A SURVEY OF DATA FLOW MACHINE ARCHITECTURES

by

DAVID ANTHONY MEAD

B. A., Kalamazoo College, 1962

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

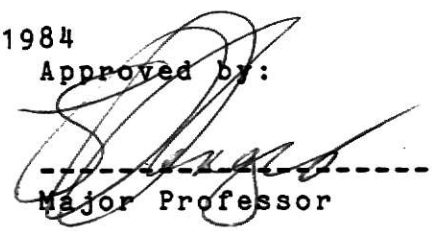
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:



Major Professor

TABLE OF CONTENTS

LIST OF FIGURES.....ii

1.0 INTRODUCTION.....1

2.0 COMPARISON OF COMPUTER ARCHITECTURES.....5

2.1 Motivations for a New Architecture.....5
2.2 A Basis for Comparison.....6
2.3 The Control Flow Model.....7
2.4 The Data Flow Model.....11
2.5 The Reduction Machine Model.....15

3.0 DESIGN OF A GENERALIZED DATA FLOW MACHINE.....18

3.1 Data Flow Graphs.....18
3.2 Major Hardware Components.....31
3.2.1 Storage Units.....32
3.2.2 Instruction Cells.....33
3.2.3 Processing Units.....34
3.2.4 Communication Units.....35
3.2.5 Control Units.....37

4.0 DATA FLOW PROGRAMMING LANGUAGES.....38

5.0 A SURVEY OF DATA FLOW PROJECTS.....42

5.1 The M.I.T Project.....42
5.1.1 The Data Flow Graphs.....42
5.1.2 The M.I.T Machine.....45
5.2 The University of Manchester Machine.....48
5.2.2 The Data Flow Graphs.....48
5.2.2 The Manchester Machine.....50
5.3 The University of Utah Machine.....53
5.3.1 The Data Flow Graphs.....53
5.3.2 The Structure of the DDM1 Machine.62

6.0 CONCLUSIONS.....65

REFERENCES.....67

BIBLIOGRAPHY.....68

LIST OF FIGURES

2.1 Structure of a generalized Data Flow Unit.....13

3.1 A Simple Data Flow Graph.....19

3.2 The SELECT Instruction Cell.....25

3.3 The DISTRIBUTE Instruction Cell.....25

3.4 An Example of Alternation.....26

3.5 An Example of Iteration.....29

3.6 A 2 x 2 Router.....36

3.7 A 4 x 4 Router.....37

5.1 The actors of the M.I.T graph.....44

5.2 The M.I.T Instruction Cell.....45

5.3 The M.I.T Ring.....47

5.4 The Manchester Ring.....51

5.5 The Utah PSE.....63

ACKNOWLEDGEMENTS

I would like to express my appreciation to the Computer Science Department of Kansas State University, and in particular to Dr. Beth Unger for her assistance in the preparation of this paper.

1. INTRODUCTION

In the decades since the invention of the computer, the industry has seen a dramatic growth in the power of these machines. Storage capacities and execution speeds have been increased by many orders of magnitude, while the physical size, power consumption, and cost have shown a similar decrease.

This improvement has been due primarily to advances in the hardware technology, not to any change in the basic conceptual model. The computers of today, as were those of thirty years ago, are based upon the control flow model developed by John von Neumann during the 1940's. Recent demands for achieving greater speed through greater parallelism, supported by advances in the hardware technology such as LSI and VLSI, have prompted certain researchers to seriously investigate the possibilities of alternate models. Treleaven,et al.(1) have stated that "There is growing agreement, particularly in Japan and the United Kingdom, that the next generation of computers will be based on non-von Neumann architecture."

Two of the alternative architectures are the data flow (or data driven) machines and the reduction (or demand driven) machines. The data flow model received its principal stimulus from the work of J. B. Dennis of M.I.T , and is currently being pursued at a number of universities. The work on reduction models has stemmed primarily from the work of John Backus and Klaus Berkling.

The purpose of this paper is to present an overview of

the data flow model; including the conceptual foundations, the machine architecture, langauge considerations, and a survey of three of the university projects. Although reduction machines are not the primary topic of the paper, the basic model will be presented for purposes of comparison.

The second chapter of the paper will be devoted to identifying the motivations for a new architecture, and to presenting and comparing the three computing models.

The third chapter will present a detailed description of the data flow model. First, data flow graphs, also called data dependency graphs, will be presented. These graphs,which are based upon the flow of data, rather than upon the explicit flow of control in the program, are the conceptual foundation of a data flow machine. This will be followed by a description of a generalized machine architecture to illustrate how data flow machines execute programs represented as data flow graphs.

As language design is directly affected by the computing model on which the programs are intended to execute, any change in the basic computing model has an impact on the languages. Data flow machines, in order to obtain the desired parallelism require languages of a type known as nonprocedural languages. In these languages, the order of execution is dependent upon functional relationships of the program, rather than upon the strict lexical ordering of the program statements. Data flow machines also require lan-

guages that are functional; that is, languages which generate programs which are not dependent upon the global storage concepts of the von Neumann machine. Chapter 4 of the paper will discuss these language considerations as they apply to data flow machines.

The final chapter of the paper will survey three of the specific university projects in data flow research. The ongoing project at M.I.T is the earliest of these projects and is based upon the Dennis Data Flow Nets (DDF) developed by Jack B. Dennis. The M.I.T project has also been one of the primary seeds to a number of other data flow projects.

The project at the University of Manchester in England is one of the projects which received its early stimulus from the work of J. B. Dennis at M.I.T. The primary researchers are Ian Watson and John Gurd. The programming language LAPSE was developed in conjunction with this project and is based on DDF Nets.

The project at the University of Utah is based upon the Data Driven Nets (DDN's) developed by Al Davis. This architecture is based upon a hierarchical tree of computing elements and differs in many respects from the ring type archictectures of the other projects. A detailed description of the three projects just listed will be given in Chapter 5 of this paper.

A number of additional projects have been pursued, or are being pursued, at this time. A project is underway at Toulouse University in France which is based upon the single assignment language LAU. In this effort the architecture

started with the language design, and the machine was then designed to support the language.

The project at the University of California at Irvine, the Irvine Data Flow Machine, is one of those which received its early stimulus from the work of Dennis at M.I.T. Arvind and Gostelow were the primary researchers at Irvine, and developed the language Id based upon the DDF nets.

Other projects include those at Newcastle, in England; and one at Texas Instruments in the United States. In addition to these projects, certain universities in Japan, the United Kingdom, and the United States have done some work with data flow concepts; although, some of this work is only of an academic nature.

2. A COMPARISON OF COMPUTER ARCHITECTURES

In this chapter the motivations for a new technology will be presented. Each of the three basic types of architecture, control flow, data flow, and reduction; will then be compared and evaluated in terms of the desirable criteria for the next generation. This comparison of the computer models is primarily from Treleaven,et al.(1).

2.1 MOTIVATIONS FOR A NEW TECHNOLOGY

The first reason that traditional architectures are being questioned as a viable model for the future computers is the need for achieving higher performance. As control flow computers have developed over the past decades, modifications have been introduced which allow varying degrees of concurrency to be realized; however, the basic nature of the von Neumann machine does not inherently support concurrency. The move toward increasing performance through increased parallelism is stimulating research in those architectures which, by their basic design, exploit the capability for parallel processing at the lowest level of the program.

A second consideration in the design of future computers is the necessity of utilizing, to its fullest capability, the technology of LSI and VLSI. This technology realizes its greatest potential for cost effective use when a large number of units, each of low cost, can be combined into the final machine.

2.2 A BASIS FOR COMPARISON

As a basis for comparison between the various computer architectures, Treleaven,et al.(1) have organized the comparison into the following areas.

1). Computational Organization

2). Program Organization

- Data Mechanisms
- Control Mechanisms

3.) Machine Organization

The computational organization of the machine refers to the actual steps required to execute a single instruction. Three phases are recognized. In the select phase, an instruction becomes selected for execution from all of the instructions in the program. In the examine phase, the instructions which have been selected are examined to ensure that all requirements for execution have been satisfied. The third phase is the execute phase, in which the instruction is actually executed. The primary differences between the different models are in the first two phases.

Program organization is compared for both data mechanisms and control mechanisms. The data mechanisms refer to the manner in which data is passed from one execute cycle to succeeding cycles; pass by literal, pass by value, and pass by reference. The control mechanism refers to the manner in which control is passed from instruction to

instruction. In sequential control, each instruction is followed by one, and only one, instruction; the processing proceeding in a step like manner. In a parallel control mechanism, control may split into multiple control paths all of which are active concurrently. In a recursive control mechanism, a function is continually being subdivided into its components, all of which may be executing concurrently.

The machine organization refers to the way in which the resources are organized. In a centralized machine, the the resources are concentrated into a single centralized processor. In a packet communication organization, various machine resources are organized as distributed asynchronous elements which transfer control and data by passing packets. In an expression manipulation organization, multiple resources are demanded as required during the resolution of an expression. The resources in this organization will be organized in some sort of regular order, such as an hierarchical tree.

2.3 THE CONTROL FLOW MODEL

In its simplest form, the von Neumann machine, or control flow machine, is characterized by the following features. First, it consists of three basic hardware units; a central processing unit, or CPU, a memory storage unit which is composed of an array of identical storage units called words, and a connecting link between the two. The CPU, which is the unit where all processing is actually

performed, contains a special register, the location counter, which is used by the processor to retain the current point of processing in the program.

The global storage unit is used to store both the program and the data upon which it is operating. The data is actually processed by changes made one at a time on the data in global storage. To change a word of data, and therefore change the state of the machine, the location counter is used to obtain the next instruction from memory across the connecting link. The instruction contains the operation which is to be performed and some designation of the data upon which the operation is to be performed. The data is usually contained in the instruction as an address which points to the memory location of the data. The data is then obtained from memory, transformed in the processor by applying the designated operation, and then returned to the global storage unit.

After the operation is completed, the processor returns to the location counter to obtain the address of the next instruction to be executed. Normally the location counter is incremented as each instruction is obtained from memory; control, therefore, passing from one instruction to the one which follows it in the storage unit. Conditional processing is supported by certain branching instructions which exert their effect by altering the contents of the location counter, thus causing control to be passed to some point in memory other than the following instruction.

Comparing the control flow model to the criteria for computational organization, the select phase occurs when the location counter selects the next program instruction in memory. Only one instruction may be selected at one time in a control flow model. The examine phase does not exist in this model, and as soon as an instruction becomes selected, it is immediately executed. No examination or verification of the state of the data operands is performed.

In terms of program organization, the data mechanisms may be any of the three possibilities; pass by literal, pass by value, or pass by reference. The most common method is by reference, in which the address of the data is contained in the instruction being executed. The data being accessed is resident in the global storage unit. The control mechanism is sequential, control being passed in a single step manner under control of the location counter.

The machine organization is essentially that of a centralized processor. Certain modifications have been made which support varying degrees of parallelism such as independent I/O channels and parallel processors; however, each of these extentions requires special hardware and software features, such as an interrupt mechanism or a centralized processor to allocate work to other processors. The essential nature of the machine is still sequential.

A number of characteristics become evident when this computer model is examined and compared to the desirable criteria for the next generation computer. First, it contains no inherent mechanism to support concurrency.

Execution of program statements occurs on a one at a time basis and the order of processing is determined solely by the contents of the location counter.

Second, the effect of the machine is applied by passing data forward from one point in the program to the next by storing it in the global storage. This means that the machine is history sensitive, the state of the program being dependent not only upon the current point of control, but also upon the specific ordering of prior execution. As each instruction is capable of accessing data without any verification that the data is in fact at the correct point in its processing, side effects are introduced. It is, in fact, through the side effects that the main effects of data manipulation are applied. It is solely the responsibility of the programmer to ensure that all prior required operations have been applied to the data.

Third, the control flow machine contains a design characteristic which Backus(2) has called the "von Neumann bottleneck", and which leads to inefficiency in the utilization of the hardware. The bottleneck is the connecting link between the CPU and the memory storage. If the execution of a single instruction is examined in terms of its usage of the link, it will be seen that much of the information passing between memory and the CPU is not the actual data. It is other information which is necessary in order to perform the function. First the link must be used to obtain the instruction, using the location counter as an

address source. The instruction will usually refer to the data by its address; therefore, it is necessary for the CPU to use the connecting link a second time in order to retrieve the actual data. The result of this arrangement is that the connecting link is actually spending more time sending control information than it is spending in transferring the actual data to be operated upon.

2.4 THE DATA FLOW MODEL

The theoretical basis of a data flow machine is a directed graph in which the nodes of the graph represent simple operations to be performed (addition, subtraction, etc.), and the arcs of the graph represent the flow of data from instruction to instruction. The graphs are variously known as data flow graphs or data dependency graphs. Each node in the graph may be logically conceived of as having a processor dedicated to it, and when all of the required data input values have been received, the node will execute, placing the result of the operation on the output arcs. The output arcs from one node are the input arcs of the nodes which logically follow in the computation being performed.

In a typical data flow machine, such as the M.I.T machine, the basic machine organization is based on a series of asynchronous components arranged in a ring, with the data and instruction packets being passed from one component to the next in a pipelined fashion. As an instruction cell in

memory becomes fireable due to the presence of the input data on its arcs, an instruction packet is built containing the operation, the operand values, and the designation of the destination arcs. This packet is distributed through a communication network to an available processor. The processor performs the operation and returns the result through a second communication network to the proper destination arc in memory.

In a data flow machine there will typically be many processors available for the execution of instructions, each capable of independent asynchrousnous execution. Figure 2.1 represents a simple data flow machine element consisting of a single memory unit and multiple processors. The broken lines represent multiple lines, one for each additional processor in the unit. A unit such as this would be duplicated many times in a complete data flow machine with some communication logic to connect them.

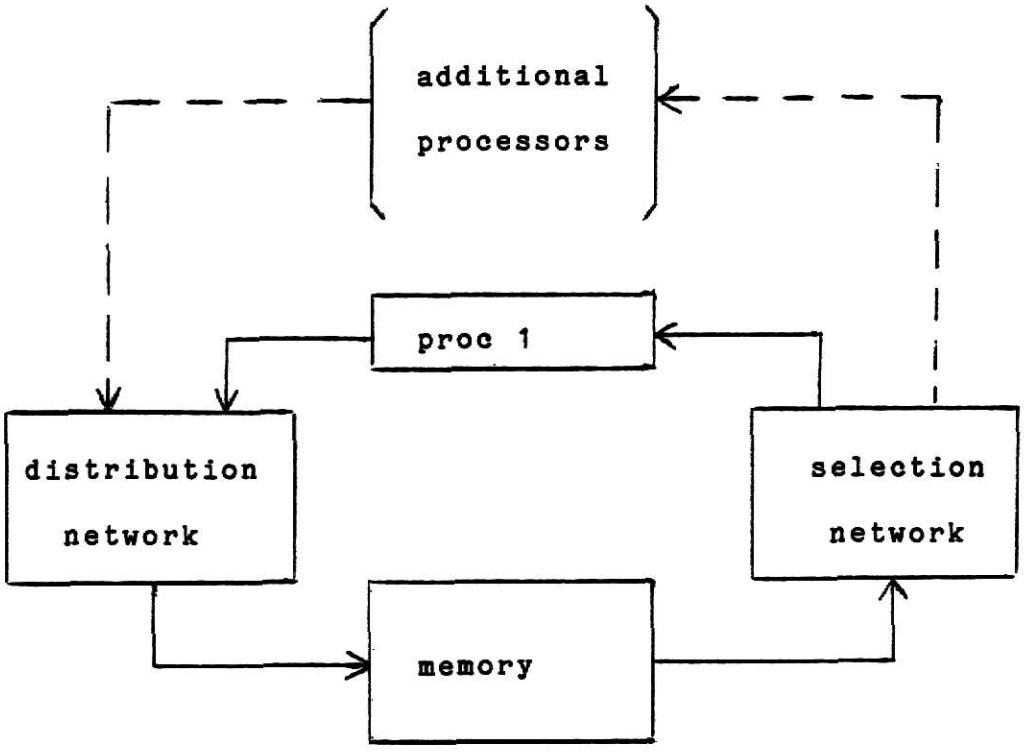


Figure 2.1 Structure of a generalized data flow unit.

Evaluating the data flow machine in terms of its computational organization, each instruction is selected at all times. This is true if we think of each instruction as logically having a processor allocated to it at all times. The examine phase of computation is the phase which actually initiates execution. In the data flow model, only those instructions which have received all of the required data pass the examine phase and become executable. That is, the examine phase is controlled by the data availability firing rule.

In evaluating the program organization, the data mechanisms are by literal values which are contained in the

instructions, or by values which are flowing on the arcs of the graph. There is no concept of a global storage, hence no accessing of data by reference. The control mechanism is parallel. There is no location counter and all sequencing constraints depend solely on the data availability firing rule of the examine phase.

The machine organization is usually packet communication with the instruction packets and data flowing in a circular system of asynchronous components. The Data Driven Machine (DDM1) developed at the University of Utah is an exception. It is built on an hierarchical tree organization and is classified by Treleaven,et al.(1) as an expression manipulation model.

In evaluating the data flow model against the criteria which has been set for the next generation machine, it becomes apparent that a very high degree of concurrency is inherent in the basic design. The lack of a location counter, and sequencing constraints which derive solely from the availability of data, allow all possibility for concurrency which is inherent in the program to be revealed automatically. The large number of processors and the asynchronous nature of the numerous components allows this concurrency to actually be achieved.

The potential for cost effective utilization of VLSI technology is also high. Due to the large number of identical units from which the machine is built.

2.5 THE REDUCTION MACHINE MODEL

The basic concept of the reduction machine is that the computation is viewed as a process of reduction of nested expressions. All instructions may be viewed as expressions in which the arguments are values or subexpressions. In the processing of an expression, if a subexpression is encountered, the resolution of the expression first requires resolution of the subexpression. Each of these subexpressions may be thought of as a function call which is resolved to a final value. The machine architecture has a large number of processors arranged as a hierarchical tree. Each processor may pass subexpressions to any of the processors which exist below it in the tree.

Two broad types of reduction are recognized, string reduction and graph reduction. In string reduction a copy of the expression is made and passed to a processor for resolution. When the subexpression has been resolved, the final value is returned to replace the subexpression in the original expression. In graph reduction, a program graph exists only once for each expression, and is shared by all expressions which require it. The final result is the resolution of the highest level expression in the program.

The computational organization of the reduction machine may be either of two organizational schemes which differ in the select and examine phase. Machines which are based on the innermost computational rule resolve the most deeply embedded expressions first. The innermost expressions, or

most deeply embedded, are those which have only values as their arguments. The resolved values are then passed to all expressions which require them. Machines using this computational rule are data driven. Machines using the outermost computational rule begin by attempting to resolve the outermost expression. As subexpressions are detected in the arguments, a demand is issued for resolution of the subexpression. Reduction machines using the outermost computational rule are also called demand driven machines.

In the machines using the innermost computational rule, the computational criteria is similar to that of data flow machines. All instructions may be viewed as selected, but only those which have received all of the required values from subexpressions pass the examine phase and become executable. In reduction machines using the outermost computational rule, only those which have been demanded are considered selected. Selection of the instruction may result in the issuing of more demands, but the examination phase will not allow execution until all subexpressions have been resolved.

In terms of the program execution, the data mechanisms may be by value or by reference. String reduction machines pass by value, and graph reduction machines use references to access the shared program graphs. The control mechanism is recursive in both types.

Comparing this architecture to the criteria for the next generation machine, it becomes apparent that the reduction machine, like the data flow machine, allows a high

degree of concurrency to be realized. Reduction machines, like data flow machines, are based upon a design which supports a large repetition of like components, and therefore have the possibility of cost effective utilization of VLSI technology.

3.0 DESIGN OF A GENERALIZED DATA FLOW MACHINE

The foundation for understanding a data flow machine is in understanding data flow graphs. A data flow machine is one which has been specifically designed to execute programs represented as data flow graphs. Understanding of the properties of these graphs reveals many of the advantages which data flow offers over traditional control flow machines. This section of the paper describes these graphs and their characteristics, and then presents a generalized description of the data flow hardware components.

This description of data flow graphs is primarily based upon the Data Driven Nets (DDN) of Davis(3,4).

3.1 DATA DEPENDENCY GRAPHS

A data flow graph, also called a data dependency graph, is a directed graph of nodes and arcs. Each of the nodes represents a basic operation to be performed, and the arcs represent the flow of data through the graph. Figure 3.1 represents a simple graph which computes ((a*b)+(c*d))/2. The initial values appear on the input arcs of the two multiply nodes. Each of these nodes performs its operation and then delivers the result to its output arc. The output arcs from the multiply nodes then supply the values a*b and c*d as input to the add operation. This node then performs its operation, and places the result on the arc as input to the divide node. Following the divide, the final result of the expression is placed on the output arc of the graph.

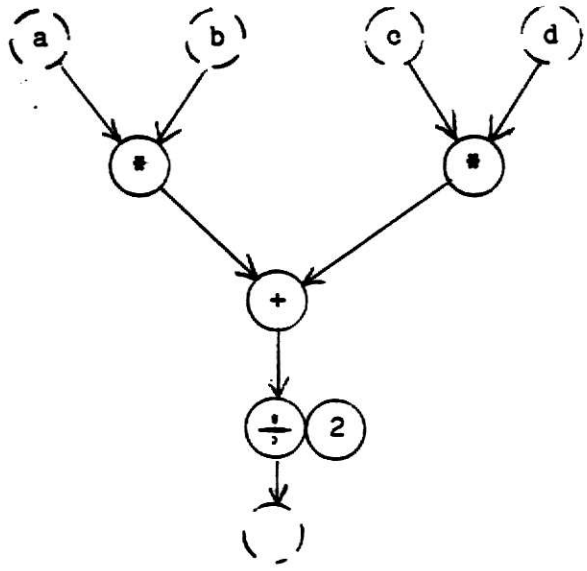


Figure 3.1 A simple Data Flow Graph

The flow of data through the graph is synchronized by the data availability firing rule. Each node automatically becomes enabled and performs its function when the data becomes available on the input arcs. Until all of the data required is present, the node remains unfireable and will continue to wait until its input data becomes available.

The data which is flowing through the graph is the actual computed result to that point in the graph. There is no concept of a global storage in a data flow machine and all data flowing on the arcs is passed by value from one operation to the next, and not by reference as is the usual method for a control machine. This characteristic of pass by value avoids the problems associated with the "von

Nuemann bottleneck", but presents other problems in the handling of arrays and other data structures. It is also one of the sources of problems in adapting programs written in conventional languages to data flow machines.

Literal values, such as the constant 2 in the divide node of figure 3.1, may be placed on one of the input arcs and marked as a constant when the program is compiled. They represent a data value which is permanantly available. The data availability rule applied to the divide node would require only input on the other arc to become fireable. After firing, the constant would not be removed from the arc and would be immediately available for the next execution.

Each of the arcs in the graph may be thought of as a queue of theoretically infinite length; the data ,therefore, flowing in a pipeline manner through the graph. Each node is the consumer of data on its input arcs and the producer of data on its output arcs. Although this view of an arc as a queue of values is sufficient for the explanation of data flow graphs, in actual implementaion it may not be true.

The pipeline nature of the arcs, combined with the data availability rule, reveals the potential for concurrency which is inherent in data flow machines. Referring again to figure 3.1, it becomes evident that if all the initial input is present, the two multiply nodes may execute concurrently. In a data flow machine with two or more processing units available at the time the two cells become fireable, this concurrency could actually be realized. The important point

to note is that the data flow graph automatically reveals the concurrency which is inherent in the algorithm being executed. This type of concurrency is called spatial, or horizontal concurrency.

A second type of concurrency, called temporal or vertical concurrency, is also present in the graph in figure 3.1. This occurs when multiple executions of the graph occur in succession. If the four initial input arcs each contains more than one data value, the top value on each arc is selected and the two multiply cells fire. Following placement of the result on the output arc, however; each multiply cell will remain fireable due to the presence of data on the input arcs. This will cause them to fire again, and this second execution can potentially occur concurrently with the add cell which is adding the results of the first execution.

It will be noted that in the graph there is no predetermined execution sequence other than the data availability rule, as would be the case in a von Nuemann machine. In the control flow machine, each of the two multiply nodes would have to be executed sequentially due to the basic nature of the machine architecture and its dependency upon the location counter to determine the next instruction to be executed. It is this property of execution based solely upon data availability which allows data flow machines to exploit concurrency at the lowest levels of the program, with no special hardware or software

required to detect the presence of an operation which could occur concurrently.

In the graph illustrated in figure 3.1, the nodes which are supplying the initial inputs and consuming the final output may be thought of as phantom nodes. Phantom nodes exist within the graph of the total program, but are outside the specific portion of the graph which is being represented. The graph in figure 3.1 could be represented in a higher level graph as a single node with four input arcs and one output arc, whose function is the expression which is being performed in figure 3.1.

All of the data items represented so far have been simple numeric values; however, this is not necessarily the case. An item of data flowing through the graph, referred to as a data token, could be a single value of type integer, real, character, boolean, etc. or could be a more complicated data structure such as an array or record or, at least in theory, an entire file. For simplicity, integer values are used in this paper to illustrate the concepts of data flow; however, it should be remembered that the actual unit is a data packet which may contain many individual data items flowing together.

Data flowing on an output arc may be delivered to as many destination arcs as required by the program logic. As a single node is usually restricted in the number of output arcs it contains, one of the simplest ways of achieving this distribution is to design into the machine a type of node whose function it is to receive a single input and pass it

unchanged to two or more output arcs. Multiple occurances of these distribution nodes may be used to distribute the data packet to as many arcs as required, without destroying the determinate nature of the flow of data through the graph.

This capability of data fanning out in the net does not mean, however, that data may be arbitrarily distributed to any arc in the graph. Each arc may receive its data only under controlled conditions, and arbitrary fan in is explicitly disallowed. To allow an input arc to receive data in an uncontrolled manner would destroy the determinancy of the graph, and the final result would vary depending upon the specific timings in the order of execution in previous nodes. There are instances in which data must be selected from multiple possible sources in the graph, but these must be carefully controlled by certain types of nodes.

In the example of the data flow graph illustrated so far, one glaring deficiency remains. All data is flowing from node to node and no capability for conditional processing exists. In fact, all of the program structures which support conditional flow can be implemented on data flow machines. This requires only the implementation of a few basic types of nodes which use booleans as one of their data values.

The first type of node is one in which a test is performed and a boolean is generated as the output token.

Nodes of this type typically will have two inputs, namely the two values to be compared, and a function which specifies the type of comparison (equal to, greater than,etc.) and a single output, true or false, which may then be distributed to as many nodes in the graph as required.

The second type of conditional nodes are those which consume a boolean as one of their inputs. Two of these will now be discussed briefly to illustrate their functions. The examples given here are actually an abbreviated version of the SELECT and DISTRIBUTE cells of the Data Driven Nets (DDN) of the Utah machine.

The SELECT node is used to conditionally select one of two input arcs and pass the data to a single output arc. A third input, which contains a boolean value, determines which of the two input arcs is selected. The data availability firing rule for the SELECT node states that a boolean value must be present at the control input, and data must be available at the input arc selected by the boolean value. Data items which are available at the arc which is not selected by the boolean value at the control input are ignored and remain waiting on the arc to be selected by another control input.

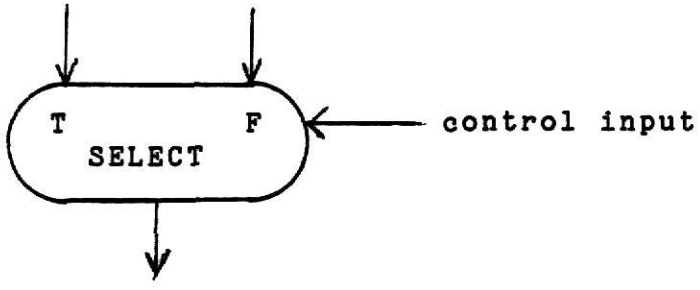


Figure 3.2 The SELECT Instruction Cell.

The second type of conditional node is the DISTRIBUTE node, in which a single input arc is distributed to one of two possible output arcs. Which output arc is selected depends upon the value of a boolean input on a control input arc. The data availability firing rule for a DISTRIBUTE node states that a boolean value must be present on the control input and a data item must be available on the input arc.

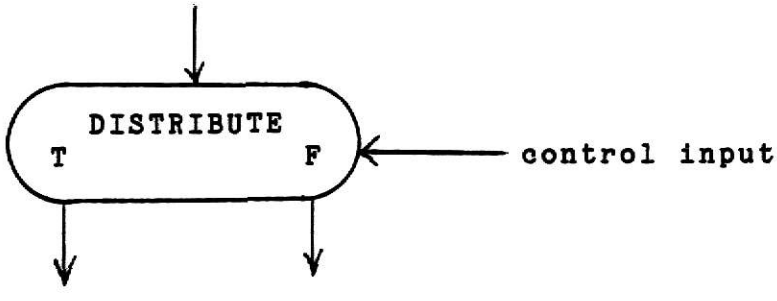


Figure 3.3 The DISTRIBUTE Instruction Cell.

These conditional nodes will now be used to present the basic graphic representation of alternation and iteration. As an example of alternation represented as a data flow graph the following if-then-else example will be used.

```
IF x > 3
   THEN  x - 1;
   ELSE  x + 1;
ENDIF
```

This example, though trivial, illustrates the basic concepts of this type of programming structure realized in data flow concepts. Figure 3.4 shows the data flow graph to execute this programming construct. For simplicity, fan out nodes are represented as branching arcs.

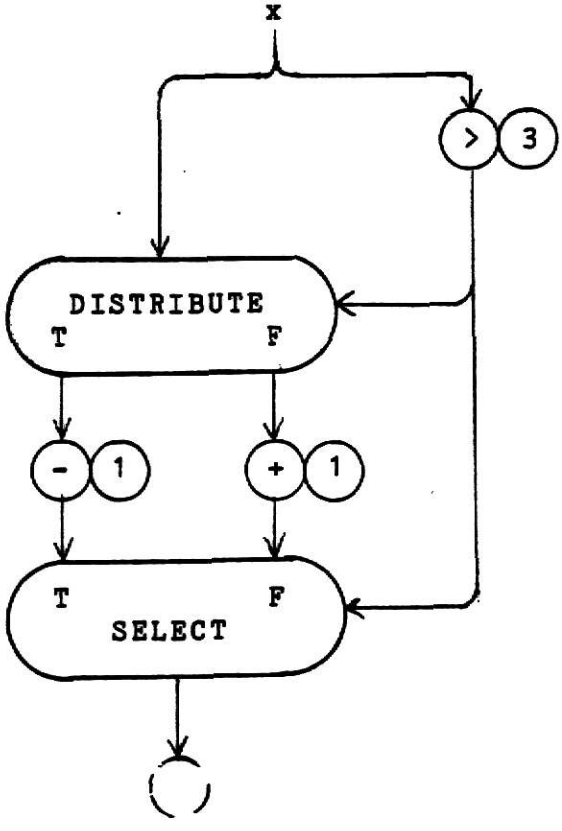


Figure 3.4 An example of alternation in data flow

The initial value of x enters from the phantom node at the top of the graph, and is distributed to the input to the DISTRIBUTE node and the test node. The test node then generates a boolean token which is delivered to both the DISTRIBUTE node at the point where control splits, and to the SELECT node where control converges. When the boolean value arrives at the DISTRIBUTE node, the presence of the control input and data at the selected data input will cause the node to execute. Depending upon the value of the controlling boolean, one of the two possible output arcs will be selected. The data item will be passed to the selected arc, and both the data item and control input will be dequeued from the DISTRIBUTE node input arcs.

Following the execution of the selected path through the graph, the result will arrive at the SELECT node. The boolean value which was delivered from the test node has been waiting at the control input of the SELECT node, and when the result appears at the proper input arc, the node executes, placing the computed result onto the single output arc flowing to the phantom node at the exit.

A close examination of this graph reveals that it is potentially capable of both the spatial and temporal concurrency described above. Multiple instances of x on the input arc may enter the graph, and due to the presence of the booleans controlling the SELECT node, the results will be retrieved from the two alternate paths in the same order in which they entered the graph. This is true even if one of the alternate paths is much faster and its result arrives

at the SELECT node before the result of another instance which entered the graph before it.

Iteration can be implemented using the SELECT and DISTRIBUTE nodes to conditionally control entry to, and exit from, a loop which is controlled by a generation of a new boolean with each iteration. The data flow graph in figure 3.5 performs the following example of a while loop.

```
WHILE    x > 5
    (x/2)+1;
END WHILE
```

Unlike the example for alternation, the initial control input to an iteration structure requires a boolean token which is present before the loop is accessed. In this example an initial false boolean token exists at the control input to the SELECT node. Initial entry to the graph occurs when x is placed on the false arc input of the SELECT. The presence of the false boolean control token, and the data item on the false input, satisfies the data availability rule of the SELECT node, and the incoming item is placed on the output arc of the node. At this time the boolean token is consumed, effectively blocking new input to the loop until a new false token appears at the control input of the SELECT node.

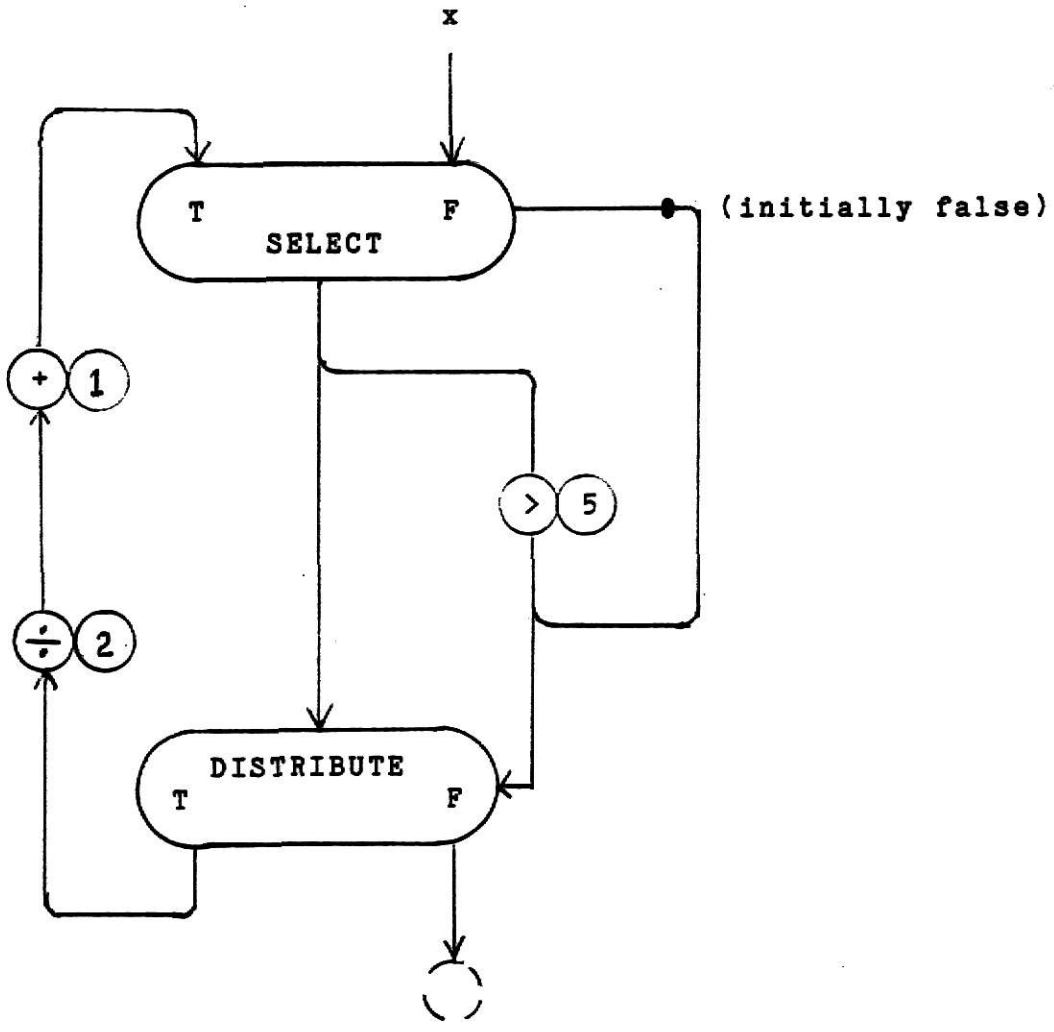


Figure 3.5 An example of iteration in data flow

Next, the output of the SELECT node delivers the data item to a test node which determines if the exit condition has been satisfied. The data item is also delivered to a DISTRIBUTE node, which will pass the data token to the body of the loop, or to the final exit. The boolean value

arriving from the test node completes the data availability rule for the DISTRIBUTE node, and the data is passed to the body of the loop or to the final exit.

If the test is true, the data item is delivered to the body of the loop, and the true token delivered to the SELECT node allows the result of the loop body to reenter the test logic for the next iteration. If the test is false, the data item is delivered to the final output arc, and the false boolean is returned to the SELECT node, thus reenabling the initial input path for the next data item to arrive from the phantom input node.

It should be noted that the iteration structure does not allow temporal concurrency within the loop. If new data items arrive from the phantom node of a busy loop, they will remain on the false input arc of the SELECT node until the boolean false token is generated at the time of exit.

Other special nodes can be designed to control the data flow counterpart of a subroutine call. In this instance, the simplest implementation is that using a return address which is passed as data at the time of the call. The identity of the output arc is appended to the data item and passed to the called routine. The called routine will terminate at a return node, which places its output value on the arc specified by the data packet. The data specifying the return arc is removed from the data packet during execution of the return node.

It must be remembered, that although simple numeric values have been used in these examples to illustrate the

basic points of data flow graphs, data packets may consist of a stream of separate tokens passing through the graph as a single data packet. Certain types of nodes can be designed to concatenate two data items into a single output data packet, and others can be designed to separate the packets into its components.

3.2 MAJOR HARDWARE COMPONENTS

A data flow machine is simply one which has been specifically designed to execute programs which are represented as data flow graphs. In this section of the paper a generalized description of the hardware components of a data flow machine will be presented. The purpose of the section is to provide a basic understanding of the architecture before presenting the specific architectures in chapter 5.

A typical data flow machine will consist of a number of asynchronous elements, connected by communication logic, which combine to form an elemental processing unit. Frequently the elements will be arranged in a ring with the information packets passed in a circular manner from element to element. In most data flow machines, a number of these basic processing units will be interconnected by another layer of communication logic. This allows for easy extention of the machine, as well as providing additional capability for concurrency. At a minimum, each basic processing unit will consist of at least one storage unit, a

set of processing elements, and the communication logic to connect them. Certain architectures also contain a separate control unit.

3.2.1 STORAGE UNITS

As in traditional machines, data flow machines need storage in which to place the program graph and the data which is currently flowing along the program arcs. The memory units in these machines, however, will usually contain additional functions not traditionally associated with memory. It is in the storage devices that the presence of fireable instructions is actually detected, and it is in the storage units that data values and node functions are combined to create an execution packet to be sent to the processing element. In those data flow implementations which allow multiple data items to reside on the same arc at the same time, some method of queue management is also required. The fact that memory units are now being built using the same basic logic as processors is one of the technological advances which have made data flow machines feasable.

There are basically two approaches which may be taken in designing the storage units; both data and the program graph may be stored in a common storage unit, or a separate unit may be used for the graph and the data. If the two are stored in the same unit, the problem of detecting fireable instructions is simplified; however, the unit has to be

capable of storing two separate types of information. On the other hand, if the data and the graph are stored in separate units, each unit has to maintain only one type of information; however, the information required for determining a fireable node must be passed with the data, or some standard set of firing rules must be supplied. Both types of storage design are presented in the specific implementations described in chapter 5.

3.2.2 INSTRUCTION CELLS

The physical representation of the data flow graph exists in the storage unit as a set of instruction cells, each corresponding to a single node in the graph. Each cell will typically contain the following information.

1. A function code indicating the specific function which the processor is to perform.
2. A firing set which indicates what inputs are required for the cell to execute.
3. Input values which may have been inserted into the cell, or which may be merged with the cell information to form an execution packet.
4. A destination address specifying at least one program arc where the result packet is to be placed.

There are certain basic functions which must be supported to implement a data flow machine. Although the specific design may vary from machine to machine, the

following functions will usually be present.

1. Operators - Instruction cells which accept at least two input values, with one possibly a literal, and which perform an arithmetic or logical operation.
2. Comparison - Instruction cells which accept at least two inputs, one of which may be a literal, performs a comparison, and produces at least one copy of a boolean as output.
3. Conditionals - Instruction cells which accept a boolean value as one of the inputs, and which conditionally accepts input or produce output.
4. Synchronization - In certain implementations, cells may be supplied to synchronize the flow of multiple data tokens which are logically related.
5. Duplication and Merge - Instruction cells which are used to create additional instances of data items, or which collect a data item from a set of possible input arcs.

3.2.3 PROCESSING UNITS

The individual processors receive the instruction packets of fireable instruction cells, perform the function and generate the output data packets to be returned to the destination program arcs in the data storage unit. Due to

the slow speed of a single processor relative to the other components, and in order to fully realize the potential concurrency in the program, multiple processors are usually contained within a processing unit.

In certain data flow machines, some of the functions are not assigned to all of the processors. These machines are called heterogenous machines, and some routing mechanism must be present to ensure that each instruction packet is distributed to a processor which is capable of performing the required function. In other data flow implementations, all of the processors are identical, and the distribution of instruction packets to processors in based solely on processor availability. These implementations are called homogenous machines.

3.2.4 COMMUNICATION UNITS

Due to the distributed nature of data flow machines, the communication units are a signiicant part of the machine architecture. These components are usually packet switching units which link together the various components of the machine. Communication elements will exist in the basic processing element to route instruction packets from the storage units to the processing units, and to route result data packets back to the data storage units. In those designs which combine multiple basic processing units to form the complete machine, additional communication logic is required to support transmission between the basic

processing elements.

One method of designing a communication unit with multiple input paths and multiple output paths, is to design the unit based on a number of 2 x 2 routers. A 2 x 2 router has two input paths and two output paths. Packets will be accepted from either input, and based on a single bit in the destination address of the packet, the packet will be passed to one of the two output paths. Figure 3.6 illustrates a 2 x 2 router.

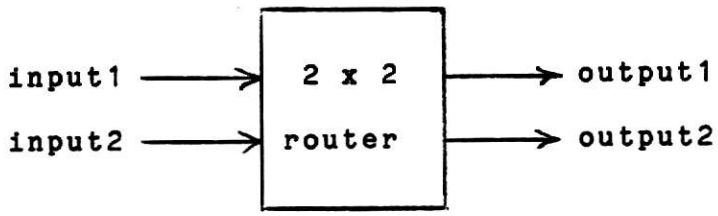


Figure 3.6 A 2 x 2 Router

In an actual routing network, a number of 2 x 2 routers are combined to form more complex units. Each 2 x 2 router is an asynchronous unit; therefore, the unit is capable of passing a number of messages concurrently. Figure 3.7 illustrates a 4 x 4 router built of four 2 x 2 routers. In this router 2 bits would be required in the destination address to determine the complete routing.

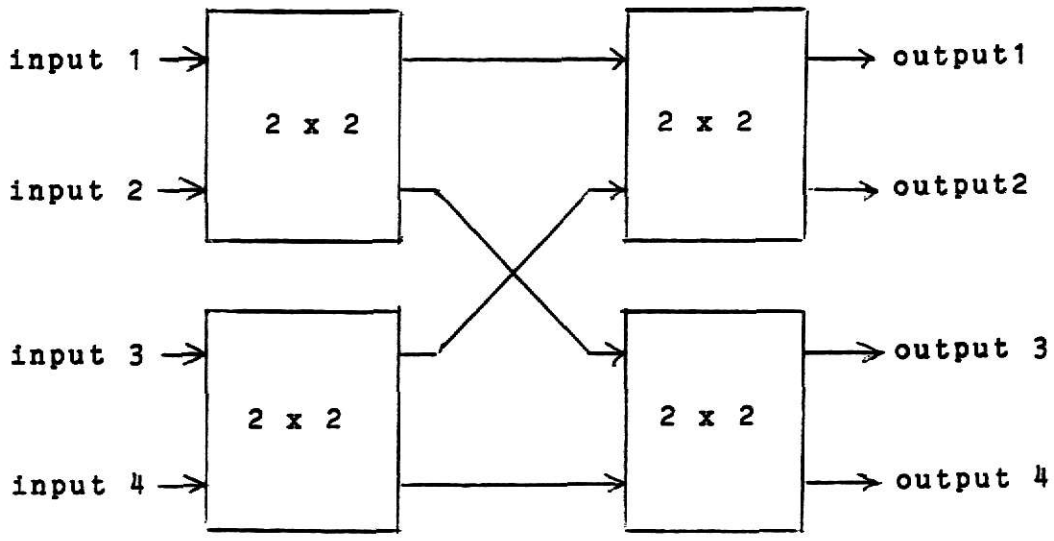


Figure 3.7 A 4 x 4 Router

3.2.5 CONTROL UNITS

Certain data flow machines also contain a separate control unit. Depending upon the design of the machine, this unit may not be required. In machines which do not have a control unit, the control information is passed in the same mannar as normal data tokens. In those machines which do have a control unit, it may be used to pass boolean values, or status information regarding the state of program arcs.

4. DATA FLOW PROGRAMMING LANGUAGES

It should be evident from the preceding description of data flow machines, that considerable effort is also required in the development of languages for programming the machines. It is not intended that anyone program at the level of the data flow graph, and each of the data flow projects in progress has an associated higher level language.

Most of the conventional languages which have been developed for the von Neumann machines could be adapted for data flow machines only with great difficulty, and most of the programs developed for von Neumann machines would have to be completely rewritten. This section of the paper examines the properties of languages and identifies the types of languages which are compatible with data flow.

The problems with conventional languages arise primarily from the two basic hardware features of a control flow machine; namely, the global storage of data and the sequential nature of program execution. The control flow machine exerts its influence by altering the contents of the global storage, and in a data flow machine the global storage is not present. Although certain optimizing compilers have been written which seek out data dependencies, primarily to optimize register usage; the application of this technique is not feasible to adapt current control flow programs to data flow.

Certain programming constructs, such as the FORK-JOIN, have also been used to in many conventional languages in order to implement concurrency; however, the burden for recognizing the possibility for concurrency, and implementing it, is the responsibility of the programmer instead of being a natural property of the language. What is needed is a language which has been specifically designed to support data flow concepts.

Ackerman(5) has identified six language properties which are essential to the development of data flow languages. Many of these properties have already been incorporated in certain conventional machine languages, although for reasons other than supporting data flow. The six language properties are:

1) Freedom from side effects.
2) Locality of effect.
3) Equivalence of instruction scheduling constraints and data dependencies.
4) Single assignment convention.
5) A special convention for iterations.
6) Lack of history sensitivity.

Side effects, as previously noted, arise from the use of global storage in the conventional machines. Data flow machines, which lack the global storage, are by their basic nature free of side effects. Languages which are suitable for data flow must reflect this difference in architecture,

and exert their effect by direct manipulation of values, rather that by the manipulation of global data.

Locality of effect means to restrict the scope of a value to a localized portion of the program. This is accomplished to some extent by local variables within subroutines of conventional languages such as PASCAL or C.

The equivalence of instruction scheduling with data dependencies requires that the execution order must be dependent solely upon the data flow and availability of data. The lexical ordering of program statements should impose no restrictions, other than those imposed by the data dependencies. This will automatically be achieved if both freedom from side effects and locality of effect are present.

The single assignment rule states that a variable will be assigned a value only once in a program context. In a single assignment language, an expression such as I = I +1 becomes meaningless, as it is assigning a value to I in a context which presupposes I. The only program construct in which the single assignment rule causes problems is iteration. This is solved by having two variables ("I and NEW I"). Each iteration is viewed as a separate program context, with "NEW I" automatically becoming "I" at the start of each iteration.

The lack of history sensitivity means simply that the order of prior instruction execution has no effect on current execution. The presence of history sensitivity implies some type of global storage device.

There are two groups of languages, both developed before data flow machines became feasible, which contain the necessary properties for data flow languages.

The first are functional, also called applicative, languages. These are languages which exert their effect solely by the application of functions to values. Pure LISP, a subset of LISP, is a functional language. By applying their functions solely to values, they break the binding between data and global variables which are accessed by reference.

The second type of language is the nonprocedural language. A nonprocedural language is one in which the execution order is not strictly dependent upon the lexical ordering of the program text. Nonprocedural languages break the binding between the language and the traditional concept of instruction flow determined by a location counter. Data flow languages, due to the basic nature of the machine on which they will execute, are naturally compatible with languages that are both functional and nonprocedural.

5. SURVEY OF THREE DATA FLOW PROJECTS

In this section of the paper, three development projects on data flow machines are presented. Two different designs of data flow graphs are included (the DDF of the M.I.T. and Manchester machines and the DDN of the Utah machine). Both the ring configuration and the hierarchial configuration of basic hardware components are represented.

5.1 THE MIT PROJECT

The data flow project at M.I.T was the earliest project in the implementation of data flow machines. Started in the early 1970's by Dennis and Misunas, this work has resulted in a fairly large effort at M.I.T, as well as serving as the initial stimulus for a number of other efforts. This description of the M.I.T project is taken from the original paper of Dennis and Misunas(6).

5.1.1 THE DATA FLOW GRAPH

The machine language representation of the program graph consists of Dennis Data Flow(DDF) nets, a directed graph scheme developed by Dr.Jack Dennisat M.I.T. The University of Manchester machine (section 5.2) also uses a machine representation based of DDF. The high level language VAL was developed at M.I.T to support program development.

The graph is composed of nodes, which may be either actors or links, and arcs along which the tokens flow from

node to node. There are two separate types of links, data links and control links. The data links carry data items and the control links carry boolean values. A link may also serve as a copy site, generating multiple instances of a data value.

There are six basic types of actors in the DDF nets.

1) Operators - nodes which accept two data input tokens and perform an operation such as addition or subtraction. The result is then placed on a data link.

2) Deciders - nodes which accept two data tokens as input and then perform a test, delivering a control token to a control link.

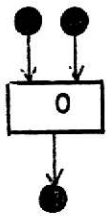
3) T-Gate - a node which accepts a single data token and a control token. If the control token is true the data token is passed to the output data link. If the control token is false, the data token is removed and no output token is produced.

4) F-Gate - a node which accepts a single data token and a control token. If the control token is false the data token is passed to the output data link. If the control token is true, the data token is removed and no output token is produced.

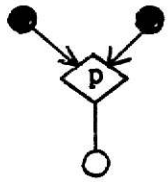
5) Merge - a node which accepts a control token and selects a data token from one of two possible inputs. A token which may be present at the input which was not selected is not affected.

6) Boolean Operator - a node which accepts two control inputs, performs a boolean function, and then places a control token on a output arc.

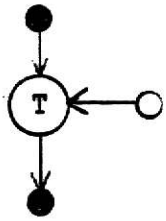
Figure 5.1 shows the actors of the M.I.T graph. The solid arcs represent data links, and the other arcs represent control links.



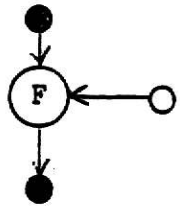
a) operator



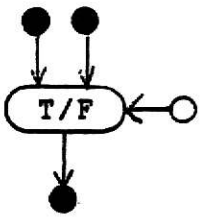
b) decider



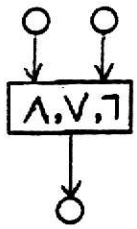
c) T-gate



d) F-gate



e) merge



f) boolean operator

Figure 5.1 The actors of the M.I.T graphs.

5.1.2 THE M.I.T MACHINE

The machine architecture of the M.I.T machine is a ring composed of five separate subsystems. The storage Unit is used to store both the data and the program graph. It is composed of instruction cells, each of which contains 3 input registers to accept input values from the links. Due to this design feature of storing data within the actual instruction cell, rather that in a separate storage unit, the machine allows only one value to be on a link at one time. The firing sets only allow a cell to become fireable if all required input values have arrived and the output link is available. Figure 5.2 shows the internal structure of an instruction cell.

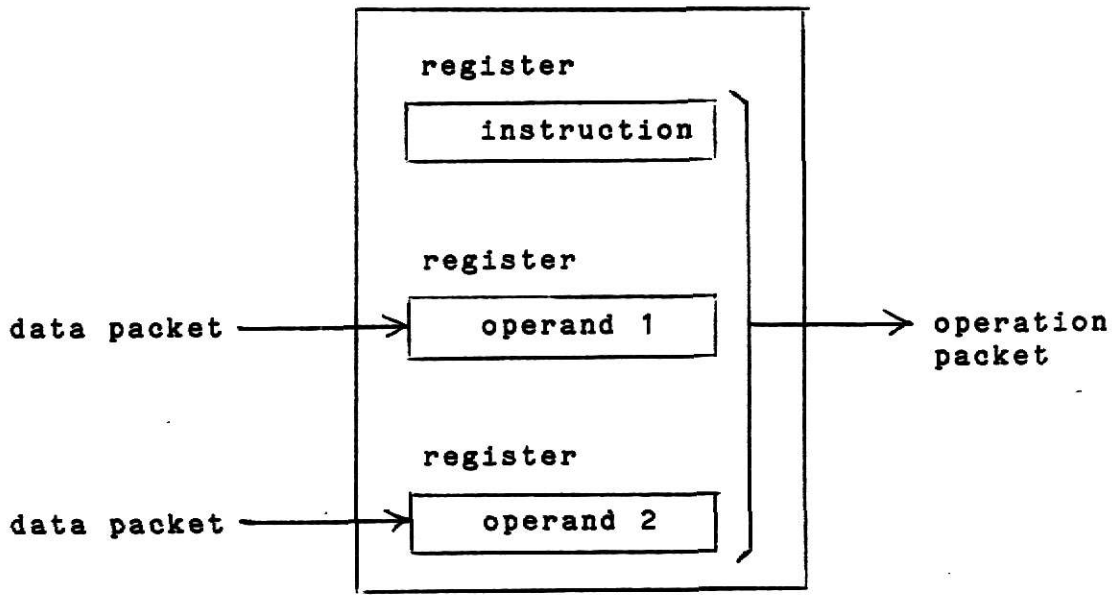


Figure 5.2 The M.I.T instruction Cell.

In the instruction cell, the first register will always contain the instruction and single destination. The second register may either accept an input token or it may be used to specify a second destination. Each register contains a code which identifies the format in which the register is being used. Associated with each input register there may be additional control information. The gating code in the register defines whether a control token is to be used, or to mark an input value as a literal value. If a control token is to be used, a control code within the register will indicate whether or not the control has been received, and the value of one which has been received.

In addition to the storage unit, the ring contains an array of heterogenous processors and three communication units. The arbitration unit routes instruction packets from the storage unit to an available processor which is capable of performing the required function. The distribution unit accepts the data packets from the processing unit and routes them back to the storage unit. The control unit routes the control packets from the processing unit to the storage unit. Figure 5.3 shows the structure of the M.I.T ring.

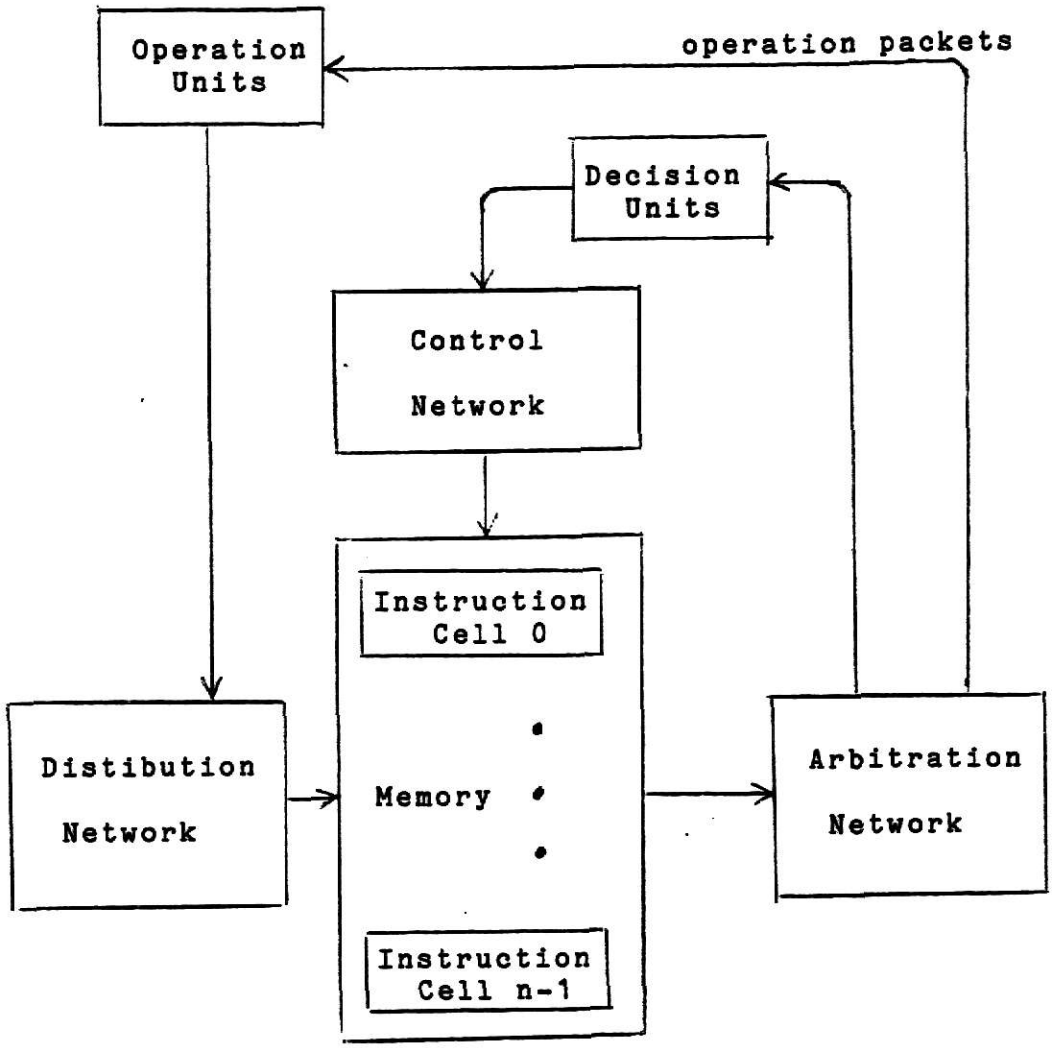


Figure 5.3 The M.I.T Ring

5.2 THE UNIVERSITY OF MANCHESTER

The data flow machine project at the University of Manchester received its early stimulus from the work of Dennis and Misunas at M.I.T. This project, one of a number of data flow projects in England, is supported by the British Science Research Council. The main researchers are Dr. Ian Watson and Dr. John Gurd. The description in this section is based upon their publications(7,8,9).

The high level langauge LAPSE was developed for writing the data flow programs. It is a high level language which generates object derived from the Dennis Data Flow(DDF) Nets. The syntax of the language is patterned after Pascal, and contains a Pascal type of procedure call.

The basic data types supported by the machine are boolean, integer, and floating point numbers. These basic data types flow as data tokens in the machine and may be combined to form records and arrays.

5.2.1 THE DATA FLOW GRAPH

The basic instruction cells which the machine supports are derived from DDF, and like the M.I.T machine there are six basic types. Each may have two input arcs, and may generate one or two output tokens. The six types of instruction cells are as follows.

1. Operators which perform operations such as addition, subtraction,etc.
2. Comparitors which compare two input values and

generate a boolean value as output.

3. Merge - which passes unchanged a single token received from one of two input arcs.
4. Pass on True - which passes a token when a boolean true token is the second input.
5. Pass on False - which passes a token when a boolean false is the second input.
6. Duplicate - which accepts a single input token and generates two identical output tokens. Repetitive use of this function may be used to create as many copies of a token as needed.

In addition to the six primitive functions listed above, LAPSE is also supported by compound functions. A compound function is actually a data flow graph which is built of the primitive functions. All of the language features of LAPSE are supported by the primitive or compound functions.

The procedure calls in LAPSE may accept many inputs, but will generate only one output token. This output token, however, may be a record; therefore, a mechanism exists for multiple data items to be returned from a procedure. Unlike the M.I.T machine, a new copy of the procedure is not created for each invocation. The procedure actually consists of three sections, only the first of which is duplicated. The first section performs the call, and each invocation creates a distinct label and return destination arc. The actual procedure exists as a single copy, the

unique labels assigned during the call ensuring the correct processing of each call. The final section performs the return. This consists of creating a destination address from the return address which was passed with the token at call time. This method is called Dynamic Tagged Data Flow.

5.2.2 THE MANCHESTER MACHINE

The basic hardware unit of the Manchester machine is a ring structure of five asynchronous components. Data token packets and instruction token packets flow in a circular pipeline through the ring. There is no control unit in the machine and the flow of control information and data tokens is identical. One of the components, the Input/Output switch, is capable of accepting tokens from outside of the ring and sending tokens to destinations outside of the ring. Figure 5.4 shows the structure of the Manchester ring. Distinct units are used for storing data tokens (token queue and matching store) and the program graph (node store).

The basic clock beat of the ring is 200 nsecs. As the actual execution time of a single processor averages around 4.5 microseconds, a parallel array of 15 processors is provided to maintain the basic ring rate of 200 nsecs. The machine is homogenous, with each individual processor capable of performing all functions. The ring will now be discussed in detail starting at the Input/Output switch.

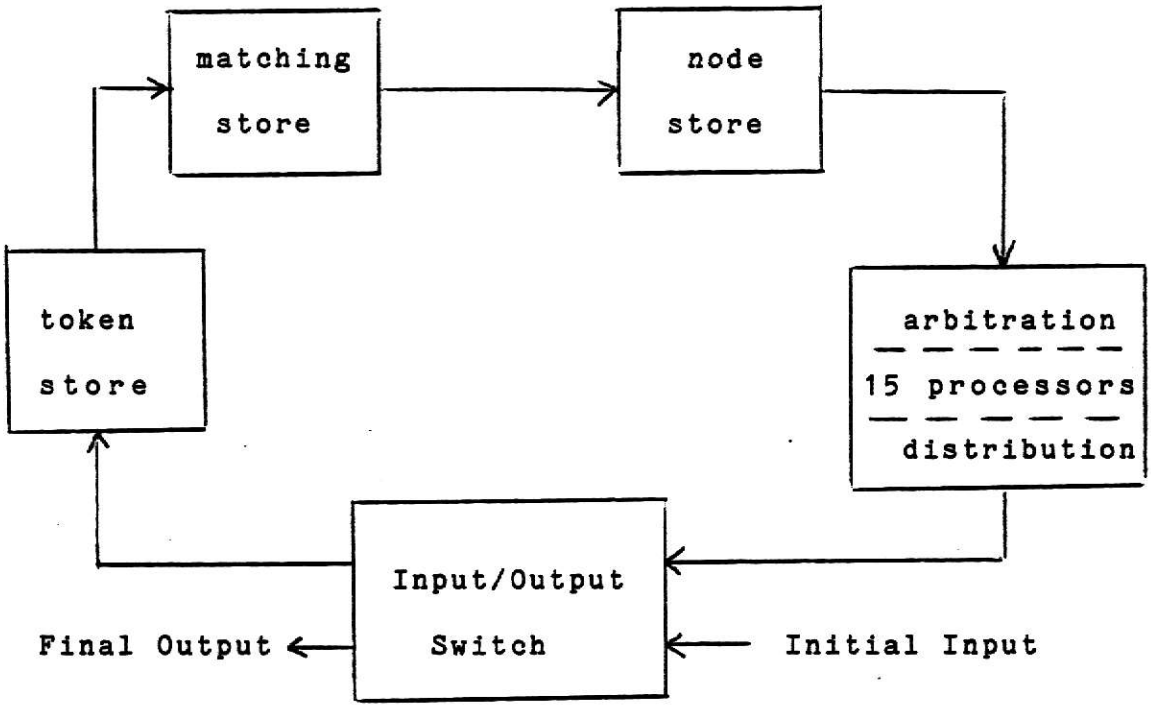


Figure 5.4 The Manchester Data Flow Ring

In the basic single ring configuration, the I/O switch is connected to a DEC LSI-11. This is used to load initial input into the ring and to accept the final output from the ring. The basic token packet which is generated by the processing unit is a 96 bit packet. The DEC LSI-11 is a 16 bit processor. Within the I/O switch at the point of initial input there is a 96 bit buffer and logic to convert six 16 bit inputs to a single 96 bit token packet. Similar logic exists at the final output to convert the 96 bit token back into 16 bit words.

The I/O switch will select a 96 bit token packet

arriving either from the processing unit or from the initial input buffer and pass it to the token queue. Tokens arriving from the processing unit have priority over those in the input buffer. The unit contains buffers and asynchronous input/output logic which allows it to receive and send at the same time.

The token queue is primarily a rate leveling mechanism to store data tokens which are waiting to be selected by the matching store. It consists of 16k, 96 bit words plus parity. The memory is managed as a circular FIFO queue with hardware pointers. When the matching store is available it selects the head item on the queue of the token store.

The matching store is the unit where data tokens destined for the same node are paired together. When the matching store detects a match, it builds a token packet of 133 bits containing both values. This packet is then sent to the node store. If a token arrives from the token queue and cannot be matched, it is placed in the matching store memory to wait for the matching token. Matching must occur on both the label and the destination fields.

The eventual design of the matching store is to use an associative memory in order to achieve rapid matching; however, due to cost considerations, the current implementation uses traditional memory units with a hardware hashing technique based on the label and destination fields. There is also an overflow storage area to handle tokens in the event of a hashing collision.

The matching process described above is actually one of eight matching criteria which may be used. Three control bits in the token packet are used to determine which matching criteria to use. If a token is the sole token input to a node, this is determined from the control bits and the token is immediately passed to the node store without attempting a match. The other matching criteria differ primarily in what action is taken if a match fails to occur, or in what special handling may be required by one of the tokens.

The node store receives token packets from the matching store, uses the destination address to obtain the instruction cell, and sends a 167 bit instruction packet to the processing unit. The node contains the program graph. The word length is 35 bits; each word containing 3 control bits and 32 data bits. Each node in the graph is represented by a one or two word instruction cell. The first word always contains the node function and a single destination address for the result packet. The second word, if present, may contain either a literal value, or a second destination address.

The processing unit recieves the 167 bit instruction packet, selects one of the 15 processors in the unit, performs the requested node function, and delivers one or two 96 bit result packets to the I/O switch. All of the individual processors in the processing unit are capable of all node functions, the distribution of the instruction packet being determined solely by processor availability.

The complete Manchester machine is a multilayered machine consisting of many of the basic Manchester rings described above. Each of these rings is independent of the others and additional concurrency is obtained by distributing the program graph across the total machine. Each ring may receive token packets from other rings through the initial input of the I/O switch, and send token packets to other rings through the final output of the I/O switch.

The rings are connected by a layer of communication logic which routes the packets from ring to ring. The machine may be extended by adding more rings, as long as the communication time between rings does not become prohibitively long.

5.3 THE UNIVERSITY OF UTAH

The Data Driven Machine #1 (DDM1) was originally developed at the Burroughs Corporation. The first system became operational in July of 1976. In September of 1977 the project was moved to the University of Utah where it continues under a grant from the Burroughs Corporation. The primary researcher, Dr. Alan Davis, has been involved with the project since its beginning. This overview of DDM1 is based upon the papers of Davis(4,10).

5.3.1 THE DATA FLOW GRAPH

The machine language of DDM1 is the Data Driven Nets (DDN's). It is not intended that anyone would program directly in DDN, and a high level Graphical Programming Language (GPL) is being developed which translates into DDN's.

The DDN's are data driven graphs composed of cells (nodes) and data paths (arcs). In these nets there is no distinction made between control information and the data items. All data paths are FIFO queues of variable but finite length, and a packet communication with request/acknowledge is used for data flowing on the data paths. The cells may contain any number of inputs (unless restricted by cell type) and may deliver the output to any number of output arcs.

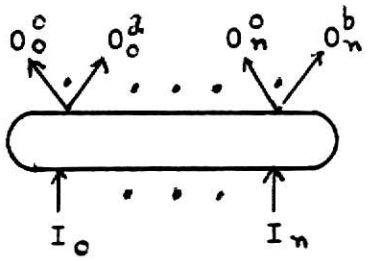
There are seven types of cells in DDN which were chosen

for their simplicity and generality. Four of these cells have conjunctive firing sets in which data must be present on all input arcs for the cell to become fireable. The other three have disjunctive firing sets which require only a subset of all possible arcs to become fireable. One of the cell types (the GATE) is dependent upon an internal state to determine the function and firing set.

In the following diagrams used to describe the cells, each of the cell types is represented by a unique graphical symbol, and the following notational conventions are used:

- Each type of data path is named: I for input, O for output, F for feedback, C for condition, and X for index.
- Subscripts indicate data paths which may recieve different valued tokens.
- Superscripts indicate data paths which will carry identical valued copies of output data items.

THE SYNCH CELL



Inputs: I_j

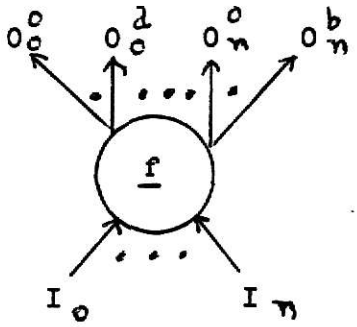
Outputs: O_j

Firing Set: $(I_o,\ldots,I_n)$

Cell Function: for every i,j: $O_j^i := I_j$.

In the SYNCH cell, there are a variable number of input arcs, and data must be present on all arcs to make the cell fireable. The cell function is to pass the data item from each input arc to one or more associated output arcs associated with the input arc. It is used to synchronize the flow of data arriving at different times.

THE OPERATOR CELL



Inputs: I_j

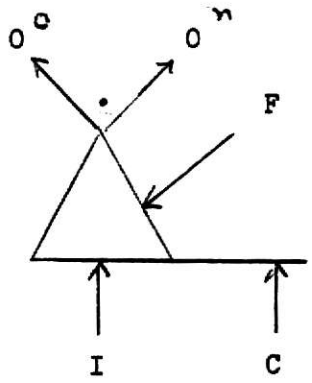
Outputs: O_j

Firing Set: $(I_0,\ldots,I_n)$

Cell Function: for every i,j: $O_j^i := \underline{f}(I_0,..,I_n)$

The OPERATOR cell is a generalized cell which performs a number of functions. A variable number of input arcs are present and all must contain data for the cell to fire. When the cell fires, the function (f) is applied to the input and one or more output arcs are delivered to one or more destination arcs. The functions performed by the operator cell include arithmetic functions, boolean logical functions, relational tests, merging and decomposing data items, and indexing into composite data structures.

THE GATE CELL



Initial Input: I

Feedback: F

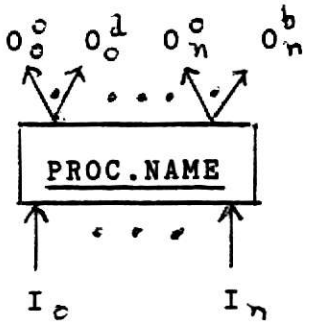
Condition: C

Output: O^i

Firing Set and cell function described below.

The GATE cell contains an internal state table. When it is waiting for initial input the firing set is I. This allows the data to pass and the cell enters an active state. In the active state, a false condition requires a feedback input and another iteration is started. A true condition causes the cell to return to the initial state. It is used, in conjunction with the OPERATOR and DISTRIBUTE cells to control iteration.

THE CALL CELL



Inputs: I_i

Outputs: O_j^k

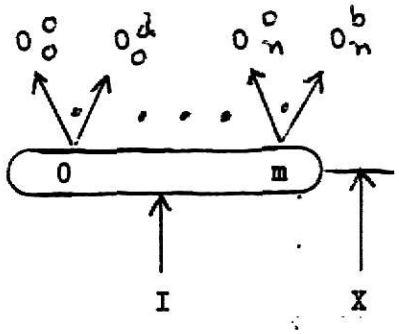
Firing Set: $(I_0,\ldots,I_n)$

Cell function: for every a,b:

O_b^a:=PROC. NAME$(I_c,\ldots,I_n)$

The CALL cell is used for subroutine calls. All of the input arcs must contain data to make the cell fireable. As the cell fires, the input data is passed to the named procedure.

THE DISTRIBUTE CELL



Inputs: I

Outputs: O_j^i

Index: X

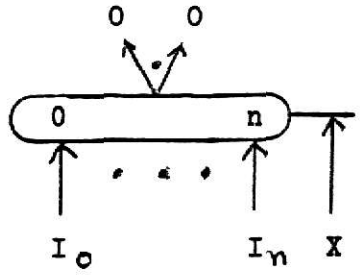
Firing Set: (I,X)

Cell Function: $O_x^i := I$

for all i and where x is the value of X.

The DISTRIBUTE cell contains a single data input and an index. The data item selects one or more output arcs which are associated with the index value. The simple boolean T/F condition presented in chapter 3 may be created by configuring the cell with two output groups and an index with only two possible values.

THE SELECT CELL



Inputs: I_j

Outputs: O^i

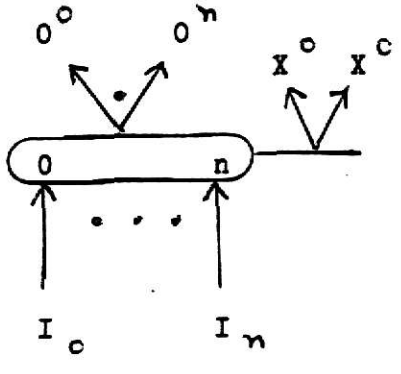
Index: X

Firing set: (I_x, X)
where x us the value of X.

Function: $O^i := I_x$ for all i

The SELECT cell will have multiple data input arcs and an index. The firing set is satisfied when input is available on the arc specified by the index. The data is then passed to one or more output arcs. Like the DISTRIBUTE cell, a simple boolean condition may be configured using two input arcs and an index of only two possible values.

THE ARBITER CELL



Inputs: I_j

Outputs: O^i

Index output: X^k

Firing set: At least
one input: I_j.

Cell function: O^i := first I ;

$x^a := j$ for all i,a.

The ARBITER cell controls entry to a net on a first come first served basis. Any input satisfies the firing rule and the input passes to enter the net. An index is generated and passed into the net to allow exiting along selected arcs. This cell function must be used in conjunction with the DISTRIBUTE cell.

The internal machine representation of the cells in DDN's is a variable length character string. The number of input arcs and the number of output arcs is variable with the specific cell unless the cell type places a restriction. The basic structure of a DDN is best defined by the following BNF description(8).

```
<DDN>::=(<NET NAME>(<CELL LIST>))
<CELL LIST>::=<CELL>|<CELL><CELL LIST>
<CELL>::=(<CELL TYPE>(<INPUT SLOT><OUTPUT LIST>))
<INPUT SLOT>::=(<INPUT MAP>(<INPUT LIST>))
<INPUT LIST>::=<INPUT>|<INPUT><INPUT LIST>
<OUTPUT LIST>::=<OUTPUT>|<OUTPUT><OUTPUT LIST>
<OUTPUT>::=(<DESTINATION LIST>)
<DESTINATION LIST>::=<DESTINATION>|
                     <DESTINATION><DESTINATION LIST>
<DESTINATION>::=(<CELL LOCATION>)
```

where <CELL LOCATION>,<INPUT>,<INPUT MAP>,<NET NAME>, AND <CELL TYPE> are all variable length nested strings.

The <INPUT MAP> contains one character for each input arc of the cell and contains status information regarding the data on the arc.

5.3.2 THE STRUCTURE OF THE DDM1 MACHINE

The basic unit of the DDM1 is the Processor Store Element (PSE). The PSE's in the system are organized in a hierarchical tree structure with each capable of accessing up to 8 PSE's at the next lower level. The total system is a totally distributed system is which the PSE's at one level are identical to those at any other level, except that those at a higher level will contain a larger storage capacity.

All communication in the system is asynchronous using a standard four phase request/acknowledge protocol. The packets flowing on the communication lines are variable length character strings. The link connecting a PSE to its father PSE contains separate queues for incoming character strings and outgoing character strings, thus allowing for concurrent transfer of the two strings.

Internally each PSE is composed of an atomic processor(AP),an atomic storage unit(ASU), the input and output queues connecting the PSE to its father PSE, and a 1 x 8 switch which allows the PSE to communicate with its son PSE's. Figure 5.5 shows the form of a PSE.

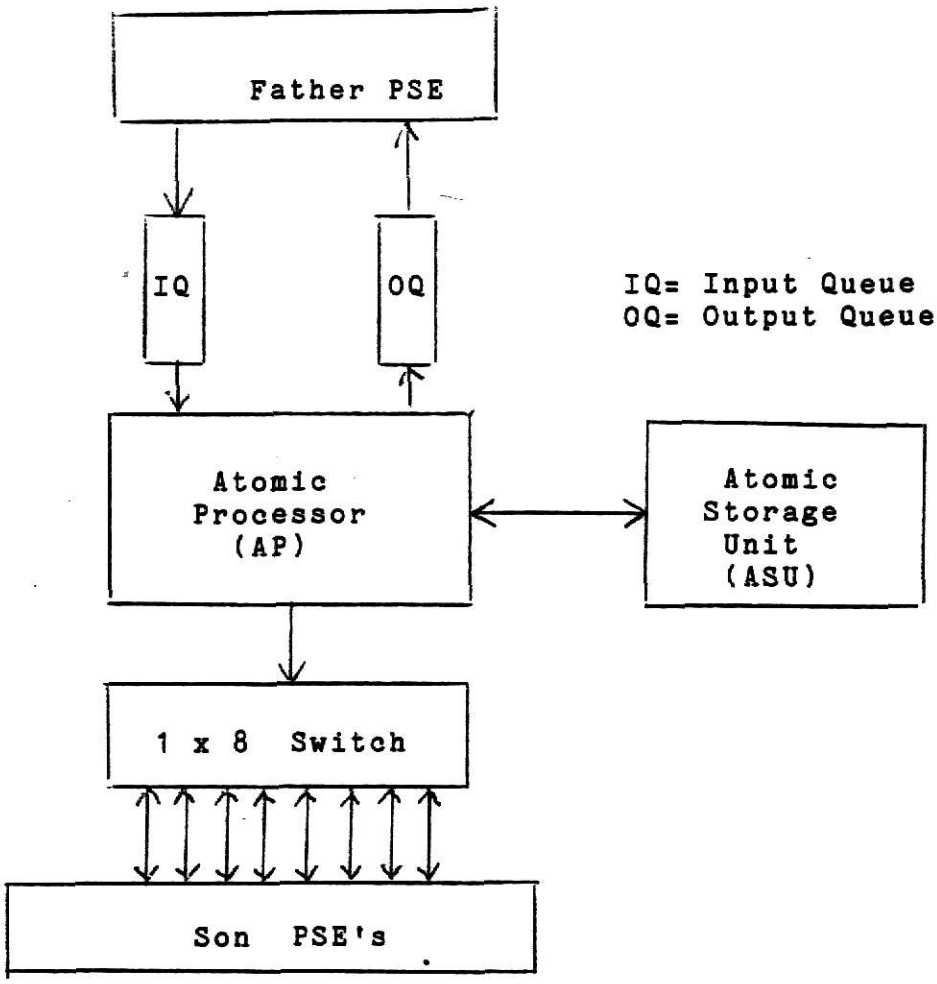


Figure 5.5 Structure of a PSE

All of the atomic processors are capable of executing all of the instruction cells in DDN's, and only the atomic processors are capable of cell execution. The AP can also allocate processing to any of its son PSE's.

The atomic storage unit (ASU) contains all of the data and DDN's which are currently residing at the node. All of the management of the storage is controlled by the ASU. The data is internally managed as a tree structured file.

All of the components of the PSE are asynchronous

elements which are capable of concurrent processing. During the actual execution of a program on the Utah machine, as a program net arrives at a Processor Store Element, the PSE may take one of two possible actions. Which action is taken depends upon the nature of the net and the position of the PSE in the hierarchy.

1). Decomposition and Allocation: If the PSE has a substructure and concurrency exists in the net, the net will be decomposed and sent down.

2). Execute locally: If the PSE has no substructure, or if no concurrency exists in the net, the net will be executed in the PSE.

VI. CONCLUSIONS

There is little doubt that data flow machines are capable of realizing a high degree of concurrency, and the modular nature of the machines allows for easy extension and cost effective utilization of VLSI. These are, however, only a few of the factors which will determine the success of the machines in the marketplace.

The greatest obstacle is in the cost of reprogramming current applications to run on data flow machines. The majority of computers are not being used for highly complex scientific problems such as weather forecasting; they are being used for payroll, inventory, billing applications, and other business functions. Millions of dollars have been invested in the developement of these applications, and it is extremely doubtful that the advantages of data flow architectures over conventional architectures justify the cost of reprogramming.

On the technical side, data flow machines, as currently designed, have some problems in the area of data structures such as arrays and records. These are problems which will probably disappear as the designs become more sophisticated and new techniques for manipulating data are devised.

On the other hand, computers are moving into a number of newer areas which are not burdened by thirty years of application development. In such areas as artificial intellegence and highly complex problems such as weather forecasting, the potential for utilizing data flow machines

is excellent.

What will probably happen is that data flow machines will be developed and used in those areas to which they are best suited. Meanwhile, the realm of business applications will continue to be dominated by more traditional architectures.

REFERENCES

1. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Archictecture," Computing Surveys,Vol. 14,No. 1,March 1982.

2. J. Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs." CACM, Vol. 21,NO. 8, Aug. 1978, pp 613-641.

3. A. L. Davis, and R. M. Keller, "Data Flow Program Graphs," Computer, Vol. 15,No. 2,Feb. 1982, pp. 26-41.

4. A. L. Davis, Data-Driven Nets: A Maximally Concurrent Procedural, Parallel Process Representation for Distributed Control Systems. Tech. Rept. UUCS-78-108, Univ. of Utah, Comp. Sci. Dept., 1978.

5. W. B. Ackerman, "Data Flow Languages," Computer, Vol. 15, No. 2, Feb. 1982, pp 15-25.

6. J. B. Dennis, and D. P. Misunas, A Preliminary Architecure for a Basic Data-Flow Processor. Tech. Rept CSG Memo 102, M.I.T, Aug. 1974.

7. I. Watson, and J. Gurd, "A Practical Data Flow Computer," Computer, Vol. 15, No. 2, Feb. 1982, pp. 51-57.

8. J. Gurd, I. Watson, and J. Glauert, A Multilayed Data Flow Architecture. Dept. of Comp. Sci., University of Manchester, July 1978.

9. J. Gurd, and I. Watson, "Data Driven System for High Speed Parallel Computing-Part 1: Structuring Software for Parallel Execution," Computer Design, June 1980, pp. 91-106.

10. A. L. Davis. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. Tech. Rept. UUCS-77-113, Univ. of Utah, Comp. Sci. Dept., 1977.

BIBLIOGRAPHY

1. W. B. Ackerman, "Data Flow Languages," Computer, Vol. 15, No. 2, Feb. 1982, pp. 15-25.

2. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Fucntional Style and its Algebra of Programs," CACM, Vol. 21, No. 8, Aug. 1978, pp. 613-641.

3. A. L. Davis, The Architecture of DDM1: A Recursively Structured Data-Driven Machine, Tech. Rept. UUCS-77-113, Univ. of Utah, Comp. Sci. Dept., 1977.

4. A. L. Davis, Data-Driven Nets: A Maximally Concurrent Procedural, Parallel Process Representation for Distributed Control Systems. Tech. Rept. UUCS-78-108, Univ. of Utah, Comp. Sci. Dept., 1978.

5. A. L. Davis, and P. J. Drongowski, Dataflow Computers: A Tutorial and Survey. Tech. Rept. UUCS-80-109, Univ. of Utah, Comp. Sci. Dept., 1980.

6. A. L. Davis, and R. M. Keller, "Data Flow Program Graphs," Computer, Vol. 15, No. 2, Feb. 1982, pp. 26-41.

7. J. B. Dennis, and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor. Tech. Rept. CSG Memo 102, M.I.T., Aug. 1974.

8. J. B. Dennis, "Data Flow Supercomputers," Computer, Nov. 1980, pp. 48-56.

9. D. D. Gajski, D. A. Padna, D. J. Kuck, and R. H. Kuhn, "A Second Opinion on Data Flow Machines," Computer, Vol. 15, No. 2, Feb. 1982, pp. 58-69.

10. J. Gurd, I. Watson, and J. Glauert, A Multilayered Data Flow Architecture. Dept. of Comp. Sci., University of Manchester, July 1978.

11. J. Gurd, and I. Watson, "Data Driven System for High Speed Parallel Computing-Part 1: Structuring Software for Parallel Execution," Computer Design, June 1980, pp. 91-106.

12. J. Gurd, and I. Watson, "Data Driven System for High Speed Parallel Computing- Part 2: Hardware Design" Computer Design, July 1980, pp. 97-106.

13. E. J. Lerner, "Data Flow Architecture," IEEE,Spectrum, April 1984, pp. 57-62.

14. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," Computing Surveys, Vol. 14, No. 1, March. 1982,

15. I. Watson, and J. Gurd, "A Practical Data Flow Computer, Vol. 15, No. 2, Feb. 1982, pp. 51-57.

A SURVEY OF DATA FLOW MACHINE ARCHITECTURES

by

DAVID ANTHONY MEAD

B. A., Kalamazoo College, 1962

AN ABSTRACT OF A MASTER'S REPORT

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

ABSTRACT

In recent years, the need for greater parallelism and recent advances in computer technology, have prompted some researchers to examine alternatives to the traditional computer architecture.

One of these alternative architectures is the data flow machine, in which the sequencing constraints are determined by the flow of data, rather than by the explicit flow of control. Traditional control flow machines, data flow machines, and another alternative architecture called reduction machines, are defined and compared.

Data flow graphs, which form the foundation of data flow concepts, are described, with examples of alternation and iteration in data flow. The major hardware components of a data flow machine are then described. The properties of programming languages which are compatible with the developement of high level languages for data flow machines are also discussed.

The final section of the paper describes three university projects in data flow. The projects described are those from the Massachusetts Institute of Technology, the University of Manchester, and the University of Utah.