computer systems operate under software control.

Cell Block Operation

For application of the cache principle to the basic data-flow processor, an Instruction Memory address is divided into a major address and a minor address, each containing a number of bits of the address. One Cell Block of the processor is associated with each possible major address. All instructions having the same major address are processed by the Instruction Cells of the corresponding Cell Block. Thus the Distribution and Control Networks use the major address to direct data packets, control packets, and instruction packets to the appropriate Cell Block. The packets delivered to the Cell Block include the minor address, which is sufficient to determine how the packet should be treated by the Cell Block.

Operation and decision packets leaving a Cell Block have exactly the same format as before. Instruction packets leaving a Cell Block have the form {m, x} where m is a minor address and x is the contents of an Instruction Cell. The major address of the Cell Block is appended to each instruction packet as it travels through the Arbitration Network. In the same way, memory command packets leave the Cell Block with just a minor address, which is augmented by the major address of the Cell Block during its trip through the Memory Command Network.

Fig. 15 shows the structure of a Cell Block. Each Instruction Cell is able to hold any instruction whose major address is that of the Cell Block. Since many more instructions share a major address than there are Cells in a Cell Block, the Cell Block includes an Association Table which has an entry {m, i} for each Instruction Cell: m is the minor address of the instruction to which the Cell is assigned, and i is a Cell status indicator whose values have significance as follows:

status value	meaning
free	the Cell is not assigned to any instruction
engaged	the Cell has been engaged for the instruction having minor address m, by arrival of a data or control packet
occupied	the Cell is occupied by an instruction with minor address m

The Stack element of a Cell Block holds an ordering of the Instruction Cells as candidates for displacement of their contents by newly activated instructions. Only Cells in occupied status are candiates for displacement.

Operation of a Cell Block can be specified by giving two procedures -- one initiated by arrival of a data or control packet at the Cell Block, and the other activated by arrival of an instruction packet from the Instruction Memory.

Procedure 1: Arrival of a data or control packet {n, y} where n is a minor address and y is the packet content.

step 1. Does the Association Table have an entry with minor address n? If so, let p be the Cell corresponding to the entry, and go to step 5. Otherwise continue with step 2.

step 2. If the Association Table shows that no Instruction Cell has status free, go to step 3. Otherwise let p be a Cell with status free. Let the Associa-

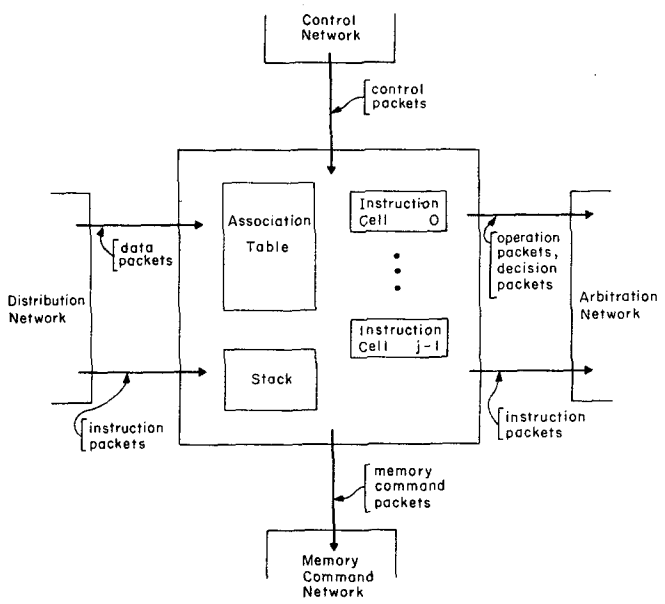


Figure 15. Structure of a Cell Block.

tion Table entry for p be {m, free}; go to step 4.

step 3. Use the Stack to choose a Cell p in occupied status for preemption; let the Association Table entry for p be {m, occupied}; transmit the contents z of Cell p as an instruction packet {m, z} to the Instruction Memory via the Arbitration Network; transmit the memory command packet {m, store} to the Instruction Memory through the Memory Command Network.

step 4. Make an entry {n, engaged} for Cell p in the Association Table; transmit the memory command packet {n, retr} to the Instruction Memory via the Memory Command Network.

step 5. Update the operand register of Cell p having minor address n according to the content y of the data or control packet (the rules for updating are those given in Fig. 12). If Cell p is occupied the state change of the register must be consistent with the instruction code or the program is invalid. If Cell p is engaged, the changes must be consistent with the register status left by preceding packet arrivals.

step 6. If Cell p is occupied and all three registers are enabled (according to the rules of Fig. 12), the Cell p is enabled: transmit an operation or decision packet to the Operation Units or Decision Units through the Arbitration Network; leave Cell p in occupied status holding the same instruction with its operand registers reset (receivers empty with the gate and value flags set to off). Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

Procedure 2: Arrival of an instruction packet {n, x} with minor address n and content x.

step 1. Let p be the Instruction Cell with entry {n, engaged} in the Association Table.

step 2. The status of the operand registers of Cell p must be consistent with the content x of the instruction packet, or the program is invalid. Update the contents of Cell p to incorporate the instruction and operand status information in the instruction packet.

step 3. Change the Association Table entry for Cell p from {n, engaged} to {n, occupied}.

step 4. If all registers of Cell p are enabled, then Cell p is enabled: transmit an operation or decision packet to the Operation Units or Decision Units through the Arbitration Network; leave Cell p in occupied status holding the same instruction with its operand registers reset. Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

Conclusion

The organization of a computer which allows the execution of programs represented in data-flow form offers a very promising solution to the problem of achieving highly parallel computation. Thus far, the design of two processors, the elementary and the basic data-flow processors, has been investigated. The elementary processor is attractive for stream-oriented signal processing applications. The basic processor described here is a first step toward a highly parallel processor for numerical algorithms expressed in a Fortran-like data-flow language. However, this goal requires further elaboration of the data-flow architecture to encompass arrays, concurrent activation of procedures, and some means of exploiting the sort of parallelism present in vector operations. We are optimistic that extensions of the architecture to provide these features can be devised, and we are hopeful that these concepts can be further extended to the design of computers for general-purpose computation based on more complete data-flow models such as presented by Dennis [4].

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. First version of a data flow procedure language. Symposium on Programming, Institut de Programmation, University of Paris, Paris, France, April 1974, 241-271.
5. Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. November 1973 (submitted for publication).
6. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974.
7. Dennis, J. B., and D. P. Misunas. The Design of a Highly Parallel Computer for Signal Processing Applications. Computation Structures Group Memo 101, Project MAC, M.I.T., Cambridge, Mass., July 1974.
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. Rodriguez, J. E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.