DATAFLOW COMPUTERS:
A TUTORIAL AND SURVEY

by

A. L. Davis
and
P. J. Drongowski

UUCS - 80 - 109

September 1, 1979

Revision: July 15, 1980

Table of Contents

List of Figures

1. Introduction

The demand for very high performance computers has encouraged some researchers in the computer science field to consider alternatives to the conventional notions of program and computer organization. The dataflow computer is one attempt to form a new collection of consistent systems ideas to improve both computer performance and to alleviate the software design problems induced by the construction of highly concurrent programs.

This report discusses both the dataflow computer concept in general, and specific dataflow computer designs and implementations in particular. Our intent is to introduce computer science professionals to the current results and terminology in the dataflow field. Serious readers should be able to acquire the knowledge necessary to assimilate and analyze in some depth, both recent and future research results. The subsections which follow describe the notion of dataflow programs and present some of the architectural features required to support these programs. The dataflow computer concept is then compared with the conventional von Neumann computer followed by a brief discussion of the disadvantages and benefits offered by the new model. Next, a survey of contemporary dataflow computer organizations is presented. The similarities and differences of these machines are discussed. Finally, a detailed case study of one particular dataflow computer is presented. Throughout the report, we have included references to recent work in this field to help the reader find additional information on both the dataflow concept and ongoing research.

1.1 The Data-Driven Computing Model

Each machine system in existence can be thought of as being based on some computing model. The same can be said of hypothetical machines which exist only in a conceptual sense. Often such conceptual or "paper" machines are called computer architectures. The choice of the computing model fundamentally affects features in programming languages, operating systems, machines, and architectures. We present here a fundamental hypothesis about why the von Neumann systems ideas have been so successful: they are self-consistent. Self-consistent systems ideas provide a framework in which system sub-components are not in conflict with each other. This results in more efficient system operation for the cost. In order to correct what are currently perceived as limitations in the von Neumann model, it is insufficient to modify only a single aspect of this traditional set of ideas. A single modification may result in an inconsistent set of systems ideas. The inconsistency may cause tremendous additional complexity which will adversely affect both cost and performance.

In an attempt to find a new self-consistent set of systems ideas, a number of researchers have proposed the data-driven computing model. The basic idea behind a data-driven program is that activities should be initiated asynchronously by the arrival (or availability) of the necessary information required to perform that activity. This data-driven model (also termed dataflow) is then in direct contrast with the von Neumann model which is control-driven, in that some control mechanism (namely the clock and the program counter) specifies which actions will take place and when.

Figure 1-1 illustrates the operation of a data-driven program which computes the expression "a*b + c*d". At the machine language level, dataflow programs may be conveniently represented as directed graphs. In the data-driven model, the nodes in the graph (called

cells) correspond to machine instructions which operate upon a sequence of operand tokens which flow to the cells along the arcs of the program graph. In the von Neumann computing model, the instructions cause the operand data to be fetched from memory to the processor; in the data-driven model, the data simply flows to the instructions cells. The arcs between cells can be viewed as FIFO storage pipes which queue the data items until the cell is ready to remove them for cell execution. When all the operands are present (each queue has transported a data item to the cell), the cell removes a data item from the head of each queue, computes the result, and passes the resulting data value along the output arc from the cell. Note that the expressions "a*b" and "c*d" may be computed concurrently. Due to the queueing behavior of the arcs, the "+" cell may also operate concurrently with the two "*" cells provided that sufficient data arrives to drive both the "*" and "+" cells. This type of concurrent operationis commonly called pipelining.

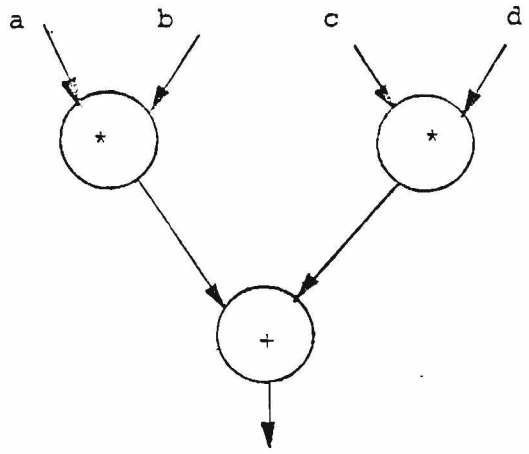


Figure 1-1: Dataflow program for the expression "a*b + c*d"

Another interesting model for highly concurrent computation is the demand-driven model proposed by a number of other researchers

[17, 6, 32, 9]. A brief description of the demand-driven model is presented here in order to acquaint the reader with some concepts related to data-driven programs, which will be expanded upon in the sequel. In demand-driven programs some actions are initiated by data-driven rules and others are initiated by demands which are generated by other actions. Typically initial allocation of low level tasks is initiated by demand, and subsequent evaluation is performed in a data-driven manner. Using Figure 1-1 once again, we can describe the operation of the demand-driven model. Initially, a demand item arrives at the arc labelled "a*b + c*d" representing a need to compute the expression. Because the arcs from the "*" cells to the "+" cell are empty (i.e., no data tokens are waiting) the "+" cell sends a demand token to both "*" cells. Since data items are available at the inputs of the "*" cells, they remove the incoming data items, compute their respective results, and send the resulting data items back to the "+" cell. The "+" cell senses that its demands have been met by the arrival of the data items on its inputs. The "+" function is performed and the result is sent over the arc "a*b + c*d" satisfying the initial demand on the program graph.

The following characteristics of the demand-driven model should be noted:

- Unlike the data-driven model, two types of information are carried over the program arcs: demand and data items.
- Demands flow back through the graph while data items flow forward.
- Both demand and data items may be queued on the program arcs.

The principle advantage of the demand-driven model is the avoidance of unnecessary computation. In larger more complex dataflow programs with conditional selection of results (i.e., a dataflow version of "if-then-else" or "case" selection), the data-driven model will evaluate all expressions for which data items are available, while the

demand-driven model will only evaluate expressions for which there is a demand or need. This is usually a subset of the data-driven expressions. In either model, a computation utilizes some physical resource. Hence, a program running under the demand-driven model should use less physical resources than its counterpart executing under the data-driven rules. A demand-driven program, however, incurs higher communication costs because demand items must be transmitted around the graph in addition to data items. The seriousness of these costs are usually cited as the primary reasons for choosing the dataflow (data-driven) model over the demand-driven model. A data-driven program may be viewed semantically as a demand-driven program in which a demand always exists at the output ports of the program cells.

All three computing models (von Neumann, data-driven and demand-driven) have their advantages and disadvantages. We concentrate here on an in-depth presentation of data-driven languages and machines. Due to the research nature of the data-driven field, the vocabulary and terminology has not yet standardized. We will therefore attempt to indicate synonyms wherever possible. The following section introduces additional terminology relevant to data-driven computing formalizing some of the concepts discussed above. It may be skipped by the casual reader.

1.2 A More Formal View of Data-Driven Computing

There are a number of ways to conceptualize the data-driven program model. The most commonly used and perhaps the easiest method, is to view a data-driven program as a directed graph. The vertices (also called nodes or cells) correspond to actions which are performed in the program. The directed arcs correspond to data paths over which information is transmitted from the producer of the information to the consumer. The information is carried in quantum units which can be

thought of as messages. In a message passing environment, the directed arcs lead from the sender to the receiver. A node of a program graph may have any number of arcs. The actual number of arcs for any specific node will depend on the type of action associated with that node.

There are usually no restrictions on the operation of an arc which specifies the number or the type of messages which the arc can carry. If more than one message (also referred to as token, data item, data token, or packet) exists on an arc then the arc logically operates as a FIFO (first-in first-out) storage device. These arcs (queues) conceptually have infinite length. This notion of infinite length queues should make most pragmatists quite nervous. It will be shown in section 3.4 that this notion of infinity is a nice conceptual device, and does not necessarily present operational difficulties. The lack of a type restriction on messages implies that a message can be almost any data structure (e.g. literal, integer, vector, program, etc.)

There are two aspects of data-driven cells which are important in understanding how actions take place in program graphs.

1. What activity takes place? This is specified by a cell function. Each cell in a program graph has a specific cell function associated with it. The method by which the cell function is specified varies from one data-driven language to another. Typically it is a combination of a graphical shape and a tag or name associated with the cell.

2. When does the activity take place? This is specified by the firing rule. Each cell has an associated firing rule. The firing rule specifies which set of input arcs must contain at least one message before the cell function can be performed. This set is called the firing set.

When the firing rule of a cell is satisfied then that cell is said to be fireable. The data-driven model is an asynchronous one, and therefore a fireable cell is executed (fires) at some finite (but undetermined) time after it becomes fireable. When a cell fires, the

firing set data items are destroyed, and a set of resultant data items are placed on the output paths. The order in which the output data items appear on the output paths is unknown. The time at which the outputs appear after a cell fires is finite but unspecified, and no assumption can be made about the order or the relation between the times at which the output items appear. This implies that cell behavior is completely asynchronous, and this is essential to a schema which is to be easily implemented in a distributed control environment. A cell is said to have fired only after all of the firing set data items have been removed and all output data items have been placed on the output paths.

An example of a cell firing is shown in Figure 1-2, where a cell performs a simple integer addition. In this case, the firing set is the set of all input data paths. The result of firing this cell would be: all output paths receive the sum of the input path items.

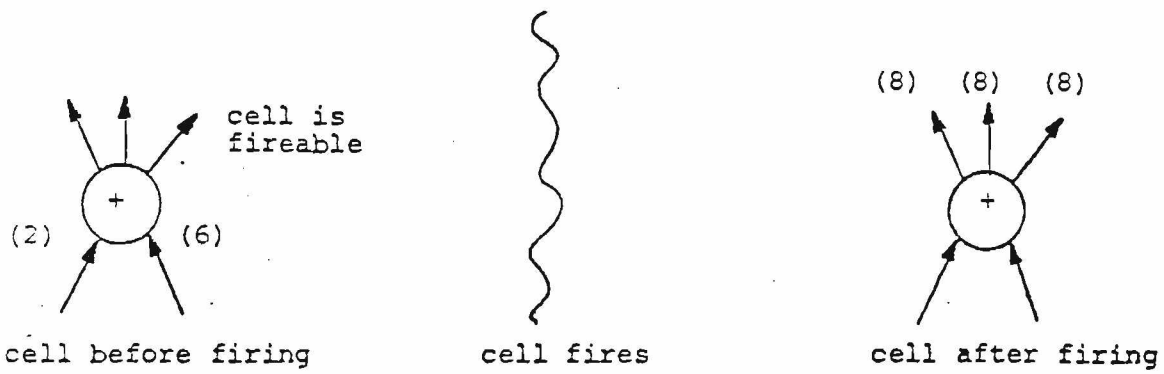


Figure 1-2: A Sample Cell Firing

In the case where all of the input arcs contain more than a single message, the cell may remain fireable. In this case, the cell may continue to fire as long as the input paths (pipes) can supply the cell with input data at a sufficient rate. This type of operation is caled pipelined execution, and is illustrated in Figure 1-3.

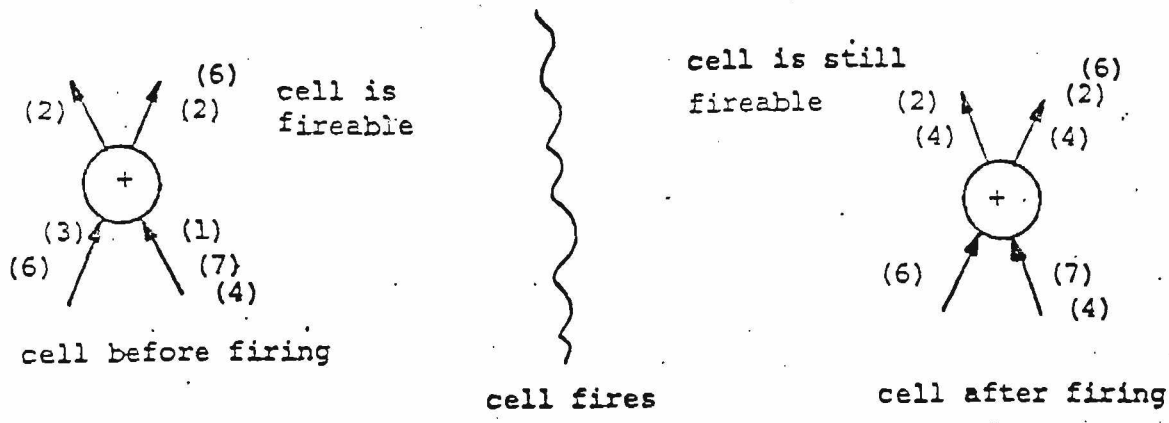


Figure 1-3: Pipelined Execution

There has been considerable discussion in recent literature concerning the benefits of functional programs [7, 17, 31]. Functional programs do not use the von Neumann global variable (storage location) concept. This allows data to be used as a value rather than as a storage location. This also implies that the nets are not history dependent (i. e. they are "memoryless"). This storage management discipline removes side-effects, which have proven to be very difficult to deal with in the formal verification of programs. Programs in which the order of statement execution is not strictly dependent on the lexical ordering of the statements in the program, but where the order is dependent upon functional relationships are called nonprocedural languages. The semantics of a nonprocedural program can be described in denotational terms rather than in an operational sense. "Denotational semantics" can considerably simplify the task of program verification. The von Neumann program model tightly binds the notion of physical location of program and data elements with the execution of program elements and the use of data elements. Functional languages break the tight binding for data, and nonprocedural languages break the binding for program elements. Data-driven programs are inherently functional and nonprocedural.

Data-driven programs also naturally represent two forms of concurrency. The only sequencing rule for a data-driven program is that of data dependency. That is, if cell B depends on cell A for data (i.e. a directed path exists from cell A to cell B) then A and B are sequenced, and B follows A. If A does not depend on B for data (and vice versa) then A and B are independent activities and can therefore be executed in parallel. This type of parallelism is often called horizontal or spatial concurrency. Two spatially concurrent activities may be distributed to two distinct points in space, where each point is capable of independent execution. The other form of concurrency is pipelining (vertical or temporal) concurrency. The data-driven program illustrated in Figure 1-4 represents both types of concurrency.

In Figure 1-4, notice that at time t2, both cell B and cell C are fireable. They are spatially concurrent. Cells A, B, and C are all fireable at time t1 even though for a given message, B and C follow A. The value produced by A between time t0 and t1 can be consumed after time t1 by B and/or C. After time t1, cell A is still fireable and can therefore be evaluated concurrently with B and C. This is due to the pipelined execution which is possible for data-driven programs. In order for B and C to both use results from A, two distinct copies of A's result values need to be produced. One copy is sent to cell B, and the other is sent to cell C. If A's result were to be stored in some single physical location and then addressed by B and C, then some "hidden" sequencing may result from storage access conflict. It is possible to construct multiport storage units, but this would not help when the same cell is accessed by multiple tasks asynchronously.

There are a number of data-driven languages (schemas) in existence. Many of these schemas have been used only as models for thematic development. Three of them are currently being used as a

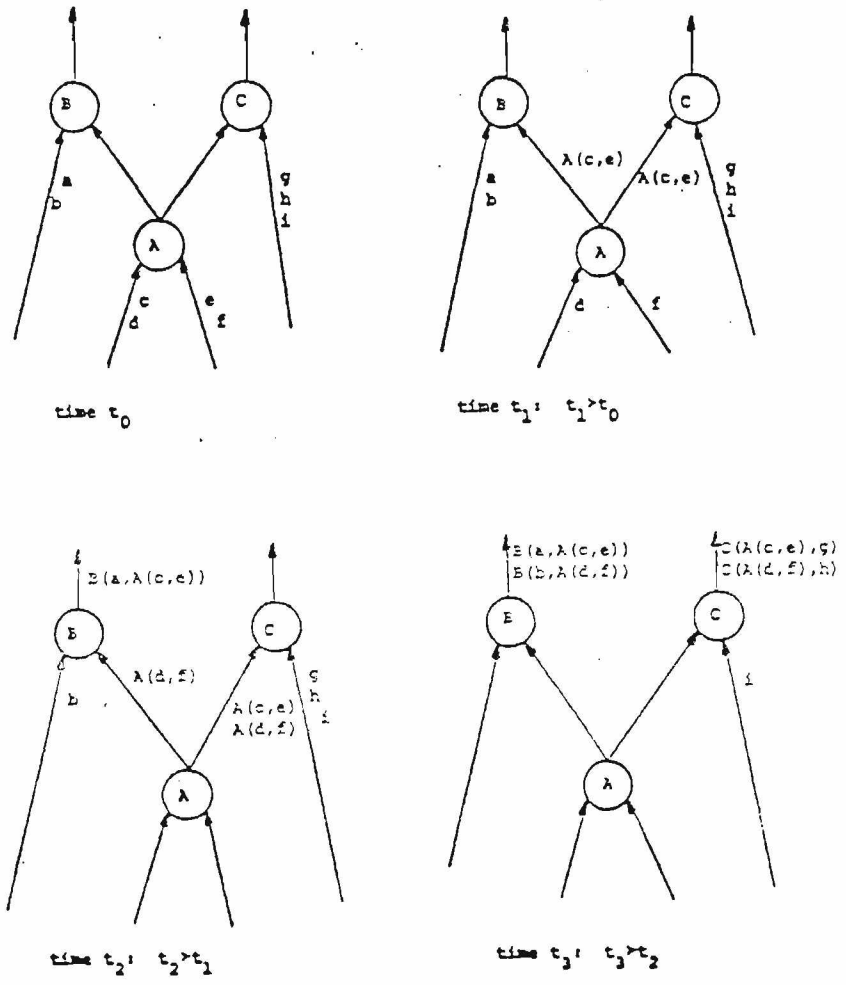


Figure 1-4: Two Types of Concurrency

basis for serious system development. These are:

1. DDF, Dennis Data Flow, a graph schema developed by J. B. Dennis [13] at MIT. This schema is the most widely known and is used as a starting point for research work being carried out at MIT, Manchester University, U. Cal. Irvine, and Iowa State University.

2. DDN's (Data-Driven Nets), a graph schema developed by A. L. Davis. This schema is used as the base language for research being done at the University of Utah, the Novosibirsk Computing Center, and Burroughs Corporation.

3. Another data-driven representation is being used at Toulouse University in the development of a prototype computing system.

A detailed description of all of these variants would be out of place here, and we therefore will restrict our presentation to a conceptual discussion of the data-driven computing model, and a later description in section 3.4 of one particular schema.

1.3 Structural Concepts of Dataflow Architecture

The preceding section presented the dataflow computer concept from the programmer's perspective. This section discusses at an abstract level the kind of machine structure required to support the data-driven programming environment. Four major elements are necessary to support data-driven computation:

- instruction and data store,
- processing elements,
- communication links (possibly non-trivial) between the stores and the processing elements, and
- control (including the allocation of logical program structures to physical resources).

1.3.1 Instruction and Data Store

As in conventional von Neumann computers, a store must be provided to maintain instructions (program graph) and data (tokens). Each cell in a machine language program graph represents a single instruction. This instruction must contain the name(s) or address(es) of the destination(s) for the data tokens which the instruction will produce for eventual distribution of results. Similarly, data tokens contain the name of their destination so that the communications element of the architecture can properly route the tokens to their destination instruction.

Two approaches may be taken for data token storage: they may either be stored separately in their own memory unit, or the tokens may be stored with the instructions. If stored in a separate memory, the job of finding fireable instructions is more difficult because the

information necessary to determine whether the firing set of the instruction has been satisfied or not may be physically distributed throughout the token memory. Furthermore, information describing the traversal order of the tokens along an arc must be maintained. Separate storage tends to make the storage management task easier because only one type of structure needs to be allocated from the memory pool. If data tokens are stored with the instructions, it is much easier to find fireable instructions. However, the management of space in the store becomes more difficult because:

- two types of storage element must be allocated from the same physical memory,
- tokens must be maintained on some sort of list to simulate the queueing behavior of the program arcs.

1.3.2 Processing Elements

To take maximum advantage of the concurrent nature of data-driven programs, more than one processing unit should be provided. These units may be specialized for performing certain functions (e.g., integer or floating point arithmetic) or homogeneous, performing all possible cell functions. Instructions are routed from the instruction store to the processing units with their operands. If special purpose processing units are employed, the communication network must route instructions to the particular processor which is capable of executing them. If general purpose processors are used, instructions may be sent to any available processing unit. In either case, the instruction contains a field indicating the operation to be invoked by the instruction.

1.3.3 Communications

The communications component of a dataflow computer is responsible for binding together the different functional elements of the computer. Communication is normally asynchronous (reflecting the asynchronous nature of dataflow programs) between units permitting the different subsystems to operate concurrently and independently of each other. Asynchrony also serves the expandability requirements of the system because synchronous signals (which are sensitive to skew over long physical distances) are not required. Hence, modules may be added somewhat more easily.

When token or instruction routing is required, the communications component switches the tokens and instructions along the appropriate physical data path to the locus of their logical destination.

1.3.4 Control and Resource Allocation

The control portion of the dataflow processor is responsible for deciding which instructions are fireable and for initiating their execution. It must also find and utilize the available processing resources. Due to the distributed control nature of most dataflow processors, locating and exploiting available resources is a difficult problem due to the lack of global knowledge about the system state.

1.4 A Brief History of Data-Driven and Related Efforts

Work in the data-driven area is still in the research stage. The history of efforts in this area is difficult to relate, as the data-driven model resembles other directed graph schemas, most notably Petri Nets [36]. A number of true data-driven projects are probably unknown to the authors due to their research or proprietary nature. No technical purpose would be served in cataloging every known effort. We therefore present a brief history of major public developments in the data-driven field.

There is considerable debate concerning the identity of the

inventor of the data-driven computation model. It seems most likely that it was discovered in several independent incarnations. In 1964, Bahrs delivered a lecture at the Novosibirsk Computing Center on the topic of "Operation Patterns" [8]. Operation patterns are definitely a type of data-driven schema. A seminar paper discussing the properties of concurrent programs in a theoretical context was presented by Karp and Miller in 1966 [28]. Although the model employed in this work is not strictly data-driven, it demonstrated the suitability of graphs as a formal tool for the analysis of parallel computation. The first major thesis on dataflow ideas was published in 1968 by Adams at Stanford [1]. At MIT, Rodriguez completed his thesis on dataflow concepts in 1967, although the thesis was not published as a report until 1969 [27]. Dennis described a dataflow computation model in 1969 [25]. This work by Dennis evolved into a rather large (by academic standards) research effort by the "Computation Structures Group" at MIT. This group has been active since that time and is extremely prolific. The most important of the MIT publications are:

- A description of the DDF schema [13].
- A description of the MIT architecture [26].
- A description of a high-level dataflow based programming language [42].

Another important data-driven research project is underway at the University of California at Irvine. This effort received its seed in the work of Dennis at MIT. The principal founders of the UCI effort are Professors Arvind and Gostelow. They subsequently developed an architecture [2] and a high-level language [24] using a base representation derived from DDF. Their group then revised their architecture and built simulation tools which were subsequently used to produce program measurements [19].

A serious system building project is underway in France at Toulouse University. This project is developing what is known as Systeme LAU [29]. This work includes the design of a programming language and the design and construction of a prototype multiprocessor system. Completion of the hardware is expected in late 1979. The French efforts have been guided by the work of Tesler and Enea on single assignment languages. The Toulouse effort also received early motivation by the work of Dennis at MIT.

Another project started with the doctoral thesis of Davis [5] which proposed both a language and a supporting architecture in 1972. In the next 5 years, these ideas were refined under the support of the Burroughs Corporation. This led to the design of a base language [4] and the construction of a prototype machine [3] which became operational in 1976. A description of the base language and a detailed case study of this machine will be presented in section 3.

In 1977, a number of Universities in England were stimulated by funding from the British Science Research Council to undertake research efforts in the area of data-driven computation. These include projects at Manchester University [21], Newcastle University, and Westfield College. The Manchester group have a hardware prototype of their machine under construction at this time.

A number of other (mostly academic) projects are involved to some extent with data-driven computation. Due either to the unpublished nature of this work, or the "youth" of the projects, they will simply be listed here as sites of effort:

- Iowa State University
- Xerox PARC
- Texas Instruments
- Clarkson College

- University of Southwest Louisiana
- Purdue University
- University of Kansas

In addition, there has been considerable recent interest in Japan, primarily at the University of Tokyo, Hitachi and Fujitsu laboratories.

The data-driven computational model has a number of ties to better established lines of theory. In particular, formal graphical network properties proven by Petri [36], Holt and Commoner [23], and other Petri net workers [16, 22, 35, 11] play an important role as a theoretical foundation for much of the work in the data-driven area. It has already been mentioned that data-driven programs are both functional and non-procedural, and as such are related to more general work in these two areas of programming language and program verification research. Since the data-driven computing model is primarily a model for highly concurrent, distributed control systems; the work is further influenced by other work on parallel processing.

1.5 Observations About Data-Driven Computing

1.5.1 Comparison with the von Neumann Organization

In 1945, John von Neumann described the organization for a computing device which has become the most prevalent computer architecture (with few exceptions). The pure von Neumann computer has a single processing unit which is connected to the primary memory unit through a set of parallel information paths which exchange control signals, addresses, and data between the two units. We shall call this collection of information paths the processor to memory channel, or simply, the channel. The channel operates in a sequential fashion, i.e., the protocol which the processor and memory use to control the exchange of information is a sequential one. Information is exchanged between the processor and memory by changing the state of the control

signals and passing an address from the processor to the memory unit. In the case of a memory read operation (as specified by the control signals) a data word is retrieved from the memory and transferred to the processor. If a write operation is selected, the data word is sent from the processor to the memory. Other features of the von Neumann computer are:

- They contain a single centrally clocked processor.
- The processor executes a single instruction in step with the clock.
- The processor is the system master and plays the role of a control and communications center for the entire system.
- Storage is organized as a linear array of basic units called words.
- The processor accesses memory by specifying the physical position (address) of some word in the linear array.

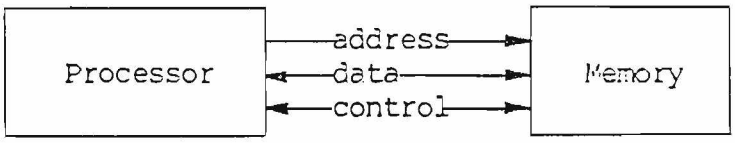


Figure 1-5: Simple von Neumann computer organization

It is true that many modern computers exhibit some features which are not typical of the von Neumann computer, for example:

- The Burroughs B5500, B6800, B7700 machines are based upon stack addressable storage rather than linearly addressed memory.
- The Burroughs B1800 does not have a fixed logical word length.
- The IBM360 and 370 series have special I/O processors called channels which can act independently of the main processor in performing certain operations.
- The Honeywell 6050 has a number of peripheral I/O processors which specifically handle I/O tasks.
- The CDC6600 and the CRAY-1 contain a number of arithmetic

processors which perform high speed arithmetic functions as requested by a separate control processor.

Each of these systems has departed in some significant way from the von Neumann organization. However, program and data access remains primarily sequential making the temper of these machines von Neumann in nature.

Five major differences exist between the von Neumann computing model and the data-driven model. They are:

1. Von Neumann programs are strictly sequential; sequencing in a data-driven program is determined wholly by the data dependencies inherent in the solution of the problem.

2. Data-driven programs permit greater freedom to exploit the potential concurrency in a given program.

3. The co-ordination of parallel processes is directly supported by the data-driven execution model, i.e., through the firing set rules. The strict sequencing in von Neumann programs is conceptually incompatible with the processing of asynchronous events.

4. The structure of dataflow programs and computers admit to decentralization because the constituent subsystems communicate asynchronously, eliminating lock step operation with a central system clock.

5. The von Neumann architecture is viable with respect to the implementation technology of the fifties and sixties. Dataflow ideas are oriented toward system implementation in very large scale integration (VLSI), the implementation technology of the future.

The subsections which follow discuss and attempt to justify each of these differences.

1.5.2 Differences in Programming Style

To illustrate the differences in programming style between von Neumann programs and dataflow programs, figure 1-6 contains a von Neumann program equivalent to the dataflow program of figure 1-1. Note that the von Neumann program is rigidly sequenced, i.e. the instructions must be executed in sequence from top to bottom for the program to be meaningful and execute correctly.Memory cells are

```
move      a, register-0
multiply  b, register-0
move      c, register-1
multiply  register-0, register-1
move      register-1, destination-address
```

Figure 1-6: von Neumann Style Program for the Expression "a*b + c*d"

explicitly referenced via the addresses "a", "b", and "c". The processor resident registers, "register-0" and "register-1", are employed as high speed scratch-pad storage cells to avoid the higher transfer cost of repeatedly moving information across the channel from the memory to the processor where it can be acted upon and transformed. The dataflow program does not employ memory cells for storage and does not require the use of explicit addresses; the memory is implicitly embedded in the storage semantics of the program arcs. The only sequencing demands made upon the dataflow program is that the addition must be performed after the two multiplication operations. The multiplication operations may be performed in parallel if sufficient processing resources are available.

1.5.3 Exploitation of Concurrency

As the computing field progressed and the solutions of larger problems were attempted through the use of computers, system architects attempted to improve the performance of the von Neumann organization through various methods. The speed of access to stored information was improved through the use of cache memories, for example. Direct transfer paths were placed in the system to move data from input/output devices to the memory, rather than forcing all external data transfers through the processing unit. Processors were internally pipelined to overlap the fetching and execution of instructions. These improvements, however, did not change the conceptual nature of the computer - that of sequential data access and program execution.

The use of cache memories and pipelined instruction processors increase the exploitation of concurrency within a von Neumann program to a limited extent. Through instruction pipelining, more than one program instruction may be under execution at any given time by buffering successive levels of combinatorial logic with memory elements (e.g., registers). The maximum number of instructions which may be under simultaneous execution, is determined by the number of stages in the pipeline. The operation of the pipeline is disrupted by:

- instructions which divert the flow of program control (e.g., goto's), and
- instructions which must read the information stored in memory cells or registers whose contents have not yet been computed and stored by a more advanced stage in the processor pipeline.

These disruptions are both artifacts of the explicit sequencing and memory cell features of the von Neumann computer. Because data-driven programs do not use memory cells and sequencing is only determined by the relative position of the function nodes in the directed program graph, additional instruction and program level concurrency may be exploited, provided that the physical processing resources are available within the dataflow computer to concurrently execute the instruction nodes.

1.5.4 Co-ordination of Parallel Processing

The construction of multiple processor systems was intended to further improve machine performance by interconnecting a number of processor-channel-memory elements into a network of interacting computing systems. Large problems are separated into pieces which can be managed by the individual computers. The programs must co-ordinate their access to shared data in order to correctly perform the aggregate task. Because the computing elements are based upon the von Neumann computing model, program execution and data transmission is

performed sequentially between processors and memory. Although multiprocessor systems significantly increased the problem solving power of modern computing, the mixture of explicit sequencing within programs and asynchronous system level behavior remains a problem, as the following discussion illustrates.

Programs (or portions of a program) which are executed in parallel and share information in some way must be co-ordinated to guarantee that the information used in any particular computation has in fact been completely defined before its use, i.e. that the computation is deterministic [10]. With the introduction of asynchronous input/output channels in about 1955, programmers were confronted for the first time with the synchronization problem; how to synchronize the operation of two or more devices and processors such that each works at maximum speed, but periodically communicates and shares results. The hardware interrupt was the first (and for quite some time, the last) mechanism proposed to assist processor and device synchronization. As a theory of operating systems developed, it became clear that the interrupt was incompatible with the modern notion of process structure causing a conceptual mismatch between software processes, the operating system, and the interrupt mechanism of the hardware. The hardware "wish list" of an operating system designer usually contains an entry requesting the virtualization of the device hardware to that of the process structure intended for the machine.

The data-driven execution model conveniently provides a mechanism whereby concurrently executing portions of a dataflow program can co-ordinate processing. In figure 1-7, the result of the entire program cannot be formed until both the upper right and upper left subgraphs have produced their data tokens. One might anticipate that part of the inputs to the subnetworks could be obtained from a file or

device. Hence, the program quite naturally and conveniently suspends execution until the file or device data arrives. Elaborate changes in physical and logical context are not necessitated as in interrupt based systems.

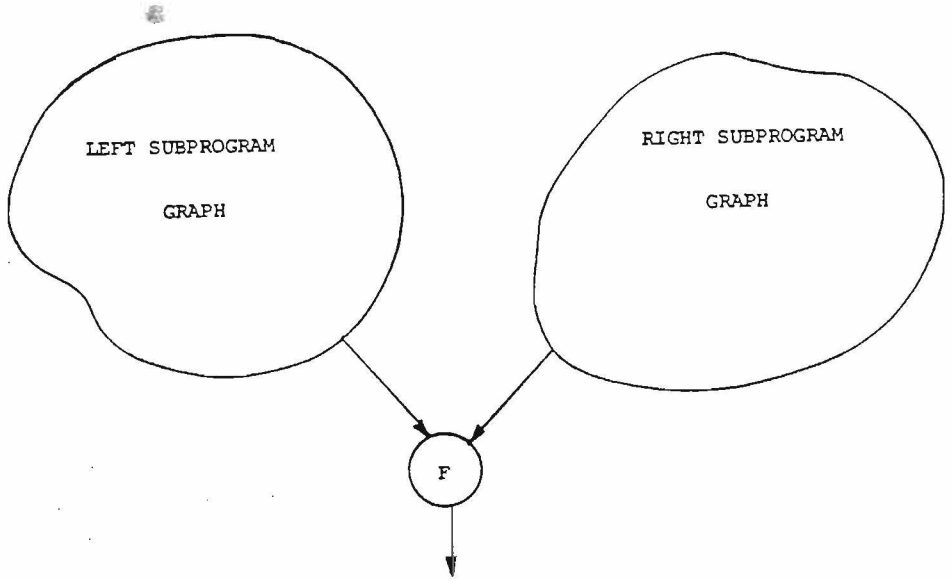


Figure 1-7: Co-ordination of Subnetworks

1.5.5 Decentralization and Buildability

In addition to performance improvements, computer users need increased computer reliability for large applications such as telemetry, anesthesiology, information services, etc. Requirements for higher reliability have placed greater emphasis on software correctness. The von Neumann memory cell is hard to characterize mathematically, making formal verification of programs for those machines quite difficult [41]. Although some good verification techniques have evolved for sequential programs expressed in higher level programming languages, the proof of correctness for parallel programs remains a hard problem. For computer applications which include information privacy as a required capability, the lack of

elegant, formal machine properties further inhibits the mathematical proof of programs with high reliability as a goal [14].

Current multiprocessor architectures encourage software designers to perceive the system in terms of global state information, rather than more localized and less complex computational abstractions. The notion of global system state runs against the human inclination to parcel problems into smaller, more manageable units. Without the ability to decompose and abstract problems, our ability to deal with complexity and to make convincing mathematical statements about systems is limited.

Centralization impacts the buildability of systems, let alone the ability to understand them. Centralized systems use a common clock which must be distributed to system components. As the physical size of centralized multiprocessors increase, signal propagation times become longer unless the multiprocessor can be packaged in a novel way which reduces the length of the clock transmission paths, e.g. the CRAY-1. Longer propagation times result in slower execution speeds. Inventive packaging has its physical limits and often reduces the physical and logical expandability of the system. Central bus architectures suffer from similar signal distribution problems. Hence, the key to expansion, performance improvement, and comprehensibility may be decentralization and asynchronous operation.

1.5.6 Impact of Implementation Technology

A computer architecture depends heavily upon the technology available to build it. At the time that von Neumann proposed his computer organization, computers were constructed with vacuum tubes and the hardware was very expensive. During the 1960's and early 70's circuit technology progressed from discrete transistor circuits to integrated circuit technology and dramatically reduced hardware costs. The prevalent computer architecture of our time is therefore based

upon a set of design principles which are presently invalid. Most notable of the obsolete principles is the one that states, "Hardware is expensive; software is cheap." The move toward very large scale integration (VLSI) circuit techniques has again drastically changed computer design rules. Highly integrated von Neumann processors with minicomputer capabilities (e.g., word length, speed, address space, etc.) are commercially available at a reasonable cost. Because a fully integrated dataflow processor has not yet been developed, it is too soon to tell whether machines based upon the dataflow model can fully utilize (and avoid the pitfalls of) VLSI implementation. The results presented in the following section, however, are encouraging and show that the exploitation of the VLSI technology is possible and it is not incompatible with the model.

1.5.7 The Impact of VLSI

The advantages of high density integrated circuit technology are so overwhelming that the constraints of VLSI must be considered as a primary force on future architectures. A detailed analysis of these effects is beyond the scope of this paper, but the global influences are summarized here.

Modern integrated devices are primarily built from either junction transistors (bipolar integrated circuits) or from field-effect transistors (metal oxide semiconductor or MOS integrated circuits). Due to the tremendous commercial emphasis that is currently being placed on MOS VLSI, the following discussion will mainly be concerned with the properties of MOS device integration. The qualitative aspects of the following argument applies to bipolar devices but the numbers would be somewhat different.

The most highly publicized VLSI benefits are those involving cost. A single custom VLSI chip (64 pin package) currently costs about $80,000 to $300,000 to produce. Even then, production typically

must be guaranteed for about a quarter of a million parts at an additional cost of $7 to $10 per part. This clearly indicates that VLSI cost advantages can be obtained only if any given chip can be used in very large volumes. If a part does not have universal appeal, then the use of such a part in a new architecture brings about a number of high pressure constraints. Either the part must be used a large number of times in a single system, or a single system must have a very high sales volume, or some combination of the two. The number of part types in a given system is also a major concern in that it becomes a multiplicative factor in the system development cost.

Another factor heavily influenced by a VLSI implementation is speed. The dominant speed factor for integrated circuits is the amount of delay which is incurred whenever a transistor tries to drive a signal level onto a conducting path. The size of this delay is proportional to the amount of capacitance which the signal path contains. The amount of effective capacitance which is attributed to any output is often called the load of that output. Typical off chip loads are on the order of 100 picofarads, while on chip loads are approximately one picofarad. Since delay times are proportional to the capacitive load (for constant output current from the driving transistor), this implies that signals which can remain on the chip will be driven about 2 orders of magnitude faster than those which must be driven to destinations off the chip. Additional speed-up can be obtained from the decreased geometries of the switching elements and the conductor path lengths on an integrated circuit chip. This is a very strong argument for architectures which attempt to maximize locality of processing. For architectures in which activity cannot be done at the same physical locus, massive off chip delays must be incurred as a result. The only way around the slow off chip drive problem is to drive more current off the chip. This requires a series of relatively large output drivers (implemented using physically large

transistors), which are very costly in terms of chip area and chip power consumption. As the integrated circuit technology advances and the size of individual circuit components is decreased, this disparity between on and off chip capacitances will increase. In addition, locality will reduce the amount of contention for a given system transmission path. This contention is important in a highly parallel system in that the resultant sequencing due to transmission conflict will yield reduced system efficiency.

The number of pins is an important VLSI metric. The pin count is a primary factor in determining whether a given system module is nicely implementable as a VLSI circuit. Techniques to decrease physical pin count, such as time division multiplexing, are applicable in certain situations, but cannot be considered a general solution. If chip types are used in sufficient quantities to amortize the initial layout cost, then the physical cost to manufacture the system becomes approximately linear with pin count. Increasing the number of pins on a particular chip causes decreased yield due to bonding problems. Increased pin count implies that even more silicon area must be allocated to connection pads and pin drivers.

A VLSI implementation also yields the more commonly discussed advantages such as:

- Increased system reliability due to reduced part count,
- Decreased power consumption since voltages on a given chip scale with physical feature size, and
- Decreased system maintenance cost as chip replacement policies become more effective in highly integrated systems.

The extent to which these VLSI advantages can be realized is proportional to the logic/pin ratio of the proposed system modules. If the logic/pin ratio is relatively small then the situation is very much that of an SSI (small scale integration) machine. If the

logic/pin ratio is very high then true VLSI advantages can be obtained. This is a challenge to computer architects to devise systems which can be modularized into high complexity modules which communicate with their environment using relatively few signals. Furthermore as integration technology advances causing feature sizes to shrink even more, these new architectures must remain viable.

1.5.8 Benefits of the New Model

The asynchronous nature of data-driven programs makes them inherently easier to contend with in a system sense. Each elementary action can be considered independently of absolute time. Only the sequence of the actions is important. This eases the task of program verification and the task of resource allocation (section 3.7). The side-effect free and distributed treatment of storage also aids in verification and decreases the "hidden sequencing" which results from access conflicts to centralized storage. Distributed storage is physically advantageous for modern components in the following way:

- When ferromagnetic cores and vacuum tubes were used as storage components for main memory, there was an electrical advantage for centralization. Logic and memory voltage levels were very different and centralization of the storage elements allowed special purpose power supplies and level conversion circuits to be shared, thus reducing system cost. Today, main memory elements are fabricated from the same components as logic, thereby removing the former advantage of central main storage.

- The larger address space of centralized stores increases access time. This results from the delay of the address decoding logic growing approximately logarithmically with the size of the address space.

- Physically distributed storage can be used as a method to reduce the performance killing "von Neumann bottleneck" described by Backus [7]. This bottleneck is the apparent slowness of the storage unit of von Neumann systems, which results from sequential storage accesses through a single memory port.

Data-Driven programs are more intuitive and natural as a means of program expression. The main reason for this is that in von Neumann

programs:

- The programmer must order the placement of every statement in a program even if the sequence is not implied by the problem. While this may not be much of a problem to the programmer, it can become a major problem when reading that program. Sequence usually IS important, so how is the reader to know that in some cases it is of no consequence.

- When writing or reading a von Neumann program, a person needs to analyze the program in two conceptual domains: data and control. That is, the person needs to be conscious of the set of variable values, while tracing the path of the program counter through a veritable maze of program statements.

Data-driven programs do not contain unnecessary (and therefore unnatural) sequencing, global variables, or program counter concepts. Data-driven programs can be constructed and analysed by considering only one domain at a time, and each domain acts only on the basis of local influence.

Data-driven programs allow two types of concurrency to be represented in a single consistent framework. Whether or not this concurrency can actually be executed depends both on the supporting architecture and the amount of physical resources in any particular instantiation of that architecture. Fortunately the data-driven computation model frees the architect from a number of serious constraints imposed by the von Neumann model. There is not much consensus among dataflow architects as to what the best architecture should be. A survey of existing architectural ideas is given in section 2, and a detailed case study of a particular machine is presented in section 3 .

1.5.9 Problems with the New Model

The biggest problem with any new computing model is that after 30 years, there is a massive von Neumann momentum. Algorithms and programming practice have been developed on the basis of large centralized chunks of storage. A major symptom of this attitude is the tremendous influence of data-bases and their applications. Data-bases are very large global stores, and present many operational difficulties in environments where concurrent accesses are allowed. The re-discovery of good algorithms for important problems, the retraining of more than a million professional people, the re-acquisition of 30 years worth of intuition and experience all combine to limit the acceptance of any new model. For example, it would currently be ludicrous for any manufacturer to market a machine which did not run FORTRAN, COBOL, PASCAL, etc. Furthermore, most computer buyers want a new machine to run these languages at least as well as their present systems.

In designing data-driven systems, our intuition can not be of much help as it is mostly based on a traditional style of computing. The inherent "copy, use, and destroy" policy for information in data-driven systems may prove to be in direct conflict with important "institutions" such as the Infernal Revenue Service database. It is indeed a certainty that the computing community will not rewrite all existing software, and generate new data bases to accommodate a new computational model.

The promise is that the current limits to growth which are imposed by the von Neumann model will be relieved. It is also hoped that some way will be found to acceptably accommodate von Neumann artifacts into the newer model in order to avoid the reconversion of 30 years of work. Whether or not these promises can be met will depend upon:

- Whether architects can find an acceptable machine organization.

- Whether language designers can produce acceptable programming environments.

- Whether operating systems and other system services will be found to make the new environment as usable as traditional ones.

The rest of this report is devoted to an exposition of some of the attempts to solve these hard problems.

2. Survey of Existing Architectures

This section describes some of the existing data-driven computer architectures. To appreciate the similarities and differences of these machines, some architectural evaluation criteria must be established. Dataflow computers, like von Neumann machines, may be compared on the basis of two broad categories: functional (logical or behavioral) and structural criteria.

The functional characterization of a computer is a description of its behavior as perceived by the programmer. In the case of dataflow computers, logical attributes of these machines include:

- the basic execution model,
- the primitive data structures supported by the machine, and
- the operations upon instances of those structures.

This level of description is analogous to a discussion of instruction sets, words, bytes, and addressing modes for a von Neumann computer. Indeed, the execution model corresponds to a specification of the execution "cycle" of the machine; data structures for the representation of integers, reals, characters, etc.; and machine operations for instructions and addressing.

A structural description of an architecture indicates how the machine is to be physically organized at a high level. Some structural criteria include:

- the structural and logical relationship of processors to storage,
- processor organization and capabilities, and
- storage organization and management.

The structural description of a machine is the first indication of how the system architects intend to implement the functional capabilities described in the behavioral specification of the computer. The interplay between the operational behavior of programs on the machine

and the actual machine structure will largely determine the computational performance of the computer. Figure 2-1 refines the general evaluation criteria presented above and may act as a handy reference guide to the description terminology used in the following survey. An evaluation table of this type will be presented for each machine discussed in this section. It is hoped that a consistent table format will allow for a more coherent understanding of the differences between the various architectures.

Execution model:	data or demand-driven, variance in firing rules.
Primitive data types:	information units such as integers, floating point, characters, etc.
Data structures:	vectors, arrays, lists, plexes, etc.
Structure operators:	information access mechanism.
Data operators:	information transformations.
Processor organization:	functional complexity, grain (size), speed, special vs. general purpose.
Storage organization:	address structure, space management, location of manager.
Processor/store comm:	physical distance, communication protocol.
Extensibility:	possible or not.
State of implementation:	self explanatory.

Figure 2-1: Machine evaluation criteria

2.1 MIT

The MIT dataflow project has the longest history of the existing data-driven computer efforts. The MIT effort has been most heavily influenced by the early theoretical work of Scott [38]. The MIT group has proposed four machine forms:

- form 1: Special purpose computer for signal processing,
- form 2: Extension of form 1 to include data structures,
- form 3: Large store version of forms 1 and 2,
- form 4: Full service time-sharing system.

Each machine form represents a significant increase in performance, usability, and complexity. To eventually reach form 4, the MIT project is investigating user programming languages, developing the organization of a form 2 machine, and constructing a prototype of the form 1 signal processing machine [26, 33, 34].

The machine language of the MIT dataflow computer is the DDF data-driven program schema. These Dennis Dataflow Nets [DDF] are best thought of as directed graphs. The nodes of the program graph are of two kinds: actors and links. Actors accept a set of operands on incoming links, then fire and compute a result which is placed on an outgoing link. Two types of links are provided: data links which conduct data tokens and boolean links which carry control information. In addition to directing the flow of data, links are explicit copy sites where tokens may be copied and passed to two separate destinations. Links may be connected in series to provide variable fan-out from cells. There are six major varieties of actors in the MIT machine language:

- operators: perform simple arithmetic and logical functions,
- identity: passes its input arguments unchanged,
- decider: applies a predicate to its input arguments producing a truth value,

- T-gate: passes a data token when its control input receives a true token,

- F-gate: passes an input data token when it receives a false token at is control input, and

- merge: selects a data token from the data input which corresponds to the truth value of its control input (two-way selection).

The MIT Dennis/Misunas dataflow processor is divided into five major subsystems:

- memory subsystem: holds instruction cells and operands,

- processing subsystems: arithmetic and logic units,

- arbitration network: a switching network to conduct operation packets from the memory subsystem to the processing subsystem,

- distribution network: a switching network to conduct data packets from the processing subsystem to the memory subsystem for storage, and

- control network: a switching network which carries control packets from the processing subsystem to the memory subsystem.

As shown in Figure 2-2, data operations to be performed flow counter clockwise from the memory subsystem, and are switched by the arbitration network to a particular processing unit in the processing subsystem. The processing unit computes the result. The data packet generated by the processing unit is then switched by the distribution network to the memory subsystem for eventual storage within an instruction cell. Control packets flow from the processing units to destination instruction cells through a separate communication network.

The control, distribution, and arbitration networks are indeed packet routing networks. The packets contain control information which the routing networks decode, eventually switching the packets to their destination. The networks are composed of arbitration and selection elements. Arbiters join paths together while the selectors perform

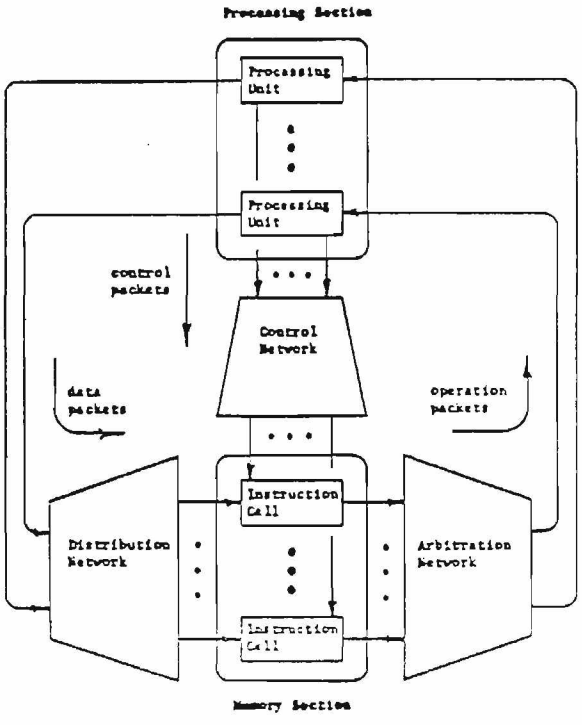


Figure 2-2: High level structure of the MIT dataflow computer

the "decode and switch" operation which directs the packets down the appropriate electrical path.

Tracing the execution of an instruction provides the best insight into the operation of the computer. Each instruction cell has the internal structure illustrated by Figure 2-3. Using cells of this type, the memory subsystem stores the internal representation of the dataflow program. One machine instruction is stored in the output control portion of each instruction cell. The three receivers store data values as they arrive at the cell. When the firing set of the cell has been satisfied, the cell transmits an operation packet. The packet which contains the instruction and operand data values, is routed by the arbitration network to the appropriate processing unit. In the MIT machine, the processing units are heterogeneous. That is, each of the processing units has a few dedicated functions which it performs very quickly. The arbitration network routes the operation packets generated by instruction cells to the processing units which

have the capability to calculate the result.

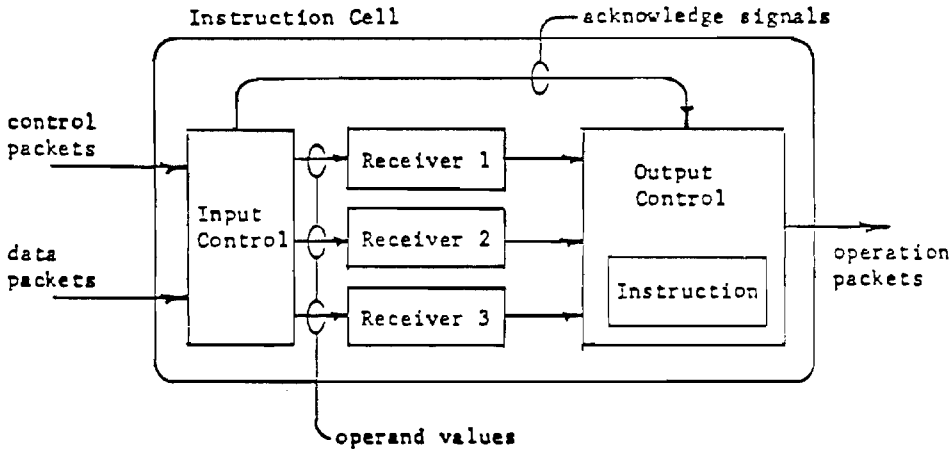


Figure 2-3: Structure of an instruction cell

Once the result has been computed, the processing unit generates one or more result packets. Result packets are either control packets bearing boolean values or data packets containing either integer or complex values. The destinations of the control and result packets are determined from destination identifiers contained in the packets. The execution cycle is completed as the newly arriving control and data packets induce additional instruction cells to fire.

The machine structure is a rather novel distribution of conventional computing functions. Instructions are not "fetched" from memory, but the operands are brought to the instructions. Processing of instructions is neatly separated from their sequencing. The transmission of packets within the machine is asynchronous, permitting the different computer subsystems to operate independently. This concurrency is effectively exploited because more than one instruction cell can be fireable at any instant. Dennis has presented a design for the arbitration network which allows many operation packets to flow concurrently through the network to the processing units.

Because the MIT machine depends heavily upon communications external to the subsystem elements and the elements themselves are of very low functional complexity, the design cannot achieve a high logic to pin ratio when realized in VLSI circuits. Packet communication speed will also suffer because the design cannot take advantage of the shorter on-chip propagation times. The extensibility of processing power and instruction storage is excellent. By expanding the arbitration and distribution networks, additional processing units and instruction cells are easily appended. However, expansion of the arbitration distribution and control networks may result in longer, serial chains of routing paths, causing somewhat longer packet transmission times. A summary table of the MIT architecture is shown in Figure 2-4.

execution model:	Data-driven
machine language:	Directed graph; computational, control, and routing operators. Form 1: scalar primitive types. Form 2&3: record, array and and self-referencing data structures.
primitive data types:	Integer, boolean, real, and character values (Form 1.)
data operators:	Functions, T-gate, F-gate, merge, identity, decider.
processor components:	5 subsystems: memory, arbitration network, distribution network, processing units, memory.
processing elements:	Special purpose dedicated processing units of low complexity.
address structure:	Instruction cells in memory are selected by decoding identifiers in the packets.
storage management:	Instruction storage is managed by the memory module.
communication protocol:	Packet switched; packets contain simple routing information.
extensibility:	Extensibility of processing function and memory is excellent at logarithmic cost.
physical extensibility:	Very good since intermodule communication is entirely asynchronous.
implementation status:	Form 1 prototype under construction. Higher level language (VAL [42]) has been designed and a translator/interpreter is operational.

Figure 2-4: MIT Dataflow Computer: Summary

2.2 Systeme LAU

Systeme LAU is a French dataflow effort at ONERA-CERT. Unlike the projects at MIT, Utah and UCI, the Systeme LAU group began their project with some specific high-level linguistic ideas [12, 29, 30, 15]. The designers were influenced primarily by the single assignment language of Tesler and Enea [40] and the work of Dennis [13]. At the highest level of abstraction, the Systeme LAU designers intend programs to be written in their higher level language. By using a higher level algebraic language, the complexity of detailed program graphs is suppressed. The Systeme LAU language uses the single assignment programming rule which states that a variable may be assigned a value at most once in a particular program context. Single assignment is semantically equivalent to the directed graph form of dataflow programs. The rule permits greater freedom to exploit the potential concurrency within a program.

The high level language of Systeme LAU is almost directly executed in the hardware. The statement forms each have a corresponding operator. Because the high level language is directly supported by the machine, the concurrency advantages of the dataflow model are not sacrificed by the algebraic nature of the programming language. The LAU architecture permits the exploitation of both pipelined and spatial concurrency. Pipelining is implemented within the individual hardware modules. Due to the single assignment rule, spatial concurrency is inherently expressed in LAU programs. It may also be explicitly denoted by the programmer with an EXPAND language construct.

All statements in the high level language are assignment statements. In place of variables, the LAU designers have substituted "objects". Objects are assumed to exist in a single assignment environment. That is, after an object has been written, it may be read

an indefinite number of times. It may not, however, be written more than once. Statement execution is ordered by data-driven sequencing. The most primitive operational notion in the LAU machine is that of the Data Production Set or "DPS". A DPS may be represented as a pair: the first component is a set of objects, and the second component is a set of statements which produces the objects in the object set component. A statement in a program is defined in terms of the DPS's which it produces, and the DPS's which the statement consumes.

There are six major instruction types: operations, loop, case, act (controlled invocation), expand (parallel execution fork and join), and call. Each instruction specifies the desired action, the identities of the operands, the destination of the result, and several condition bits. Instructions are not bound to execute on any particular processing unit. Data items in the system consist of a value and a variable number of addresses which identify the instructions which use the data value as an operand.

Systeme LAU is a multiprocessor architecture whose overall structure is given in figure 2-5. The current implementation is limited to one processor attached to a host minicomputer through an interface for input/output facilities. It is divided into three main subsystems:

1. Control units,
2. Memory subsystems, and
3. Up to 32 elementary execution processors.

The host minicomputer is used to develop programs and unload both the control unit memories and the memory subsystem.

The local memory subsystem contains several computational tasks represented as data production sets. Tasks are loaded when all inputs are available and terminated when all outputs have been produced. To increase memory bandwidth, the actual store is divided into 8

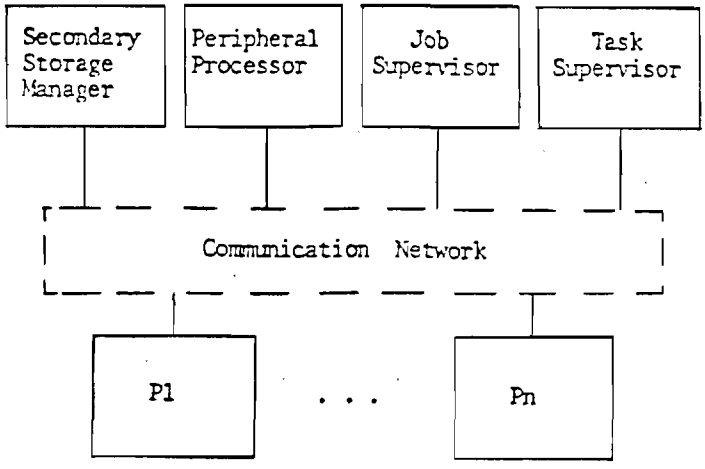


Figure 2-5: Overall structure of Systeme LAU

independent banks which operate concurrently. The memory control unit arbitrates accesses to the store.

The control unit performs the data-driven control sequencing of the processor, and its operation is rather unique. The control unit contains two memory units: the instruction control memory and the data control memory. There are three control memory bits and one data control bit for each word in the local memory. The three control memory bits signify the status (presence of operands) of the firing set for a particular instruction. When the three bits in a control memory word become enabled, the instruction at the corresponding word in the local memory becomes executable. The control memory is continually searched for for the presence of executable instructions. Ready instructions are read from the local memory and passed to the execution unit for subsequent interpretation. The data control memory indicates whether a particular datum has been calculated or is awaiting computation. When the data control memory is scanned, the

control unit determines which bits may be enabled in the control memory, i.e., which instructions have a particular operand awaiting execution of the instruction.

The execution unit is split into several independent asynchronous modules. Full parallel operation can be realized due to the data-driven nature of instruction flow. The execution unit decodes the instructions which it receives from the local memory and dispatches their execution to the appropriate instruction interpretation unit. The arithmetic unit has additional substructure consisting of several floating point execution units, fixed point arithmetic units, and a vector execution unit. These interpretation modules can operate concurrently. The control execution unit is composed of separate, asynchronous interpretation units, at least one unit for each of the control instruction types. Due to the unique properties of the machine language (i.e., data-driven sequencing and registerless instruction format) pipelining techniques are usable without any of their well-known drawbacks with respect to branching and synchronization.

The structure of the LAU machine has been influenced by a desire to use commercially available computers as components. As such it supports the most powerful set of primitive operations of any of the machines surveyed. It will also naturally have a number of support tools which will greatly aid the programming task., The machine is therefore not intended to be, and it is not reasonably amenable to a future VLSI implementation. The decision to start with a high level language was a good one. A major advantage of the LAU machine (like the MIT machines) is direct support of floating point arithmetic. Work by the Utah group has demonstrated the absolute necessity of floating point hardware to provide the performance required to solve real scientific problems [39].

execution model:	Data-driven
machine language:	Directed graph.
primitive data types:	Integers, floating point.
operators:	Arithmetic operations, loops, case, procedure calls.
processor components:	Control unit, execution unit, memory subsystem.
processing elements:	Interchangeable, homogeneous. Element substructure permits parallel operation of special instruction interpretation units.
address structure:	Linear.
storage management:	No special management technique.
communication protocol:	Simple addressing, request and acknowledge protocol.
extensibility:	Easy addition of processing elements. Memory control units permits addition of more store although speed of the unit will limit bandwidth eventually.
physical extensibility:	Extensibility not a goal; difficult.
implementation status:	As of November 1979, the Systeme LAU has been fabricated and tested at the subsystem level. The machine can currently execute programs and performance analysis is being conducted.

Figure 2-6: Systeme LAU: Summary

2.3 University of Manchester

Under the support of the British Science Research Council, the Manchester University (MU) dataflow group is currently producing a prototype dataflow machine. As a preliminary investigation into the relationship between a high level language and machine structure, the language LAPSE was defined [21]. The designers credit the Id language [24] and LUCID [31] for many of their language ideas. In particular, LAPSE is a single assignment language with many of the familiar higher level programming language constructs. The syntax is patterned after Pascal. The graph language of the machine, which is the executable program form, closely resembles LAPSE. Hence, the programming language has fairly direct support in the hardware.

The simple data types supported by the machine are Boolean, integer, and floating point values. Simple types may be combined into records and arrays similar to those of Pascal or PL/1. Some of the operators supported by the machine are:

- arithmetic operators: add, subtract, etc.,
- comparison: greater than, equal to zero, etc.,
- merge: join two arcs, passing tokens from exactly one input,
- pass on true: conditional transfer of token,
- pass on false: conditional transfer of token, and
- duplicate: replicate incoming data values along two paths.

The Manchester machine supports Pascal-like procedure calls. Unlike the MIT dataflow computer, procedures are not copied when invoked. Instead, an input interface and output interface are provided for each procedure corresponding to the usual notion of entry and return linkages. When a data token flows into the input interface, it is labelled with an identifier that indicates the source of the data token. Data tokens from the same source have the same identifier. Upon exit through the output interface, a dynamic link is created which

directs the output back to the locus of the procedure call. This mechanism effectively imitates the generation of new procedure code at the appropriate place in the program graph.

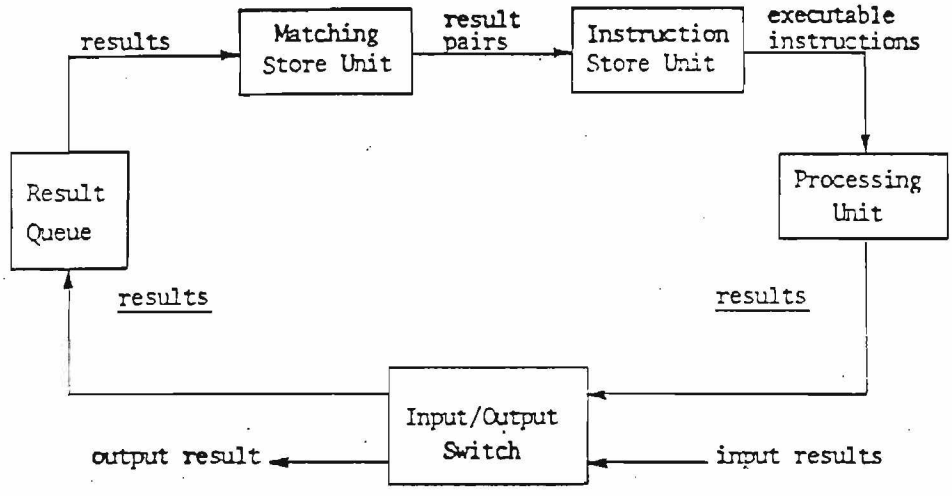


Figure 2-7: Structure of an individual Manchester processor

The Manchester architecture consists of a variable number of ring structured processors arranged around an exchange switch. The ring processor closely resembles the MIT machine in its circular flow of data (figure 2-7). However, no distinction is made between control and data information; a separate path for control information (e.g., the control network in the Dennis/Misunas machine) is not provided.

Two types of messages flow in the machine: instructions and tokens. Instructions are kept in the instruction store. Each instruction contains:

- the node function,
- the destination for each of the results that it produces which includes the instruction address and the number of operands expected at the destination.

Tokens are produced by the processing unit as a result of instruction execution or they are received via the input/output switch from some external source. They are stored either in the result queue or the matching store. Tokens contain their data value, eventual destination, and a label which is used to regulate procedure calls. Tokens are tagged with the type of the data value that they carry allowing dynamic type-checking.

The result queue is a rate balancing mechanism which attempts to smooth the rate of token production and consumption. Executable instructions for the processing unit are produced by the combined operation of the matching store and the instruction unit. The generation of an executable instruction proceeds in the following way:

1. The matching store removes a result token from the result queue.

2. If the token indicates that only one operand is expected at its destination, it is sent immediately to the instruction store unit.

3. If more than one operand is required, the memory in the matching store is searched for an entry with the same destination. The matching entry is deleted and the result pair is sent to the instruction store. Unmatched tokens are saved in the memory.

4. Token pairs are accepted by the instruction store. The destination instruction is read and transmitted to the processing unit for execution.

The interfaces between units are asynchronous permitting greater operational concurrency with all units operating in parallel.

In order to capitalize on the potential spatial concurrency of the dataflow programs, several rings are forged into a multi-ring structure. The Manchester computer is a multilayered integration of many ring processors arranged around an exchange switch. Within the context of the ring processor, the exchange switch corresponds to the input/output switch that provides external communications to and from

a given ring. Because rings are autonomous the processors may operate asynchronously and in parallel. None of the storage units are shared, removing performance limitations due to finite memory bandwidth. Input and output channels are naturally accommodated by this design. They simply mimic the input and output behavior of tokens.

The exchange switch has an uncomplicated structure. It consists of successive layers of token distribution, buffering and arbitration. Tokens are routed at each distribution layer according to a particular bit in the name field. By altering the routing bits, faulty processor units can be isolated until repair. Buffer layers decrease the effects of address interference or token "clashes" within the distribution stages.

The MU machine inherently requires a high amount of communication around each ring. In a VLSI implementation, the slow off chip speeds will tend to reduce system performance. This may be mitigated somewhat by the ability of each ring processor to be pipelined. Unlike the MIT machine, the MU machine's component units are rather complex and can utilize the scale supported by VLSI densities. The MU machine (like Systeme LAU) has rather direct support of high level language constructs: floating point operations, and structure operators. The logical structure of the exchange switch is simple and can be implemented with order log(N) circuit elements. However, the physical interconnection cost of these elements and delays through the switching network will significantly impact overall cost and performance. Figure 2-8 shows the summary table for the MU machine.

execution model:	Data-driven.
machine language:	Directed graph; direct support of LAPSE higher level language.
primitive data types:	Boolean, integer, floating point values.
structure operators:	Records and arrays.
data operators:	Arithmetic, comparison, token distribution and selection, function calls.
processor components:	Instruction store, processing unit, input output switch, result queue, and match-store unit arranged into a ring.
processing elements:	Processing unit has substructure of several homogeneous executable instruction processors.
address structure:	Instruction store is linearly addressable. Matching store has an associative memory organization.
storage management:	Matching store performs its own management.
communication protocol:	Packet switched; tokens contain simple routing information. Exchange switch performs interprocessor message transmission.
extensibility:	Exchange switch permits good extensibility through addition of more token routing circuitry and ring processors. Ring processors may be extended through addition of instruction memory and executable instruction processors.
physical extensibility:	Exchange switch interconnections tend to be complex. Asynchronous token transmission eliminates sensitivity to physical transmission distances.
implementation status:	Simulation of machine is complete. LAPSE to program graph translator is complete. A single ring prototype is under construction.

Figure 2-8: University of Manchester Dataflow Computer: Summary

2.4 University of California Irvine

The goal of the dataflow project at the University of California, Irvine (UCI) is to develop a machine which:

- fully exploits the advantages of large scale integration,
- utilizes a very large number of processors (greater than one thousand),
- significantly improves the structure and construction of software by rejecting those features of the von Neumann model which adversely affect programming.

UCI has proposed a new programming language called Id which supports the single assignment programming concept. The unique aspect of the UCI work is the "unfolding" interpreter which attempts to obtain still more execution concurrency through manipulations applied to the iterative and procedural constructs of the program graph.

The UCI dataflow system supports two classes of values: elementary and structured values. Elementary values correspond to the usual notion of primitive types, e.g., integers, reals, etc. Structured values, however, are recursively defined trees from which vectors, arrays and other complex structures may be defined. Two operators may be applied to structured values: select and append. The expression "select(x,i)" returns the ith subtree of the structure value x. The append operator joins structure values together to create instances of yet more complex structure values. In addition to the usual arithmetic functions, operators are provided for procedure application, conditional expressions, and loops.

The asynchronous operation permitted by a program graph is termed static parallelism. The "unfolding" interpreter attempts to find instances of additional dynamic concurrency by "unfolding" the graph.

Figure 2-9 shows an add cell "s" which has two sets of input operands queued at its inputs. Following the usual data-driven execution rules, the add cell will fire twice in succession producing

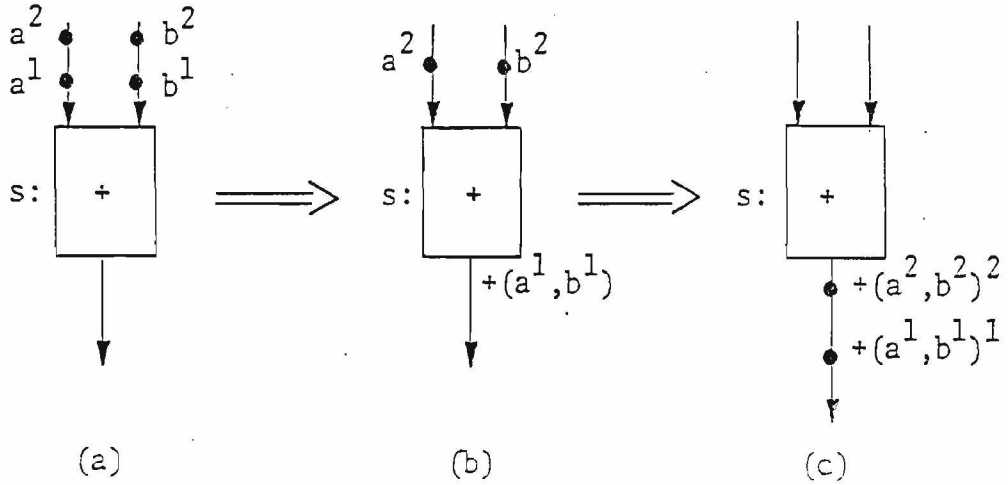


Figure 2-9: Example of static parallelism

two result tokens. (The superscripts indicate the logical traversal ordering of tokens on the graph arcs. Because an add cell is functional in the sense that future results do not depend upon the execution history of the cell (i.e., it is memoryless) the second addition could also be performed immediately. The addition of the second pair of operands could be physically completed before the first addition due to the asynchronous firing rules of the data-driven model. This single cell could be replaced by two physical add cells provided that the result tokens leave the expanded graph in the same order in which they arrive, as indicated by the subscript order. Subscript order can be maintained during execution through the use of tags which retain information regarding the token ordering. If the add cell is replaced by a loop or procedure invocation, even more concurrency can be realized. The interpreter is permitted to unfold all the expressions within the loop or procedure for parallel execution. If a large number of physical processors are available,

the unfolded expression will execute much faster than the regular form of the expression. This form of concurrency is called dynamic parallelism, since often the degree of unfolding cannot be determined until execution time, as in the case of loops.

The design of the UCI dataflow computer was guided by four major principles:

1. concurrency: "... It is more important to design for large numbers of slow but concurrent accesses than to design for a few accesses that are fast but sequential." [41]

2. distribution: Activities are spread over the available processors.

3. locality: Logically related activities should execute within close physical proximity, presumably minimizing communication time.

4. redundancy: Local copies of a particular structure can be used to improve the speed of access.

Because the designers view the existing dataflow architecture as a testbed for machine and language ideas, the architecture is changing. Future proposals will address modular expansion and fault-tolerance, issues which have been temporarily deferred.

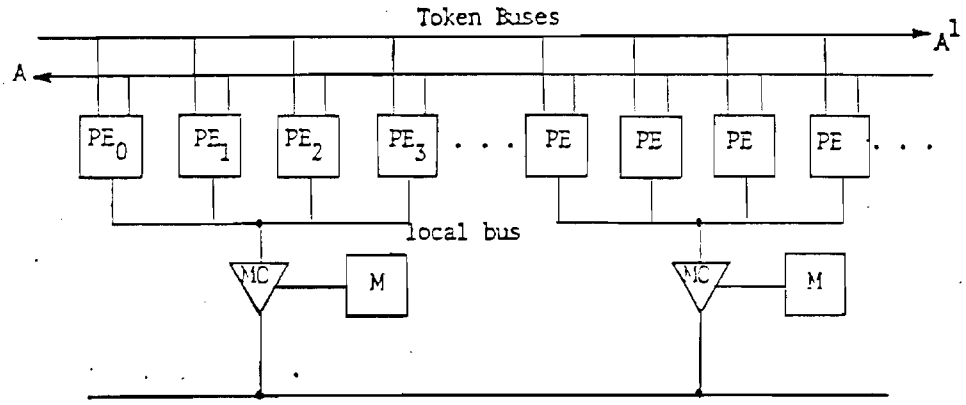


Figure 2-10: UCI processor - physical domain

The UCI dataflow computer consists of one or more physical

domains. A physical domain (Figure 2-10) is a network of one or more processing elements, a memory controller, and a memory storage module. The processing elements are connected to two shift register token busses which, together with the global memory bus, connect a number of physical domains to form a ring domain. The token busses are organized as counter-rotating rings. The ring is divided into a fixed number of token slots (one per ring per processing element), each of fixed length. The processing elements monitor the busses for tokens with their physical address. When a match occurs, the token is removed creating an empty slot. Any empty slot facing a processing element may be filled with an output token.

When a result has been computed by a processing element, the logical destination address for the result is mapped into a physical address through the use of an assignment function. Because more than one computational activity can be assigned to any given processing element, tokens must be sorted into activity groups. Tokens are labelled with an activity name. When the firing set of a waiting activity has been satisfied, the computation will be performed. Output tokens are queued to be transmitted in the next empty token slot.

Structure values reside in the local memories. Tokens need to carry only structure pointers, eliminating undesirable copy operations. The processing elements within a physical domain are connected to the memory controller through a local bus. Address interpretation and memory arbitration is performed by the controller. Although distinct memory units exist within the physical domains of the computer, the address space of the machine is unified. Hence, if a structure is contained within the physical domain which generated the access request to it, the structure value can be quickly accessed. If the structure exists within another physical domain, the local

memory controller can forward the access request to the appropriate distant controller along the global memory bus. Global communication can be reduced through the use of local structure copies, although the regulation of updates to a globally accessed structure (in the logical sense) becomes more difficult because all copies must be modified as the result of a single update.

The UCI machine, like that of MIT and MU, is communication intensive. This causes possible system contention for transmission. It also indicates that some of the speed advantages of a VLSI implementation will be lost. The fine grain search for parallelism creates a certain amount of additional overhead to do the "unfolding" style of interpretation. It remains to be seen whether this overhead is worth it in terms of total system performance. A summary table for the UCI machine is shown in Figure 2-11.

execution model:	Data-driven, unfolding interpreter.
machine language:	Directed-graph translation of Id programs.
primitive data types:	Integer, floating, Boolean, and string values.
structure operators:	append and select applied to recursive tree structures.
data operators:	Arithmetic, loops, conditionals, procedure application.
processor components:	Machine is partitioned into separate physical domains. Each domain contains several processing elements, a local memory controller, and a local store.
processing elements:	Non-specialized processing elements within a physical domain.
address structure:	Unified address space over the entire computer. Structure requests are satisfied locally if possible. Otherwise, structures are transferred from distant memory units via the global communication bus.
storage management:	Performed locally by control unit.
communication protocol:	Slotted ring network.
extensibility:	Easily extensible along token and global busses. Additional traffic may cause communication to degrade without the use of locally cached structures.
physical extensibility:	Ring busses are sequential shift registers. May be hindered by sequential timing of the net.
implementation status:	Extensive simulation.

Figure 2-11: UCI Dataflow Computer: Summary

2.5 University of Utah

The University of Utah Data-driven Machine (DDM1) is the subject of the case study presented in section 3. We therefore restrict our presentation here to a brief description of DDM1 in terms of the criteria set forth in the machine summary tables.

The machine language of DDM1 is a directed graph [4, 3]. The data tokens which circulate within the machine are list structures of arbitrary complexity. Using lists, both complex data structures (e.g., plexes, trees, etc.) and regular structures (e.g., vectors and arrays) can be represented. The operator set includes functions that concatenate, decompose and index the list structure of the data tokens. A high-level Graphical Programming Language (GPL) has been proposed for higher level programming. GPL supports a number of common programming constructs which can be translated to the low level machine language.

Physically, the DDM1 is a tree structured multiprocessor. Computational tasks, called data-driven processes (DDP's), are partitioned among lower level physical processors if the resources are available and the computational pay-off exceeds the amount of work required to dissect and transfer the program subnet to a subordinate processor. The processors in DDM1 are homogeneous, capable of executing any of the operator cell types. The processors consist of a processing element and a storage unit pair. This organization permits the exploitation of program locality.

One of the primary goals of the University of Utah effort is to develop a machine which is compatible with VLSI implementation and can be simply extended through the addition of more processor-store elements. Communications in DDM1 are packet switched and entirely asynchronous. Hence, the system does not need to be tuned when more computing elements are added. The summary table is shown in Figure

2-12.

execution model:	Data-driven, recursive machines.
machine language:	Directed graphs.
primitive data types:	Integers.
structure operators:	Tokens are list structures which may be concatenated, decomposed, and indexed.
data operators:	Arithmetic, and token routing.
processor components:	Processors are implemented as processing element and storage pairs.
processing elements:	Processors are homogeneous and capable of interpreting all cell types.
address structure:	Storage is organized as a list structured file.
storage management:	Management is performed within each storage element.
communication protocol:	Packet switched and fully asynchronous.
extensibility:	Addition of processor-store pairs is easily accomplished. Extension of local storage capacity has not yet been adequately solved.
physical extensibility:	Asynchronous communication eliminates sensitivity to physical separation.
implementation status:	Prototypes 1 and 2 have been constructed, debugged, and tested. New languages and machines are currently under definition and construction.

Figure 2-12: University of Utah DDM1: Summary

3. A Case Study of The DDM1 Machine

3.1 Introduction and Chronology

Data-driven machine #1 (DDM1) is part of an ongoing research effort to produce a working set of systems ideas for a highly concurrent, distributed control, computing environment. The project began around 1970 with an attempt to find a language for concurrent structured programming [5]. These ideas were refined in the following year to produce a data-driven program schema known as DDN's [4]. A translator was then written which would translate programs written in a lexically simplified subset of Algol to the functionally equivalent DDN programs. During the next 4 years, ideas about resource allocation and implementation strategies were developed. The DDM1 prototype [3] became operational in July of 1976. This prototype became the nucleus for an experimental data-driven environment where systems software and new hardware could be developed. In September 1977, the project moved to the University of Utah where it continues under a grant provided by the Burroughs Corporation.

The remainder of this section is a detailed case study of this project, and in particular the prototype DDM1 machine. We will (in a perhaps futile attempt to give an unbiased presentation) describe only work which has either been completed or is underway. We will deliberately avoid making claims about future wonders which we hope to produce. In attempting to improve substantially upon the von Neumann world, it is necessary to create a new set of self-consistent systems ideas. We outline this new set of ideas here, and then present the details and critique of the DDM1 implementation.

There are two primary ways in which the performance of traditional single sequential processor systems can be improved:

1. To use faster components in existing architectures, and
2. To design new architectures and programming methods, which

are capable of exploiting high degrees of concurrency.

The first approach is inherently limited in that the effects of reduced integrated circuit geometry and new logic families can reasonably be expected to increase overall system performance by only two orders of magnitude. While this is initially impressive, it does not meet the desired machine performance estimates necessary to solve large physics problems, or that needed for accurate weather prediction [37]. The second approach is not inherently limited by the physical properties of switching devices. The bound on the performance increase which can be obtained by exploiting concurrency is only inherently limited by the problem and human imagination (both on the part of the programmer and by the machine system designer).

There are numerous levels at which concurrency can be exploited in digital computers, i.e. multiple data paths, more concurrent realization of low-level circuit functions, overlapped and pipelined processing within a single processing element, multiple processors, etc. In developing any new "fast as possible" machine, it is important to attempt to implement all of these suggestions. We feel however that raw speed is not the primary reason for investigating the class of machines presented here. In fact, judgment of the merit of the data-driven approach on the basis of raw speed measures made on machines such as DDM1 is somewhat unfair to the new approach. Data-driven systems ideas are still in their infancy when compared to the long development history of Von Neumann structures, and therefore it is unlikely that the data-driven ideas will have reached comparable levels of sophistication. In addition, the actual performance of any of the proposed data-driven architectures is based on the assumption that the programs being executed contain very high levels of concurrency, and that the machine is composed of a large number of functional units. Such data-driven machines have not yet been

constructed in even a prototype form. Furthermore, the types of programs which are currently being run on prototype hardware or simulation systems are quite simple and do not contain tremendous amounts of exploitable parallelism. The best way to judge data-driven system concepts at the current time is to analyze the various system approaches in a qualitative sense, and decide whether or not these systems are likely to meet their stated goals. The DDM1 project has been mainly concerned with solving the problem of how to utilize and organize systems containing large numbers of independent processors. The result of this concern is that a number of low level issues relevant to the creation of single, very high performance processors have been neglected. Among these are, high speed arithmetic, fast internal circuit design, the use of high speed circuit families, etc.

The influence of integrated circuit technology permeates the entire spectrum of commercially available digital systems. It is clear that any machine architecture intended to have a general commercial appeal must be viable with respect to the changing constraints of integrated circuit technology. For architectures which fit nicely into the VLSI (very large scale integration) realm, the advantages are numerous. Among these are lower cost, increased reliability, increased speed, and decreased power consumption. The architectures which do not fit well into the VLSI world are at most only ideological points of interest or they may be successful only in a very special purpose sense.

There are a number of additional design goals which heavily influence the class of machine structures presented here. Namely it is intended that these machines be general purpose, extensible, reliable, easily programmable, support very high levels of concurrency, and also be economical with respect to their performance and existing technology. In particular this effort is not concerned

with one of a kind or special purpose machines. Special purpose machines are perhaps ideal for a given environment, but suffer from inherent limits in their applicability to other problems.

3.2 Architectural Principles

The implications of VLSI implementation constraints imply that machines which consist of a number of similar part types are attractive economically. In addition, if the systems can exploit a high degree of program locality, then higher performance can be obtained because on chip transmission paths are both short and fast (small capacitive load). Drastic additional speed up can be obtained if these systems can also support very high levels of concurrency. One approach to meet these objectives is to create an architecture which consists of a large number of identical processing sites. This approach has been taken by many architects and is particularly prevalent today where microprocessors are used as a replicated processing element in a number of different topologies. The DDM1 project had a number of design goals which made the use of existing microprocessor modules impossible. The design goals did require that the system support very high levels of concurrency and consist of a set of processing sites capable of performing localized storage and computation of a reasonable complexity. These sites should be essentially the same physical module, which can be constructed from one (ideal) or a set of chip types. An additional goal of the architecture presented here is that of extensibility, namely that DDM1 should consist of a finite but unbounded number of processing elements. More specifically, the architecture should be indefinitely extensible in the following way:

- Machine power should be enhanced by the addition of more processing modules (i.e. allow greater concurrency due to the increased number of processing sites);

- The addition of new modules should not require any change to the existing operating system in order to manage the

resulting larger system;

- Additional resources should be added simply by "plugging in new modules" without any special tuning of the existing hardware to create consistent system timing and communication for the expanding system; and

- Extensions should be available in small quantums.

The first and last points indicate that a user should be able to purchase only the power needed, rather than much more or much less than the amount desired. The other points demonstrate that the manufacturer should only need to support a single module, rather than a large number of system configurations and size ranges. If such goals can be met in practice then such systems would have an enormously attractive economic appeal to both the user and the system vendor.

Extensible systems cannot be implemented in a synchronous, centrally controlled manner. Central control of arbitrarily extensible systems implies that the control must be able to function on an arbitrarily large amount of state information, which either slows system timing drastically or requires controller modification to access the new state information. One major purpose of the multiprocessor approach is to increase performance, and a drastic slowing of system timing would therefore be considered unacceptable. Controller modification is equally unacceptable in that it conflicts with the stated design goals. In an arbitrarily extensible synchronous system the problem of unbounded clock skew (maximum difference in the perceived clock time between any two processing sites in the system) will cause failure. The systems described here will therefore be asynchronous, fully distributed systems. Fully distributed systems have the following characteristics:

- No module of a fully distributed system can determine the total system state, and

- No module of a fully distributed system can enforce

simultaneity in other modules.

Holt [13] has shown that the notion of total system state in complex asynchronous systems is intellectually counter productive. Furthermore the enforcement of simultaneity in physically separate, asynchronous devices is impossible.

There are many ways to organize an extensible set of modules in a distributed control system. One possible choice is a hierarchy. The advantages of hierarchical organizations are:

- A simplification in the amount of complexity to be dealt with at a given level.
- Verification by inductive methods can be done for uniform hierarchic systems.
- The natural superior-inferior relationships of elements in hierarchic systems can be utilized to resolve important multi-resource system problems such as contention for shared resources and deadlock.

It will be seen that hierarchy can be further exploited and facilitates a nice resource allocation policy. Recursive hierarchies are of particular interest. In a recursive hierarchy the structure of the system at one level of the hierarchy is the same as the structure of the system at any other level in the hierarchy. Recursion inherently implies that some element is defined in terms of itself. In the case of DDM1 it will be seen that this implies that the same module (and ultimately the same chip) can be used at each level. Recursive systems are nicely extensible. Clearly physical recursion must terminate at some point. This point will be seen to be the logically deepest set of resources in the physical hierarchy. Additional advantages of recursively structured systems have been demonstrated by Glushkov [18]. It will be shown that the width of a level in these recursive hierarchic structures can be used to execute independent operations, while the depth of the hierarchy will be used to facilitate pipelined operations.

It is also our basic belief that machine ideas and language ideas should be based on the same fundamental concepts to provide a nice "fit". This fit is an important property of a self-consistent set of systems ideas. In this case the local, hierarchic, asynchronous behavior of the architecture nicely fits intrinsic properties of data-driven programs.

3.3 The Machine Language

The machine language of DDM1 (Data-Driven Machine #1) is the DDN (Data-Driven Net) representation. It is not intended that anyone should program directly in the DDN representation, but rather that the language used for the actual programming be translated into DDN form for execution. It is possible to translate well-structured programs in conventional languages (ALGOL, FORTRAN, etc.) into DDN's, but these languages are not well suited to the specification of parallel algorithms. Some recently developed high-level languages are particularly well suited for the specification of concurrency, and can be easily and efficiently translated into DDN form [31, 42, 24, 29, 20].

The main advantage of DDN's over the other data-driven schemata [13, 8, 1, 27] is that no distinction need be made in DDN's between control and data. The lack of this distinction yields increased simplicity in DDN processes with no loss of representational power. In addition the DDN primitives, while not being any more numerous than those of the other low-level dataflow languages, are more general.

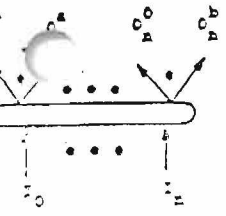
Seven distinct cell types are used in data-driven nets. A DDN cell type corresponds to the choice of statement types in conventional programming languages, and as such reflects a particular style and area of emphasis. Often times such choices are guided by theoretical considerations to find a minimal set or a maximally powerful collection of types. DDN cell type selection was influenced primarily

by pragmatic considerations. The DDN cells were chosen for simplicity and generality, and each cell type was chosen to clearly characterize a particular type of activity that was felt to exist in parallel programs. Each cell type is represented by a unique graphical symbol. Figure 3-1 shows the cell types, with their firing sets (the set of inputs which are required to be present before the cell type can be executed), and cell functions (a description of the functional action taken by each cell type). Notational conventions adopted here are:

- 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 receive different valued tokens.
- Superscripts indicate data paths which will carry identical valued copies of output data items.

Since each data item of a firing set is destroyed when the cell fires, any time a data item is to be used in more than one place (due to either pipelining or concurrency requirements), more than one copy of that output item will need to be be produced. This implies that the destination for any output may be a list of destinations. If there are n elements in the list, then n copies of the result data will be produced and sent to the respective destinations. Note that input paths will never have superscripted names, but outputs always do, indicating that any output result may be copied many times.

The SYNCH cell allows parallel streams of data to be synchronized. Such synchronization may indicate that a number of concurrent activities have reached a certain point or state. For example, the inputs to some process may be produced by a number of parallel processes. The inputs could be passed through a SYNCH cell to the called process. If it were desirable to determine whether the called process is ready to be executed or not, then it would be possible to look at the input SYNCH cell and determine the answer to the question. If the SYNCH cell were fireable then the called process



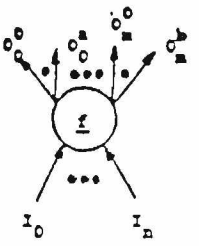
FORK CELL

I_j: inputs

O^i_j: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every i, j: $O^i_j := I_j$



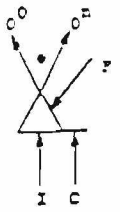
OPERATOR CELL

I_j: inputs

O^i_j: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every i, j: $O^i_j := f(I_0, \ldots, I_n)$



GATE CELL

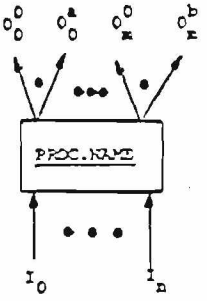
I: initial input

F: feedback input

C: condition input

O^i: outputs

for the cell function and firing set see Fig. 4.



CALL CELL

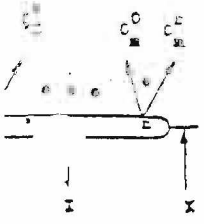
I_i: inputs

O^k_j: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every a, b

$O^a_b :=$ PROC.NAME $(I_0, \ldots, I_n)$



DISTRIBUTOR CELL

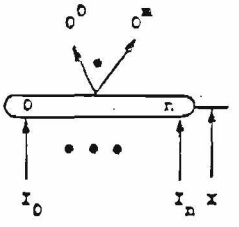
I: input

O^i_j: outputs

X: index

firing set: (I, X)

cell function: $O^i_x := I$ for all i and where x is the value of X



SELECT CELL

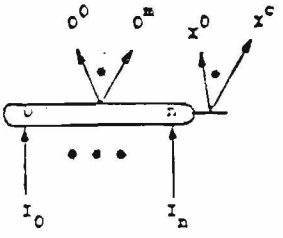
I_j: inputs

O^i: outputs

X: index

firing set: (I_x, X) where x is the value of X

cell function: $O^i := I_x$ for all i



ARBITER CELL

I_j: inputs

O^i: outputs

X^k: index outputs

firing set: at least one input: I_j

cell function: $O^i :=$ first I_j; $X^k := j$ for all i, k

(Note: in case of a tie any input I_j which is present is chosen).

Figure 3-1: DDN Cell Types

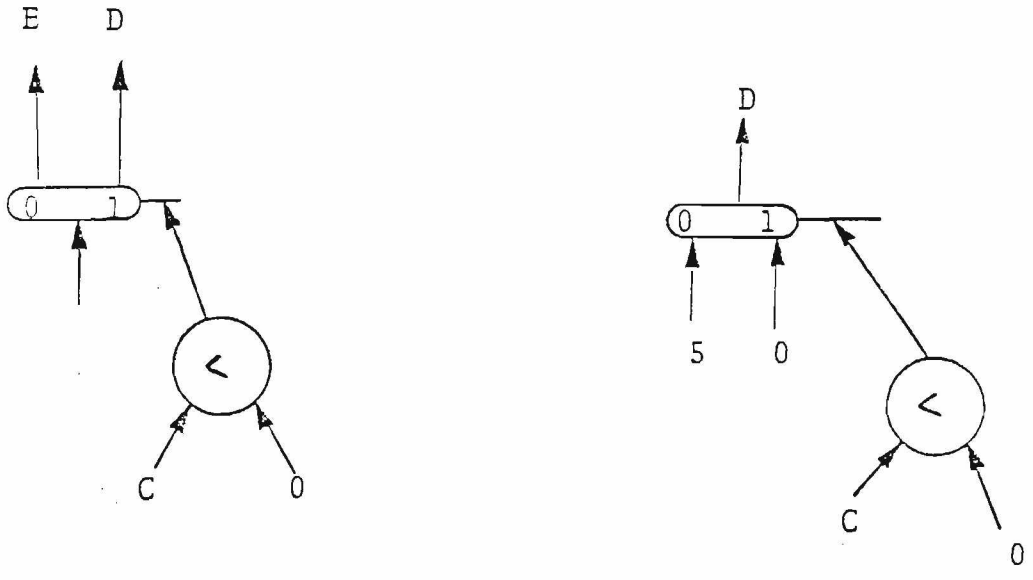
could be considered to be fireable. If no such input SYNCH cell existed, then the decision would be much more complicated, in that it would require the ability to perceive action in all of the distributed parallel producers of the called process' inputs. This information would then have to be correlated to create some consistent view of the state of the system. It has already been stated that such a global state view is counter productive. In this sample case, the state information is very expensive to obtain, and a better solution exists (the SYNCH cell).

The OPERATOR cell is used to perform normal operations such as arithmetic, relational tests, etc. The OPERATOR cell type is actually a class of individual OPERATOR cells. The actual operator definition is made by a small symbol placed in the OPERATOR cell, which indicates the actual operation (e.g. '+' indicates add, etc.). Each operator type will inherently indicate how many actual inputs and outputs will be required. For example addition is normally thought of as a 2 input and 1 output operation. The LISP operation CAR, on the other hand, is a single input and double output operation. Any DDN output may be further replicated for the reasons previously discussed.

The CALL cell is used to invoke a named data-driven process, and in that respect acts just like call constructs in languages such as ALGOL. If the firing sets and the cell functions of the CALL cell and the OPERATOR cell in Figure 3-1 are compared, then it is apparent that the OPERATOR and CALL cells act in very similar manners. This similarity allows CALL cells and OPERATOR cells to be semantically interchangeable in DDN programs. In essence they both perform functions, the only difference being that an OPERATOR cell indicates that the function is a machine or system provided function whereas the CALL indicates that the function being performed has been defined by the programmer. The CALL cell also allows the usual program hierarchy

to be constructed.

Conditional expressions in DDN programs are implemented by the conditional routing of data items to the desired parts of the program graph. There are two DDN cell types which can be used to perform this task, the DISTRIBUTE cell, and the SELECT cell. The DISTRIBUTE cell allows an input data item to be routed onto one of n (programmer defines n) possible outputs. The desired output is specified by the index input. If the index value is out of range then an error condition exists. The method for handling errors will be described later. The SELECT cell allows one of n inputs (again n is a variable and is specified by the programmer) to be selected and placed on the single output. The input to be selected is specified by the index input to the SELECT cell. Two simple programs are shown in Figure 3-2 which illustrate the use of SELECT and DISTRIBUTE cells to represent conditional program constructions.



a) If C < 0 then D:=1 else E:=1 b) If C < 0 then D:=0 else D:=5

Figure 3-2: Two DDN methods for representing condition statements

In a concurrent environment, there are times when two parallel

events will need to be merged on a first come first served basis. This action is performed in DDN programs by the ARBITER cell. The primary problem with such a merge is that information is lost about the sources of the merged data items. The index output of the ARBITER allows the information about this choice to be preserved. This information can be used in DDN programs to create program structures which are deterministic in their behavior. A sample program which exhibits such a construction will be shown later in Figure 3-8.

Another form of merge operation is required in iterative DDN programs and is provided by the GATE cell. Unlike the other cells, the GATE cell operates on the basis of an internal state. The cell function and firing set are functions of this internal state. Iterative DDN's appear as a net with a set of input paths I, a set of feedback paths F, and a set of output paths O. The recycling of data on the F paths is done once for each iteration of the iterative program. The program is initially started with the values which arrive on the I paths, and when the iteration terminates the results of the program are sent out on the O paths. The basic function of the GATE cell is to perform a merge operation on inputs I and F, as specified by the input C. The value on the C path corresponds to a condition which controls the iteration. A TRUE value indicates that another pass will be taken through the body of the iterated code. This will result in a set of feedback values on the F paths. If the C value is FALSE then the iteration is terminated and no new feedback values will be produced. Initially the GATE cell's state is set "open" to pass a single I data item, then the GATE "closes" to inhibit further I's and allow F's to pass. After a C=FALSE input arrives the GATE again opens. The general form of an iterative DDN program contains the following:

- A net or process to be iterated,
- A set of initial data paths,

- A set of feedback data paths, and
- A set of output data paths.

This structure is illustrated in Figure 3-3.

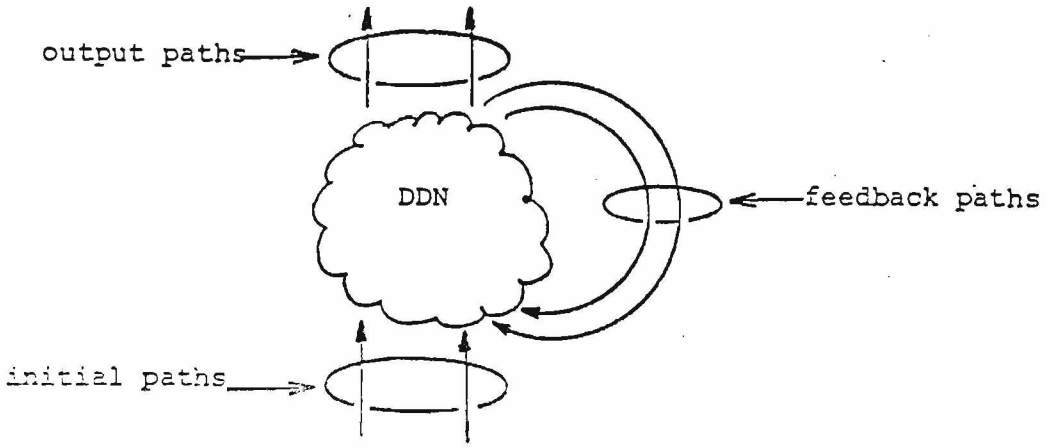


Figure 3-3: Data-Driven Iteration

Proper sequencing for such an iteration would be:

- When each initial data path has an item, the net fires.
- When the net has fired, output items are placed on the feedback paths, and the net is then primed to iterate.
- Step 2 is repeated until the iteration started by the first set of initial inputs terminates and produces outputs.
- The sequence is then restarted.

The GATE cell is used to prevent non-deterministic merging of data paths in iterative situations. Loop termination is implemented by the joint use of DISTRIBUTE, and OPERATOR cells. A sample iteration is shown in Figure 3-4, which increments a value iteratively until it becomes 3, and outputs it. Data items not delimited by parentheses are of type CONSTANT and are considered to be part of the program definition. Constants are therefore not destroyed by the firing of a cell.

The situation where several data paths terminate at a single

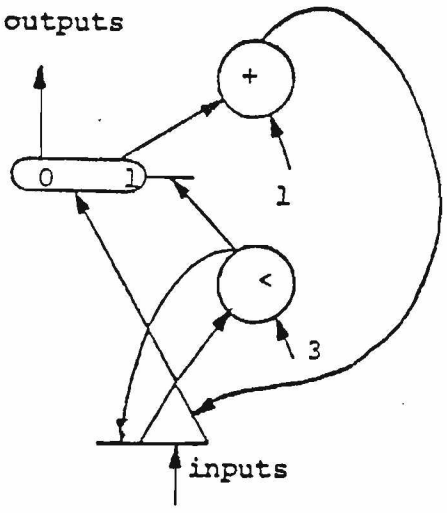


Figure 3-4: A Simple Iterative Net

destination is not allowed, as this would imply that non-deterministic merging could occur at such a junction. Merging of data paths is allowed only in well-controlled instances as provided by the GATE, SELECTION, and ARBITER cells.

As in other data-driven schemata, DDN's naturally represent two types of concurrency (pipelining and independent operations). DDN's, due to their asynchronous behavior, are non-deterministic with respect to execution histories. However the important point is that in a functional sense, DDN structures are deterministic. That is, if a sequence of values is sent into a DDN program any number of times, the resulting sequences of output values will always be the same. This property is known as output functionality. Certain DDN program topologies are not output functional and correspond to a programmer error. It is possible in a "DDN compiler" to find all such non-deterministic topologies and flag them as errors.

The inherent operational asynchrony of distributed control systems makes it virtually impossible to recreate an error situation for debugging purposes. It is therefore important to be able to guarantee correct system operation through analytic techniques.

Analysis of DDN structures is in general very difficult. For example it is difficult to answer such questions as:

- When does a DDN begin execution?
- When does a DDN terminate?

Answers to these questions require global perception of the net activity. Since DDN mechanics are defined in local terms, global behavior cannot be perceived. The general problem lies in the lack of ability to observe state phenomenon in fully distributed systems.

If all of the input paths to a DDN are synchronized at an input SYNCH cell, then important state information can be observed locally at that point. A similar argument can be made about the output paths. The "delimiting" of a DDN by SYNCH cells yields a form for a data-driven process (DDP). This process form is shown in Figure 3-5.

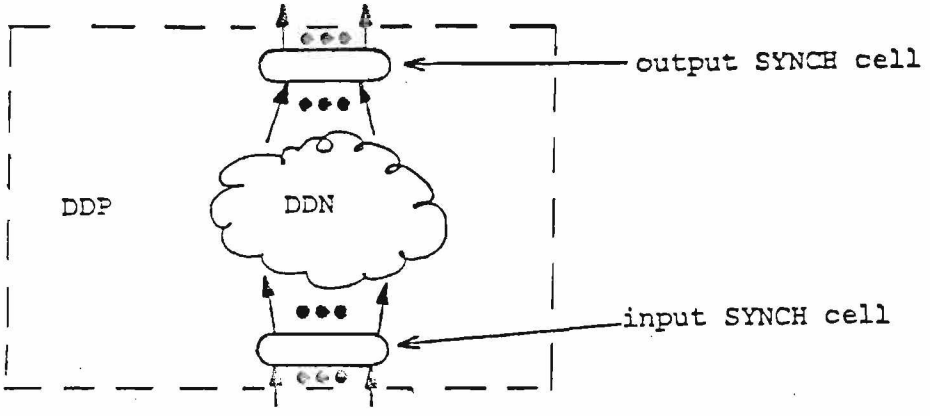


Figure 3-5: Data-Driven Process Form

The DDP form simplifies comprehension of the program considerably. The single input SYNCH cell acts as a collector for the process working set, and the output SYNCH cell acts as a termination point for the process and a distributer for the results of the process. In addition, DDP's have the same operational characteristics as CALL and OP cells. In fact, CALL cells invoke only DDP's.

When the input SYNCH cell becomes fireable, then the DDP is said to be fireable. When the output SYNCH cell has fired, then the DDP is said to have terminated. Between these two times, the DDP is said to be active. Under pipelining, the definitions of active, fireable, and terminated have to be modified as there may be many instances of each. A new definition for termination may be made by counting the number of input SYNCH cell firings and the number of output SYNCH cell firings. When these numbers are equal then the pipelined DDP can be said to have terminated.

Since a DDP exhibits the same behavior as a simple OPERATOR cell, a clean substitution rule can be formulated. Within any DDN, a DDP which performs a function F may be substituted for any OPERATOR cell performing F without changing the functional operation of the original DDN. This substitution rule allows a call mechanism to be defined (the CALL cell), which allows for recursive and/or hierarchically defined DDN's and DDP's. The name of the called DDP is indicated inside the CALL cell box.

While the DDP model has some very nice properties with respect to abstraction, substitution, and hierarchical structure, it is more limited in what it can represent than the more general DDN's. The general problem is that DDP structures use one input set for each output set produced. Many problems use an arbitrary number of inputs to produce a single output (or vice-versa). This problem can be corrected using a type of message structuring discipline known as streams [43]. A discussion of streams and the operations of DDN programs using streams as an intrinsic data structure type is not presented in this chapter, as it is felt that such a discussion is not necessary to understand the material presented here. Due to the functional form of DDP's, the formal verification of program properties is simplified.

Figure 3-6 shows two DDP's, each containing two parallel recursive calls to calculate the nth Fibonacci number, for positive integers n. Figure 3-6a shows the obvious net, while 3-6b shows a net which will execute as fast with two processors as 3-6a does with three processors (assuming that > and - operations require equal time to compute).

Find the nth Fibonacci number:
where F(1) = 0
F(2) = 1
F(n>2) = F(n-1)+F(n-2)

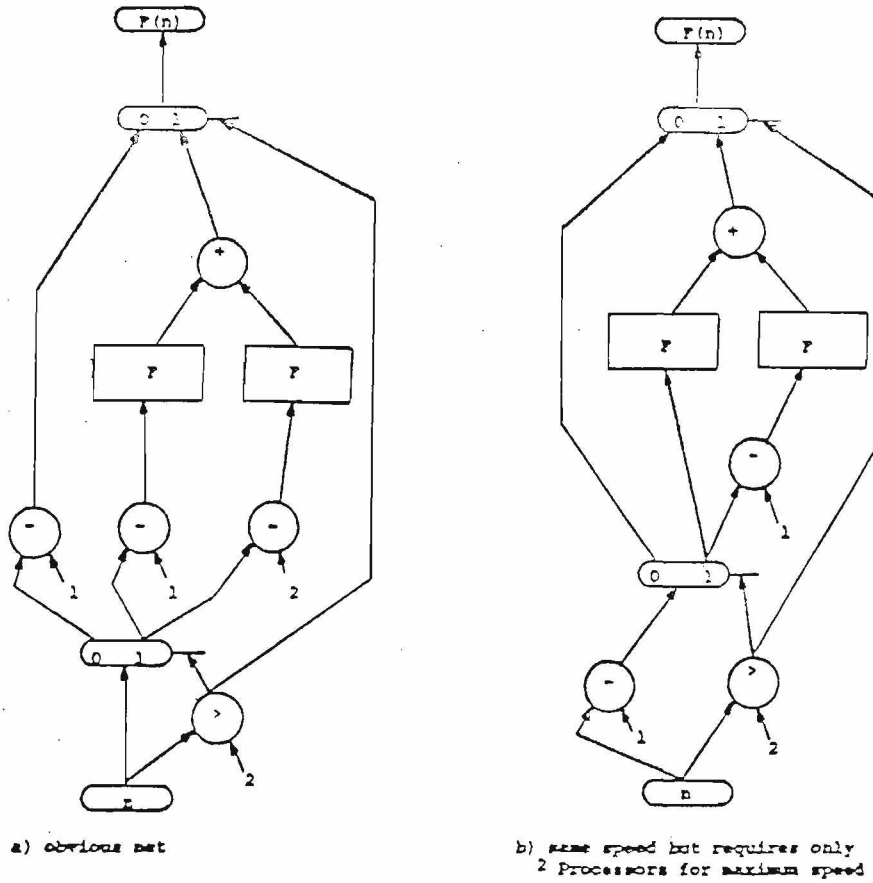


Figure 3-6: Fibonacci DDP's

A detailed discussion of data structure handling is beyond the scope of the issues discussed here. It is appropriate to mention a few considerations relating to better program structure. One can consider DDN's to consist of two files:

1. A static file (so far - the net description), and

2. A dynamic file (until now - the data items).

A more general approach is to allow the data item file to be either static or dynamic (and similarly for the net description). The basic nature of data-driven computation indicates that the dynamic file elements will be destroyed upon cell firing, and some copying will be inherently necessary. The proper choice for the dynamic file would be the file (data item or net) which would minimize the copying requirements. In instances where large data structures are used, the static file would be the data structure and the net description would be the dynamic file. In this instance the data structure would be treated as a static resource which could then be shared by a number of concurrent processes. To avoid the possibility of access conflicts to the structure, an ARBITER cell can be used to guarantee first come first served (but sequential) access to the structure.

Figure 3-7 shows a net for controlling read access to a shared vector. The inputs P1, P2, and P3 are the indices from concurrent processes 1, 2, and 3 respectively. The vector input is the vector to be loaded into place. It is assumed that the load input arrives before any of the Pn inputs. The SHARED RESOURCE box of this net acts as a sequential interpreter for instructions flowing into it. The net also shows how order-preserving parallel to serial to parallel conversion takes place using the ARBITER and DISTRIBUTE cells. The DDN ARBITER cell does not perform just the normal arbiter function, but also generates an index indicating which input was selected. This index preserves enough information to allow the sequenced items to be correctly "reparallelized". Any time an ARBITER cell is used in a net, it must be used in exactly the same ARBITER - DISTRIBUTE cell pair topology as shown in Figure 3-7. Otherwise the ARBITER cell will cause non-deterministic sequencing and the result will be a net which is not output functional. Figure 3-8 illustrates how order-preserving

serial to parallel to serial conversion is handled.

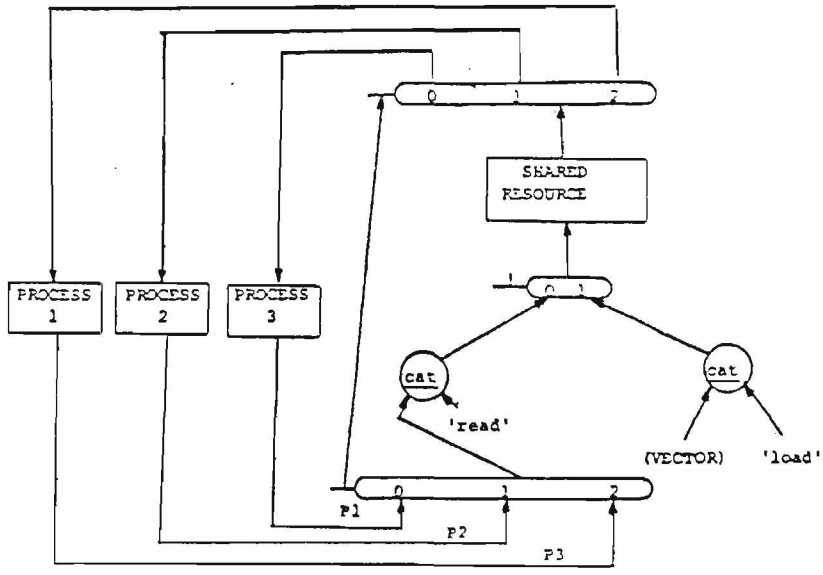


Figure 3-7: Shared Resource DDN

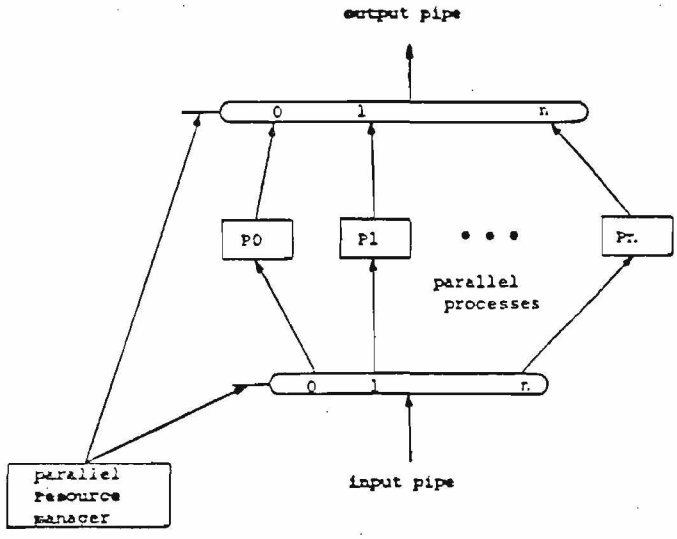
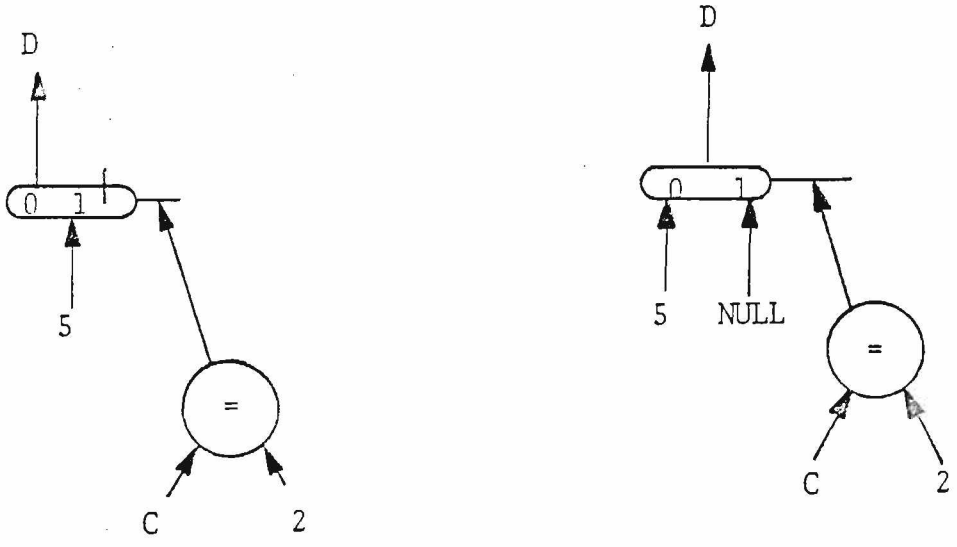


Figure 3-8: Serial to Parallel to Serial Conversion

The basic nature of data-driven processes is that operations are pushed into action by the arrival of the required set of inputs. If one of these inputs is prevented from arriving at the intended destination (due to a programming problem or other type of error), then that destination cell will never fire. Consequently, all cells having firing sets with outputs from the unfireable cell will never

fire and so on. A cell or a net which can never fire is said to hang. A cell or net which can never hang is said to be live. Since no cell can determine whether it is waiting for an input that will never arrive (i.e. whether it is live or not), it is important to be able to guarantee liveness from a topological examination of the net process. This examination can be performed in a compiler-like operation and a special ERROR data item can be inserted where necessary to maintain liveness. For example, if a DDP contained a conditionally defined output, then that DDP could hang if the condition was never met. The compiler (or the programmer) could produce a net which, on failure of the condition, would send an ERROR item. An example of such a net is shown in Figure 3-9.



A net which can hang

A corrected net which can not hang

Figure 3-9: Correcting a hangable net

When a conditional expression is described as a DDN, only one path of the condition will fire for a given set of inputs. For this reason, the notion of whether a particular cell is live or not is not of much practical worth, and in fact, it is impossible to determine topologically. Similarly for general DDN's the notion of liveness is somewhat nebulous, but for a DDP, liveness is an important and

topologically verifiable property.

Two other important characteristics of DDP's are whether they are safe or clean. A DDP is said to be clean if when it terminates, there are no non-constant data items existing in the DDP. DDP's are clean when they are defined. If they were not, then the output values would be history dependent upon the values of the existing non-constant data items. A live DDP which terminates without error and is always clean is said to be safe. It can be shown that safe DDP's execute in an output functional manner under pipelining.

It is possible to determine by topological analysis of any DDN whether it is safe or not. The machine algorithms for such analysis are lengthy and will not be presented here. Such analysis would be an important part of a DDN compiler, and should be performed before execution of any DDP.

3.4 The Architecture

The architecture consists of a set of asynchronous modules which communicate by passing messages. The fundamental computational unit of the architecture is a processor-store element (PSE). A PSE consists of a processor module (P) and its associated local storage module (S). Any PSE can execute any machine language program, providing that it has a sufficient amount of local storage. The architecture is a recursively organized set of these PSE's. The recursive definition of the structure is:

$\langle PSE_n \rangle$ ::= $\langle P_n \rangle \langle S_n \rangle$

$\langle S_n \rangle$ ::= $\langle ASU_n \rangle$

$\langle P_n \rangle$ ::= $\langle AP_n \rangle$ | $\langle AP_n \rangle \langle PSE.GROUP_{n+1} \rangle$

$\langle PSE.GROUP_{n+1} \rangle$ ::= $\langle PSE_{n+1} \rangle$ | $\langle PSE_{n+1} \rangle \langle PSE.GROUP_{n+1} \rangle$

Subscripts denote the recursive level at which the module physically resides. <AP> is an atomic processor module, which has no further sub-structure (contains no PSE's). Similarly an atomic storage unit <ASU> has no PSE substructure. The width of a <PSE.GROUP> has a physical bound. For the DDM1 prototype, this bound is eight. The structure is depicted in Figure 3-10.

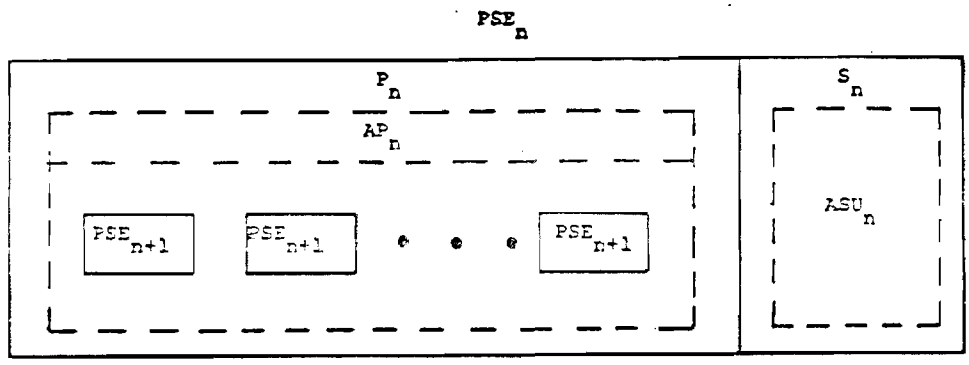


Figure 3-10: Recursive definition of a PSE at level n

This structure allows for a hierarchical distributed storage organization. Any S or ASU may consist of an arbitrary amount of storage of any desired medium. Higher levels of PSE's are considered logically superior to lower level PSE's. It is advantageous if higher level stores (ASU's) are slower and larger than the stores of lower levels. This reflects the notion that the functional ability of the father node should be greater than or equal to the abilities of all of the son nodes. The interface and functional ability of any ASU (regardless of size, speed, and level) is the same. The structure also allows for an arbitrary number of processors to be used concurrently. It is important to note that all AP's are identical regardless of level. However, the processors at higher levels will be more powerful, in that they contain more substructure than the processors at lower levels. More substructure implies more internal

concurrent processing capability.

Viewing the DDM1 system in this recursive sense is the best way to understand the operational nature of the system. The recursive view is somewhat disconcerting when it comes to understanding what the system looks like physically. When viewed non-recursively this structure is simply a tree structure with a single root and a possibility for up to eight sons at any node. Each node of the tree is a PSE and is capable of executing any machine language program. The leaf nodes have no substructure and therefore consist of an AP and an ASU. At each node the fan-out is fixed but the depth of the tree is arbitrary. In this manner the architecture allows any desired number of PSE's to be included into a given machine. Each PSE is a completely asynchronous module, and therefore as new PSE's are added to the system no system tuning need be performed. The desired goal is for machine performance to improve with the addition of more PSE's.

There are a number of ways to map this logical tree structure onto a collection of PSE's. All involve some form of a connection network to implement the desired communication paths. A number of general interconnection networks have been considered: busses, crossbars, Banyan nets [12], and permutation networks [17]. For tree-like machines, full connectivity is not required. The expense of crossbar switches vary as the square of the connected elements. Bus conflict (and therefore bus contention) would drastically reduce actual parallelism in the machine. Permutation networks present a tremendous problem in that they may need to be totally reconfigured when a single new connection is necessary. This is difficult to do reliably in a multi-path distributed control environment. Banyan networks have some merit, but do not allow the desired hierarchic pipelined communication. Therefore in the DDM1 prototype, a simple 1 to 8 switch was chosen as the interface unit between successive levels

of PSE's. The result is that the physical and logical recursive structures are the same. The structure is fixed and cannot be dynamically changed.

Information is passed between PSE's as messages which are variable length character strings. Upward traveling messages are passed by the switch in an arbiter like manner. Downward going messages contain header fields which indicate their destination. This header is processed by the switch hardware in order to enable the desired routing through the switch. The header is also deleted by the switch hardware as the message is passed. Downward and upward messages are manipulated by independent hardware in the switch, and therefore are capable of being controlled concurrently.

The character serial nature of the machine has the following advantages:

- Hardware modules are made simpler and are more applicable for VLSI implementation due to the reduced pin count.

- Hardware communication paths are more general in that information units can be transmitted as varying numbers of fixed-width base characters. This facilitates a hardware substitution strategy for modules. Each module can interpret the variable length message and perform the indicated function.

These advantages aid in greatly reducing the cost of the hardware modules. Some low-level performance is lost by doing everything serially. The philosophy for this architecture is to regain that lost performance many times over by providing a systems organization that allows for many highly concurrent levels of activity.

Physical queues are placed between levels of PSE's in order to facilitate pipelining and increase physical module independence. Without queues, the sender of a message would need to wait on receiver availability. If a queue becomes full, only then must the sender wait until the receiver has released some queue space. If queue sizes are

adjusted so that a sender is rarely required to wait for space, then the system would be well tuned for efficient processing. Optimal queue size depends on the average message length. It is therefore impossible to guarantee that no waiting will occur. It can be shown that the strict hierarchical control and a restricted process structure insures that the system does not deadlock. A block diagram of the PSE structure is shown in Figure 3-11. In the DDM1 prototype, all communication paths except for the path between the ASU and the AP, consist of 6 wires (a 2 wire request-acknowledge control link and a 4 wire, character-width data bus).

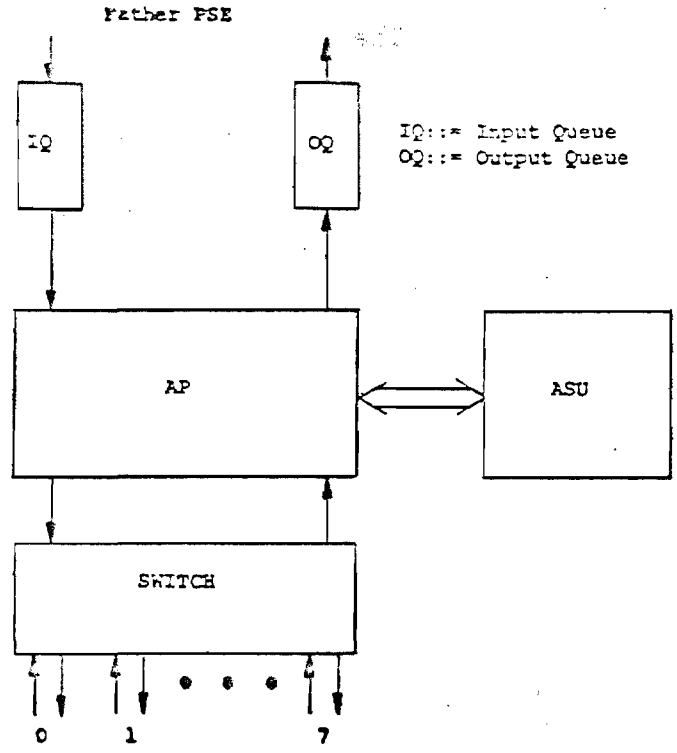


Figure 3-11: PSE Structure

The variable length, character serial message structure and DDN representation requires a highly flexible storage structure. This needed flexibility has been implemented in the low level hardware functions of the ASU. In order to increase efficiency of the PSE, all storage management functions are performed internally by the ASU. The ASU appears to the AP as a variable field length file system, which

directly executes commands, such as: initialize, skip, insert, read, write, delete, and index. The free space is managed automatically by the ASU.

The PSE structure allows for a high degree of processing locality in that any PSE can execute any DDN program (assuming that there is sufficient storage in its local ASU). In addition the PSE admits nicely to VLSI implementation. The 1 to 8 switch can be implemented using a cascaded set of 1 to 2 switches. Using 1:2 switches, module complexities for the DDM1 prototype (pin and gate count) are shown in Figure 3-12. These pin counts include connections for power, ground, initialization, and extension. The indicated module pin counts are rounded up to coincide with standard package sizes. A gate in these counts corresponds to a 2 input NAND gate.

Module	Gate Count	Pin Count
IQ, OQ (4K)	3,000	16
Ap	20,000	64
ASU (4K)	47,000	64
1:2 Switch	2,000	64
Ap + ASU	67,000	64
AP+ASU+IQ+OQ	73,000	64
AP+ASU+switch	69,000	64
PSE	75,000	64

Figure 3-12: PSE Module Complexities

These counts are well within the limits of modern circuit and packaging technology, especially since much of the logic is storage and therefore has a highly regular geometry.

3.5 Internal structure of DDM1

The internal structure of DDM1 is a rather flexible prototyping architecture. A number of decisions were made to give a large degree of flexibility to what is a special purpose data-driven machine. Some of these decisions result in decreased performance, but are considered to be worth it as the hardware is continually being upgraded to reflect new implementation ideas. In this section we present the fundamental structure of the atomic storage unit (ASU) and the atomic processor (AP). The other components in the PSE are simple queues and switches of a very straightforward design.

Inherent in a DDN process is the need for the net to grow and shrink in size during execution. This, and a general belief in other advantages associated with a variable field size storage representation, led to the adoption of The Storage Model (TSM) as the basis for storage in DDM1. TSM is a storage structuring discipline invented by R. S. Barton. DDN processes exist in DDM1, as "Storage Model" files. The goal of the TSM discipline is to provide a location independent method for dealing with an arbitrary structure of variable length fields.

The TSM structure is a field. A field is a variable number of characters enclosed between two reserved characters. These special delineation characters may not appear in the data, and will be denoted by left and right parentheses. A field may also be a sequence of any number of fields enclosed in parentheses.

TSM structures appear as well nested parenthesized expressions. One view of these structures is that they represent data structures, which are generalized trees. Straight forward mappings of scalars, tuples, strings, lists, vectors, and n-ary arrays can be made onto these tree structures. Any field which does not contain subfields is called a record, while any field which contains some subfields (i.e.

has substructure) is called a file. In any file the first subfield (which may itself be a file) is the descriptor for the remaining subfields. The remaining subfields are the contents of the field. A simple TSM vector may be represented by the template:

```
((ordered vector)(element value 1)(value 2) ... (value n))
```

Similarly a matrix may be represented by:

```
(((ordered matrix)
  ((ordered row 1)(value 1) ... (value n))
  .
  .
  .
  ((ordered row m)(value 1) ... (value n))
                                          ))
```

The non-descriptor fields of any file may be ordered or unordered. Unordered fields must be named and are accessed by name. Ordered fields may also be named, but can be accessed either by name or by position via an index. TSM fields may contain an arbitrarily deep substructure and are indexed by an access vector. Each element of the access vector may be either a name or an index number. If the nth element of the access vector is a number then the TSM file must be ordered or an error will occur. If the nth element of the access vector is a name, then there must exist an equivalently named subfield in the TSM structure being accessed. If no matching name exists at that level, an error condition will also occur. One notational convenience in specifying TSM structures is allowed. Any pair of back to back parentheses may be replaced by a comma. Hence ((6)(4)(5)) can also be written as (6,4,5).

TSM also specifies how free space is stored and used, but these details are not germane to the programmer's world and are omitted here.

The ASU is simply a TSM file system. The file commands which the

ASU performs are:

- Initialize - initializes memory contents.
- Skip - cursor skips over field currently under the cursor.
- Insert - inserts field prior to <the character or file pointed to by the cursor.
- Delete - deletes field pointed to by the cursor.
- Assign - assigns a character or a file to the character or file pointed to by the cursor.
- Read - reads the field pointed to by the cursor.
- Head - positions the cursor to the leading "(" of the father field of the character currently under the cursor.
- Shift - increment cursor.
- Aindex - (absolute index): does indexing operations starting at the first character in the store.
- Rindex - (relative index): does indexing operations starting from the current cursor position.

The ASU is a 4K 4 bit character store using random access storage chips. The ASU is organized so that dynamic storage can be easily accommodated. RAM storage was picked to minimize the number of variables affecting performance measures. A black box view of the ASU is shown in Figure 3-13.

A 4 cycle self-timed signaling convention (handshaking) controls the inputs to the ASU. The two wires which are necessary to support this convention carry request and acknowledge signals. These two wires are called a link. The input link controls both the command and the input busses. Commands are placed on the command bus and data characters are sent via the input bus. Similarly, output characters (output bus) and error conditions are controlled by the output link. Eighteen lines are also present to communicate with the mapping unit which is strictly a speed-up device used during index commands. If no mapper is present, no change to the ASU is necessary.

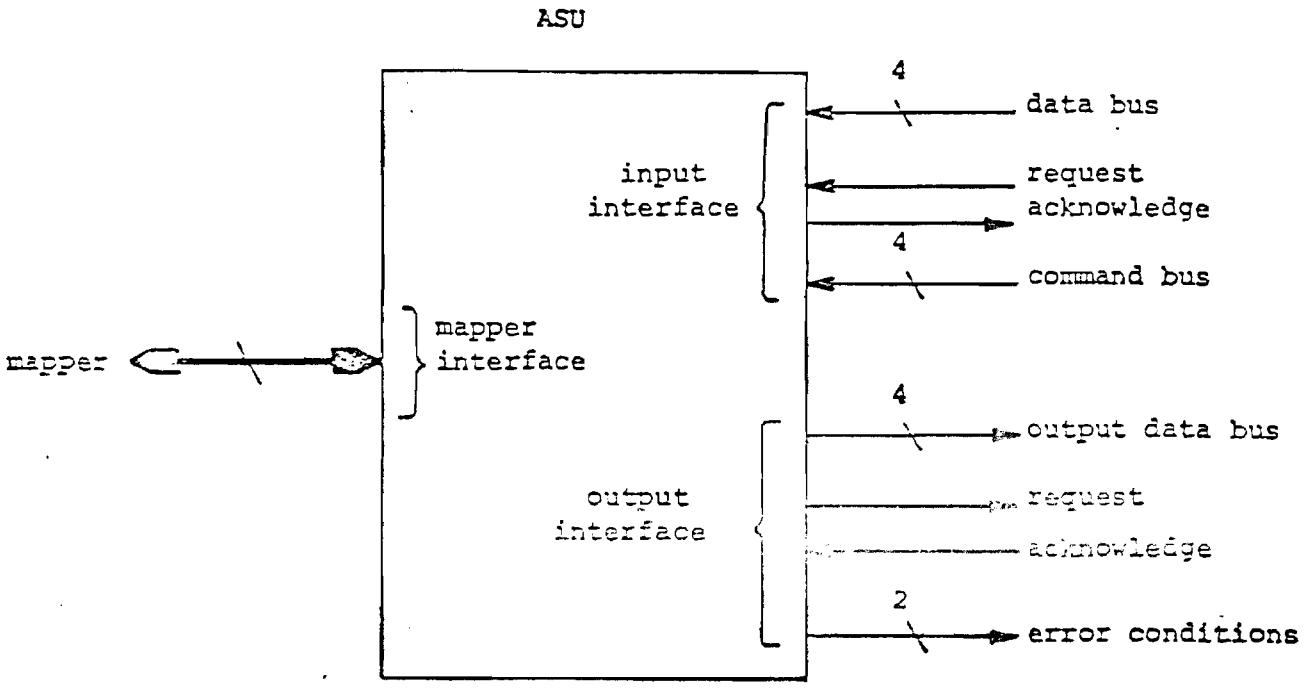


Figure 3-13: ASU Black Box Model

The internal architecture of the ASU is shown in Figure 3-14. This architecture has proven to be a nice structure for prototyping. Submachines may be added, deleted, and modified with negligible impact on the rest of the modules.

The soft control consists of an asynchronous micro-controller, a read/write microcode store, and a condition-select unit. The condition-select unit selects a condition line and places it in a condition register.

The Ap architecture is the same flexible prototyping structure that is used in the ASU. Different submachines and microcode formats are used to support Ap functional requirements. The Ap is intended to execute the DDN program structures, with cells of type; SYNCH, CALL, OP, GATE, DISTRIBUTE, SELECT, and ARBITER. The sub-types of OP cell

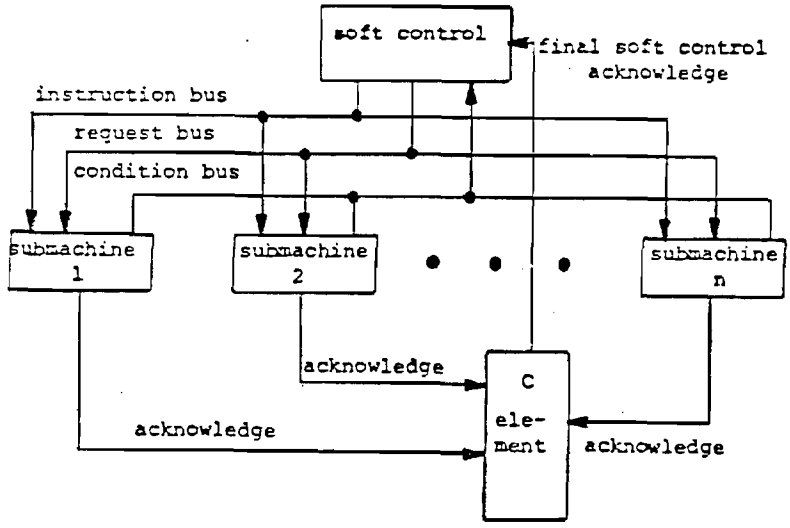


Figure 3-14: Block Architecture of ASU

which are allowed reflect the machine operation primitives which the DDM1 hardware supports. These operators perform actions such as add, subtract, relational tests, indexed reads and writes on TSM structures, maximum, minimum, not, negate, and absolute value.

DDM1 operates on sign magnitude decimal integers. TRUE is represented as a 1 digit while FALSE is a 0 digit. The presence of a "-" sign indicates negative numbers, no sign indicates positive numbers. Integers may be of arbitrary length. All data transmissions are in storage model format and are performed in a character-serial fashion. Numbers are transmitted low-order digit first followed by the sign.

The combination of the Ap, ASU, two asynchronous queues (IQ and OQ), and the 1:8 switch are a PSE.

3.6 Automatic Resource Allocation and Evaluation

When a message corresponding to a DDN program enters a PSE at any level, the PSE may take one of two actions:

1. DECOMPOSITION AND ALLOCATION: If the PSE has substructure and if there exists some set of concurrent subnets in the DDN process, then the PSE may split the DDN and send concurrent subnets to PSE's at the next lower level.

2. EXECUTION: if the PSE has no subresources, or if there is no exploitable concurrency in the DDN, then the PSE executes the DDN at that level.

To aid the decomposition process, a structural descriptor may precede the incoming DDN in the message. This additional storage can greatly reduce the time required for decomposition decisions in the PSE. In addition, each PSE must contain information about the number of available PSE's and the sizes of their respective stores. Problems would result if a DDN were sent to a PSE that did not have sufficient memory to store the DDN. Only the local store sizes of immediate subresources are known. This insures the recursive nature of the decomposition process.

The decomposition process takes some time. It is important that the speed-up gained by the extra concurrency resulting from decomposition is not overshadowed by the time to decompose. Experiments have indicated that a "first fit" decomposition is generally better than a "best fit" decomposition strategy. It also appears not to be worthwhile to completely decompose a DDN on this architecture. At fine granularities, the slowdown resulting from loss of locality is not regained by the concurrent execution of very small subtasks. An exception to this rule would be in the case of pipelining, where subtasks remain allocated for relatively long periods of time and sustain high activity at each site.

If decomposition and resource allocation occur at run-time, it is important that they be simplified as much as possible. It is possible

to perform these tasks completely at compile-time. This however is inadvisable since it is based on an assumption of the run-time availability of PSE's in the system. In a system containing large numbers of PSE's, the probability is high that some PSE's will fail or be busy doing other things. In addition, large portions of a process may only be evaluated conditionally. A compile-time allocation would have to allocate tasks which may never be executed. The strategy is taken here to split the decomposition task into two phases:

1. At compile time: do all of the resource and condition independent work, and
2. At run-time: dynamically make the actual allocation of executable tasks to available physical resources.

DDN's are quite irregularly structured graphs and DDM1 is a very regularly structured set of resources. Direct run-time allocation would be too slow, due to the structural disparity between program and machine. At compile-time, the two-terminal DDN process structure is transformed into a well structured and functionally equivalent series parallel graph (SP-graph). "Two-terminal" means that the graph contains a single "first" cell and a single "last" cell. This matches the DDP form and facilitates determination of net termination and initiation. SP-graphs are acyclic, two terminal, directed graph structures which can be formed by successively combining cells and/or SP-graphs in series or in parallel. The SP-graph structures are then allocated as necessary at run-time. Dataflow graphs in general admit nicely to arbitrary restructuring due to their asynchronous and local control characteristics.

The allocation of SP-graphs onto tree-structured physical resources is an easy task. If the SP-graph of Figure 3-15 is folded back onto itself about the middle, the result is a tree-structured SP-graph. The SP-graph, its folding, and the allocation onto a tree of physical resources are all shown in Figure 3-15. In this way full

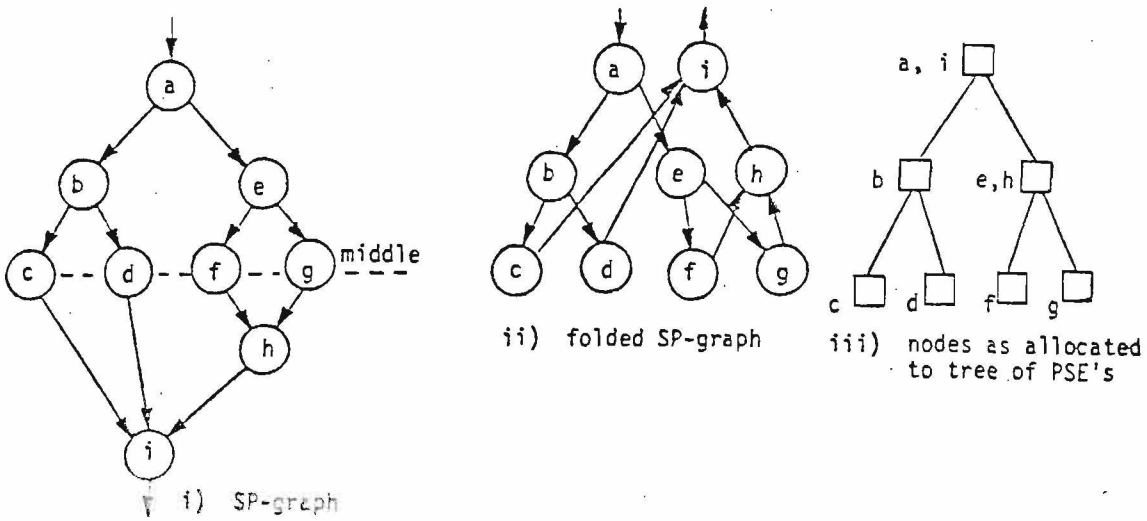


Figure 3-15: The allocation of SP-Graph programs onto a PSE tree

upward and downward communication can be carried on concurrently to achieve pipelining. Horizontal parallelism can be achieved by spreading independent subtasks across a given level of the architecture. Resource allocation is performed automatically by the hardware in DDM1 to achieve very high degrees of parallelism. The amount of obtainable concurrency is a function of available hardware resources and the program structure.

3.7 DDM1 in retrospect

DDM1 represents a particular architecture and evaluation scheme for dataflow programs. The architecture exploits a recursive hierarchy to reduce complexity and allows for the arbitrary expansion of system resources. Physical resources are organized such that they can be used to exploit both pipelined and independent tasks. The system exploits the notion of locality which is important for both the increased speed and decreased cost aspects of a VLSI implementation. This notion of locality also indicates that this system is not

intended to exploit concurrency at the lowest possible level. It is felt that the additional overhead involved would actually reduce overall performance levels.

DDM1 is operational and executes DDN programs. DDM1 communicates with a DECSYSTEM 20/60, which is used to support conventional software tools such as compilers, simulators, and measurement programs. The current programming language for the DDM1 system is the DDN representation. Programs can be created by working at a Tektronix storage tube display terminal and simply drawing the program. The programs which support this graphical programmer station run on the DECSYSTEM 20/60. The main problem with this station is the lack of interaction which results from the slow storage tube graphics terminal. An interactive graphical programming language is in progress (in both a high-level and a low-level form). A simulator is being written on the DEC-20 which will manage any specified tree of resources (virtual) and use the DDM1 for actual evaluation. A number of large application programs are being written for DDM1. Detailed statistics will be taken during the execution of these programs to aid in formal evaluation of the DDM1 hardware. There is also a graphical high-level language GPL [20] which has been defined (preliminary form) and is currently being implemented in the form of an on-line graphical programming environment and a language compiler.

The main points of departure of the "Utah" approach and that of Dennis [26] is the use of a recursive hierarchy of physical resources, the exploitation of physical locality to decrease message frequency and increase the speed of VLSI implementations, dynamic hierarchical resource allocation, the lack of specialized functional modules to reduce the chip type count, and a slight difference in the structure of the low-level schema. The architecture of DDM1 differs from that of Arvind and Gostelow [2] in that it does not try to

achieve concurrency at all possible levels (because of the locality issue), the interconnection scheme is much simpler and no bus contention is possible, no special address space management needs to be done, allocated tasks may consist of many cells rather than just a single operation, and tasks are allocated only when all of their necessary input operands are present.

The disadvantages of the system described are:

- The current ASU design is not nicely extensible to allow more storage capacity to just be "plugged in" at a PSE site.
- The fixed, hard-wire tree structure is not flexible and results in certain PSE's in one subtree remaining idle when another heavily loaded subtree badly needs more resources.
- There is currently not enough empirical data from test runs on very large programs to accurately quantify the overhead involved in decomposition.
- Failure of a PSE will cause the entire subtree below the failure to become unusable. In general, the issues of fault tolerance have not been properly attended to.
- Certain "perverse" SP-graph topologies can not easily be allocated such that full pipelining can be supported.

4. Conclusions

We have presented a rather detailed view of a non von Neumann computing model. The biggest problems with this model lie in overcoming the tremendous intellectual and commercial momentum of von Neumann structures. The parts of the cult which seem most likely to succeed are the highly functional programming style, the method by which a multiplicity of tasks can be coordinated in a concurrent environment, and the basic notion that local control is an important property for a module of a large system. It is possible to build these ideas into existent commercial systems which run on conventional hardware structures. The weakest point of the work done to date is the failure of all researchers to find acceptable new mechanisms for von Neumann artifacts like data structures and file systems. Given time and some good ideas perhaps these problems can be solved. If they can, the payoff is potentially enormous.

5. Acknowledgments

We would like to acknowledge the help of fellow researchers Jack B. Dennis, Arvind, Kim Gostelow, Ian Watson, and Jean Claude Syre for their comments and corrections of this manuscript. If the manuscript still contains some errors about their work, then it is solely the fault of the authors. We also greatly appreciate the efforts of our incredible secretary, Kathy Burgi, who always seemed capable of any request (no matter how unreasonable). Most importantly, we would like to thank our sponsor, Burroughs Corporation, whose support allows us to continue our work.

REFERENCES

1. D. A. Adams. A computation model with data flow sequencing. Tech. Rept. CS117, Stanford University, Computer Science Dept., 1968.

2. Arvind, K. P. Gostelow. A computer capable of exchanging processors for time. Information Processing 77, AFIPS, 1977, pp. 849 - 854.

3. A. L. Davis. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. Tech. Rept. UUCS-77-113, University of Utah, Computer Science 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, University of Utah, Computer Science Dept., 1978.

5. A. L. Davis. SPL - A structured programming language. Ph.D. Th., University of Utah, 1972.

6. R. M. Keller, G. E. Lindstrom, S. S. Patil. An architecture for a loosely coupled parallel processor. Tech. Rept. UUCS-78-105, Univ. of Utah, Computer Science Dept., 1978.

7. J. Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." CACM 21, 8 (1978), 613-641.

8. A. Bahrs. Programming language semantics and closed applicative languages. Proc. of the ACM Symposium on Principles of Programming languages, ACM, 1972, pp. 71-86.

9. K. J. Berkling. Reduction Languages for Reduction Machines. Proc. 2nd Annual Symposium on Computer Architecture, IEEE, 1975, pp. 133-140.

10. Per Brinch Hansen. Operating Systems Principles. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

11. C. A. Petri. Fundamentals of a theory of asynchronous information flow. Information Processing 62, IFIPS, 1962, pp. 386-391.

12. D.D. Charberlin. A Parallel Implementation of a Single Assignment Language. Ph.D. Th., Stanford University, 1971.

13. J. B. Dennis. First version of a data flow procedure language. Lecture Notes in Computer Science, SPRINGER-VERLAG, 1974, pp. 362-376.

14. P. J. Drongowski. Application of Hardware Description Languages to Microprogramming: Method, Practice amd Limitations. MICRO-12 Proceedings, ACM and IEEE Computer Society, Hershey, Pennsylvania, 1979, pp. 55-60.

15. G. Durrieu. Extension of the LAU System: global specification of synchronizations in a data driven language. 1st European Conference on Parallel & Distributed Processing, AFCET, CNRS and IEEE, February, 1979, pp. 149 - 155.

16. T. Agerwala, M. Flynn. Comments on capabilities, limitations, and correctness of Petri Nets. Proc. First Annual Symposium on Computer Architecture, IEEE, 1973, pp. 81-86.

17. D. P. Friedman, D. S. Wise. "The impact of applicative programming on multiprocessing." IEEE TC C-27, 4 (1978), 289-296.

18. V. M. Glushkov, et al. Recursive Machines and Computing Technology. Information Processing 74, IFIPS, 1974, pp. 65-70.

19. K. Gostelow and R. Thomas. Performance of a Dataflow Computer. To appear in IEEE Transactions on Computers

20. A. L. Davis, K. Boekelheide. GPL - a Graphical Programming Language. Preliminary manuscript

21. J. Gurd, I. Watson, and J. Glauert. A Multilayered Data Flow Architecture. Dept. of Computer Science, University of Manchester, July, 1978.

22. M. Hack. Petri net languages. Tech. Rept. 161, MIT Laboratory for Computer Science, 1976.

23. A. Holt, F. Commoner. Events and Conditions. Record of the Project MAC conference on concurrent systems and parallel computation, MIT Project MAC, 1970, pp. 3-52.

24. Arvind, K. P. Gostelow, W. Plouffe. The Id Report: An Asynchronous Programming Language and Computing Machine. Tech. Rept. 114A, Univ. Calif. Irvine Comp. Sci. Dept., 1978.

25. J. B. Dennis. Programming generality, parallelism, and computer architecture. Proceedings IFIPS Congress, IFIPS, 1969, pp. 484-492.

26. J. B. Dennis, D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proc. of the 2nd Annual Symposium on Computer Architecture, ACM, IEEE, 1974, pp. 126-132.

27. J. D. Rodriguez. A Graph Model for Parallel Computation. Tech. Rept. TR-64, MIT Project MAC, 1969.

28. R. M. Karp, R. E. Miller. "Parallel program schemata." Journal of Computing and System Sciences 3, 2 (1969), 147-195.

29. D. Comte, N. Hifdi. LAU Multiprocessor: Microfunctional Description and Technological Choices. First European Conference on Parallel and Distributed Processing, IFIPS, AFCET, CNRS, 1979, pp. 8-15.

30. D. Comte, G. Durrieu, O. Gelly, A. Plas, J. C. Syre. Etude at specifications d'une architecture de calculateurs a controlle decentralise exploitant le concept d'assignation unique. Centre d'Etude et de Recherches de Toulouse, Department d'etude at de recherches en INFORMATIQUE, October, 1976.

31. E. A. Ashcroft, W. W. Wadge. "Lucid, a nonprocedural language with iteration." CACM 20, 7 (1977), 519-526.

32. G. A. Mago. "A Network of Microprocessors to Execute Reduction Languages, Part I." International Journal of Computer and Information Sciences 8, 5 (March 1979 revised), 349-385.

33. J. B. Dennis, D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM National Conference, ACM, 1974, pp. 402 - 409.

34. J. B. Dennis, D. P. Misunas, and C. K. Leung. A Highly Parallel Processor Using a Data Flow Machine Language. MIT LCS, January, 1977.

35. J. L. Peterson. "Petri Nets." Computing Surveys 9, 3 (1977), 223-252.

36. C. A. Petri. General Net Theory. Conference on Petri Nets and Related Methods, MIT Project MAC, 1975, pp. 26-41.

37. T. E. Rudy. Megaflops from Multiprocessors. Proceedings of the Second Rocky Mountain Symposium on Microprocessors, Colorado State University, 1978, pp. 99-107.

38. D. Scott. "Data types as lattices." SIAM J. Comput. 5, 3 (September 1976), 522-587.

39. J. A. Stanek. Exploration of Concurrent Digital Sound Synthesis on a Prototype Data-Driven Machine. Master Th., University of Utah, September 1979.

40. L.G. Tesler and H.G. Enea. A Language Design for Parallel Processes. Proceedings of the 1968 SJCC, AFIPS, 1968, pp. 403-408.

41. K. P. Gostelow and R. E. Thomas. Performance of a Dataflow Computer. preprint

42. W. B. Ackerman, J. B. Dennis. VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual. Tech. Rept. LCS/TR-218, MIT, Computer Science Department, 1979.

43. K. S. Weng. Stream-Oriented Computation in Recursive Data-Flow Schemas. Tech. Rept. MIT/LCS/TM-68, MIT LCS, 1975.