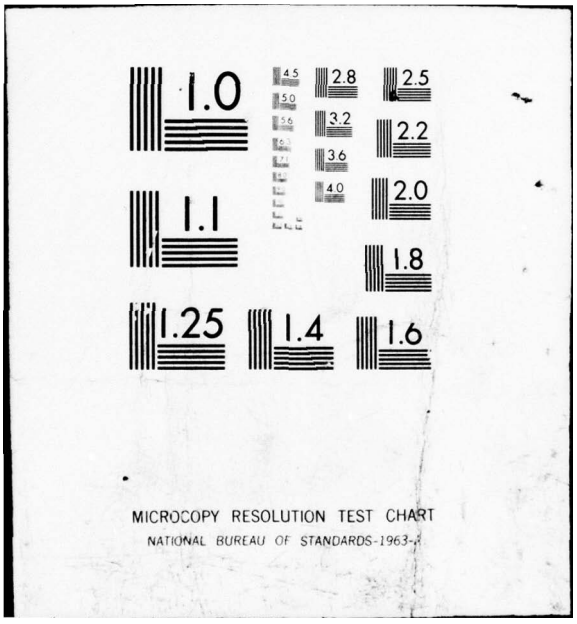
AD-A052 538 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
A COMPUTER ARCHITECTURE FOR DATA-FLOW COMPUTATION.(U)
MAR 78 D P MISUNAS N00014-70-A-0362-0006
UNCLASSIFIED MIT/LCS/TM-100 NL

1 OF 2
AD
A052538


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-100

A COMPUTER ARCHITECTURE FOR DATA-FLOW COMPUTATION

DAVID P. MISUNAS

MARCH 1978

THIS RESEARCH WAS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY OF THE DEPARTMENT OF DEFENSE AND WAS MONITORED BY THE OFFICE OF NAVAL RESEARCH UNDER CONTRACT No. N00014-70-A-0362-0006

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TM-100	2. GOVT ACCESSION NO: Master's thesis,	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Computer Architecture for Data-Flow Computation		5. TYPE OF REPORT & PERIOD COVERED: S.M.Thesis, May 16. 1975
		6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TM-100
7. AUTHOR(s): David P. Misunas		8. CONTRACT OR GRANT NUMBER(s): N0014-70-A-0362-0006
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, Ma 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va 22217		12. REPORT DATE: MAR 78
		13. NUMBER OF PAGES: 109
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer architecture
data-flow
parallelism

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The structure of a computer which utilizes a data-flow program representation as its base language is described. The use of the data-flow representation allows full exploitation by the processor of the parallelism and concurrency achievable through the data-flow form. The unique architecture of the processor avoids the usual problems of processor switching and memory/processor interconnection by the use of interconnection networks which have a great deal of inherent parallelism. The structure of the processor

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 648

20. allows a large number of instructions to be active simultaneously. These active instructions pass through the interconnection networks concurrently and form streams of instructions for the pipelined functional units.

Due to the cyclic nature of an iterative computation, the possibility of deadlock can arise in the performance of such a computation within the data-flow architecture. A deadlock is caused by the interaction of several simultaneously active cycles of the same iterative computation. The use of a recursive rather than iterative representation of a computation avoids the deadlock problem and provides a more efficient implementation of the computation within the architecture. For this reason, a program executed by the data-flow processor is restricted to an acyclic directed graph representation.

MIT/LCS/TM-100

A COMPUTER ARCHITECTURE FOR DATA-FLOW COMPUTATION

David P. Misunas

MARCH 1978

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

A COMPUTER ARCHITECTURE FOR DATA-FLOW COMPUTATION

by

David Peter Misunas

ABSTRACT

The structure of a computer which utilizes a data-flow program representation as its base language is described. The use of the data-flow representation allows full exploitation by the processor of the parallelism and concurrency achievable through the data-flow form. The unique architecture of the processor avoids the usual problems of processor switching and memory/processor interconnection by the use of interconnection networks which have a great deal of inherent parallelism. The structure of the processor allows a large number of instructions to be active simultaneously. These active instructions pass through the interconnection networks concurrently and form streams of instructions for the pipelined functional units.

Due to the cyclic nature of an iterative computation, the possibility of deadlock can arise in the performance of such a computation within the data-flow architecture. A deadlock is caused by the interaction of several simultaneously active cycles of the same iterative computation. The use of a recursive rather than iterative representation of a computation avoids the deadlock problem and provides a more efficient implementation of the computation within the architecture. For this reason, a program executed by the data-flow processor is restricted to an acyclic directed graph representation.

Key Words: computer architecture, data-flow, parallelism

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, M.I.T., on May 16, 1975, in partial fulfillment of the requirements for the degree of Master of Science.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the invaluable patience, assistance, and support of his thesis supervisor, Professor Jack Dennis. The author is also greatly appreciative of the valuable interactions he has had with members of the Computation Structures Group at Project MAC. Special thanks go to Jim Rumbaugh, Bob Jacobsen, and Dave Isaman.

The work reported in this thesis was carried out at Project MAC, M.I.T. and was supported by the Advanced Research Projects Agency, Department of Defence, under Office of Naval Research contract number N00014-70-A-0362-0006.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1.1 Architecture of Parallel Systems

Highly parallel computer systems have evolved in a manner which often necessitates the placing of unusual constraints on programs and data. Parallel machines such as the Illiac IV [7] and the CDC Star [18] can realize their full potential only for data represented in array or vector format. In these architectures, the programmer is forced to use such data representations, even when inappropriate, to achieve highly parallel execution.

A number of methods have been developed to exploit simultaneous or concurrent operation, however, the implementation of these techniques within a traditional von Neumann architecture has not fully utilized their potential. This applies both to the various procedures for increasing the performance of a single processor and those for exploiting multiple processors in a computer system.

Three techniques are currently popular for increasing the parallel activity within a single processor. These are:

1. pipelining of operations
2. overlapped memory access
3. instruction lookahead

The pipelining of an arithmetic operation distributes the performance of the operation over time rather than space. That is, rather than utilizing several functional units of a specific type to increase the processing rate, one larger functional unit is employed, and the operation is broken

into a number of smaller operations which are performed simultaneously upon a stream of values. Although the performance of a single operation can actually take longer in a pipelined functional unit, the fact that a large number of operations are being performed concurrently can produce a very high processing rate.

In order to fully utilize the technique of pipelining, the data must be represented as a vector; if there are gaps in the stream of values supplied to the pipeline, the processing rate can actually be decreased from that of a single conventional functional unit. Current stream-oriented processors as the CDC Star [18] and the TI ASC [34] do not have the capability to form data into streams, that burden must be born by the programmer.

The technique of overlapped memory access merely extends the concept of pipelining to the fetching of instructions from memory. If the memory of a computer system is interleaved; that is, if the memory is divided into a number of sections, and the instructions and data of a program are distributed over the sections, then several items can be accessed simultaneously. If the instructions of a program are arranged so consecutive instructions are contained in separate memories, then instruction fetching can be pipelined, and instructions can be supplied at a very fast rate. However, a problem arises when a conditional is encountered because the system does not know which of the set of possible succeeding instructions to fetch until after the conditional has been executed.

The use of instruction lookahead in a processor allows the exploitation of multiple arithmetic units by decomposing the instruction stream into independent elements. For example, consider the arithmetic expression A + B +

(C * D). The two computations A + B and C * D can be performed simultaneously in separate functional units. The IBM 360 model 91 [5, 33] and the CDC 6600 [32] have developed techniques for exploiting this property for short instruction sequences; however, once again, any branching in the program will disrupt the flow of instructions to the functional units and decrease the processing capability of the architecture.

In illustration of the problems encountered in exploiting these techniques, consider the IBM 360/91 [16]. The functional capability of this processor is 70 million instructions per second (MIPS). However, the instruction decoder can only supply instructions at a rate of 16 MIPS using the technique of lookahead. An average incidence of conditional instructions reduces the performance of the processor to 6 MIPS. Thus, the processing capability of the architecture cannot be fully realized, and with the lookahead of eight instructions which is used, it is difficult to have an adequate instruction mix to fully utilize the multiple functional units.

Another common technique for increasing the performance of a processor is that of separating the memory system of the processor into levels of memory. This allows the expensive fast memory, known as a cache, to contain only the most active instructions and data. In order to exploit this technique, instructions and data are organized into blocks known as pages, and these pages are transferred between levels of storage. The utilization of pages is wasteful in that often not all space in the page is filled (known as breakage). Also, in order to reference one item on a page, the entire page must be transferred to the cache, moving a great deal of unnecessary information. In a multiple processor environment another problem arises,

that of several copies of a page existing in the caches of different processors. If one copy is altered, the other processors will not have knowledge of that fact.

In addition to increasing the performance of a single processor, there has been a trend toward connecting several processors together to form a multiprocessor system. Such a system can be either homogeneous; that is, each processor can be replaced by any other, or non-homogeneous, each processor has its own special function. However, in either case there is a degradation in performance of a multiprocessor system from the possible performance of the individual processors working on separate problems due to memory conflict and the necessary operating system. It is claimed that, in general, for a two processor system, 2.2 times as much hardware is required as for the two separate processors, and the resulting performance is 1.8 times the performance of a single processor [2].

Homogeneous multiprocessor systems such as array and pipeline processors have the drawback that they can only achieve their full capability for special data representations, as has been previously discussed. Let us consider more conventional methods of connecting processors in a multiprocessor system. There are three techniques which are commonly utilized:

1. time-shared or common bus
2. crossbar switch
3. multiport memory

A time-shared bus provides a single path interconnecting all units of a system (Figure 1.1). Any unit which wishes to transfer data to another unit must obtain control of the bus, transfer the desired information, and

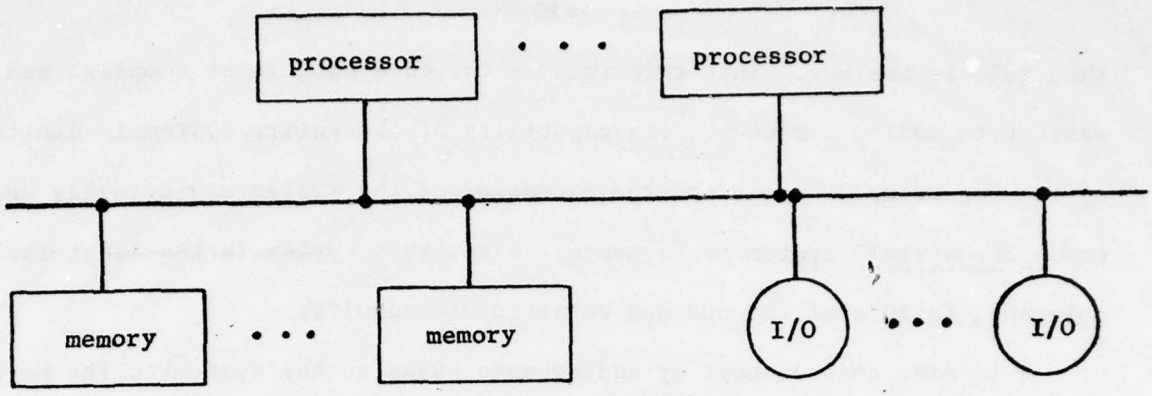


Figure 1.1. Structure of a time-shared bus.

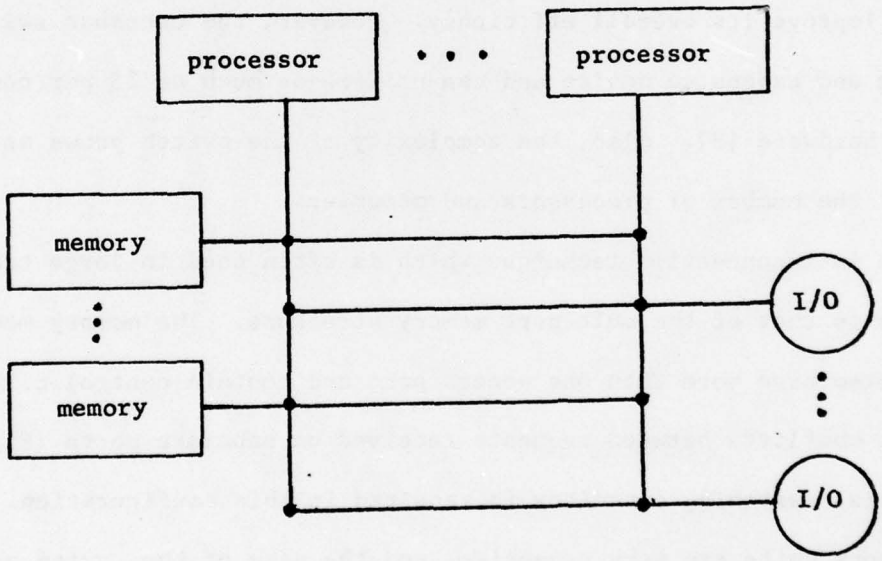


Figure 1.2. Structure of a crossbar switch.

then release the bus. This technique is the cheapest, least complex, and easiest to modify. However, the capability of the entire system is limited by the bus transfer rate, and the expansion of the system may actually degrade the overall system performance. Also, this system is the least fault tolerant, failure of the bus has catastrophic results.

Extending this concept by adding more buses to the system to the point where each memory has a bus which can be connected to any processor, we realize a structure known as a crossbar switch (Figure 1.2). Although the possibility of conflict still exists in this configuration, it is possible for a number of transfers to occur simultaneously. This structure has the potential for the highest system transfer rate, and expansion of the system should improve its overall efficiency. However, the crossbar switch is a complex and expensive device and can utilize as much as 25 per cent of the system hardware [8]. Also, the complexity of the switch grows as the product of the number of processors and memories.

An interconnection technique which is often used in large time sharing systems is that of the multiport memory structure. The memory modules of the system have more than one access port and contain control circuitry to resolve conflicts between requests received on separate ports (Figure 1.3). No special switching circuitry is required in this configuration. However, the memory units are very expensive, and the size of the system and configuration options are limited by the number and type of memory ports available.

The methods of structuring multiple processor systems and improving the performance of a processor all have serious drawbacks to the full exploitation of the capabilities of the processors. In this thesis an approach to

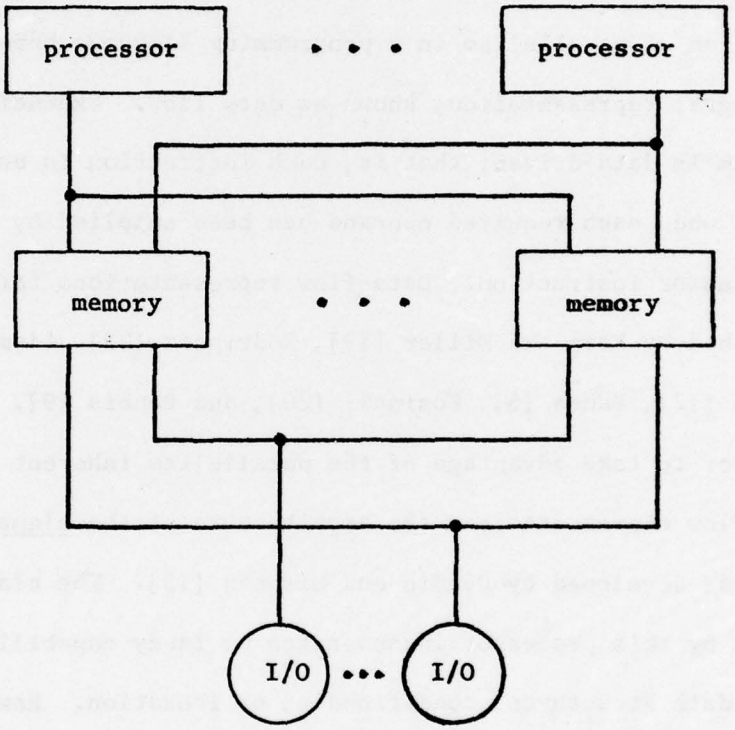


Figure 1.3. Structure of a multiport memory system.

the structuring of a computer system which offers attractive solutions to many of these problems is presented.

1.2 The Data-Flow Approach

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 [19], Rodriguez [28], Adams [1], Dennis and Fosseen [12], Bährs [6], Kosinski [20], and Dennis [9].

In order to take advantage of the parallelism inherent in an elementary data-flow representation, the architecture of the elementary data-flow processor was developed by Dennis and Misunas [13]. The class of programs implemented by this processor incorporates no fancy capabilities such as recursion, data structures, conditionals, or iteration. However, the language and its corresponding architecture are well-suited for the representation of signal processing computations such as filtering, waveform generation, and fast Fourier transforms, in which a group of operations is to be performed once for each sample (in time) of the signals being processed.

In the development of the basic data-flow processor we added conditional and iterative constructs to the language and architecture [14]. Also, the basic data-flow architecture incorporates a multi-level memory system in which the active memory is operated as a cache, and individual instruc-

tions are retrieved from the auxiliary memory as they are required for computation.

An extension of the architectural concepts to a general-purpose computer, incorporating procedures, recursive program activation, and data structures has been completed recently [11, 24, 26, 27] and is the subject of this thesis.

The problems of processor switching and memory/processor interconnection are avoided within the data-flow architecture by the use of interconnection networks which have a great deal of inherent parallelism. Sections of the machine communicate by means of fixed size information packets, and delays in packet transmission within the networks do not affect the utilization of the hardware. The interconnection networks are large, but grow at a much slower rate than a crossbar switch and require none of the global control circuitry necessary for the switch.

The structure of a data-flow processor allows a large number of instructions to be active simultaneously. These active instructions pass through the networks concurrently and form streams of instructions for the pipelined functional units.

The processor does not utilize an instruction register or instruction decoder; an instruction proceeds on its own when its operands are ready and delivers its results to other instructions which are waiting for them. No software operating system is necessary within the architecture. Processor allocation, the formation of instructions into streams for the functional units, and the transfer of information between levels of memory is efficiently

accomplished by the hardware of the machine.

The configuration of component parts in a data-flow processor is readily chosen to fit a desired application, allowing the architecture to be applicable to a wide variety of problems. The processor is also expandable, and the performance of the processor increases linearly with an increase in size.

The exploitation of data dependencies in programs has been investigated previously by Shaprio, Saint, and Presberg [30], Miller and Cocke [22], Seeber and Lindquist [31], and Rumbaugh [29]. Indeed, such is the goal of the look-ahead techniques utilized in architectures such as the IBM 360/91 and the CDC 6600. The approach taken in the data-flow processor differs from these approaches in that it utilizes a radically different concept of computer organization which offers attractive solutions to many of the problems encountered in adapting von Neumann machines for parallel computation, an architecture in which parallelism and concurrency are inherent in the structure of the processor.

1.3 Outline of the Thesis

The organization of the remaining chapters of the thesis is as follows:

Chapter 2 presents the data-flow language chosen as the base language of the processor. A program in the language is represented as an acyclic directed graph, eliminating the problems associated with the execution of cyclic program structures within a highly parallel architecture. The data-flow language has a capability greater than that of Algol 60 and is a generalization of pure Lisp.

Chapter 3 describes the architecture of the data-flow processor. The machine is presented in stages, examining first the execution of instructions within the processor and then the implementation of data structures. The structure of a multi-level memory system is considered next, and to complete the presentation, the implementation of procedures is described.

Chapter 4 illustrates the reason for restricting a program executed within the data-flow processor to an acyclic directed graph representation. The performance tradeoffs within the processor between cyclic and acyclic program structures are examined to demonstrate the necessity of such a restriction.

The final chapter summarizes the architectural concepts and discusses topics for future research.

Chapter 2

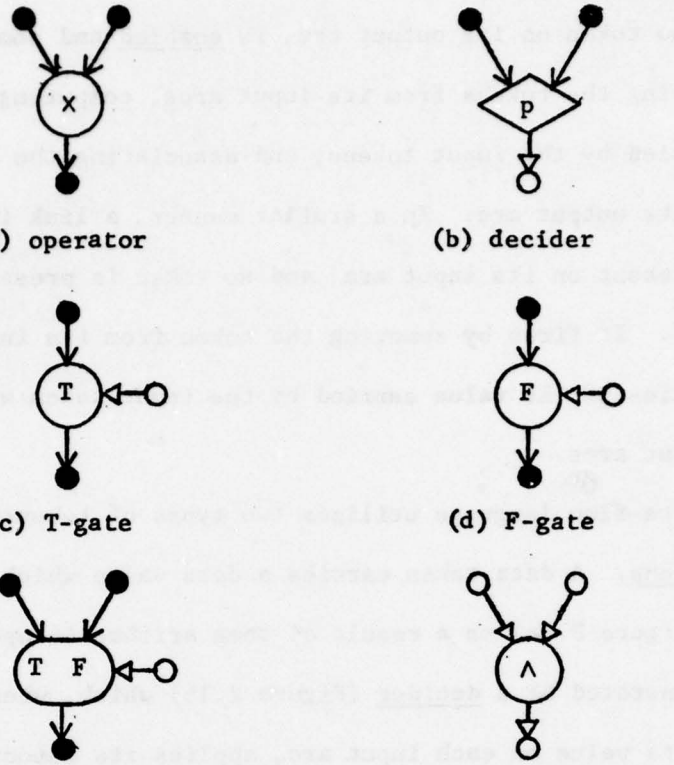
THE DATA-FLOW LANGUAGE

The data-flow language presented in this chapter serves as the base language for the architecture to be described in the following chapter. The semantics of the language is identical to that of the data-flow procedure language developed by Dennis [9]. The only major difference arises in the manner in which simultaneous procedure activations are differentiated. The model proposed by Dennis distinguishes between the data utilized by separate activations of a procedure through the use of a unique color associated with all data of each activation. The model described in this chapter is implememtation oriented and therefore uses a technique similar to the "copy rule" of Algol.

2.1 Data-Flow Programs

A program in the data-flow language is composed of two kinds of elements, called actors and links. An actor of the language can be either an operator, a decider, or a gate (Figure 2.1). Each actor has a number of input arcs which supply values necessary for its execution and one output arc upon which results are placed. A small dot or circle represents a link which has one input arc upon which it receives results from an actor and a number of output arcs over which it distributes copies of the result to other actors (Figure 2.2).

Values are conveyed over the arcs of the program by tokens which are represented by large solid dots. An actor with a token on each of its input



(a) operator (b) decider

(c) T-gate (d) F-gate

(e) merge (f) Boolean operator

Figure 2.1. Actors of the data-flow language.

(a) data link (b) control link

Figure 2.2. Links of the data-flow language.

arcs, and no token on its output arc, is enabled and sometime later will fire, removing the tokens from its input arcs, computing a result using the values carried by the input tokens, and associating the result with a token placed on its output arc. In a similar manner, 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 removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs.

The data-flow language utilizes two types of tokens: data tokens and control tokens. A data token carries a data value which is produced by an operator (Figure 2.1a) as a result of some arithmetic operation. A control token is generated at a decider (Figure 2.1b) which, when the decider receives a data value on each input arc, applies its associated predicate and produces either a true- or false-valued control token on its output arc.

Control tokens direct the flow of data tokens by means of either a T-gate, F-gate or merge actor (Figure 2.1c, d, e). A T-gate will pass the data token on its input arc to its output arc when it receives a control token carrying the value true over its control input arc. 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, an 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 (Figure 2.1e) 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.

In illustration of the capability of the data-flow language, consider the iterative data-flow program in Figure 2.3 for the following computation:

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

Upon exiting the body of the iteration, n is equal to the original value of x divided by y, and x is equal to the remainder.

In the data-flow program the control input arcs of the three merge actors carry false-valued control tokens in the initial configuration to allow the input values of x and y and the constant 0 to be admitted as initial values for the iteration. Once these values have been received, the predicate x>y is tested. If it is true, the value of y and the new value of x 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 x and n are delivered through the two F-gates, and the initial configuration is restored.

2.2 Structure Values

The values conveyed by tokens over the arcs of a data-flow program are either elementary values or structure values, and each value has an associated tag designating its type. The set of elementary values E contains

$$E = T \cup I \cup R \cup Q$$

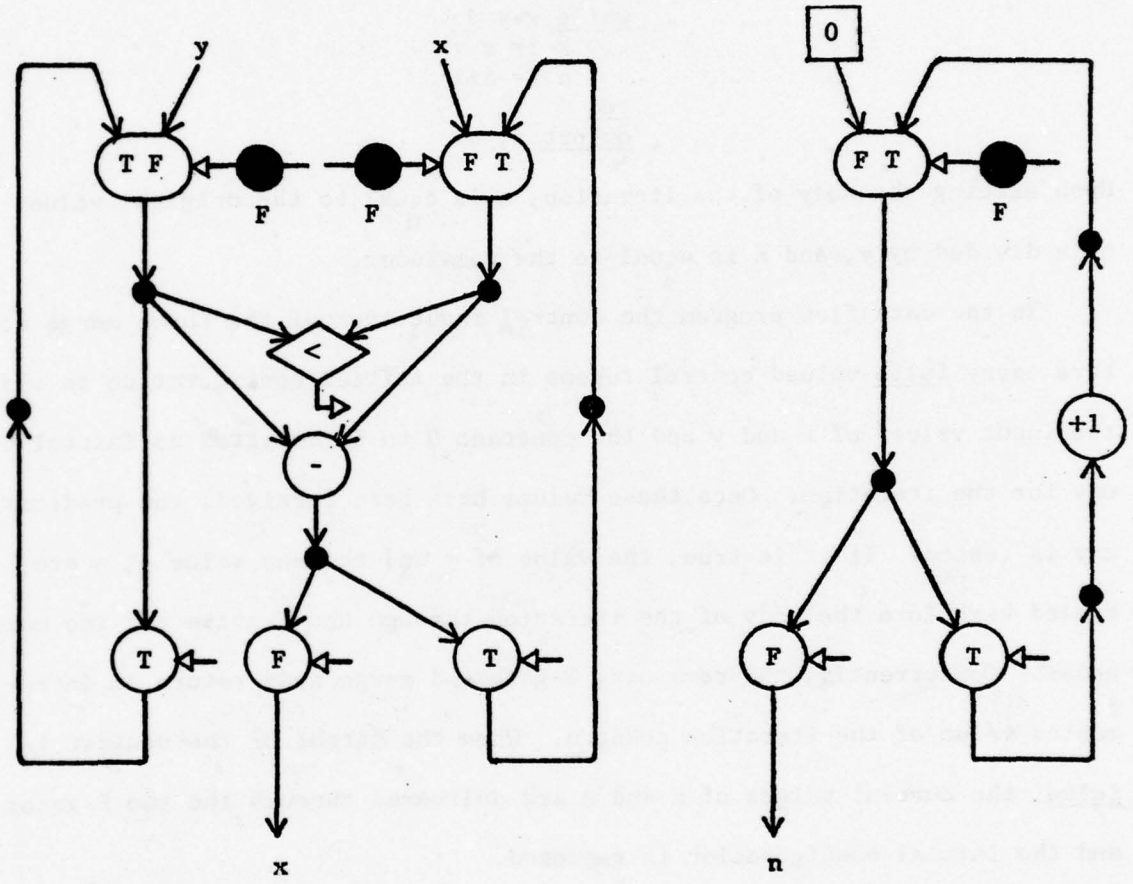


Figure 2.3. An iterative data-flow program.

where:

T = truth values
I = integers
R = reals
Q = strings

A structure value in a data-flow program is represented as an acyclic directed graph having one root node with the property that each node of the graph can be reached by a directed path from the root node. Each node of the graph is either a structure node or an elementary node. A structure node serves as the root node for a substructure of the structure and consists of a set of selector-value pairs

$$S = \{(s_1, v_1) \ldots (s_n, v_n)\}$$

where

$$s_i \in I \cup Q$$
$$v_i \in E \cup S \cup \{\underline{nil}\}$$

and s_i is the selector of node v_i. An elementary node has no emanating arcs; rather, an elementary value is associated with the node. A node with no emanating arcs and no associated elementary value has value {nil}.

A structure value is represented by a data token carrying a unique pointer to the root node of the structure. In Figure 2.4 the structure α contains three elementary values a, b, and c, designated by the simple selector L and the compound selectors R·L and R·R respectively. Structure node γ of structure α is shared with structure β and is designated by a different selector in β than in α.

A simple selector associated with a node can be either an integer or a string. A compound selector is formed by the concatenation of a number of simple selectors and specifies a path through the structure which can be

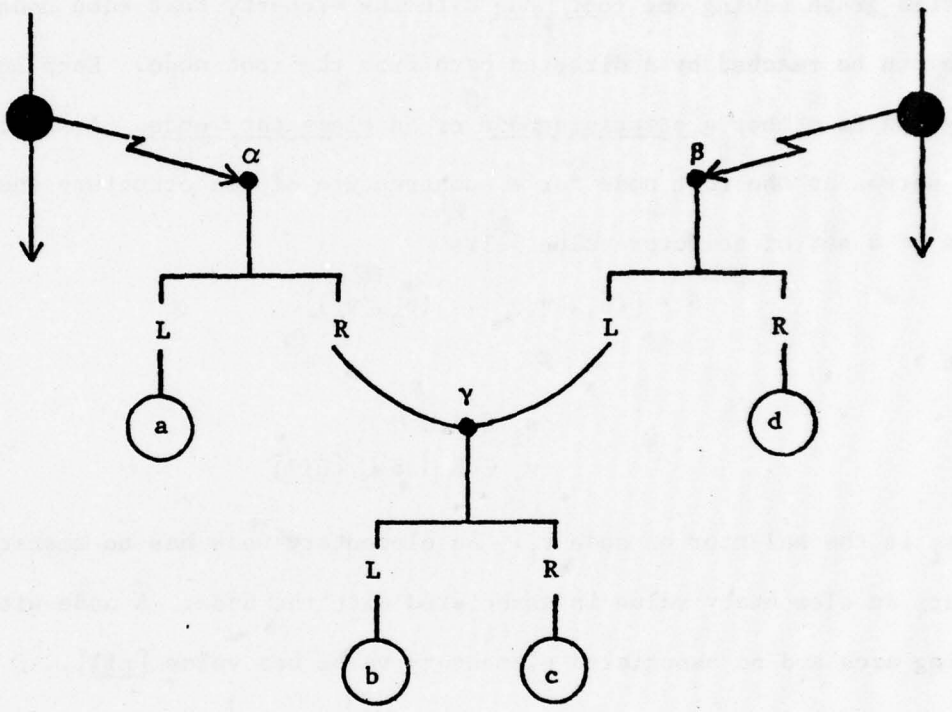


Figure 2.4. An example of two structures sharing a common substructure.

followed by applying the simple selectors in the stated order.

The structure values of a data-flow program are located within a heap [9]. The state of the program at any point in time is represented by a token distribution on the graph of the program and a heap. During each execution step of the program, some link or actor will fire, resulting in a new token distribution and, in some cases, a modified heap.

A node of the heap is accessible to a program only if some token carries a pointer to the node or the node can be reached by a directed path from some accessible node. Upon completion of an execution step of a program, any nodes of the heap made inaccessible by that step are deleted together with any emanating branches.

In order to generate and perform operations upon structure values, a number of new actors must be defined. The structure actors presented herein are not necessarily the only ones one might desire, however, they provide the necessary basic operations for the creation and manipulation of a structure and serve to illustrate the manner in which structures are handled within the data-flow architecture. The operation of each of the structure actors of the data-flow language is presented in Figures 2.5 to 2.8.

Structures are created through use of the construct actor (Figure 2.5). The actor accepts an elementary or structure value from each input and places on its output a structure containing the input values as components. Each input is labeled with the selector in the new structure to be associated with the value arriving on that input.

A value is retrieved from a structure by the select actor (Figure 2.6). The value in the input structure designated by the selector argument is placed on the output of the actor. The result can be either an elementary

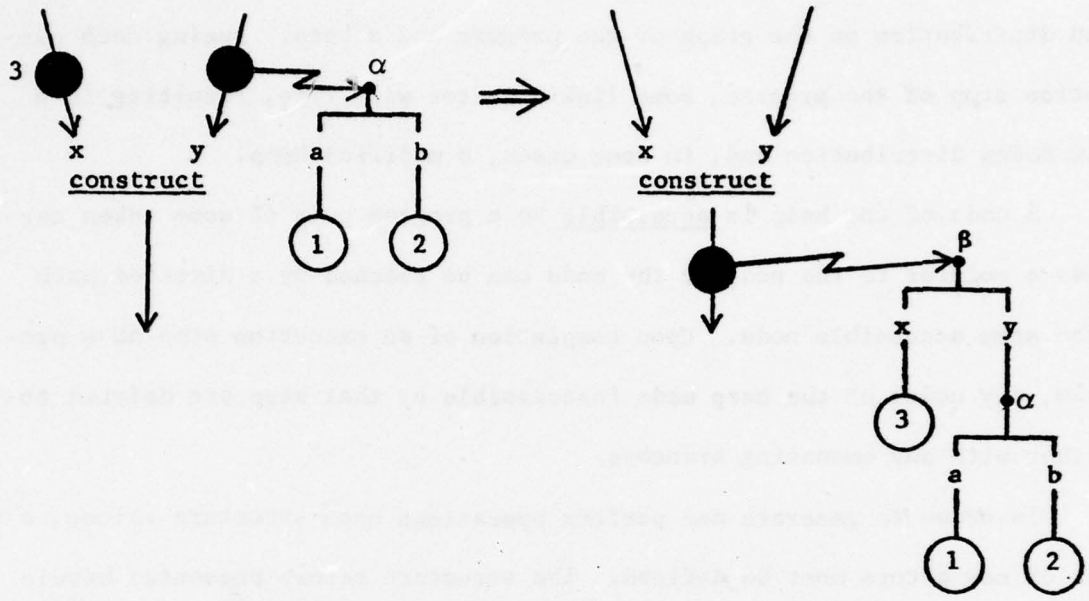


Figure 2.5. Operation of the construct actor.

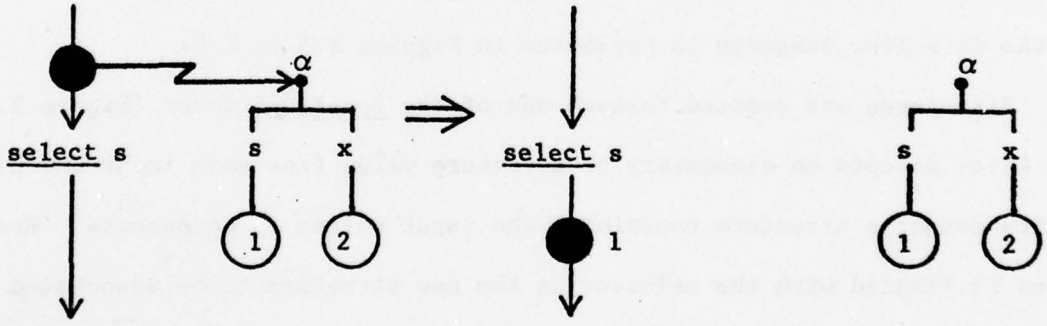


Figure 2.6. Operation of the select actor.

value or a structure value. If the argument of the actor is a multiple selector, the actor produces on its output the value at the end of the path designated by the multiple selector. The action of the actor is undefined if the input structure does not contain the specified selector(s).

Structure values in a data-flow program are not modified; rather, new structure values are created which are modifications of the original values, while the original values are preserved. The append and delete actors provide the means of creating these new structure values.

The structure produced by the firing of an append actor is a version of the input structure which contains a new or modified component (Figure 2.7). If the specified node of the input structure has a selector corresponding to the selector argument of the actor, the value designated by that selector in the new structure is the input value. Otherwise the specified selector-value pair is added to the node of the new structure. Identical elements of the input and output structures are shared between the two structures.

In a similar manner, the structure appearing on the output arc of a delete actor is a version of the input structure in which the specified node contains one fewer component (Figure 2.8). The specified node in the new structure is missing the selector-value pair designated by the selector argument. As with the append actor, identical elements are shared between the input and output structures.

2.3 Data-Flow Procedure Representation

In this section we present an approach to the description of procedures within the data-flow language. Procedures of the language are represented as acyclic directed graphs in a manner which is very attractive from both

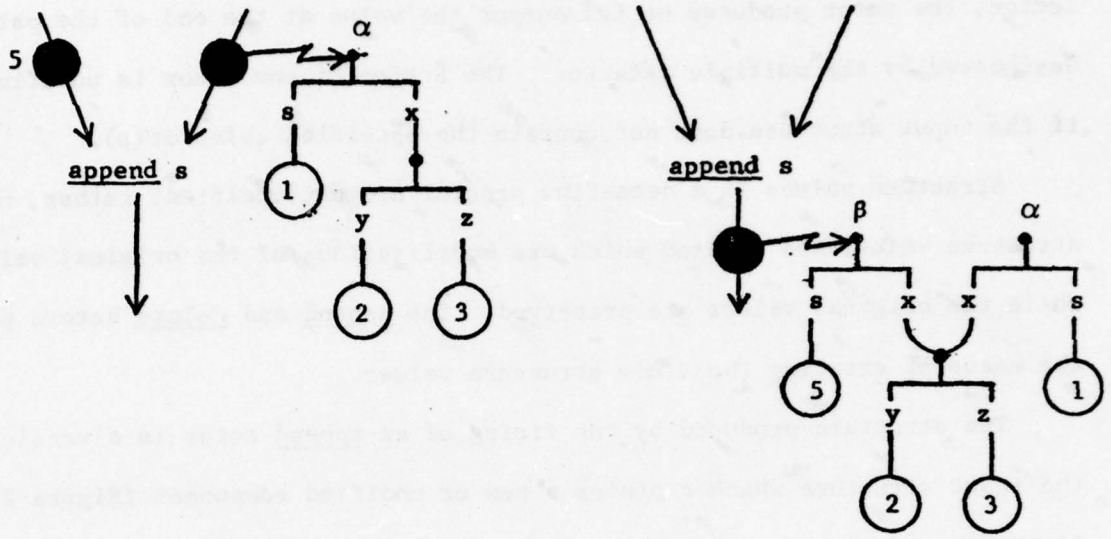


Figure 2.7. Operation of the append actor.

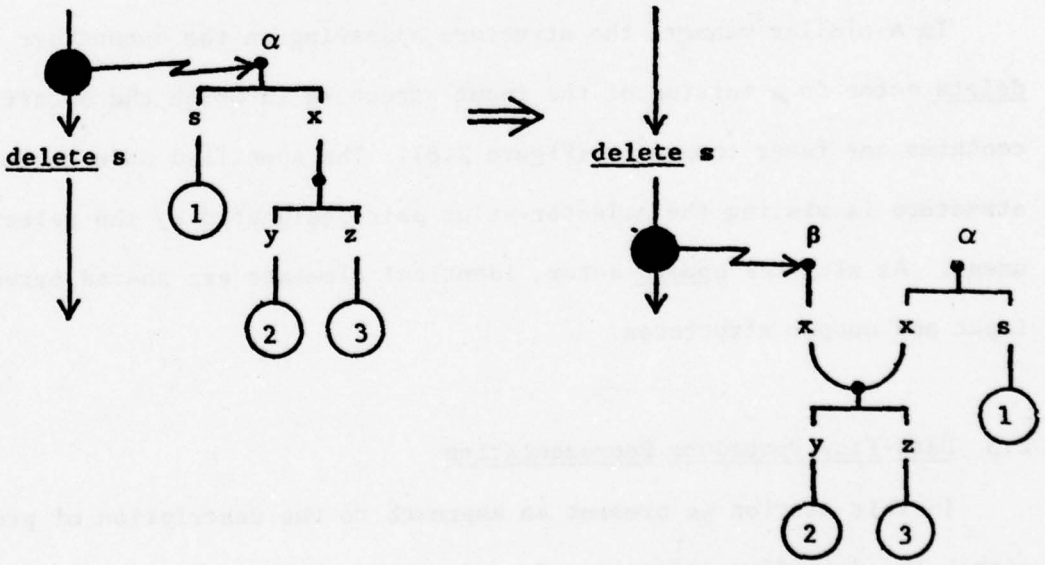


Figure 2.8. Operation of the delete actor.

a semantic viewpoint and an implementation viewpoint. Similar descriptions of a base language in terms of tree structures have been presented by Dennis [10], Amerasinghe [3], Ellis [15], and Henderson [17]. The application of such an approach to the description of a data-flow computation has not been previously considered, yet the extension to a data-flow representation appears to be very attractive.

A data-flow procedure is a data-flow program with a single input arc over which the argument arrives and a single output arc upon which the result is placed. The body of a procedure is represented as a data structure, and the procedure is referenced by a token carrying a pointer to the structured representation. Every procedure in the language is determinate; that is, the same result is produced by every activation of the procedure which receives the same input values.

To provide for procedure activation and termination, the apply and return actors are introduced into the data-flow language. The operation of these actors is shown in Figure 2.9. The apply actor receives two inputs, a procedure and an argument, which may be either an elementary value or a structure value. Upon firing, the actor creates an argument structure of the argument and the destination for the result of the application, and this argument structure is given to the procedure as input. If no instruction follows the apply actor in the program, the value designated by the destination selector in the argument structure passed to the procedure is nil. Upon completion of the execution of the procedure, the result is sent to the specified destination by a return actor within the procedure body.

The data-flow representation of the following simple procedure is shown in Figure 2.10:

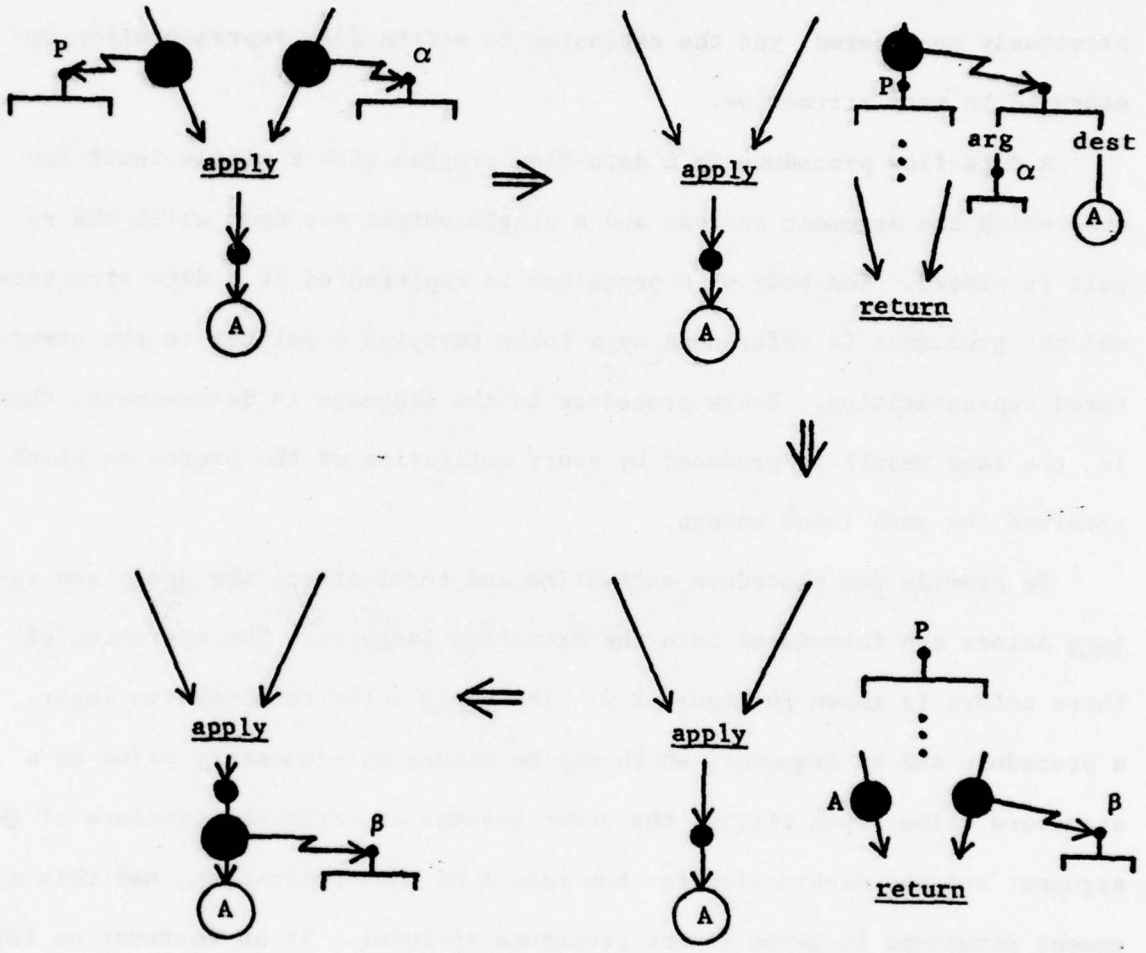


Figure 2.9. Operation of the apply and return actors.

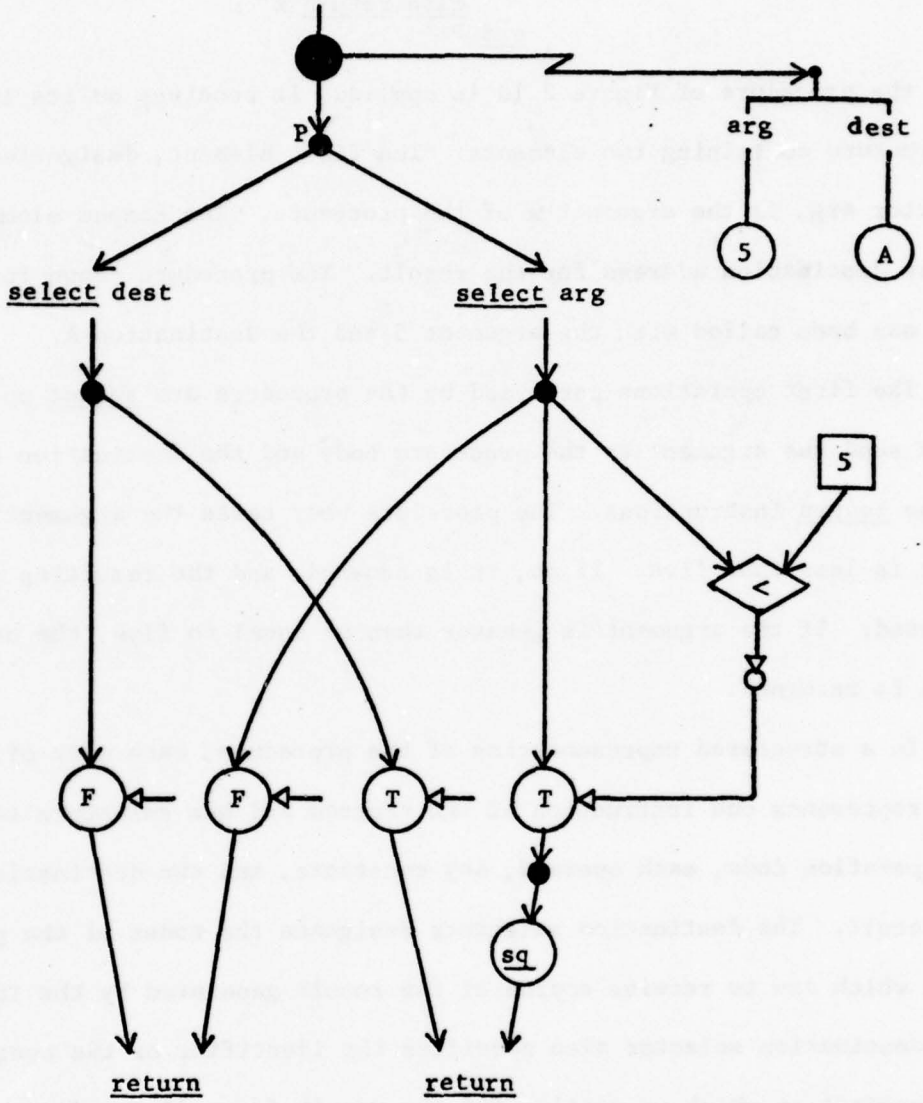


Figure 2.10. Data-flow representation of a simple procedure.

```
P: procedure (x)
   if x<5
      then return x^2
      else return x
   end P
```

When the procedure of Figure 2.10 is applied, it receives on its input arc a structure containing two elements. The first element, designated by the selector arg, is the argument x of the procedure. The second element, dest, is the destination address for the result. The procedure shown in Figure 2.10 has been called with the argument 5 and the destination A.

The first operations performed by the procedure are select operations which send the argument to the procedure body and the destination address to the return instructions. The procedure body tests the argument to see if it is less than five. If so, it is squared, and the resulting value is returned. If the argument is greater than or equal to five, the original value is returned.

In a structured representation of the procedure, each node of the structure represents one instruction of the program and has selectors to specify the operation code, each operand, any constants, and the destinations for the result. The destination selectors designate the nodes of the program graph which are to receive copies of the result generated by the instruction. Each destination selector also specifies the identifier of the operand in the destination which is waiting for the result (i.e. d1-l indicates that this is the first destination specified in the instruction, and the result is to go to the first operand of the designated destination instruction). A destination instruction may be shared if it receives more than one operand. A structured representation of the data-flow procedure of Figure 2.10 is

given in Figure 2.11.

In the structured representation of the procedure, the gate actors are not represented by separate instructions, but are incorporated into the instructions representing the return and square actors as part of the operand specification. The value within a gated operand designated by the selector gate specifies the type of gate represented, and the selectors control and data designate the control value and data value received. This method of representation allows a more efficient execution of the instruction within the data-flow processor and is discussed further in Section 3.1.1.

Initially, all operands of the program structure of Figure 2.11 have value nil to indicate that they are empty. An instruction is enabled for execution when no operands of the instruction contain the value nil, and each control value received matches the associated gate value.

Upon being enabled, an instruction is ready to be processed. Some arbitrary time later, the specified operation is performed on the operands of the instruction, and the result is sent to all instructions indicated by the destination address selectors of the instruction. At each destination, the result is appended to the correct nil-valued operand (designated by the operand number in the destination selector), and the instruction containing that operand is enabled if all operands are present and the correct control values have been received. If the destination operand is gated, the type of value received determines whether it is to be appended to the node designated by the selector control or the one designated by the selector data.

If the control values received by an instruction do not match their associated gate values, the instruction is not enabled, and it and its suc-

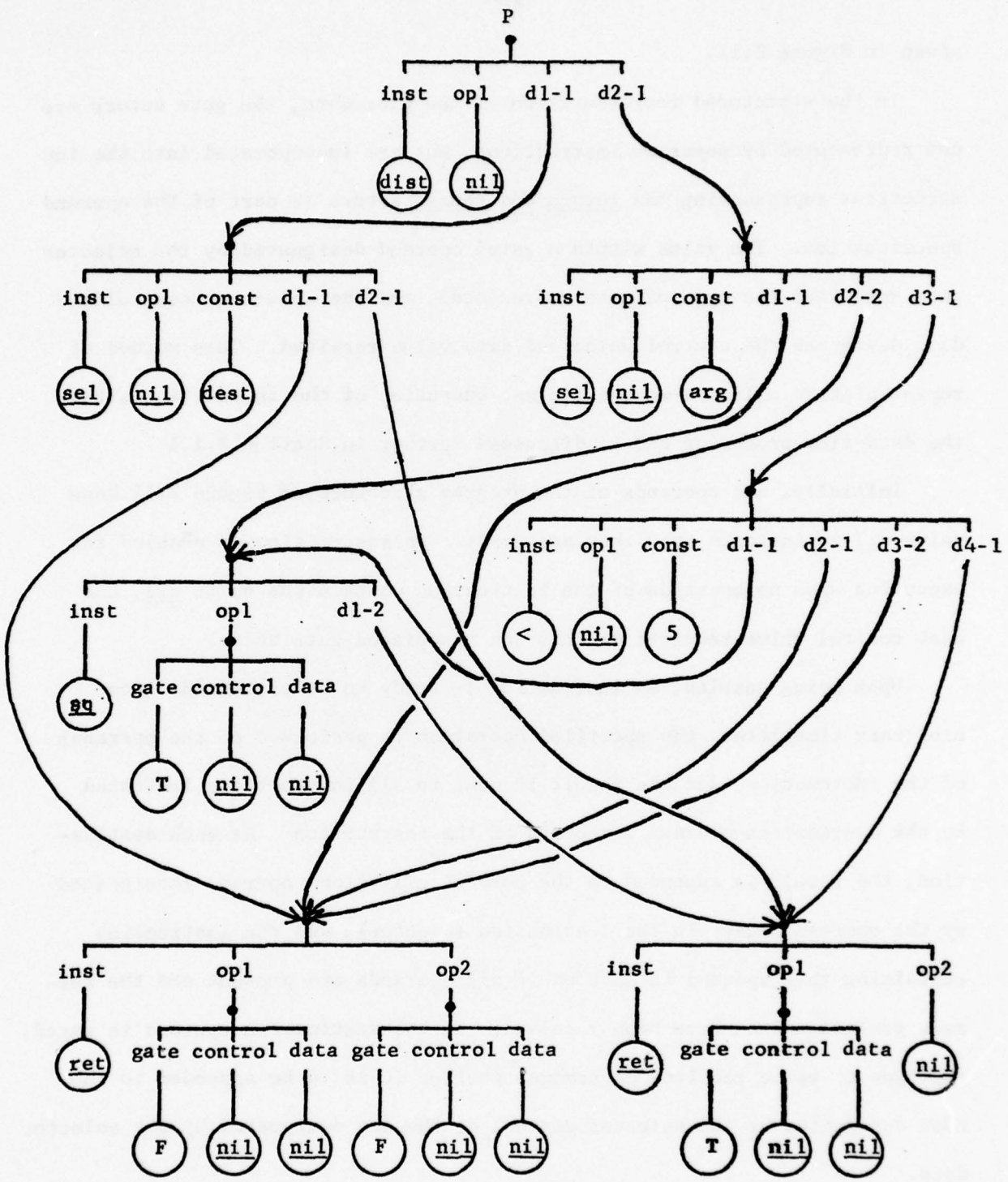


Figure 2.11. Structured representation of the program of Figure 2.10.

cessors are not executed in the activation.

Many simultaneous activations of a data-flow procedure may exist as a result of concurrent or recursive procedure application. In order to avoid the possibility of interaction between tokens from separate activations, a new copy of a procedure is created for each activation, the argument structure is transmitted to the new copy, and after a result is returned, the copy is discarded.

Chapter 3

ARCHITECTURE OF THE DATA-FLOW PROCESSOR

The data-flow processor described in this chapter is designed to directly execute programs expressed in the data-flow language presented in Chapter 2. The structure of the processor is presented in four stages. The first section of the chapter discusses the representation of instructions within the processor and the execution of individual instructions representing operators and deciders of a program. The next section extends the description to include the processing of structures. The third section presents the multi-level memory structure utilized by the processor in which the memories of the instruction and structure processing sections of the processor act as caches for the most active instructions and structure values. The final section describes the implementation of procedures within the architecture.

3.1 Instruction Processing

The instructions of a data-flow program are stored and executed in the instruction processing section of the processor (Figure 3.1). Instructions awaiting execution are contained in the Instruction Memory. Upon becoming ready for execution, an instruction enters the Arbitration Network and is conveyed by the Arbitration Network to the correct Operation or Decision Unit. The results of an operation are distributed to the desired destination instructions by a Distribution Network. Similarly, the results of a decision are distributed by a Control Network

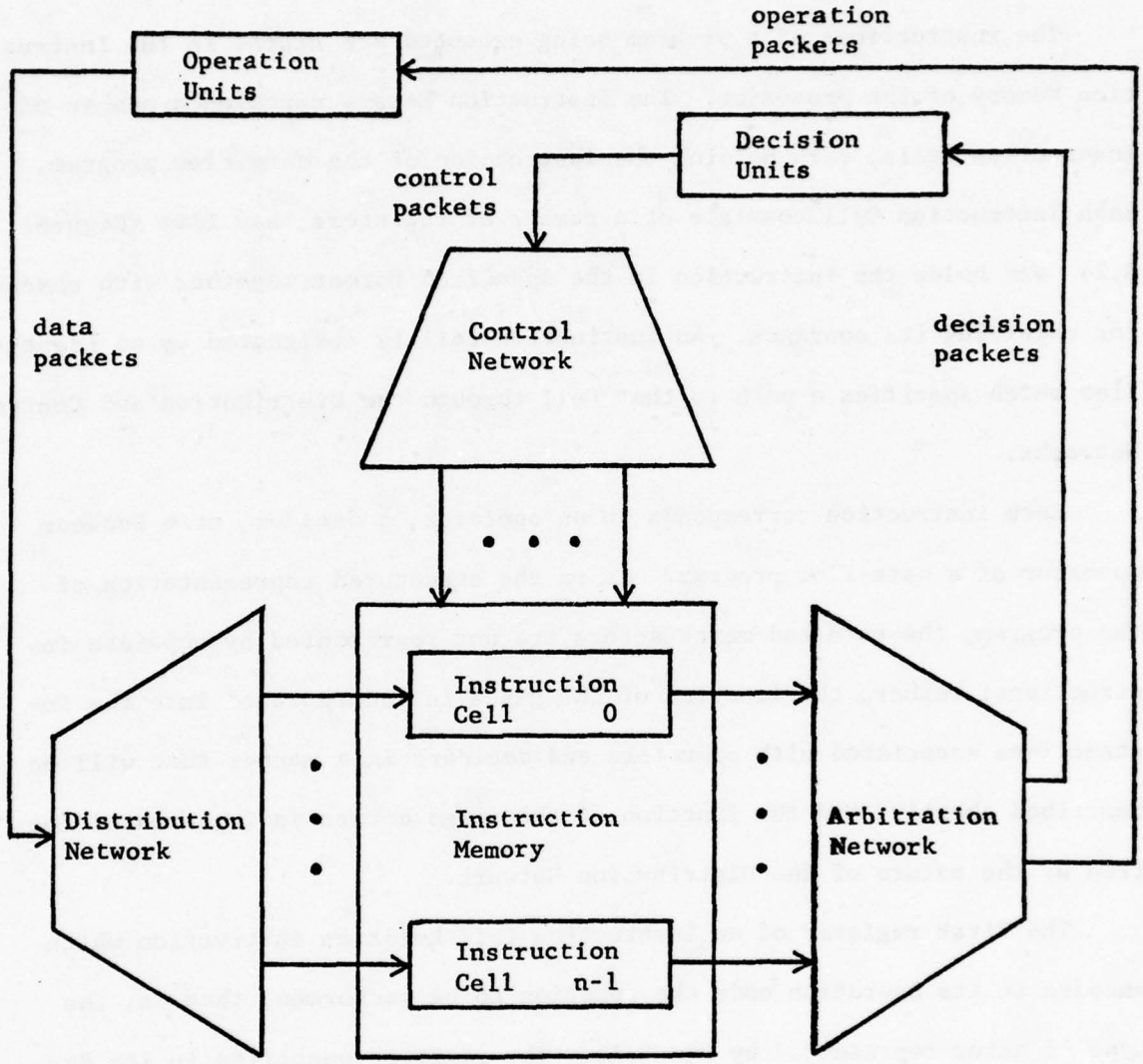


Figure 3.1. Organization of the instruction processing section of the data-flow processor.

3.1.1 Instruction Representation

The instructions of a program being executed are stored in the Instruction Memory of the processor. The Instruction Memory contains a number of Instruction Cells, each holding one instruction of the data-flow program. Each Instruction Cell consists of a number of registers, say five (Figure 3.2) and holds the instruction in the specified format together with spaces for receiving its operands. An Instruction Cell is designated by an identifier which specifies a path to that Cell through the Distribution and Control Networks.

Each instruction corresponds to an operator, a decider, or a Boolean operator of a data-flow program. As in the structured representation of the program, the gate and merge actors 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.

The first register of an Instruction Cell holds an instruction which encodes in its operation code the function to be performed, that is, the type of actor represented by the Cell. The register specifies in its destination field the Cell identifier of an instruction which is to receive one copy of the result.

Each other register of the Cell can hold either a data operand, a Boolean operand and one destination, or two destinations. A register can also be empty, indicating that it is not used by the instruction currently occupying the Cell. The use of the register is indicated by a use code in the first field of the register. If four data operands are used in an instruc-

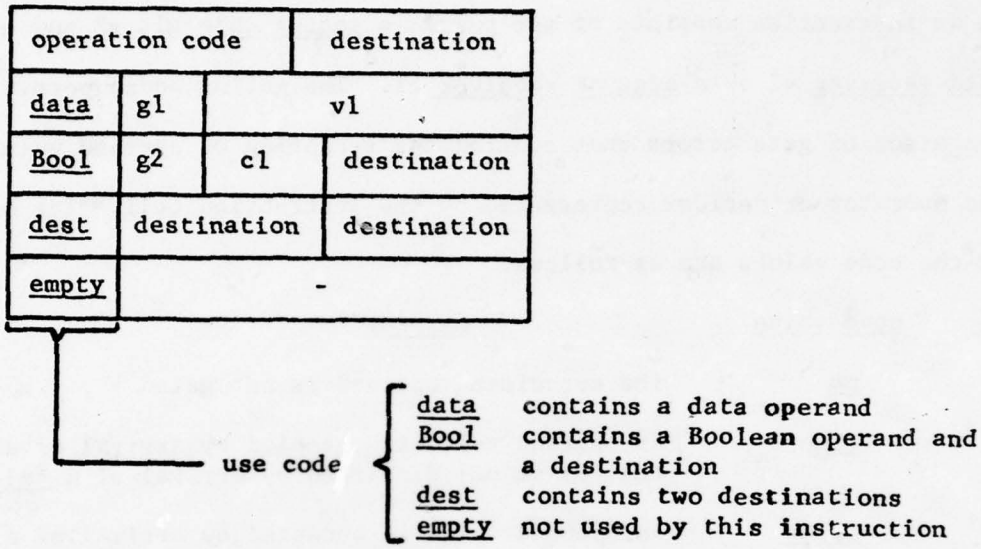


Figure 3.2. Format of fields in an Instruction Cell.

tion, only one destination can be specified, and that destination must be a distribution instruction (Figure 3.3) if more than one destination is desired for the result.

A register containing the components designated by an operand selector in an instruction consists of two parts, a gating code g1, g2 and either a data receiver v1 or a control receiver c1. 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 control value; discarded by arrival of a false value
false	an operand value is accepted by arrival of a false control value; discarded by arrival of a true value
const	the operand is a constant value

The structure of a data or control receiver (Figure 3.4) provides space to receive a data or Boolean value, and two flag fields in which the arrival of data and control values is recorded. The gate flag is changed from off to true or false by a true or false control value. The value flag is changed from off to on by a data or Boolean value according to the type of receiver.

An initial configuration of Instruction Cells corresponding to the data-flow program of Figure 2.11 is given in Figure 3.5.

3.1.2 Operation of an Instruction Cell

The function of each Instruction Cell is to receive data and control values, and, when the Cell becomes enabled, to transmit the contents of the Cell

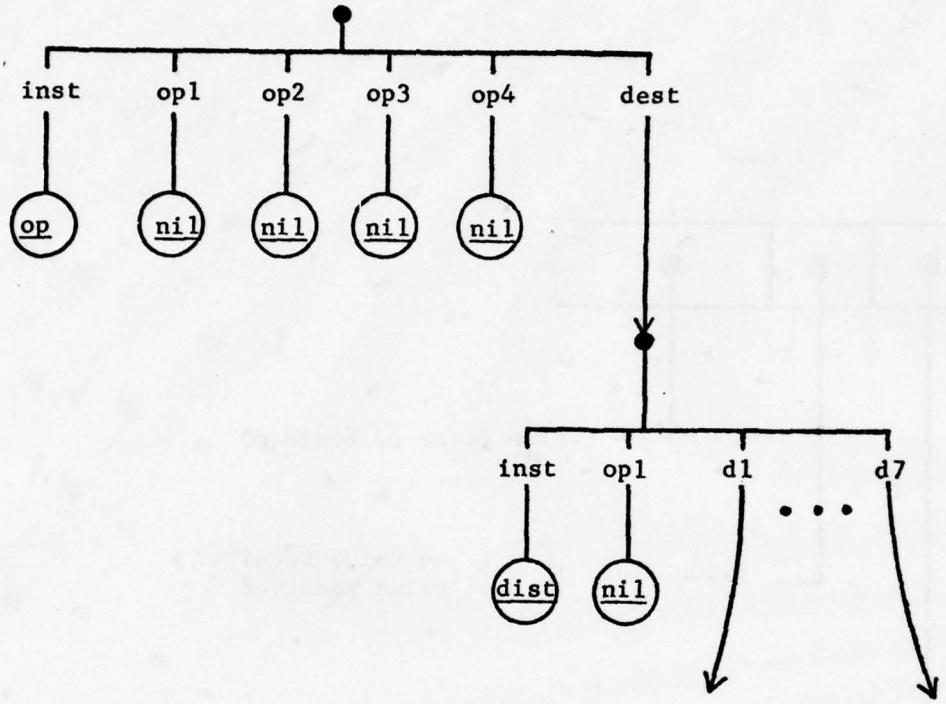


Figure 3.3. Use of the distribution instruction.

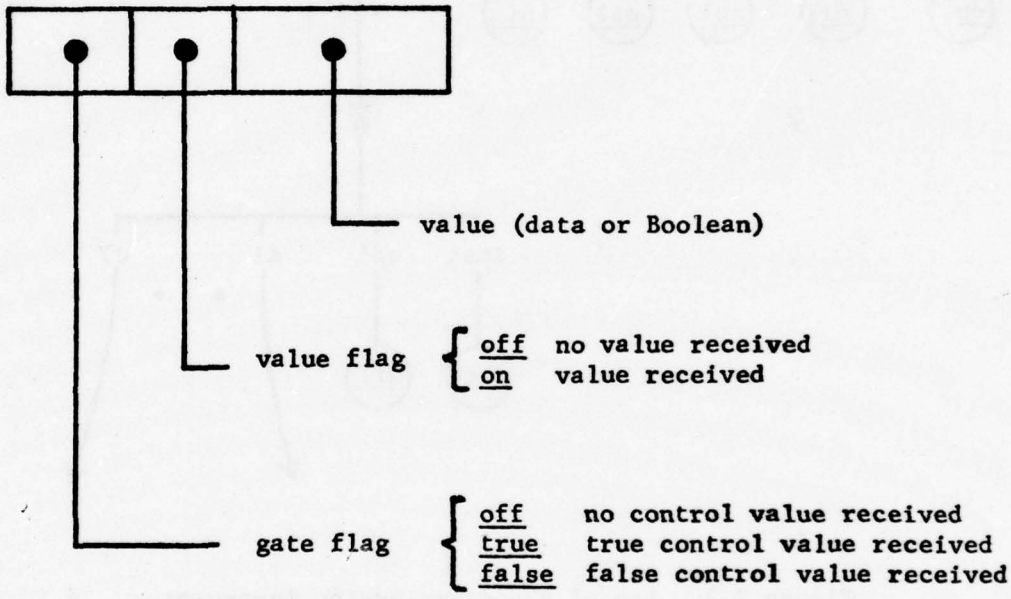


Figure 3.4. Structure of a receiver.

Cell 0

0	dist		1/2
1	dest	2/2	-
2	data	no	( )
3	empty	-	
4	empty	-	

Cell 1

0	select		3/1
1	dest	6/1	-
2	data	no	( )
3	data	const	dest
4	empty	-	

Cell 2

0	select		3/2
1	dest	4/3	5/1
2	data	no	( )
3	data	const	arg
4	empty	-	

Cell 3

0	return		-
1	data	false	( )
2	data	false	( )
3	empty	-	
4	empty	-	

Cell 4

0	less		3/1
1	dest	3/2	6/1
2	dest	6/2	-
3	data	no	( )
4	data	const	5

Cell 5

0	square		6/2
1	data	true	( )
2	empty	-	
3	empty	-	
4	empty	-	

Cell 6

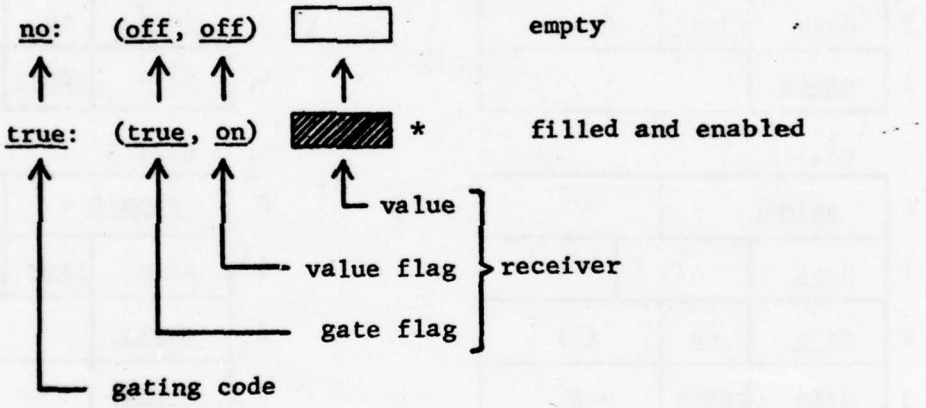
0	return		-
1	data	true	( )
2	data	no	( )
3	empty	-	
4	empty	-	

1/2 ⇔ register 2 of Cell 1

Figure 3.5. Initial configuration of Instruction Cells for the data-flow program of Figure 2.11.

to a Functional Unit (either an Operation Unit or a Decision Unit, determined by the actor type). An Instruction Cell becomes enabled just when all its registers are enabled. A register specified to act an an instruction register is always enabled. Registers specified to act as operand registers change state with the arrival of values directed to them. The state transitions and enabling rules for data operand registers are defined in Figure 3.6.

In Figure 3.6 the contents of an operand register are represented as follows:



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

d	data value
t	true control value
f	false control value

Note that upon arrival of a data value and a control value that does not match the gating code of the register, the register enters a trap state which indicates that the instruction contained in the Cell is not to be executed. In a correct program, if one register of a given Cell enters

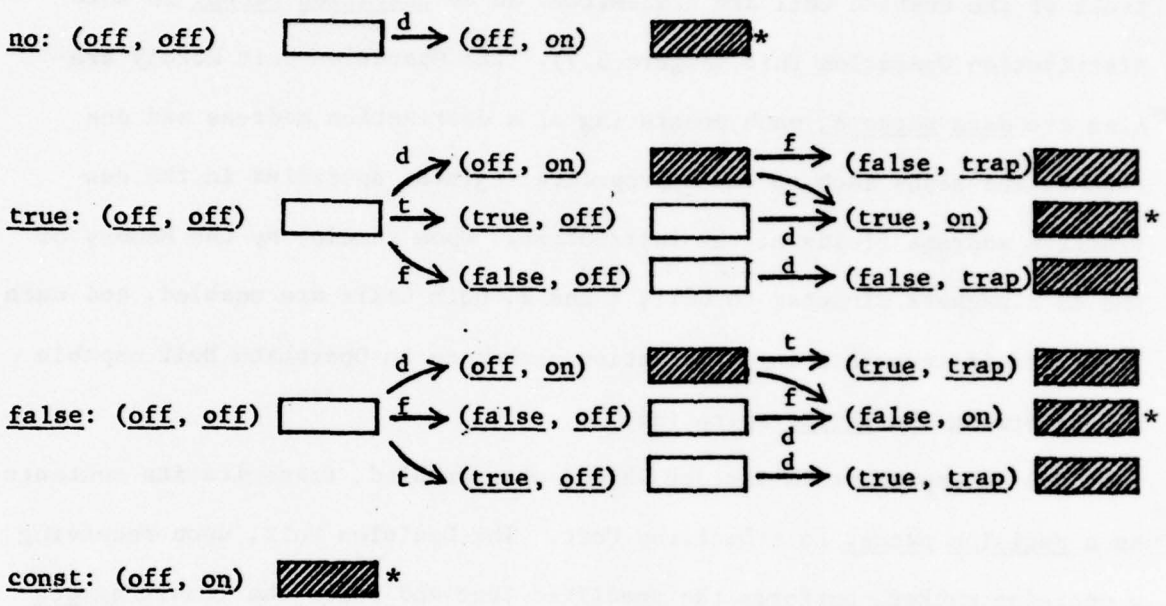


Figure 3.6. State transition and enabling rules for data operand registers.

the trap state, all will enter the trap state.

To illustrate the operation of an Instruction Cell, let us examine the data-flow program of Figure 3.5. Cell 0 of the program is enabled upon receiving the identifier of a structure α in register 2. The entire contents of the enabled Cell are transmitted as an operation packet to some distribution Operation Unit (Figure 3.7). The Operation Unit merely creates two data packets, each consisting of a destination address and one result, and sends each to the appropriate register specified in the destination address fields of the instruction. Upon receipt by the Memory of the data packets directed to Cells 1 and 2, both Cells are enabled, and each transmits its contents as an operation packet to an Operation Unit capable of performing the select operation.

Cell 4 represents a decider which, when enabled, transmits its contents as a decision packet to a Decision Unit. The Decision Unit, upon receiving a decision packet, performs the specified test and sends the resulting control packets to the designated Cells.

3.1.3 Network Structure

To connect the Instruction Cells of the Memory to the Operation and Decision Units, a network, called the Arbitration Network, provides a path from each Instruction Cell to each Operation or Decision Unit. Operation and decision packets are transmitted from Instruction Cells into the Arbitration Network. The network is capable of accepting many packets simultaneously and delivers each packet to the correct Functional Unit.

Upon receiving an operation packet, an Operation Unit performs the func-

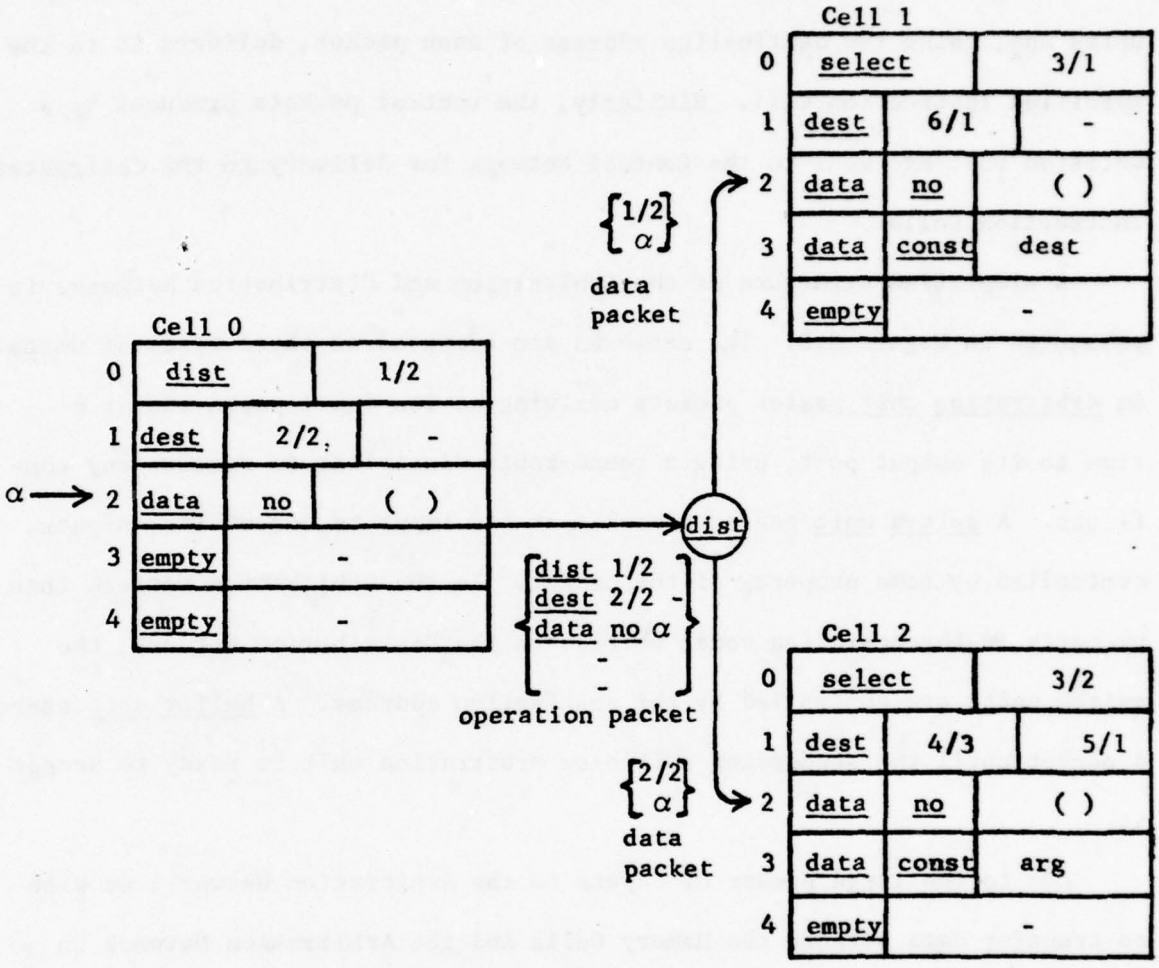
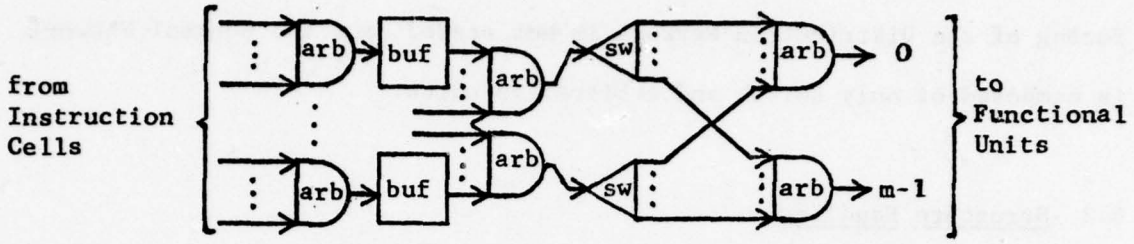


Figure 3.7. Operation of an Instruction Cell.

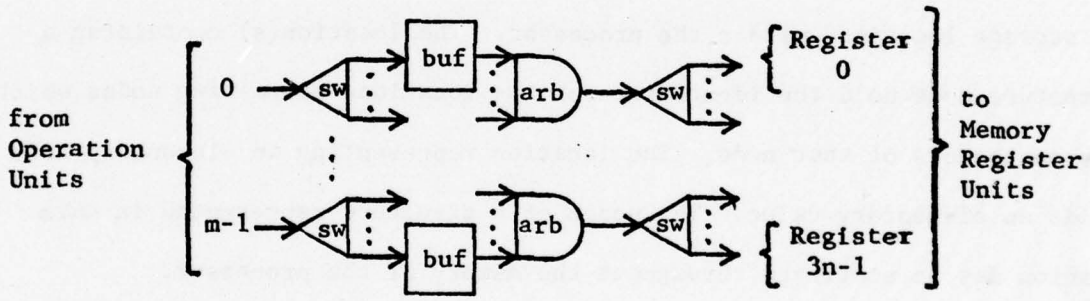
tion specified by the operation code on the operands of the packet and produces a data packet for each destination specified in the instruction. A Distribution Network concurrently accepts data packets from the Operation Units and, using the destination address of each packet, delivers it to the specified Instruction Cell. Similarly, the control packets produced by a Decision Unit are sent to the Control Network for delivery to the designated Instruction Cells.

A simplified structure of the Arbitration and Distribution Networks is presented in Figure 3.8. The networks are composed of three types of units. An 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 conflicts. A switch unit passes a packet at its input to one of its outputs, controlled by some property of the packet. In the Arbitration Network this property is the operation code, whereas in the Distribution Network, the switch units are controlled by the destination address. A buffer unit stores a packet until the succeeding switch or arbitration unit is ready to accept it.

Due to the large number of inputs to the Arbitration Network, we wish to transfer data between the Memory Cells and the Arbitration Network in serial format to reduce the number of wires necessary. However, in order to maintain a high rate of packet flow at the output ports, we wish to transfer packets to the Functional Units in parallel format. For this reason, serial-to-parallel conversion is done gradually within the buffer units as a packet travels through the Arbitration Network. Parallel-to-serial conversion is performed in the Distribution Network for similar reasons.



(a) Arbitration Network



(b) Distribution Network

Figure 3.8. Structure of the Arbitration and Distribution Networks.

The structure of the Control Network is similar to that of the Distribution Network. However, all packets within the Control Network contain simple Boolean values, and hence the parallel-to-serial conversion and buffering of the Distribution Network is not needed, and the Control Network is composed of only switch and arbitration units.

3.2 Structure Handling

The physical representation of a structure within a computer system may be viewed in several different ways. One extreme involves implementing the structure as it is represented in the data-flow model, that is, as an acyclic directed graph in which each node is either a structure node or an elementary node. In such an implementation, each node of the graph occupies a number of storage locations within the processor. The location(s) containing a structure node hold the identifiers of the locations containing nodes which are successors of that node. The location representing an elementary node holds an elementary value. The nodes of a structure represented in this fashion may be scattered throughout the memory of the processor.

Alternatively, all elementary values of a structure may be stored together in a group of locations. The first few locations of the group then contain a mapping function which allows one to find the location of a specific element within the group. This method is often used for the representation of arrays within a conventional computer system.

The first approach has the problem that the storage of a structure in such a manner can occupy a great deal of space within the memory. Not only must the data be stored, but a large number of structure nodes and associated pointers must also be located within the memory. Accessing an elementary

value in a graph can take a long time as a path is followed over the arcs of the graph to the desired node. On the other hand, a single structure represented by the second approach occupies much less room, but the representation of several structures in such a manner can be very expensive in terms of space since components of a structure cannot be shared as they can in the graph approach.

It would seem that perhaps a combination of these two methods could be efficiently utilized; that is, a structure representation in which each node of the structure is a small block of data. In the remainder of this section we present an approach in which data structures, described as acyclic directed graphs, are implemented in such a hybrid manner.

3.2.1 Simple Structures

The storage of structures and the execution of the structure actors described in Section 2.2 occurs in a separate structure processing section within the data-flow processor. The structure processing section consists of a Structure Operation Unit and a Structure Memory and attendant Arbitration and Distribution Networks. This section of the processor is viewed as an Operation Unit by the Instruction Memory; that is, packets specifying structure operations are sent to the section, and data packets are returned. The organization of the data-flow processor with the addition of the structure processing capability is shown in Figure 3.9.

Packets specifying structure operations are received by the Structure Memory and the Structure Operation Unit. Instructions which require the creation of new structure nodes are processed by the Structure Operation

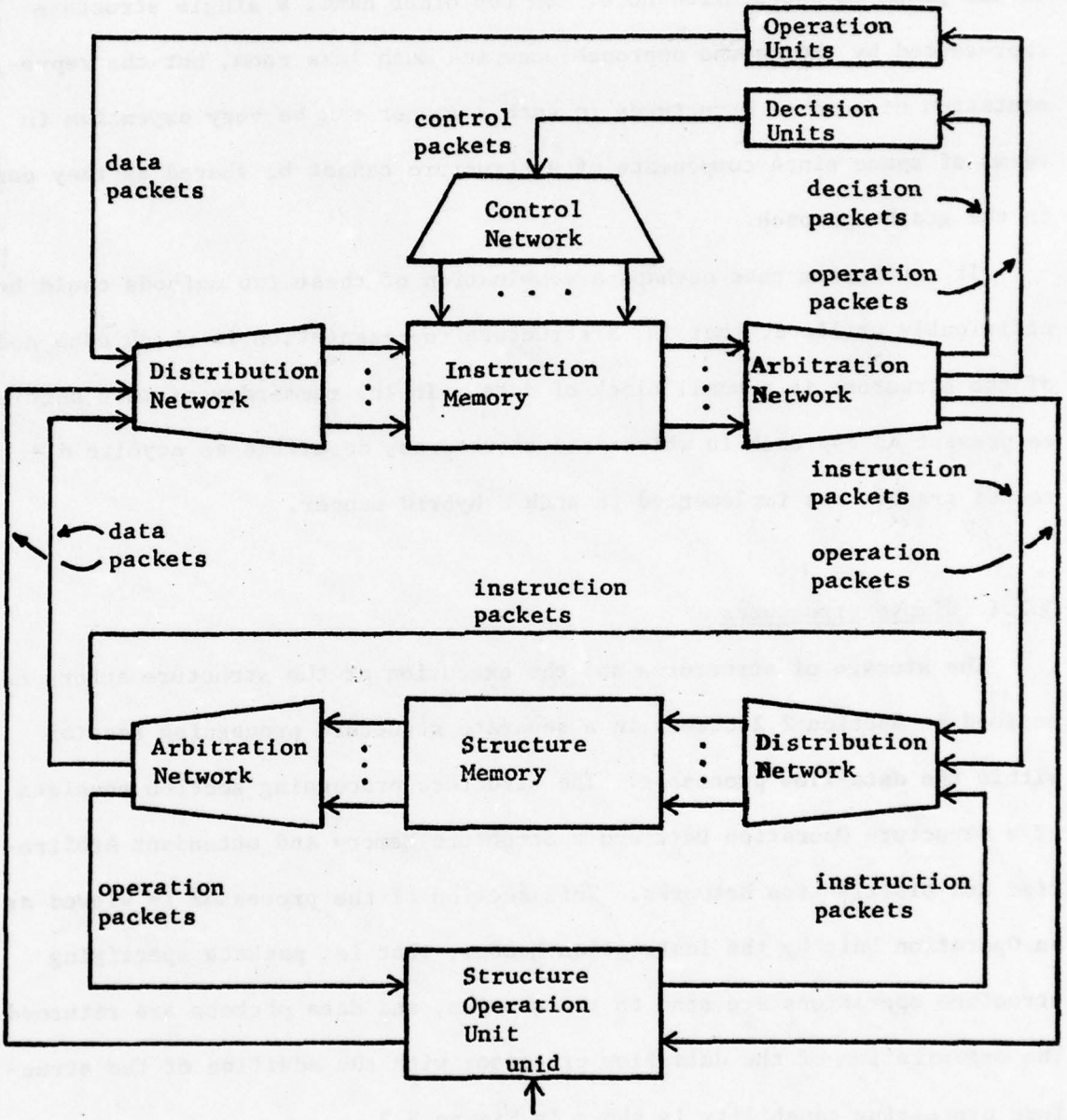


Figure 3.9. Organization of the data-flow processor without multi-level memory.

Unit. The unit controls the performance of the instruction specified in each operation packet through instruction packets sent to the Structure Memory and sends as data packets the identifiers of the resulting structures to the instruction processing section. All structure operations other than the allocation of a new node are performed within the Structure Memory.

To illustrate the operation of the structure processing section of the processor, in this section we shall limit our consideration to structures represented as binary trees. A selector of such a structure can have one of two values, L (left) and R (right). Such structures are well-known from their use in Lisp [21].

A node of a structure is contained in a two register Cell known as a Structure Cell and designated by a Cell identifier. The two registers of the Cell contain the left and right components of the structure, respectively; and hence no selector need be stored in a register. The first field of a register is a use code which indicates whether the item stored in the second field is the identifier of another Cell or an elementary value or the register is empty. A memory representation of the simple structure of Figure 2.4 is presented in Figure 3.10.

The Structure Memory is composed of a number of Structure Cells in a manner similar to the way the Instruction Memory is formed of a number of Instruction Cells. Each Structure Cell is capable of holding one node of a structure, and the identifier of the Cell specifies a path through the Distribution Network to the Cell. The Structure Memory receives instruction packets from the Instruction Memory and the Structure Operation Unit commanding a specific Structure Cell to execute some structure operation upon the node located in the Cell.

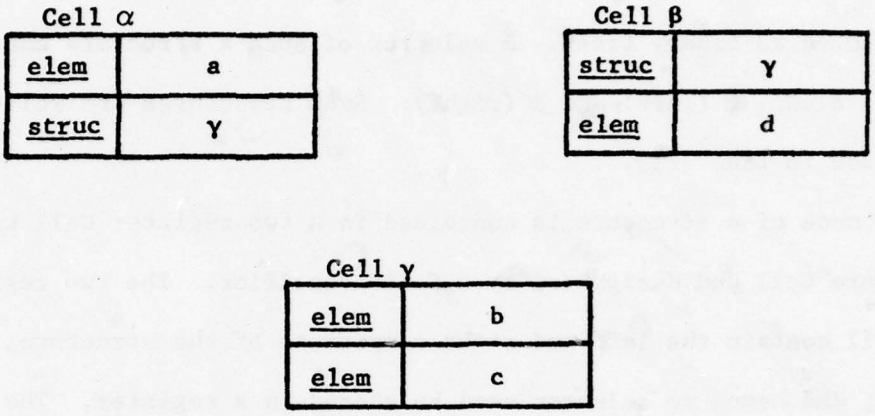


Figure 3.10. Memory representation of the structure of Figure 2.4.

Each Structure Cell within the Structure Memory is capable of performing one of two operations upon the structure node contained in the Cell. The possible operations are:

1. select. Upon receipt of an instruction packet specifying a select operation

$$\left\{\begin{matrix}\underline{\text{select}} & \text{dest} \\ \text{s} & \end{matrix}\right\}$$

a Structure Cell follows one of two procedures, controlled by whether s is a simple or compound selector.

a. If s is a simple selector, the content c of the register designated by s is used to form a data packet

$$\left\{\begin{matrix}\text{dest} \\ \text{c}\end{matrix}\right\}$$

which is presented to the Arbitration Network for transmission to the instruction processing section of the processor.

b. If s is a compound selector $s_1 s_2 \cdots s_n$, the content β of the register designated by s_1 is the identifier of some other Structure Cell and is used to form the instruction packet

$$\left\{\begin{matrix} & \beta & \\ \underline{\text{select}} & & \text{dest} \\ & s_2 \cdots s_n & \end{matrix}\right\}$$

which is presented to the Arbitration Network for transmission to the input Distribution Network of the Structure Memory. The process is then repeated with the selector s_2 at Structure Cell β.

2. alter. The receipt of an alter instruction

$$\left\{\begin{matrix}\underline{\text{alter}}\\ s\\ x\\ \beta\end{matrix}\right\}$$

indicates that the Structure Cell is to contain a copy of the node β with the component of β designated by the selector s set to x. First, a copy of node β is retrieved from the auxiliary memory through use of a retrieve command packet in the manner to be described in Section 3.3.3. Once the copy of β is present in the Cell, the value contained in the register designated by the selector s is changed to x, and the use code of the register is set to the appropriate value (elem, struc, or empty), designated by the tag of x. If β is $\emptyset$, no node is requested from the auxiliary memory; the current contents of the Structure Cell are altered in the manner described.

The format of an instruction packet received at the input Distribution Network of the Structure Memory differs from the format of an operation packet transmitted to a Functional Unit or the Structure Operation Unit due to the fact that the operation code of an instruction packet does not control the switching within the Distribution Network; rather, the Cell identifier is used to direct an instruction packet toward the correct Structure Cell. Hence, an instruction packet in the Distribution Network has the following format

$$\left\{\begin{matrix}\alpha\\ i\end{matrix}\right\}$$

where α is the identifier of some Structure Cell in the Structure Memory and i specifies one of the two operations which can be performed by a Structure Cell and contains the necessary operands.

Packets containing instructions that designate structure operations are transmitted to the structure processing section of the processor from the Instruction Memory. A packet specifying a select instruction is transmitted directly to the Structure Memory as an instruction packet. Structure operation packets representing the other structure instructions are transmitted to the Structure Operation Unit. The necessity of processing each operation packet within the Structure Operation Unit is due to the required allocation of one or more free Structure Cells for the execution of each instruction with the exception of the select instruction. The Structure Operation Unit performs the allocation of a free Cell simply by accepting the identifier of a Cell over the unid port. The manner in which these identifiers are provided and Structure Cells are freed for use by new structures is described in Section 3.3.2.

Now that we have considered the operation of a Structure Cell within the Structure Memory, we can describe the execution of each of the remaining structure actors of Section 2.2 merely by listing the procedure followed by the Structure Operation Unit in processing the instruction. For the purposes of this discussion, it is assumed that all selectors are simple selectors.

A construct instruction

$$\left\{\begin{array}{ll} \underline{\text{construct}} & \text{dest} \\ \text{s1:} & \alpha \\ \text{s2:} & \gamma \end{array}\right\}$$

specifies that a new node is to be created with components α and γ, designated by the selectors s1 and s2. The instruction is implemented by the Structure Operation Unit as a number of alter operations in the following manner:

1. Accept an identifier β from the unid port.

2. Transmit to the Structure Memory the instruction packets

$$\left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\\ \text{s1}\\ \alpha\\ \emptyset\end{array}\right\} \quad \text{and} \quad \left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\\ \text{s2}\\ \gamma\\ \emptyset\end{array}\right\}$$

transferring the values α and γ to the correct registers of β.

3. Transmit to the instruction processing section the data packet:

$$\left\{\begin{array}{c}\text{dest}\\ \beta\end{array}\right\}$$

An operation packet containing an append instruction is of the following format

$$\left\{\begin{array}{cc}\underline{\text{append}} & \text{dest}\\ \text{s} & \\ \text{x} & \\ \alpha & \end{array}\right\}$$

where s is the selector of the element in Structure Cell α which is to be replaced by x in the new structure. The procedure followed by the Structure Operation Unit in execution of the instruction is as follows:

1. Accept an identifier β from the unid port.

2. Transmit the instruction packet

$$\left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\\ \text{s}\\ \text{x}\\ \alpha\end{array}\right\}$$

to the Structure Memory to copy node α into Cell β and change the component of β designated by the selector s to x.

3. Transmit to the instruction processing section the data packet:

$$\left\{\begin{array}{c}\text{dest}\\ \beta\end{array}\right\}$$

An operation packet specifying a delete instruction

$$\left\{\begin{array}{c}\underline{\text{delete}}\ \ \text{dest}\\ \text{s}\\ \alpha\end{array}\right\}$$

is processed in a similar manner:

1. Accept an identifier β from the unid port.
2. Transmit the instruction packet

$$\left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\\ \text{s}\\ \emptyset\\ \alpha\end{array}\right\}$$

to the Structure Memory, indicating that the use code of the register designated by s in Cell β is to be set to empty.

3. Transmit the data packet

$$\left\{\begin{array}{c}\text{dest}\\ \beta\end{array}\right\}$$

to the instruction processing section.

3.2.2 Extension to More Complex Structures

The extension of the described techniques for the implementation of data structures to larger and more complex structures is straightforward. In order to implement structures with a fixed maximum number of arcs emanating from each node, the size of a Structure Cell is increased to accommodate the new node size. The use of arbitrary (to a fixed maximum size) integers or character strings as selectors can be accommodated through the addition of a selector field to each register. A Structure Cell must then have the capability to choose from the node contained in the Cell an item whose selector matches a specified selector. These extensions allow the

representation of fairly powerful structures, including the data-flow procedures presented in Section 3.4. A further extension to allow a node to have arbitrary number of emanating arcs introduces a great deal of complexity since it might be necessary to use several Cells to hold the identifiers of all Cells which contain successors of the node. To avoid this complexity, a node of a structure in the data-flow processor is of fixed size, and each arc emanating from the node has a fixed size selector associated with it.

3.3 Multi-Level Memory Structure

Each node of a data-flow procedure will fire at most once during execution of the procedure within the data-flow processor. A large number of nodes of the procedure will not fire at all if any conditionals are present in the program. Thus, it would be wasteful to assign an Instruction Cell to each instruction of a procedure when the procedure is activated. For this reason the instruction processing section of the data-flow processor incorporates a multi-level memory system such that only the active instructions of a program occupy the Instruction Cells of the processor. Similarly, in order to assure maximum use of the Structure Cells of the processor, the structure processing section utilizes a multi-level memory to insure that only active structure nodes occupy the Structure Cells. Individual instructions and structure nodes are retrieved from their respective auxiliary memories as they become required for computation. Instructions are returned to the auxiliary memory only when the Instruction Cells holding them are required for more active parts of the program. Structure nodes are sent to the auxiliary memory upon creation through execution of an append, delete,

or construct instruction.

The structure of the instruction processing section of the data-flow processor with the addition of a multi-level memory system is shown in Figure 3.11. Data-flow programs are located within the Packet Memory System, and each instruction of a program is sent to the Instruction Memory as a retrieve packet when it is needed for program execution, designated by the arrival of an operand of the instruction at some Instruction Cell. Instructions which occupy Instruction Cells needed for more active instructions are returned to the Packet Memory System as store packets. Each instruction so discarded is later retrieved from the Packet Memory System upon arrival at the Instruction Memory of another of its operands. The Memory Command Network provides the correct sequencing of store and retrieve commands transmitted to the Packet Memory System.

The multi-level memory system of the structure processing section of the processor (Figure 3.12) operates in a manner similar to that of the multi-level memory in the instruction processing section. A Structure Cell is transferred to the Structure Memory from the Packet Memory System when required for execution of an instruction. A newly created structure node is returned to the Packet Memory System after the execution of an append, delete, or construct instruction. The identifiers of free nodes are maintained within the Packet Memory System and are requested by the Structure Operation Unit at the getid port. Upon receiving such a request, the Packet Memory System returns the identifier of a free node to the Operation Unit at its unid port. The new identifier is used for the creation of a new structure node during execution of an instruction specified in a structure operation packet.

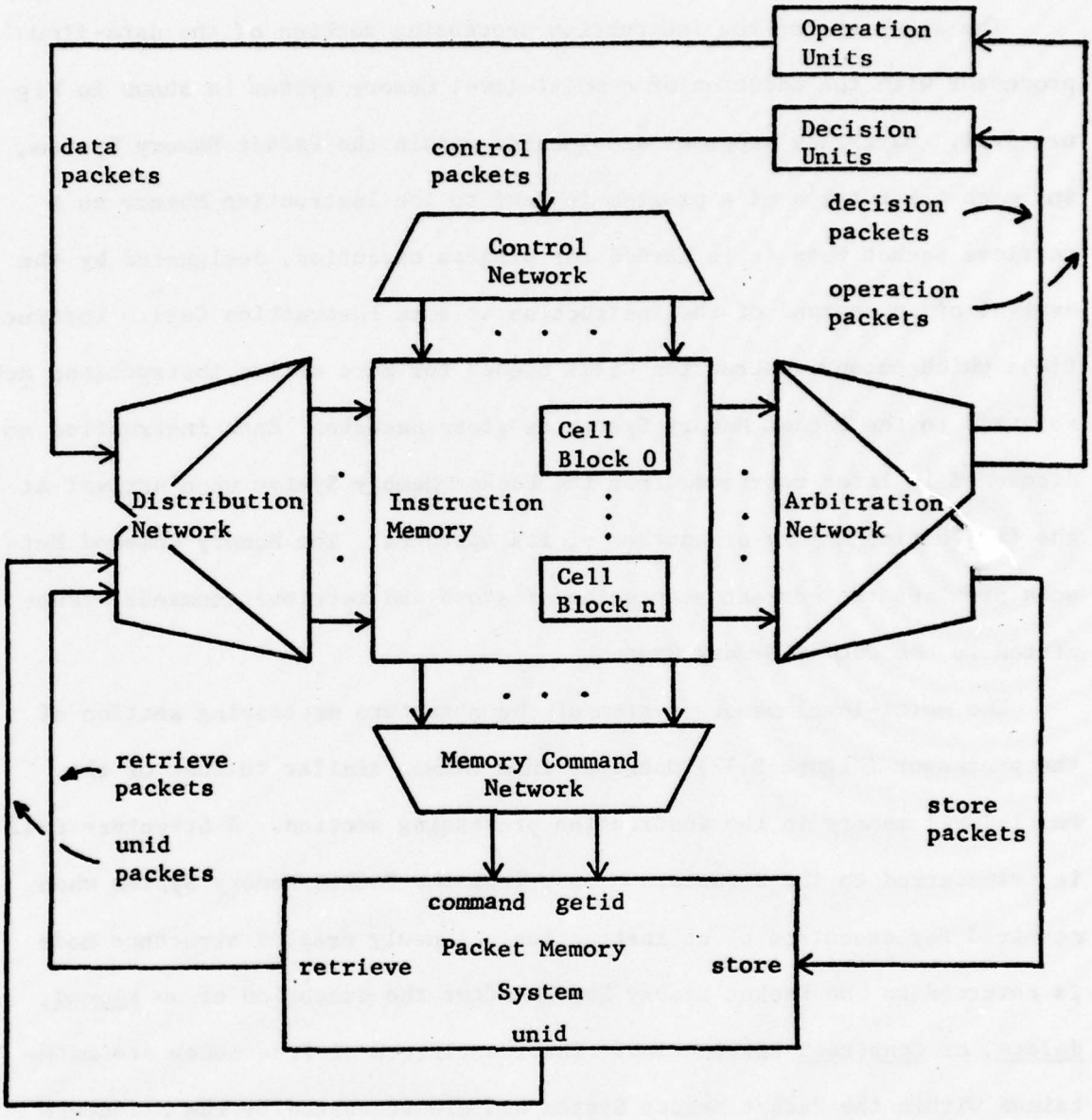


Figure 3.11. Organization of the instruction processing section without multi-level memory.

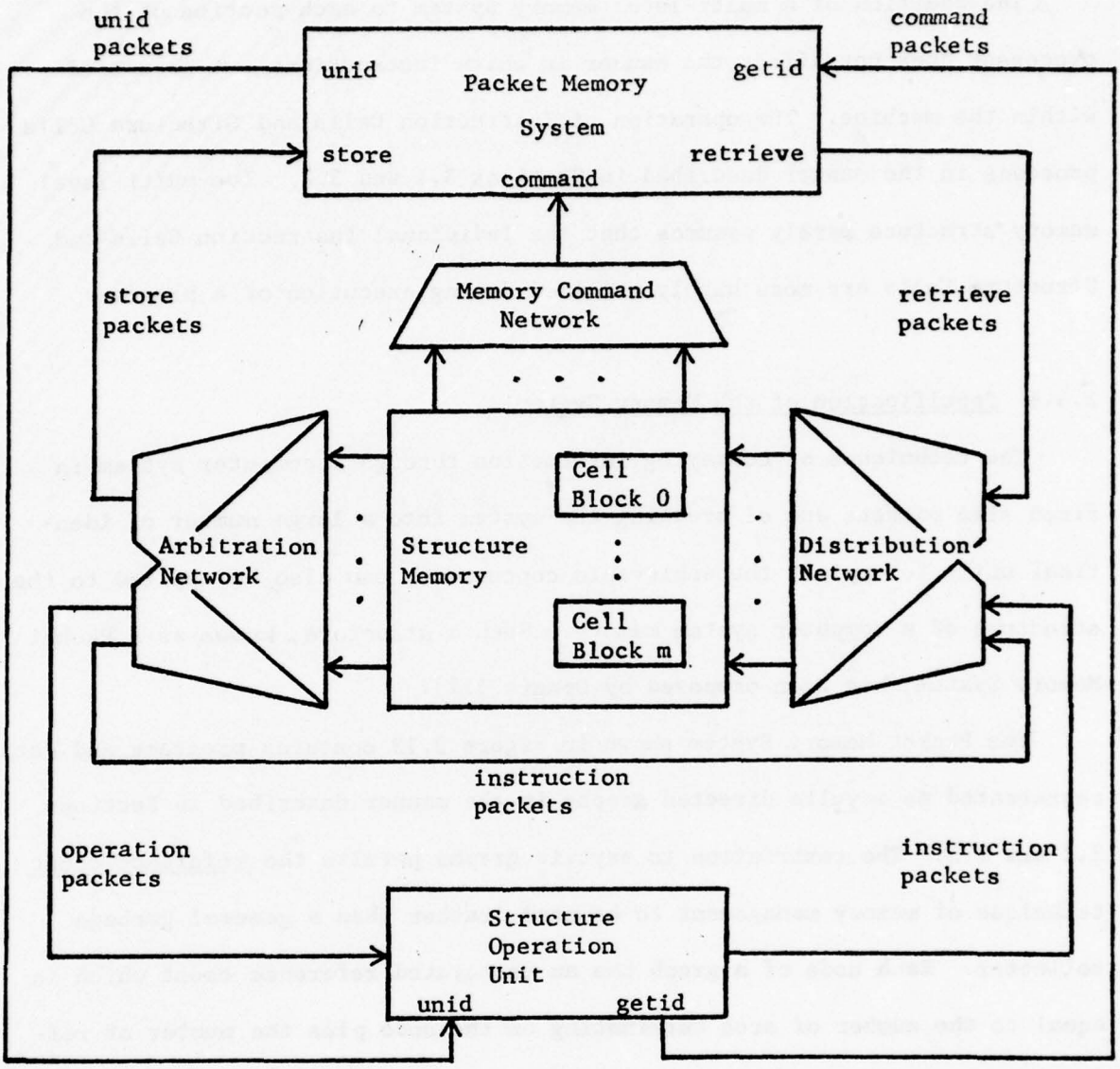


Figure 3.12. Organization of the structure processing section with multi-level memory.

The addition of a multi-level memory system to each section of the processor does not affect the manner in which instructions are processed within the machine. The operation of Instruction Cells and Structure Cells proceeds in the manner described in Sections 3.1 and 3.2. The multi-level memory structure merely assures that the individual Instruction Cells and Structure Cells are more highly utilized during execution of a program.

3.3.1 Specification of the Memory System

The techniques of conveying information through a computer system in fixed size packets and of breaking the system into a large number of identical units to exploit the achievable concurrency can also be applied to the structure of a computer system memory. Such a structure, known as a Packet Memory System, has been proposed by Dennis [11].

The Packet Memory System shown in Figure 3.13 contains programs and data represented as acyclic directed graphs in the manner described in Sections 2.2 and 2.3. The restriction to acyclic graphs permits the reference count technique of memory management to be used, rather than a general garbage collector. Each node of a graph has an associated reference count which is equal to the number of arcs terminating on the node plus the number of references to the node existing in the processor. When a node becomes inaccessible due to the execution of some instruction of the program, the reference count of the node will become 0.

The directed graphs are represented within the Packet Memory System through a collection of items I, each of which is designated by an element from a set of unique identifiers U. An item can be one of three types:

1. empty. The item contains the value nil. However, the item may or

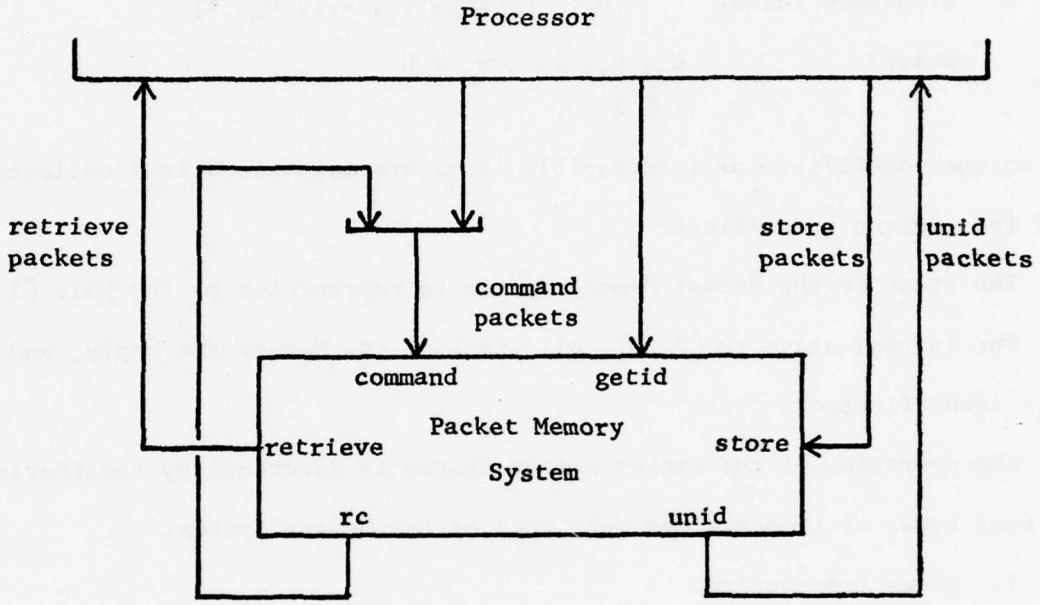


Figure 3.13. Structure of a Packet Memory System.

may not be accessible.

2. elementary value. u: {elem, e, r}

   where: $u \in U$ unique identifier of the item (unid)
   $e \in E$ elementary value
   $r \in I$ reference count

3. structure value. u_0: {struc, u_1, ..., u_n, r}

   where: $u_0, u_1, \ldots, u_n \in U$
   $r \in I$

The unique identifiers of inaccessible items are maintained in a collection T of free unique identifiers.

The state of the Packet Memory System is represented by the pair {I-T, T}. The initial state is {$\emptyset$, U}, all items of the Memory are empty, and their identifiers are free.

The operation of the Packet Memory System is described by considering the four types of transactions performed by the Memory System:

1. Store transaction.

   A store packet {i, elem, e, 1} or
   {i, struc, u_1, ..., u_n, 1}

   presented at the store port of the Packet Memory System requests storage of the item with unid i:

   {elem, e, 1} or
   {struc, u_1, ..., u_n, 1}

   However, the storage of the item is not effective until a store command packet

   {i, store}

   is received by the Packet Memory System at its command port, and any prior retrieval requests have taken affect.

2. Retrieval transaction.

An item with unid i is retrieved from the Packet Memory System by means of a retrieve command packet

$$\{i, \underline{retr}\} \text{ or}$$
$$\{i, \underline{retr}, j\}$$

presented at the command port of the Memory. A retrieve packet conveying the contents of the item identified by i is eventually delivered at the retrieve port of the Memory. In the first case, the unid i controls delivery of the retrieve packet, in the second, the identifier j specifies the destination for the packet.

3. Reference count transaction.

The incrementing or decrementing of the reference count of an item with unid i is accomplished by an

$$\{i, \underline{up}\} \text{ or}$$
$$\{i, \underline{dwn}\}$$

command packet delivered at the command port. If the new reference count of the item identified by i is 0:

a. If the item is a structure

$$i: \{\underline{struc}, u_1, \ldots, u_n, 0\}$$

the command packets

$$\{u_1, \underline{dwn}\}, \ldots, \{u_n, \underline{dwn}\}$$

appear at the reference count (rc) port.

b. Unid i is added to T.

4. Unique identifier generation.

A free unique identifier is requested by means of a command packet

{m, getid}

presented at the getid port, where m designates the unit requesting the identifier. A unid i is removed from T and appears in a unid packet

{m, i}

at the unid port of the Memory System. This transaction can only occur if $T \neq \emptyset$.

The Packet Memory System may be organized as a set of smaller Packet Memory System modules among which packets are distributed by a Distribution Network and formed into common streams by an Arbitration Network. Such an organization permits the system to handle large numbers of storage and retrieval requests concurrently and is discussed further in [11].

To insure the proper maintanence of items in the Packet Memory System, the Structure Memory and Instruction Memory must send the appropriate dwn and up command packets to the Memory System of the structure processing section as references to items are deleted and created during instruction execution. A reference to an item is deleted by the execution in the Structure Memory of any instruction representing a structure actor. A reference to an item is created in the Structure Memory by execution of a select instruction if the selected item is a structure value. A reference to an item can be created in the Instruction Memory through execution of a data distribution instruction containing a unid as its operand. We must require that upon the enabling of such an instruction, the Instruction Cell containing the instruction provide a command packet of the form {unid, up} for each destination in excess of one.

3.3.2 Organization of the Active Memories

The use of a multi-level memory system within each section of the data-flow processor requires that the Instruction Memory and Structure Memory act as caches for the most active instructions and structure nodes. For application of the cache principle to the architecture, the Instruction and Structure Cells of the processor are organized into groups of Cells, known as Cell Blocks.

A packet destined for the Instruction Memory or Structure Memory can no longer identify its destination by use of a Cell identifier. Each packet in the processor contains a node identifier which specifies a destination in the processor. A node identifier specifying a destination in either the Structure Memory or the Instruction Memory is the unique identifier of the node to which the packet is destined. The identifier is divided into two parts, a major address and a minor address, each containing a portion of the identifier. One Cell Block of each section of the processor is associated with each possible major address. However, the two memories may be of different size and hence use different sizes of major address.

All instructions having the same major address are processed by the Instruction Cells of the corresponding Instruction Cell Block. Thus, the Distribution and Control Networks use the major address to direct data packets, control packets, and retrieve packets to the appropriate Instruction Cell Blocks. Similarly, all structure nodes with the same major address are processed within the same Structure Cell Block, and the major address serves to direct retrieve and instruction packets to the correct Cell Block. The packet delivered to a Cell Block includes the minor address, which serves an an identifier for that packet within the Cell Block.

Store packets leaving a Cell Block have the form {m, x}, where m is either a complete identifier or a minor address, and x is the contents of a Cell. If m is a minor address, the major address of the Cell Block is appended to the packet as it travels through the Arbitration Network. In the same way, a minor address within a command packet leaving a Cell Block is augmented by the major address of the Cell Block as the packet travels through the Memory Command Network.

3.3.3 Operation of a Structure Cell Block

Let us consider the organization of the Structure Cell Block shown in Figure 3.14. Each Structure Cell of the Cell Block is able to hold any structure node whose major address is that of the Cell Block. Since many more structure nodes 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 Structure Cell; m is the minor address of the node 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 node
engaged	the Cell has been engaged for the node having minor address m by arrival of an instruction packet
occupied	the Cell is occupied by a node with minor address m

The Stack element of a Cell Block holds an ordering of the Structure Cells as candidates for displacement of their contents by more active nodes.

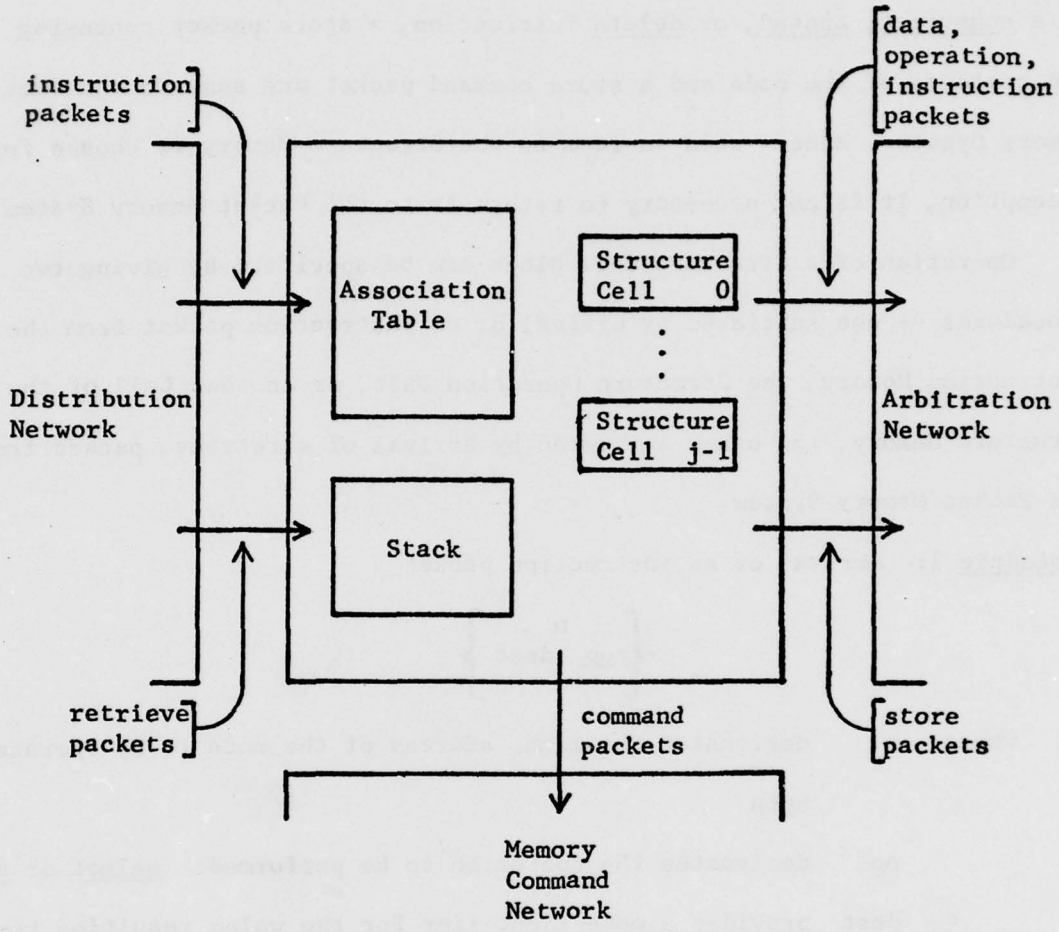


Figure 3.14. Organization of a Structure Cell Block.

Only Cells in occupied status are candidates for displacement.

Whenever a node in the Structure Memory is created through execution of a construct, append, or delete instruction, a store packet conveying the contents of the node and a store command packet are sent to the Packet Memory System. Hence, when an item in the Structure Memory is chosen for preemption, it is not necessary to return it to the Packet Memory System.

Operation of a Structure Cell Block can be specified by giving two procedures -- one initiated by arrival of an instruction packet from the Instruction Memory, the Structure Operation Unit, or another Cell of the Structure Memory, the other activated by arrival of a retrieve packet from the Packet Memory System.

Procedure 1: Arrival of an instruction packet

$$\left\{ \begin{matrix} n \\ \text{op} \quad \text{dest} \\ w \end{matrix} \right\}$$

where n designates the minor address of the node to be operated upon

op designates the operation to be performed: select or alter

dest provides a node identifier for the value resulting from a select operation

w consists of the necessary operands for the performance of the operation op

step 1. If the Association Table does not have an entry with minor address n, go to step 3. If there is an entry with minor address n, let p be the Cell corresponding to the entry; continue with step 2.

step 2. If op specifies an alter operation and the final operand in w is not Ø, go to step 6; otherwise go to step 7.

step 3. If the Association Table shows that no Structure Cell has status free, go to step 4. Otherwise let p be a Cell with status free; go to step 5.

step 4. Use the Stack to choose a Cell p in occupied status for preemption.

step 5. If op specifies an alter operation and the final operand in w is ∅, set the status of Cell p to {n, occupied}; go to step 7. Otherwise continue with step 6.

step 6. Change the entry in the Association Table for Cell p to {n, engaged}. Transmit the appropriate retrieve command packet to the Packet Memory System via the Memory Command Network.

step 7. Transmit the contents of the instruction packet to the Cell p. If the status of the Cell is engaged, await the arrival of a retrieve packet; otherwise continue with step 8.

step 8. If Cell p is occupied, perform the structure operation specified by op on the contents of Cell p. Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

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

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

step 2. Transmit the contents of the retrieve packet to the Cell p.

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

step 4. Perform the operation specified by the instruction present at Cell p upon the new contents of the Cell. Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

3.3.4 Operation of an Instruction Cell Block

The composition of an Instruction Cell Block in the Instruction Memory is similar to that of a Structure Cell Block, consisting of a number of Instruction Cells, an Association Table, and a Stack (Figure 3.15). The Instruction Cell Block acts as a cache in a manner similar to the operation of a Structure Cell Block. However, an instruction in an Instruction Cell Block is not retained in a Cell after being sent to a Functional Unit; rather, the Cell is freed for use by another instruction. Also, an item in an Instruction Cell Block is returned to the Packet Memory System only if the Cell it occupies is needed for a more active instruction.

The Association Table of an Instruction Cell Block must have greater capability than one in a Structure Cell Block in order to keep track of instructions which have been returned to the Packet Memory System. The possible values of the status indicator in the Association Table of an Instruction Cell Block are:

free
engaged
occupied
absent

The first three status values have the same meaning as the corresponding values in a Structure Cell Block. The status absent is used to indicate that the associated instruction has been displaced from the Cell Block to the Packet Memory System.

Since an Association Table in an Instruction Cell Block can contain an undetermined number of items with status absent, the structure of the Association Table must be extended beyond that utilized in a Structure Cell Block. The Association Table of an Instruction Cell Block is divided into two

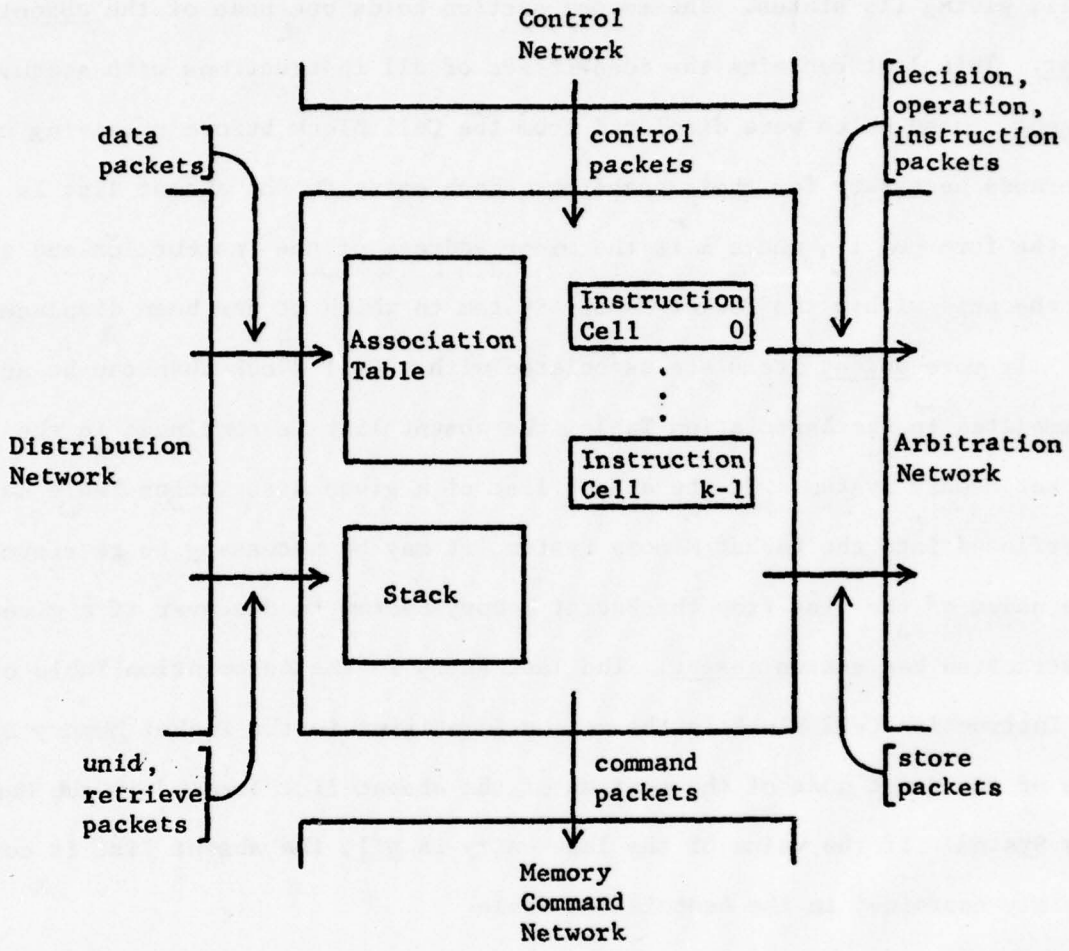


Figure 3.15. Organization of an Instruction Cell Block.

sections (Figure 3.16). The first section corresponds to the Association Table of a Structure Cell Block and contains an entry for each Instruction Cell, giving its status. The second section holds one node of the absent list. This list contains the identifiers of all instructions with status absent, those which were displaced from the Cell Block before receiving all operands necessary for their enabling. Each entry in the absent list is of the form {m, i}, where m is the minor address of the instruction and i is the unid within the Packet Memory System to which it has been displaced.

If more absent items are associated with a Cell Block than can be accommodated in the Association Table, the absent list is continued in the Packet Memory System. If the absent list of a given Association Table has overflowed into the Packet Memory System, it may be necessary to retrieve the nodes of the list from the Packet Memory System to discover if a given instruction has status absent. The last entry in the Association Table of an Instruction Cell Block is the unique identifier in the Packet Memory System of the first node of the portion of the absent list located in the Memory System. If the value of the last entry is nil, the absent list is completely contained in the Association Table.

The number of Instruction Cells in an Instruction Cell Block and the size of the Association Table must be properly chosen in a data-flow processor so that only a small number of instructions associated with a Cell Block have status absent at any one time. This insures that the absent list for a Cell Block is, in most cases, completely contained in the Association Table of that Cell Block.

A data or control packet received by an Instruction Cell Block may not be completely processed at one time if its destination instruction is not

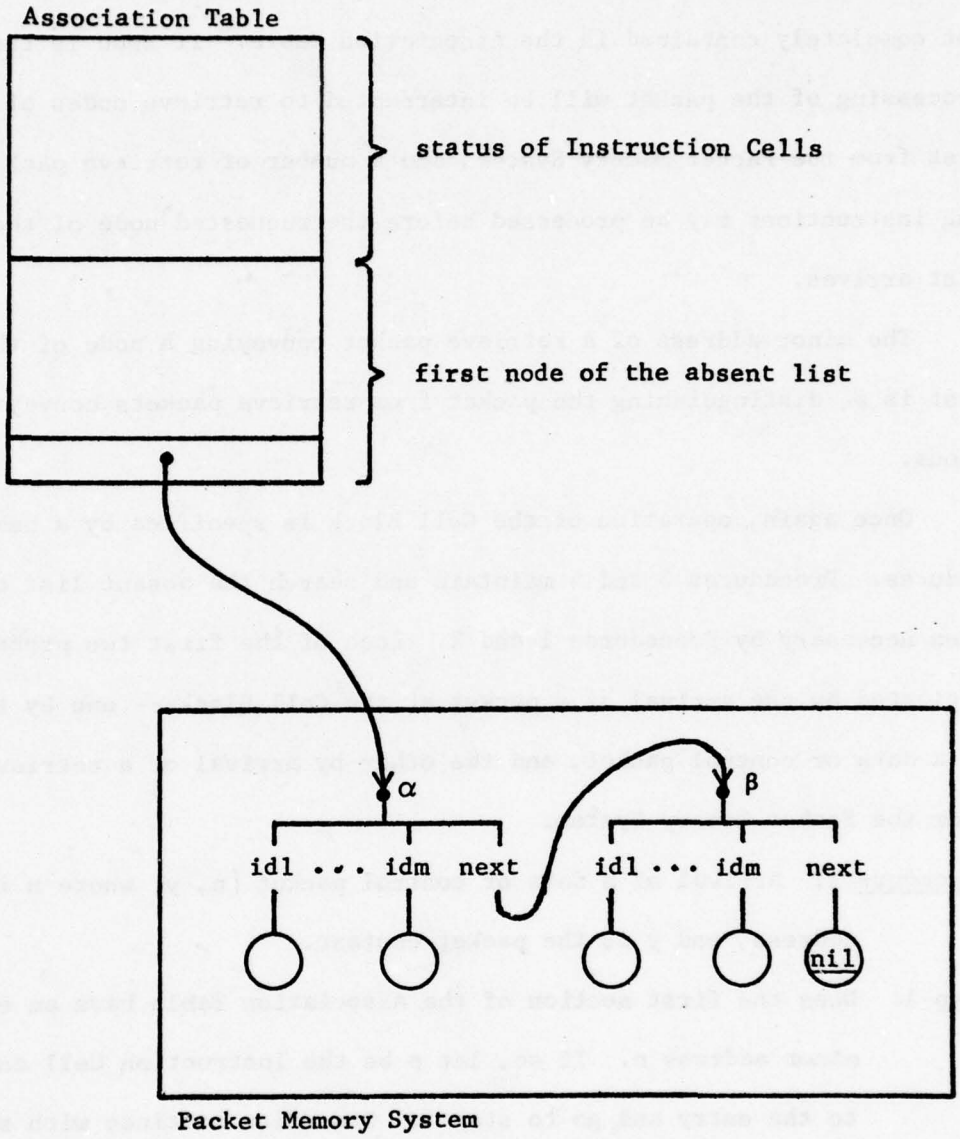


Figure 3.16. Structure of the Association Table for an Instruction Cell Block.

listed in the Association Table of the Cell Block and the absent list is not completely contained in the Association Table. If such is the case, the processing of the packet will be interrupted to retrieve nodes of the absent list from the Packet Memory System, and a number of retrieve packets conveying instructions may be processed before the requested node of the absent list arrives.

The minor address of a retrieve packet conveying a node of the absent list is $\emptyset$, distinguishing the packet from retrieve packets conveying instructions.

Once again, operation of the Cell Block is specified by a number of procedures. Procedures 3 and 4 maintain and search the absent list and are used when necessary by Procedures 1 and 2. Each of the first two procedures is activated by the arrival of a packet at the Cell Block -- one by the arrival of a data or control packet, and the other by arrival of a retrieve packet from the Packet Memory System.

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 first section of the Association Table have an entry with minor address n. If so, let p be the Instruction 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 Association Table entry for p be {-, free}; go to step 4.

step 3. Use the Stack to choose a Cell p in occupied status for preemption; return Cell p to the Packet Memory System through execution of Procedure 3; then continue with step 4.

step 4. Make an entry {n, engaged} for Cell p in the Association Table. Determine whether the instruction with minor address n has status absent through execution of Procedure 4.

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 given in Figure 3.6). 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 change must be consistent with the register status left by preceeding packet arrivals.

step 6. If all operand registers of Cell p are in the trap state as defined in Figure 3.6, set the Association Table entry for Cell p to {n, free}. Otherwise continue with step 7.

step 7. If Cell p is occupied:

a. If all three registers are enabled (according to the rules of Figure 3.6), the Cell p is enabled; transmit an operation or decision packet to the Arbitration Network and set the status of Cell p to free.

b. If the Cell p is not enabled, change the order of Cells in the Stack to make Cell p the last candidate for displacement.

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

step 1. If the minor address n is Ø, the packet is a node of the absent list and has been requested and is processed by Procedure 4. If the minor address is not Ø, continue with step 2.

step 2. Let p be the Instruction Cell with entry {n, engaged} in the Asso-

ciation Table.

step 3. The status of the operand registers of Cell p must be consistent with the content x of the retrieve packet, or the program is invalid. Update the contents of Cell p to incorporate the instruction and operand status information in the retrieve packet.

step 4. Change the Association Table entry for Cell p from {n, engaged} to {n, occupied}.

step 5. If all operand registers of Cell p are in the trap state as defined in Figure 3.6, set the Association Table entry for Cell p to {n, free}. Otherwise continue with step 6.

step 6. If all registers of Cell p are enabled, then Cell p is enabled: transmit the Cell contents to the Arbitration Network and set the status of the Cell to free.

Procedure 3: Return Cell p with status {n, occupied} to the Packet Memory System.

step 1. Transmit a {Ø, getid} command packet to the Packet Memory System to obtain a free unid. The major address of the Cell Block is appended to the minor address Ø of the command packet as it travels through the Memory Command Network.

step 2. Process all retrieve packets received (Procedure 2) until a unid packet {Ø, i} arrives at the Cell Block. Upon its arrival, transmit to the Packet Memory System the store packet {i, x}, where x is the contents of Cell p. Transmit the store command packet {i, store} to the Memory Command Network.

step 3. Is the portion of the Association Table holding the absent entries full? If not, go to step 5. Otherwise transmit another {Ø, getid}

command packet to the Packet Memory System and continue to process retrieve packets with Procedure 2.

step 4. Upon receipt of the unid packet $\{\emptyset, j\}$, transmit the content z of the second section of the Association Table to the Packet Memory System as a store packet $\{j, z\}$. Send the store command packet $\{j, \underline{store}\}$ to the Packet Memory System. Set the last entry in the Association Table to j.

step 5. Add the entry $\{n, i\}$ to the absent list in the Association Table.

Procedure 4: Search the absent list for an entry with minor address n.

step 1. Let β be the value of the last entry of the Association Table.

step 2. Does the node of the absent list located in the Association Table contain an entry with minor address n? If not, go to step 3. Otherwise let the entry with minor address n be $\{n, m\}$; go to step 6.

step 3. Let α be the value of the last entry of the Association Table. If α is nil, go to step 5. Otherwise transmit the retrieve command packet $\{\alpha, \underline{retr}, \emptyset\}$ to the Packet Memory System through the Memory Command Network and continue to process retrieve packets using Procedure 2.

step 4. Upon receipt of the requested retrieve packet, set the last entry of the Association Table to the value designated by the selector next in the retrieve packet. Let z be the content of the absent list located in the Association Table; transmit the store packet $\{\alpha, z\}$ and the store command packet $\{\alpha, \underline{store}\}$ to the Packet Memory System through the Arbitration Network and the Memory Command Network. Transmit the content of the retrieve packet to the absent list in the Association Table; go to step 2.

step 5. Set the last entry of the Association Table to β; transmit the command packet {n, retr} to the Packet Memory System via the Memory Command Network; go to step 8.

step 6. Transmit the command packet {m, retr, n} to the Packet Memory System through the Memory Command Network; transmit the command packet {m, dwn} to the Packet Memory System.

step 7. Delete the entry {n, m} from the absent items in the Association Table. If no absent items are left in the Association Table and β ≠ nil, send the command packets {β, retr, ∅} and {β, dwn} to the Memory Command Network, and upon receipt of the retrieve packet, transmit it to the absent list in the Association Table. Otherwise set the last entry of the Association Table to β.

step 8. Return to Procedure 1.

3.4 Procedure Implementation

All programs executed within the data-flow processor are data-flow procedures. Although this restriction to programs represented as acyclic directed graphs rules out the performance of an iterative computation, we shall see in Chapter 4 that this is not a serious restriction, and that iterative computation can be efficiently expressed and performed through recursive procedure activation.

The procedures of the data-flow language are represented as acyclic directed graphs in the manner described in Section 2.3. A procedure can thus be manipulated as a structure within the structure processing section of the processor through use of the structure operations construct, append, delete, and select defined in Section 2.2. A compiler can construct a data-

flow program as a structure within the Structure Memory, and modified versions of a program can be readily created.

Each instruction of a data-flow procedure is represented as one node in the program structure. An operand in an instruction, consisting of a gating code and a receiver, occupies one register in an Instruction Cell, and its components are located in the same register of a Structure Cell containing the instruction. A destination identifier occupies a complete register in a Structure Cell, whereas it occupies only a portion of a register in an Instruction Cell. This is necessary to allow a procedure to be processed as a structure in the Structure Memory, yet occupy the minimum amount of space in the Instruction Memory. The changes in format of an instruction occur as it is transferred into and out of an Instruction Cell, hence the format of the instruction is that of the structured representation at all times when it is not resident in an Instruction Cell.

The formation of programs within the structure processing section of the data-flow processor and their subsequent execution in the instruction processing section requires that the Packet Memory Systems of the two sections be combined for efficient operation. The organization of the complete data-flow processor with combined Packet Memories is shown in Figure 3.17.

Although we could eliminate the Instruction Memory and extend the capability and complexity of the Structure Memory to allow it to execute a data-flow procedure by means of the structure operations described for instruction execution in Section 2.3, there are a number of reasons for continuing to process instructions in a separate Instruction Memory. First, each instruction is only used once and then discarded from the Instruction Memory, whereas structure nodes are retained within the Structure Memory

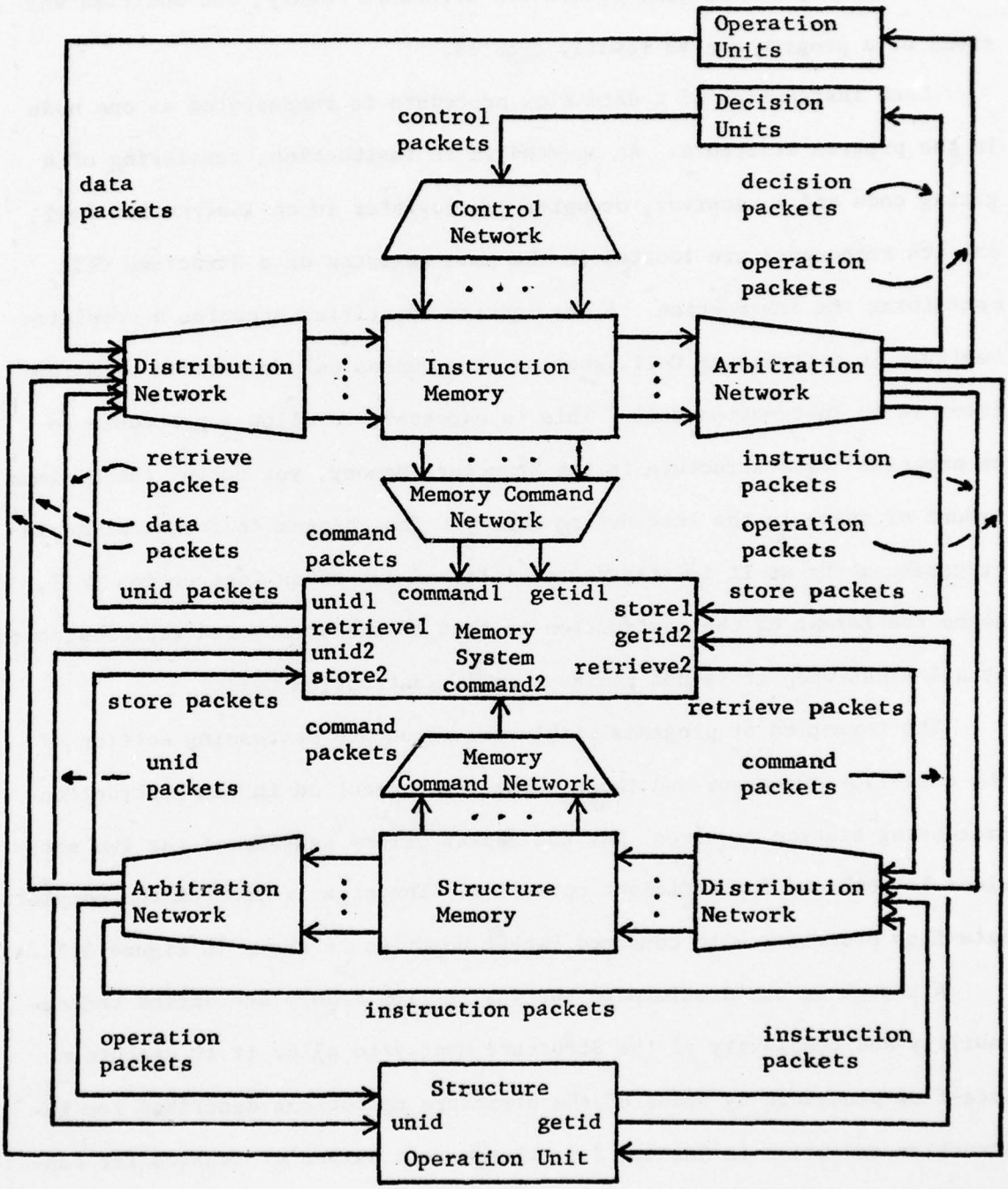


Figure 3.17. Organization of the data-flow processor.

after being used. Hence the algorithm to handle the Association Table and Stack in an Instruction Cell Block differs greatly from the one used in a Structure Cell Block. Second, the operations performed by an Instruction Cell are very dissimilar to those performed by a Structure Cell, and the complexity of a Cell designed to handle both would be excessive. Third, the use of specialized Instruction Cells allows the function of gates to be incorporated into the instructions associated with operators and deciders, rather than using separate instructions for the representation of gate actors. Last, the use of separate memories reduces somewhat the size of each Arbitration and Distribution Network, increasing the utilization of each memory [24].

Each node of a program structure representing a data-flow procedure is identified by its unique identifier within the Packet Memory System of the processor. The contents of the nodes designated by destination selectors in an instruction are the unique identifiers of the destination instructions. A procedure in the data-flow processor is identified by the unid of its first instruction.

There are a number of possible approaches to the execution of a procedure in the data-flow processor. The "copy rule" for procedure activation could be strictly followed, creating a unique copy of the procedure for each activation. Or, a unique activation record could be formed, containing the data associated with an activation. Both of these approaches have the same major problems. The copying of an arbitrary program structure or activation record is not easily accomplished within the processor due to the presence of shared nodes. Also, using a separate structure for each activation is wasteful; such a structure would not be fully utilized since the number of

unexecuted instructions in a program increases at a rapid rate with the number of conditionals in the program, and the version of a program stored in the Packet Memory System is only modified when instructions of the program are displaced from the Instruction Memory. In a properly structured processor, the number of instructions returned to the Packet Memory System should be small.

The approach presented in this thesis, while not as elegant as a strict implementation of the copy rule, is more efficient in that a new copy of an instruction is created only when necessary to avoid conflict. One copy of a procedure is maintained in the Packet Memory System, and each activation of the procedure retrieves its instructions from that copy. A procedure activation is uniquely identified by the unid of its argument structure. The node identifier of an instruction sent to the Instruction Memory is formed by a concatenation of the argument structure identifier and the instruction identifier. The argument structure identifier must be associated with all instruction identifiers during execution of the procedure.

The only possibility of conflict between separate activations of a procedure arises when a partially enabled instruction is returned to the Packet Memory System from an Instruction Cell Block. Conflict is avoided by assigning a new unid to an instruction which is returned to the Packet Memory System and changing its status in the Cell Block to absent, as described in Section 3.3.4. We are assured that in a correct program an instruction with status absent will eventually receive another operand and be recalled to the Cell Block from which it was displaced.

The Structure Operation Unit of the processor is responsible for the processing of apply and return instructions. An apply instruction can be transmitted to the Structure Operation Unit from either the Instruction Mem-

ory or from outside the processor. The instruction is of the following format

$$\left\{\begin{array}{c}\underline{\text{apply}}\ \text{dest} \\ P \\ \alpha\end{array}\right\}$$

and specifies that the procedure with root node P is to be applied to the structure with root node α. The Structure Operation Unit first creates an argument structure with components α and the destination dest. This argument structure is then presented to a new activation of P.

Upon reaching a <u>return</u> instruction in the execution of a procedure, the Structure Operation Unit is sent an operation packet of the form

$$\left\{\begin{array}{c}\underline{\text{return}}\ - \\ \text{dest} \\ \gamma\end{array}\right\}$$

where dest is the destination specified in the <u>apply</u> instruction which activated the procedure, and γ is the unid of the result structure. The Structure Operation Unit merely forms a data packet of the destination and the result and presents the packet to the Distribution Network associated with the Instruction Memory:

$$\left\{\begin{array}{c}\text{dest} \\ \gamma\end{array}\right\}$$

The process of procedure activation can be examined more closely by considering the activation of the procedure P shown in Figure 2.11. Upon receipt by the Structure Operation Unit of an <u>apply</u> instruction

$$\left\{\begin{array}{c}\underline{\text{apply}}\ \text{dest} \\ P \\ \alpha\end{array}\right\}$$

the following sequence occurs:

1. An argument structure δ is created with destination dest and argument α.

2. The data packet

$$\left\{\begin{matrix}\beta \\ \delta\end{matrix}\right\}$$

is sent to the Instruction Memory. β is an identifier formed by the concatenation of the unique identifiers δ and P. A command packet {P, retr, β} is sent from the Instruction Memory to the Packet Memory System to retrieve the instruction P and send it to destination β.

Since the initial instruction of P is merely a distribution instruction which sends the input structure to the select instructions, the Instruction Cell assigned to β is enabled as soon as the retrieve packet conveying P arrives at the Cell. Upon its becoming enabled:

1. An operation packet of the following format is sent to the Arbitration Network:

$$\left\{\begin{matrix}\underline{\text{dist}} & \text{dest1} \\ \text{dest2} & - \\ & \alpha\end{matrix}\right\}$$

2. The status of the Instruction Cell holding P is set to free.

The execution of the procedure then proceeds in accordance with the rules presented previously.

Chapter 4

RECURSIVE vs. ITERATIVE REPRESENTATION

To illustrate the reason for restricting programs within the data-flow processor to those represented as acyclic directed graphs, this chapter examines the performance tradeoff between iterative and recursive computation within the data-flow architecture. The restriction is necessitated by the threat of deadlock which immediately arises in a cyclic computation such as iteration. In this chapter we consider the impact on processor performance of the necessary modifications to an iterative program to assure freedom from deadlock. The performance of iterative and recursive program structures within the data-flow processor is then examined to demonstrate the greater efficiency of the acyclic recursive version.

4.1 The Nature of the Deadlock Problem

A close examination of the architectural basis of the data-flow processor immediately brings one face-to-face with one of the classic problems of parallel computation, that of deadlock. This deadlock problem manifests itself in a manner which affects the fundamental operation of the processor. Therefore, modifications to the architecture are necessary in order to prevent a deadlock condition from arising in a cyclic or stream-oriented computation.

The nature of the deadlock problem and its solution for stream-oriented computation is thoroughly discussed in [23]. In this section we consider the implications of this problem for iterative program structures and present a solution to the problem. In Section 4.3 we examine the impact of this

solution upon the performance of an iterative data-flow program.

The firing rule for an operator or link of a data-flow program states that the operator or link cannot become enabled unless there is no token on any output arc of that operator or link. However, the architecture of the processor described in Chapter 3 provides no mechanism by which an instruction can check that the registers specified in its destination address fields are empty. Consequently, the firing rule can be violated.

The highly parallel and cyclic nature of an iterative computation can allow several cycles of the iteration to be simultaneously active. Due to the fact that the firing rule is not observed, tokens from different cycles can then interact within the program. In illustration of how a deadlock condition arises from this interaction, consider the iterative data-flow program of Figure 2.3 which represents the following computation:

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

Upon completion of the program, n is equal to the original value of x divided by y, and x is equal to the remainder. A detailed description of the operation of this program is presented in Section 2.1.

In the program of Figure 2.3 it is possible for the decider to fire a significant time before the subtraction operator. Once the decider has fired, if the result of the decision is true, a token conveying the value y is returned to the input of the program through the left-most merge actor. If the subtraction operator has not fired by this time, then two tokens carrying the value y can be simultaneously present on an input arc of the operator. Within

the processor, a data packet conveying the new value of y destined for the subtraction operator is stored in a buffer unit of the Distribution Network since the destination register is occupied. The stored packet blocks access to succeeding switch units, and a deadlock condition arises if the data packet conveying the value x necessary for the enabling of the subtraction operator is blocked by this stored packet.

Note that such blocking can also occur within the Distribution Network of the structure processing section of the machine. If a number of operation packets are destined for the same Structure Cell, they may temporarily block other packets in the Distribution Network. However, no deadlock condition can arise since a blocking packet will eventually move on and cannot block packets which are needed by the Cell to which it is destined.

The solution to the deadlock problem in the Instruction Memory requires the addition of a form of feedback between operators of a program in order to force the program to observe the firing rule. In the case of the iterative data-flow program of Figure 2.3, the feedback assures that all operations of one cycle of the iteration are concluded before the next cycle is initiated. The feedback is accomplished by placing a decider with the nil predicate on the output link of each gate actor of the program. Each of these deciders, upon receiving a data value indicating that the gate has fired, produces a control-valued control token. All control-valued tokens so produced are ANDed, and the resulting token is used to reenable the program input. The deadlock-free version of the iterative data-flow program of Figure 2.3 is shown in Figure 4.1.

The merge actors can no longer be ignored in the implementation of the data-flow program since they are now utilized to control the initiation of

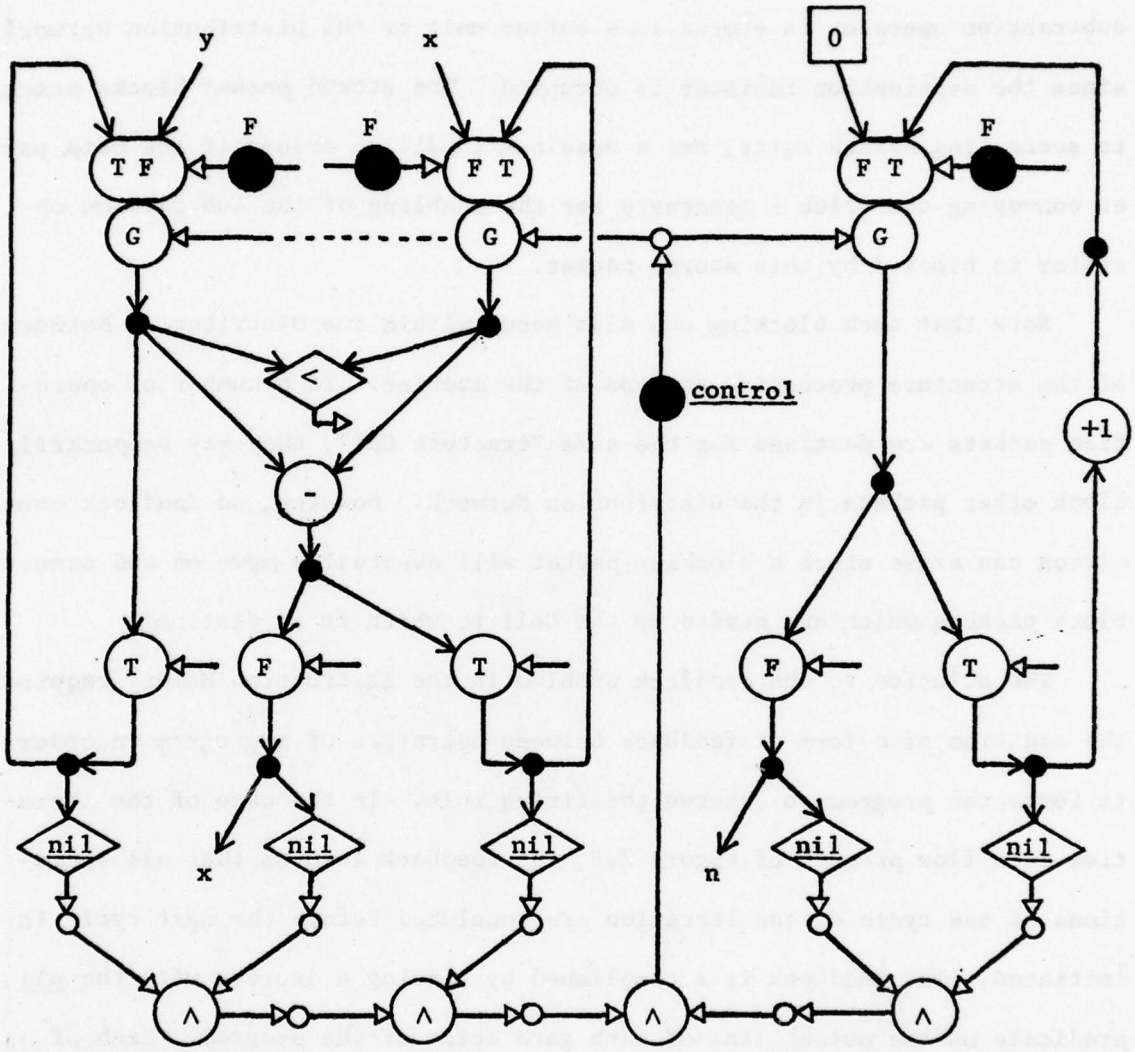


Figure 4.1. Deadlock-free version of the iterative data-flow program of Figure 2.3.

each cycle of the iteration. Each merge actor has a gate on its output. This gate allows the merge actor to become enabled only after receiving a data value and a control-valued control token. Thus, a token indicating that the previous cycle of the iteration has been completed must be received before initiation of a new cycle.

The cost of implementing this deadlock solution is high, both in terms of space and time. The instruction register of an Instruction Cell must be augmented to allow the receipt (when necessary) of a control-valued token for the enabling of that register [23]. A number of Boolean operators must be added to the program to perform the ANDing of the control-valued packets, and a merge instruction must be implemented in the architecture.

4.2 Performance of the Architecture

In this section we describe an elementary analysis of the performance of a data-flow processor. The performance of the processor is examined through consideration of the flow of packets within the networks of the processor. This section is intended to give us a framework within which to intelligently compare the performance of cyclic and acyclic representations of a computation and does not touch upon peripheral issues such as the proper structuring of the networks of the processor. A more detailed analysis of the performance of a data-flow architecture and rules for the structuring of the various networks are presented in [24].

The performance of a data-flow processor can be measured through consideration of the minimum elapsed time between the enabling of an instruction and the arrival of its results at the desired destinations. If the number of Functional Units is chosen and the networks of the processor are struc-

tured so that all instructions pass through the Arbitration Network in this time, then we are assured of maintaining a constant supply of instructions for the Functional Units. We have also discovered the processing capability of the machine.

The cycle time of an instruction within the processor is the minimum elapsed time between the enabling of the instruction and the arrival of the results of the operation specified by the instruction at their destination Cells. The cycle time for a given instruction is affected by conflict in the networks. For a simple instruction representing an operator of a data-flow program, the cycle time is equal to the passage time through the Arbitration Network, the Distribution Network, and an appropriate Operation Unit. The delay in the Operation Unit is fixed for that Operation Unit. However, the network delays can vary greatly.

The cycle time for an instruction is found by considering the passage of the operation packet containing that instruction thrcugh the Arbitration Network and the passage of the resulting data packets through the Distribution Network, assuming no conflict arises in either network. The minimum delay through a network, the Arbitration Network for example, is given by the summation over the number of stages in the network of the time required to transfer a packet through each stage:

$$\text{minimum delay} = \sum_{\text{stages}} (\text{no. bits serial} + 1)(\text{bit transfer time})$$

The transfer time for a stage is equal to the number of bits passing through the stage in serial plus one for a signal to indicate that the packet is ready to be transferred multiplied by the time necessary to transfer a bit. A similar equation applies to delay in the Distribution Network.

Let us examine the delay within a specific Arbitration Network (Figure 4.2). This network has three stages and seven arbitration units. Packets travel through stage 0 in four-bit serial format and are gradually converted to a more parallel format, passing through stage 1 in two-bit serial and stage 2 in one-bit serial format. As noted previously, the passage time for a packet through each stage is equal to the number of serial bits plus one times the bit transfer time t. For this structure, the transfer times are 5t, 3t, and 2t, respectively. The minimum delay through the network is equal to the summation of the stage delays, or 10 t.

In order to find the time necessary to process all instructions contained in the Instruction Memory of the processor, T, we must consider the maximum delay a packet can encounter in passing through the Arbitration Network. Such a maximum delay can only occur in a network which has a packet present at every node in a machine in which every Instruction Cell Block contains an enabled instruction, placing a packet on each input to the Arbitration Network (Figure 4.3). The maximum delay which can be encountered by a packet, say the triangular one, arises only when all other packets in the network and at the inputs of the network pass through the output of the network before the triangular one does. In order for this to happen, not only must the triangular packet lose every conflict, but every packet on the path it will follow to the output must also lose every conflict. Thus, finding the maximum delay involves examining how many packets flow through each stage before the triangular one.

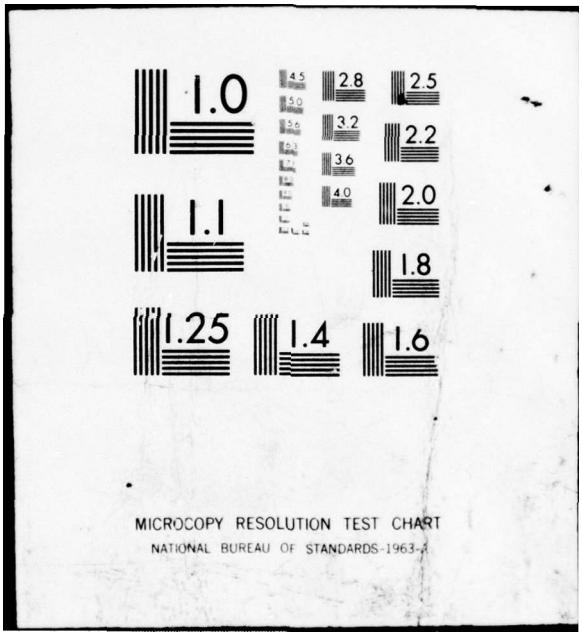
For this network the worst case packet will be the 14th through stage 2, the 6th through stage 1, and the 2nd through stage 0. Multiplying the number of packets passing through each stage by the delay in that stage,

AD-A052 538 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
A COMPUTER ARCHITECTURE FOR DATA-FLOW COMPUTATION.(U)
MAR 78 D P MISUNAS N00014-70-A-0362-0006

UNCLASSIFIED MIT/LCS/TM-100 NL

2 OF 2
AD
A052538

END
DATE
FILMED
5-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Stage Number	0	1	2
Serial Bits	4	2	1
Passage Time	5t	3t	2t

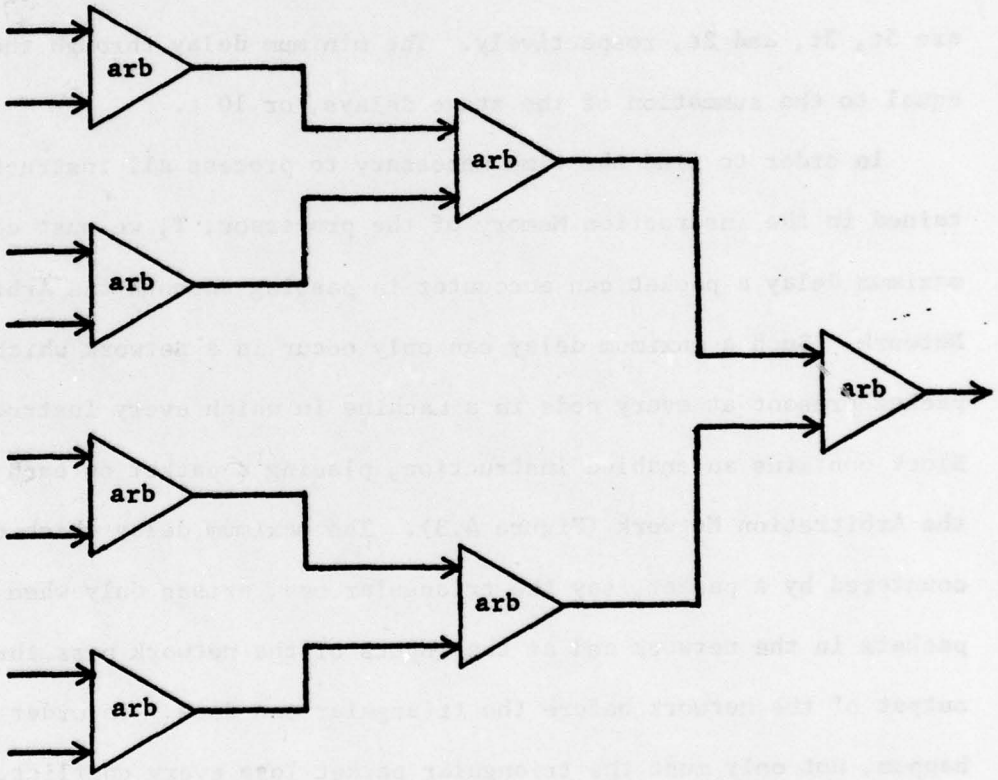


Figure 4.2. Structure of an elementary Arbitration Network.

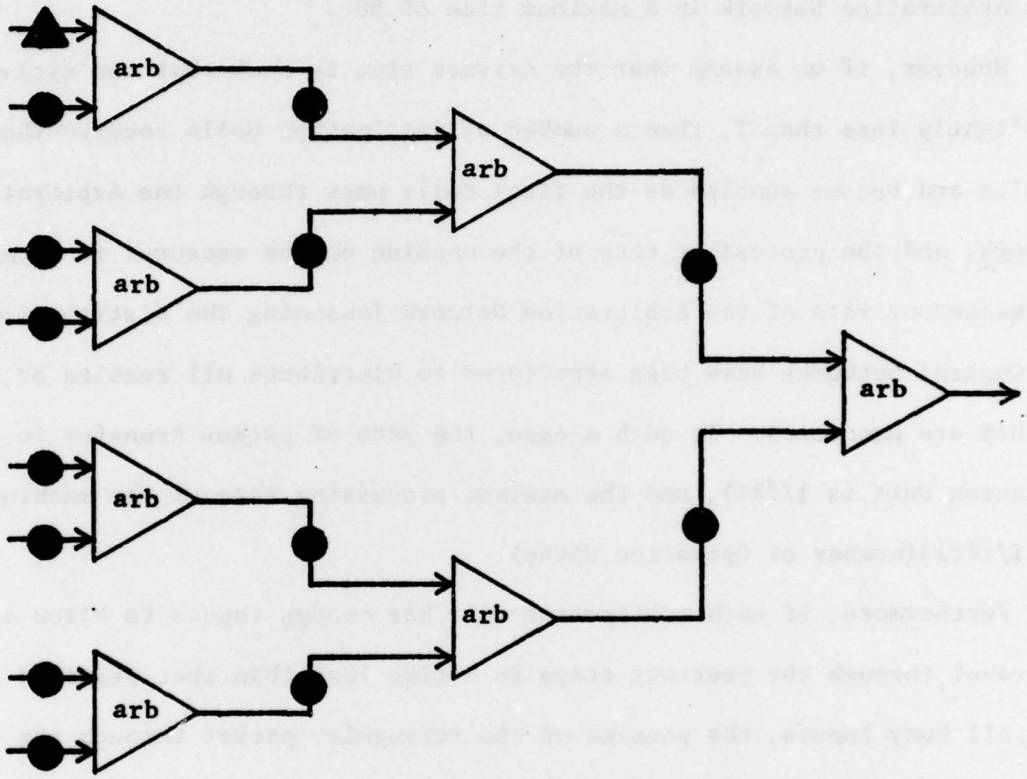


Figure 4.3. Example of a full Arbitration Network.

we find that:

$$T = \text{maximum delay} = 2(5t) + 6(3t) + 14(2t)$$
$$T = 56t$$

Thus, if all instructions of the processor are enabled, they can pass through this Arbitration Network in a maximum time of 56t.

However, if we assume that the network size is such that the cycle time is slightly less than T, then a number of destination Cells receive their results and become enabled as the final Cells pass through the Arbitration Network, and the processing rate of the machine can be measured in terms of the output rate of the Arbitration Network (assuming the Distribution and Control Networks have been structured to distribute all results as fast as they are produced). In such a case, the rate of packet transfer to each Operation Unit is 1/(2t), and the maximum processing rate of the machine is [1/(2t)](number of Operation Units).

Furthermore, if each arbitration unit has enough inputs to allow a packet to travel through the previous stage in a time less than that required to service all busy inputs, the passage of the triangular packet through the first stages of the Arbitration Network will occur simultaneously with the transmission of other packets at the output. The time T for the transmission of all packets in the network to the Operation Units is then 14(2t) = 28t.

4.3 Example: An Iterative Computation

The program modifications described in Section 4.1, necessary to assure freedom from deadlock in an iterative computation, cause a significant degradation in the performance of such a computation. Applying the analysis techniques developed in the previous section to a specific processor allows

us to examine the actual nature of the performance degradation.

In illustration of the performance of a data-flow processor executing a cyclic computation, consider a 128 Instruction Cell Block machine in which an operation packet exits a Cell Block in 16-bit parallel, 4-bit serial format. For a balanced processor structure, one in which the number of Functional Units is matched to the processing rate, the processing time T must be equal to the activation time, that is, the minimum time between successive activations of an instruction within the processor. Thus, in order to determine the performance of the processor, we must consider the structure of the networks and determine the value of the activation time.

The modifications to an iterative program necessary to assure freedom from deadlock can cause a significant delay to exist between successive activations of a given instruction. This delay is due to the fact that each Instruction Cell which contains a gated operand must, as it sends out an operation packet, return control packets to preceding actors in the program which supply values to the gated register(s). Upon receipt of these control values, the preceding operators are reenabled if their operands are present. If D is the cycle time for an operation packet, and d is the cycle time for a control packet; that is, the delay through the Arbitration Network and the Control Network, then the minimum activation time of an Instruction Cell is D+d.

In order to obtain a small activation time, and hence a greater processing capability, the networks must be structured with as few stages as possible. However, a minimum of three stages is required within the networks of this processor to perform the serial-to-parallel conversions and still maintain the necessary throughput from stage to stage. The minimum

delay analysis of the three stage network structure of the 128 Cell Block processor is identical to that described in the previous section; the minimum delay in the Arbitration Network is equal to 10t.

Assuming the delay in the Distribution Network is the same as that in the Arbitration Network, and the minimum delay in the Control Network is approximately one-half the delay in the Arbitration Network or Distribution Network, the resulting values for the cycle times and activation time AT are:

$$D = 20t$$
$$d = 15t$$
$$AT = D+d = 35t$$

If t = 150 nanoseconds, allowing 15 TTL gate delays to accomplish one ready/acknowledge cycle, the resulting activation time is:

$$AT = 35(150 \text{ nsec.})$$
$$= 5.25 \ \mu\text{seconds}$$

And the maximum processing capability of the architecture containing cyclic program structures is:

$$\text{processing rate} = \frac{128 \text{ instructions}}{5.25 \ \mu\text{seconds}}$$
$$= 24 \text{ MIPS (million instructions per second)}$$

4.4 Recursive Representation

The set of problems which can be expressed iteratively is a subset of those suited to recursive representation. Therefore, any iterative computation can be expressed in the data-flow language as a procedure and can be performed through recursive procedure activation.

Naturally, if the objective of a program is to perform the same opera-

tion on a large number of items, then in a parallel computer system, a forall construct can be much more efficiently utilized than an iterative program. However, if each cycle of an iteration depends upon values generated in previous cycles, a recursive representation can be utilized to avoid the cyclic structure and often seems to provide a more readily understandable expression of the computation.

The acyclic nature of a procedure in the data-flow language assures that even though the firing rule may not be observed, two tokens cannot simultaneously exist on one arc of a properly structured program graph. Hence, no deadlock condition can arise within an activation of a procedure or between concurrent or recursive activations of the procedure, and the procedure can be more efficiently executed in the data-flow architecture than a cyclic representation of the same computation.

A recursive description of the iterative computation of Figure 2.3 is stated as follows:

```
P: proc (label P) (x, y, n)
     if x>y then P(x-y, y, n+1)
            else (x, n)

in P(x, y, 0)
```

A data-flow representation for P is given in Figure 4.4. The argument structure for the procedure contains three elements. In addition to the argument and the destination, it holds the unid of P (label P) to allow the procedure to be reapplied if necessary. The number of gates is reduced in the recursive version, reducing the number of control distribution Cells which are needed in the program. However, a number of structure operations have been added, probably balancing the savings involved in eliminating the gates. The apply actor in the program passes as a destination the destination it received, and,

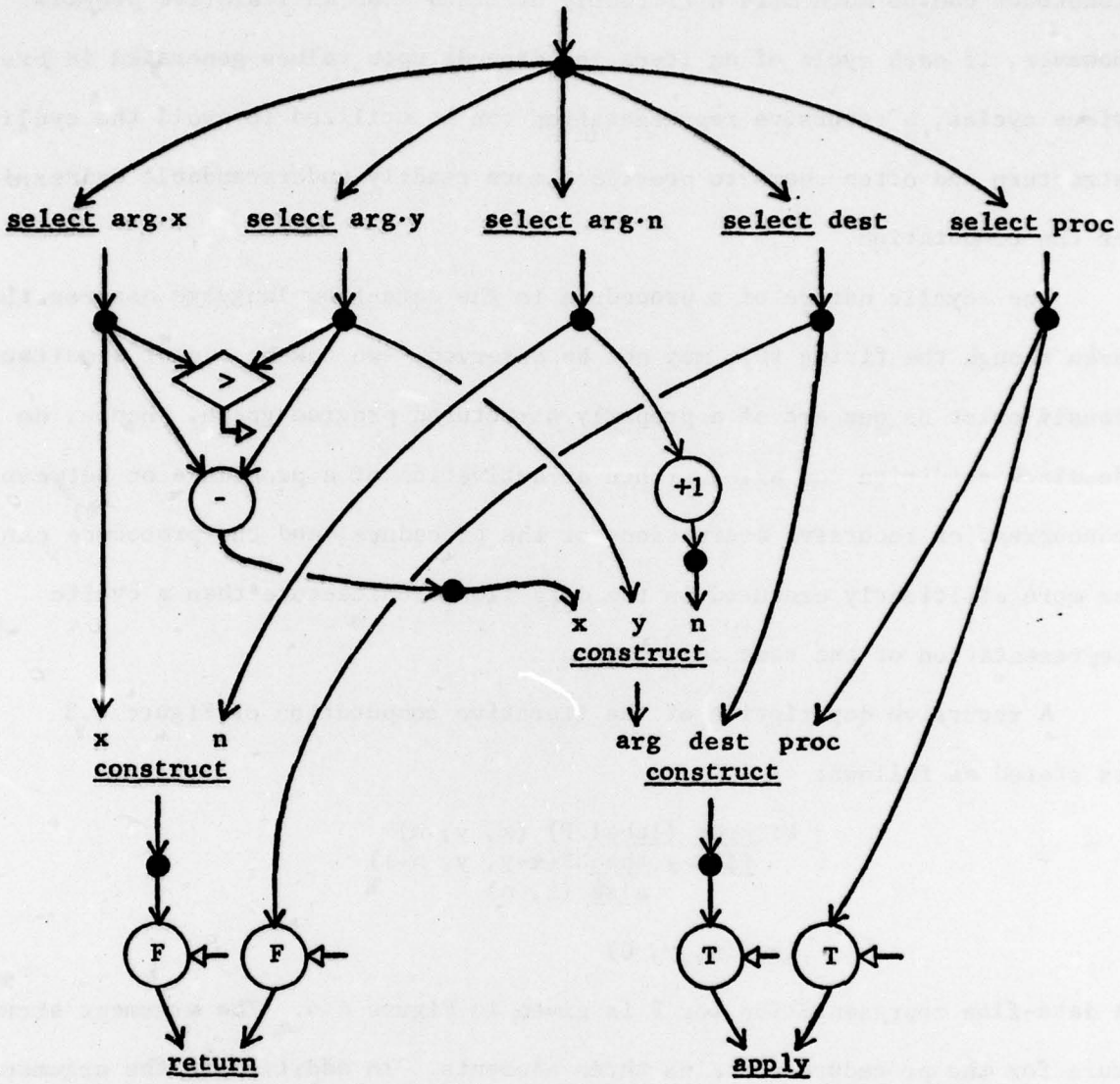


Figure 4.4. Recursive version of the data-flow program of Figure 2.3.

after the firing of the apply actor, the activation is terminated. When the result of the decision is finally false and a return instruction is executed, the destination to which the values x and n are sent is the destination specified by the original calling procedure.

Since no feedback is necessary in the recursive representation, the activation time for the 128 Cell Block processor executing a data-flow procedure is equal to the cycle time D, and the resulting performance of the processor is:

$$\text{maximum processing rate} = \frac{128 \text{ instructions}}{3\ \mu\text{sec.}}$$

$$= 42 \text{ MIPS}$$

Not only does a recursive program representation occupy less space in the data-flow processor than a cyclic program structure, but a processor executing programs represented as acyclic directed graphs can realize almost twice the performance achievable through the execution of cyclic program structures.

Chapter 5

TOPICS FOR FURTHER RESEARCH

In this thesis the data-flow architecture has been presented as a solution to many of the problems of highly parallel computer systems. The use of interconnection networks between sections of the processor provides an attractive approach to the communication of information between units comprising a large system and allows an interesting method of analyzing system performance through the examination of information flow within the networks.

Due to the radical nature of the architecture presented, undoubtably more questions have been raised by this thesis than have been answered. These questions range from mild quibblings over the use of certain methods of representation or design choices to deep semantic and philosophical issues. In this chapter we will point out some of the unanswered questions and indicate some of the issues involved.

The data-flow procedure language, while appearing to be a semantically elegant method of expressing parallelism, is wide open to further study and extension. The language needs to be expanded by the addition of such actors as a forall construct to enable it to better express concurrent processing of the elements of a structure. However, the best choice of semantics for such constructs is yet to be clearly established. Also, the language does not currently contain the capability to express nondeterminate computation, a very important feature for some applications.

Further investigation of the use of the data-flow language is also necessary. Upon initial examination, the representation of such algorithms as the fast Fourier transform in data-flow form appears very attractive [25].

However, data-flow representations for other computations need to be developed, examined, and contrasted with more conventional programs.

The data-flow language is designed to serve as the base language of the data-flow processor. It is not regarded as an acceptable user language. The development of a user language which can be readily translated into a data-flow representation is currently under study [35]. The translation of programs expressed in a language such as Algol 60 to a data-flow representation is also currently being investigated [4]. Much more work needs to be done to identify concurrency in problems and to take advantage of that concurrency through use of the data-flow representation.

An analysis of data-flow programs is necessary for a complete understanding of the operation of the data-flow processor. In order to determine the number of instructions which must be maintained in the Packet Memory System to provide the necessary number of active instructions to allow the Instruction Memory to operate at its highest rate, one must understand the parallelism achievable in a given data-flow program and the number of active instructions a program can be expected to supply at any point in time.

An important issue which was discussed briefly in this thesis is the tradeoff between cyclic and acyclic representations of a computation. Due to the highly parallel nature of the data-flow architecture, a recursive version of a computation is performed more efficiently than an iterative version. It can also be argued that the recursive version is semantically much cleaner and in many cases, much easier to understand. Further research is necessary to fully understand the use of concurrent and recursive procedure activation for the execution of cyclic and stream-oriented computation.

The data-flow representation and the architecture of the data-flow processor which have been presented in this thesis are very attractive as a means of describing parallel computation and structuring a parallel computer system. The projected performance of the data-flow processor is also very attractive, and hence, we are hopeful that these concepts will prove very useful in the construction of future computer systems.

BIBLIOGRAPHY

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. Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conference Proceedings, 30, AFIPS Press, Montvale, N. J., 1967, 483-485.

3. Amerasinghe, S. N. The Handling of Procedure Variables in a Base Language. S.M. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., September 1972.

4. Amerasinghe, S. N. PhD. Thesis in preparation, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

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

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

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

8. Comtre Corporation. Multiprocessors and Parallel Processing (P. Enslow, Ed.), John Wiley and Sons, New York, N. Y., 1974, 31.

9. Dennis, J. B. First version of a data flow procedure language. Lecture Notes in Computer Science, 19 (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York, N. Y., 1974, 362-376.

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

11. Dennis, J. B. Packet memory system architecture. Paper submitted for presentation at the 1975 Sagamore Computer Conference on Parallel Processing, August 1975.

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

13. 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, 402-409.

14. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975, 126-132.

15. Ellis, D. J. Semantics of Data Structures and References. Report TR-134, Project MAC, M.I.T., Cambridge, Mass., August 1974.

16. Flynn, M. J., A. Podvin, and K. Shimizu. A multiple instruction stream processor with shared resources. Parallel Processor Systems, Technologies, and Applications (L. C. Hobbs, et. al., Eds.), Spartan Books, New York, 1970, 251-286.

17. Henderson, D. A., Jr. The Binding Model: A Semantic Base for Modular Programming. Report TR-145, Project MAC, M.I.T., Cambridge, Mass., February 1975.

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

19. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queing. SIAM Journal of Applied Mathematics, 14, (November 1966), 1390-1411.

20. 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.

21. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3, 4 (April 1960), 184-195.

22. Miller, R. E., and J. Cocke. Configurable computers: A new class of general purpose machines. Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).

23. Misunas, D. P. Deadlock avoidance in a data-flow architecture. Proceedings of the Milwaukee Symposium on Automatic Computation and Control, IEEE, New York, April 1975, 337-343.

24. Misunas, D. P. Performance analysis of a data-flow computer architecture. Paper submitted for presentation at the 1975 Sagamore Computer Conference on Parallel Processing, August 1975.

25. Misunas, D. P. Performance of an Elementary Data-Flow Processor. Computation Structures Group Memo 115, Project MAC, M.I.T., Cambridge, Mass., February 1975.

26. Misunas, D. P. Procedure representation in a data-flow processor. Paper submitted for presentation at the 1975 Sagamore Computer Conference on Parallel Processing, August 1975.

27. Misunas, D. P. Structure implementation in a data-flow computer architecture. Paper submitted for presentation at the 1975 Sagamore Computer Conference on Parallel Processing, August 1975.

28. Rodriguez, J. E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.

29. Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Project MAC Technical Report, M.I.T., Cambridge, Mass., forthcoming.

30. Shapiro, R. M., H Saint, and D. L. Presberg. Representation of Algorithms as Cyclic Partial Orderings. Applied Data Research, Inc., Wakefield, Mass., 1971.

31. Seeber, R. R., and A. B. Lindquist. Associative logic for highly parallel systems. AFIPS Conference Proceedings, 24, AFIPS Press, Montvale, N. J., 1963, 489-493.

32. Thornton, J. E. Parallel operation in Control Data 6600. AFIPS Conference Proceedings, 26, Part II, AFIPS Press, Montvale, N. J., 1964, 33-41.

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

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

35. Weng, K. S. S.M. Thesis in preparation, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass.

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Va 22314 — 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Va 22217 — 2 copies

Office of Naval Research
Code 102IP
Arlington, Va 22217 — 6 copies

Office of Naval Research
Code 200
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 455
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Va 22217 — 1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, Ma 02210 — 1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Il 60605 — 1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, Ca 91106 — 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 — 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 — 6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 — 1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, Ca 92152 — 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Mathematics Department
Bethesda, Md 20084 — 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy