AN INTRODUCTION TO PARALLEL COMPUTER ARCHITECTURES

By

PHYLLIS JOHNSON THORNTON

Bachelor of Arts

in Liberal Arts and Sciences

San Diego State University

San Diego, California

1970

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF EDUCATION
July, 1988

AN INTRODUCTION TO PARALLEL

COMPUTER ARCHITECTURES

Thesis Approved:

Sharilyn A. Thoreson
Thesis Advisor

D. E. Hedman

M J Folk

John D Gordon

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to the members of my doctoral committee. I am grateful to Dr. Sharilyn A. Thoreson, my dissertation adviser, and Dr. George E. Hedrick, my committee chairman, for their joint efforts on my behalf, their teaching, counsel, support, encouragement, and friendship. Further, Dr. Thoreson's expert advice and direction in the study of parallel systems was invaluable. I thank Dr. John J. Gardiner, for his aid and guidance in Higher Education, and Dr. Michael J. Folk, for his insightful suggestions and assistance as a member of my doctoral committee.

Appreciation and gratitude is extended to all my professors at Oklahoma State University from whom I have gained so much. Also, I thank my fellow faculty members at Central State University for their encouragement and cooperation in class scheduling which has allowed me to pursue these studies.

A special thank you goes to my parents, Phillip and Fern Johnson, the source of my educational aspirations, for their love, concern, and encouragement in all my endeavors through the years.

Most of all I want to thank my husband, Michael, and children, Justin and Jennifer, for their patience and

understanding, for their help and encouragement, for their love. I would never have attained this goal without their support. Michael, Justin, and Jenny, thank you for being the best husband, the best son, and the best daughter a wife and mother ever had -- I love you.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.0 Statement of the Problem and Proposed Solution

The computer was developed to perform the tedious time consuming calculations required to solve problems from many areas of application. As time has passed, more and more complex problems have been identified for which solutions are desired. Solutions are needed to such problems as weather forecasting, signal and image processing, expert systems and implementation of artificial intelligence, and the implementation of current military applications and future ones such as the Strategic Defense Initiative (SDI). Feasible solutions to these problems and others will not be possible without the use of high-performance supercomputers.

The first approach to building better computers has been to build faster von Neumann computers by improving gate speeds, reducing transfer distances, and generally improving the existing architecture technologically. However, there is a limit to the amount of computing power compressible into one package. The speed of light and electricity has been determined to be a constant; it

cannot be exceeded. This constant establishes a fundamental limiting factor on the computing capacity of the standard uniprocessor architecture. The speed of light is approximately 3 x 10^8 m/sec in a vacuum and the signal transmission speed in silicon is at best about 3 x 10^7 m/sec after gate switching delays are taken into account. A three centimeter chip can propagate a signal in approximately 10^{-9} sec. Such a chip can perform in the neighborhood of 10^9 floating point operations per second (FLOPS), since a nonparallel chip can produce at best, one floating point operation per signal propagation. Thus, it appears that the standard von Neumann uniprocessor will not be able to exceed 10^9 FLOPS or one giga-FLOPS (GFLOPS). The supercomputers presently available are within a factor of 10 of this limit [22].

From the above discussion, it should be clear that the need for increased computer speed will not be met and challenging problem solutions will not be attainable within a feasible time limit without major improvements in computer organizations and programming techniques. Parallel architectures offer a partial solution to this problem.

Parallel computer architectures allow 1) instruction parallelism, the execution of two or more machine instructions within a time interval, 2) data parallelism, the processing of several data elements at a time, or 3) both data and instructional parallelism may take place

within the same time interval. These architectures were described by Michael Flynn in 1966 [30]. Under his scheme of characterization, machines with no parallelism were single-instruction-stream single-data-stream (SISD) machines. Computers with only instruction parallelism were characterized as multiple-instruction-stream single-data-stream (MISD) computers; those with only data parallelism were single-instruction-stream multiple-data-stream (SIMD) ones. And finally, those computers with both instruction and data parallelism were multiple-instruction-stream multiple-data-stream (MIMD) computers.

Some operations that would have to be performed one after another on a standard von Neumann uniprocessor can be performed concurrently on parallel computer architectures. For example, let a given job, taking T time units when executed sequentially, be partitioned into n substeps, each requiring T/n time units. If each substep can be executed concurrently on a parallel computer, then, theoretically, the result could be expected in 1/n-th of the sequential time. Although such results are currently only approximated in real applications, considerable speedup can be verified.

The study and exploitation of these parallel systems is of utmost importance if we are to attain computers with sufficient speed of computation to reach feasible solutions to many critical problems.

Currently, knowledge of parallel systems is not widely disseminated. Within most universities, education of students into the aspects of parallel architectures is not initiated until the graduate level. An informal survey of thirty-eight universities indicated this to be true. The catalogs of the thirty-eight universities listed in Table I were consulted to determine what parallel computer architecture courses are offered by the universities. Although a number of them offered courses dealing with some aspect of parallel computer architecture at the graduate level, few offered anything at the undergraduate level. Seven of the thirty-eight offered general computer logic and organization courses whose descriptions included some allusion to parallel processing topics. Only two universities offered undergraduate courses whose descriptions indicate a strong emphasis on parallel computer architectures.

This deferring of parallel computer architecture curriculum is unfortunate. The sequential nature of von Neumann uniprocessors and the procedural languages developed to execute on them frequently establish a mind-set for students which colors their view of computing for the rest of their lives. Further, if they are not introduced to parallel computing during their undergraduate experience, they may never give the matter serious thought. Different architectures give alternate ways of approaching problems; without knowledge of these

TABLE I

UNIVERSITY CATALOGS CONSULTED TO DETERMINE WHAT PARALLEL COMPUTER ARCHITECTURE COURSES ARE OFFERED AT THE UNDERGRADUATE LEVEL

University	Catalog Year	Parallel Course Offered
Air Force Academy	1986-1988	None
Baylor University	1985	None
Bowling Green State University	1987-1989	None
Central State University	1988	None
California State University, Fullerton	1985-1987	None
Case Western Reserve University	1985-1987	None
Duke University	1988-1989	A*
East Texas State University	1986-1987	None
Illinois Institute of Technology	1986-1988	None
Indiana State University	1986-1988	A*
John Hopkins University	1986-1987	None
Kent State University	1985-1986	A*
Louisiana State University	1986-1987	A*
Massachusetts Institute of Technology	1985-1986	None
Michigan State University	1987-1988	None

TABLE I (Continued)

Murray State University	1986-1988	A*
North Dakota State University, Fargo	1986-1988	None
Ohio State University	1985-1986	None
Oklahoma State University	1986-1988	None
Pennsylvania State University	1985-1986	A*
Princeton University	1987-1989	A*
Purdue University	1987-1989	None
Texas Women's University	1987-1989	None
Vanderbilt University	1987-1988	None
Virginia Polytechnic Institute and State University	1988-1989	P*
United States Naval Academy	1987-1988	None
University of Arizona	1984-1985	None
University of Arkansas at Little Rock	1985-1986	None
University of Boston	1985-1987	None
University of California, Riverside	1987-1988	None
University of Connecticut	1986-1987	None

TABLE I (Continued)

University of Idaho	1987	None
University of Miami	1987-1988	None
University of Texas at Dallas	1984-1986	None
University of Rhode Island	1985-1986	None
University of Wisconsin, Milwaukee	1987-1989	None
Washington State University	1987-1989	None
Yale University	1985-1987	P*

A* indicates course description included some allusion to parallel processing topics.
P* indicates course description implies a strong emphasis on parallel computer architectures.

alternate systems the student is locked out from a whole new perspective on problem solving. The earlier in their computer education process that students are introduced to parallel architectures the better will be their opportunity for growth.

The computer science student of today is the computer designer, engineer, analyst, and programmer of tomorrow. Current students must be the identifiers and solvers of the computational problems of the present and future. It is the responsibility of the computer educator of today to facilitate the learning of these students in the concepts of parallel architectures. Thus, the fundamental concepts of parallel systems should be introduced into the computer science curriculum as soon as possible.

The purpose for this dissertation is to produce an introductory treatise on parallel architectures which will be appropriate for study by undergraduate computer science students in their junior or senior year of academic study. By bringing these architectures to the attention of the student early in his learning experience, the student's perspective on computing will be broader and more fruitful.

1.1 Intended Audience For the Treatise

The appropriate audience for this treatise is upper division undergraduate students majoring in computer science. These students should have mastered the following:

1) programming in a high level procedural language such as Pascal or C. The student should have a clear understanding of procedures, parameter passing, pointers, and algorithms.

2) programming in an assembly language such as IBM 370 Assembly Language or VAX 11 Assembly Language. The study of a computer architecture implies the investigation of the machine at this low level. The student should be familiar with the low level workings of at least one machine so that he/she can extrapolate that understanding to new and, perhaps, more complicated architectures.

3) the fundamental concepts of computer logic and computer organization. The student should already be conversant in the integral components of a computer; such comprehension is necessary for the appreciation of the new parallel systems he/she will be studying.

4) the basic concepts of data structures including stacks, queues, linear and circular linked lists, matrices, and trees. Many of the structures studied in such a course are utilized, either in software or hardware, within parallel systems.

5) the basic concepts of operating systems.

Parallel systems frequently involve the concurrent and/or simultaneous execution of programs on the same or different processors. Such execution is controlled by an operating system. Knowledge of how program execution is managed on a nonparallel system will be helpful in the study of parallel program executions.

1.2 Specific Topics for the Treatise

Three types of parallel architectures commonly identified are array, pipeline, and multiprocessor architectures. These three architectures are extensions of the von Neumann architecture. There are two other parallel architectures that are non-von Neumann. The first of the two non-von Neumann machines is referred to as a data-driven or a data flow machine. The second is referred to as a demand-driven or reduction machine. This treatise introduces each of these five parallel architectures by examining the general aspects of each. Also, specific machines that implement these architectures are presented.

1.3 Existing Literature

The literature which presents aspects of array, pipeline, multiprocessor, data flow, and reduction architectures are plentiful in periodicals and books. Included with this dissertation is a bibliography of

sources used for the documentation and completion of this treatise. Kartashev and Kartashev (1982) [48] has an excellent review of array processors and Kogge (1981) [52] covers pipelining in great depth, but their breadth is limited. Others such as Baer (1980) [7] and Stalling (1987) [75] briefly introduce some of these topics but their primary focus is on the von Neumann architecture. Calingaert (1982) [16] and Peterson and Silberschatz (1985) [63] briefly present some aspects of multiprocessing from the perspective of the operating system but do little with the architecture. Few existing works bring together under one cover the architectural topics presented in this dissertation. Two that cover most of the topics are Hwang and Briggs (1984) [41] and Stone (1986) [76], but they are posed for presentation at the graduate level. No known work attempts to present the concepts of array, pipeline, multiprocessor, data flow, and reduction at a level appropriate for the undergraduate student. This treatise attempts to satisfy the need for a book to aid the computer science undergraduate student in the study of parallel computer architectures.

1.4 Operational Terms and Reading Aids

The following operational terms and their definitions may be of assistance to the reader of this treatise. Additional terms and definitions may be found in Appendix A, a glossary of terms used in this dissertation.

Further, Appendix B contains a list of acronyms and the words from which they are formed.

A computer architecture is the arrangement of the parts of a computer system, their interconnections, dynamic interactions, implementations, and management.

A parallel computer can perform multiple operations at the same time.

Supercomputer is a loose term for an extremely powerful mainframe computer that provides high speed computing.

A von Neumann computer is based on the work of mathematician and computer designer John von Neumann. The computers are characterized by 1) a single computing element incorporating processor, communications, and memory, 2) linear organization of fixed size random-access memory cells, 3) a sequential, centralized control of computation. A machine instruction program is loaded sequentially in main memory and executed under the sequencing of a program counter.

Data dependency is the state of being dependent or conditional on the value of the data read or written in a single instruction or in a block of code. Data dependencies exist between operations when the action of one operation on data affects the outcome of the other operation and vice versa.

An array processor is a computer with one control unit, multiple arithmetic/logic units, and multiple memory

units. The control unit fetches instructions from the memories, decodes them and broadcasts the instructions to the arithmetic/logic units. Each arithmetic/logic unit can fetch its own data for processing. An array processor performs duplicate operations on multiple data items simultaneously.

An associative processor is a computer system much like an array processor with the distinction that it operates on associative memories.

Pipelining is the process of partitioning a job into distinct steps and streaming inputs through the steps. The mechanism is like that of materials moving through an assembly line.

A multiprocessor is a computer system with more than one central processing unit. It is used to decrease the time to completion for a single job.

A data flow computer is one in which instructions are executed based on data dependencies. Programs are represented by data flow graphs. Availability of operands triggers the execution of operations.

A data flow graph is a directed graph used to represent a data flow program, where nodes are instructions or processes whose outputs pass along links to subsequent processes. A node executes, or fires, if all its input links are carrying values. The graph represents the data dependencies inherent in the computer program.

A reduction machine is a computer in which the requirement for a result triggers the operation that will generate it.

Reduction is a computation system in which programs are built from nested expressions. The nearest analogy to an instruction is a function application where the function returns its result in place (a CALL-RETURN pattern of control). A function or its arguments may be recursively defined as a primitive operation, such as add or multiply, as a constant, as an expression, or as another function. In reduction, a program is equivalent to its result in the same way that 2+2 is equivalent to 4. The main points of reduction are that 1) program structures, instructions, and arguments are all expressions, or functions; 2) there is no concept of updatable storage; 3) there are no sequencing constraints other than those implied by demands for operands; 4) demands may return both simple or complex arguments, such as a function.

Graph reduction is a form of reduction in which each instruction that accesses a particular definition will manipulate references to the definition. That is, graph manipulation is based on the sharing of arguments using pointers. When a functional value is demanded the reference is traversed in order to reduce the definition and return with the actual value.

CHAPTER II

A REVIEW OF THE VON NEUMANN COMPUTER ARCHITECTURE

2.0 Introduction and Historical Perspective

This section briefly reviews the history and earliest organization of the the von Neumann computer architecture.

The evolution of computer development has its beginnings in the 1400's when Blaise Pascal invented the first mechanical calculator. Charles Babbage, an English mathematician, inventor, and philosopher of the 1800's initiated a calculating engine which was to have a control unit, arithmetic unit, memory, and I/O devices. Unfortunately for him and his collaborator Lady Ada Augusta Lovelace, the technology for such a machine was not available and they were never able to complete their work.

In 1946, J.W. Mauchly and J.P. Eckert working at the Moore School of Electrical Engineering at the University of Pennsylvania, were credited with building the Electronic Numerical Integrator and Calculator, better known as ENIAC. For many years, the ENIAC was credited as being the first electronic computer. However, in the

1970's, it was shown that Mauchly and Eckert had drawn very heavily from the work of John Atanasoff and Clifford Berry. The Atanasoff-Berry machine built at Iowa State University in 1939 now is credited as the first electronic computer.

While working on the ENIAC project, Mauchly and Eckert collaborated with John von Neumann, a prominent mathematician of that period, on problems of machine design. In 1945, von Neumann wrote a memo as an ENIAC consultant suggesting a stored program machine, its possible implementation and implications. This important idea led to the construction of EDVAC (Electronic Discrete Variable Automatic Computer) which was begun in 1946. The EDVAC is credited as being the first stored program computer. Although it was not the first such computer to become operational, it was the first computer for which a workable plan was established to implement a stored program. During the time when the EDVAC was being constructed, von Neumann also joined with a group of scientists at Princeton University's Institute for Advanced Studies. In June 1946, they published a report entitled "Preliminary Discussion of the Logical Design of an Electronic Computer." It was a well argued paper on the many details of machine design. These documents led to the construction of the Institute for Advanced Studies computer (IAS). Both the EDVAC and the IAS became operational in 1952 [55, p. 68].

It is interesting to compare the mechanisms used by these two machines for the purpose of fetching machine instructions from the computer memory to the control unit for decoding and execution. The EDVAC whose construction was begun first, had a 1024 word mercury delay line memory. Each instruction was composed of an operation code, or opcode, and four address fields. Two of the addresses specified the locations in the memory of the operand values to be used in the execution of the instruction. The third address field specified the location in the computer memory at which the result of the execution should be stored. The fourth field contained the address at which the next instruction to be executed could be found. The instructions were not loaded sequentially in the memory; to the contrary, the instructions could be anywhere in the circulating mercury delay line memory. The instructions were related logically as nodes on a singly linked list. To place the program into execution, only the list head pointer, the address of the first instruction to be executed was required. Each subsequent instruction to be executed was fetched from the location specified in the next field of the current instruction [55, p. 65-69].

The IAS computer contained a Random Access Memory (RAM) implemented by Williams tubes. Williams tubes were developed by F.C. Williams in 1947. They were cathode ray tubes with bits stored on their face. The bits could be

capacitively sensed, and access time was a function of electron beam switching and sensing times only. The IAS memory was built on 40 1024-bit Williams tubes. These provided 1024 40-bit words. Each 20-bit instruction contained an opcode and one 10-bit address. The address specified the location of one operand value in the memory while the second operand was held in a dedicated register and the result of the operation was stored back into this register (accumulator). Because the memory was random access, given its address, each instruction could be accessed directly. The "next instruction field" of EDVAC's instruction field format was eliminated by introduction of a program counter register (PC). The program was loaded sequentially into the RAM. The address of the first instruction to be executed was loaded into the PC and the instruction to be executed was fetched from that location; then, the PC content value was incremented by the length of the instruction giving it the address of the subsequent instruction to be executed [7, p. 3-4, 55, p. 65-69].

2.1 Von Neumann Computer Organization

This section identifies the essential elements of a von Neumann computer architecture.

The EDVAC and especially the IAS computer were the prototypes for what has become the basic structure for most sequential machines in use today. This basic

architecture is called the von Neumann architecture. In a von Neumann architecture, each machine operation that is under programmer control is specified in a machine instruction. Each instruction is composed in some format (determined by the machine designers) of an opcode which specifies the nature of the operation and address fields. Each address field contains the address in the RAM at which the operand value(s) to be acted on by the opcode may be found. These instructions are loaded sequentially in the computer memory. Von Neumann architectures are called control flow computers because the flow of execution is sequential and is controlled by a program counter [Figure 1].

A general-purpose von Neumann architecture digital computer has the fundamental elements illustrated in Figure 1. The following discussion gives an outline of its operation from the beginning of an instruction cycle to the beginning of the next cycle. All these events may be carried out asynchronously; that is, each activity is performed by a designated module, and the activities are performed sequentially; when the first module has finished its work, it signals the second module to begin its work, etc. EDVAC and IAS were asynchronous machines; however, computer designers soon realized that the extra control hardware and time for acknowledge signals between elementary operations required too much overhead as

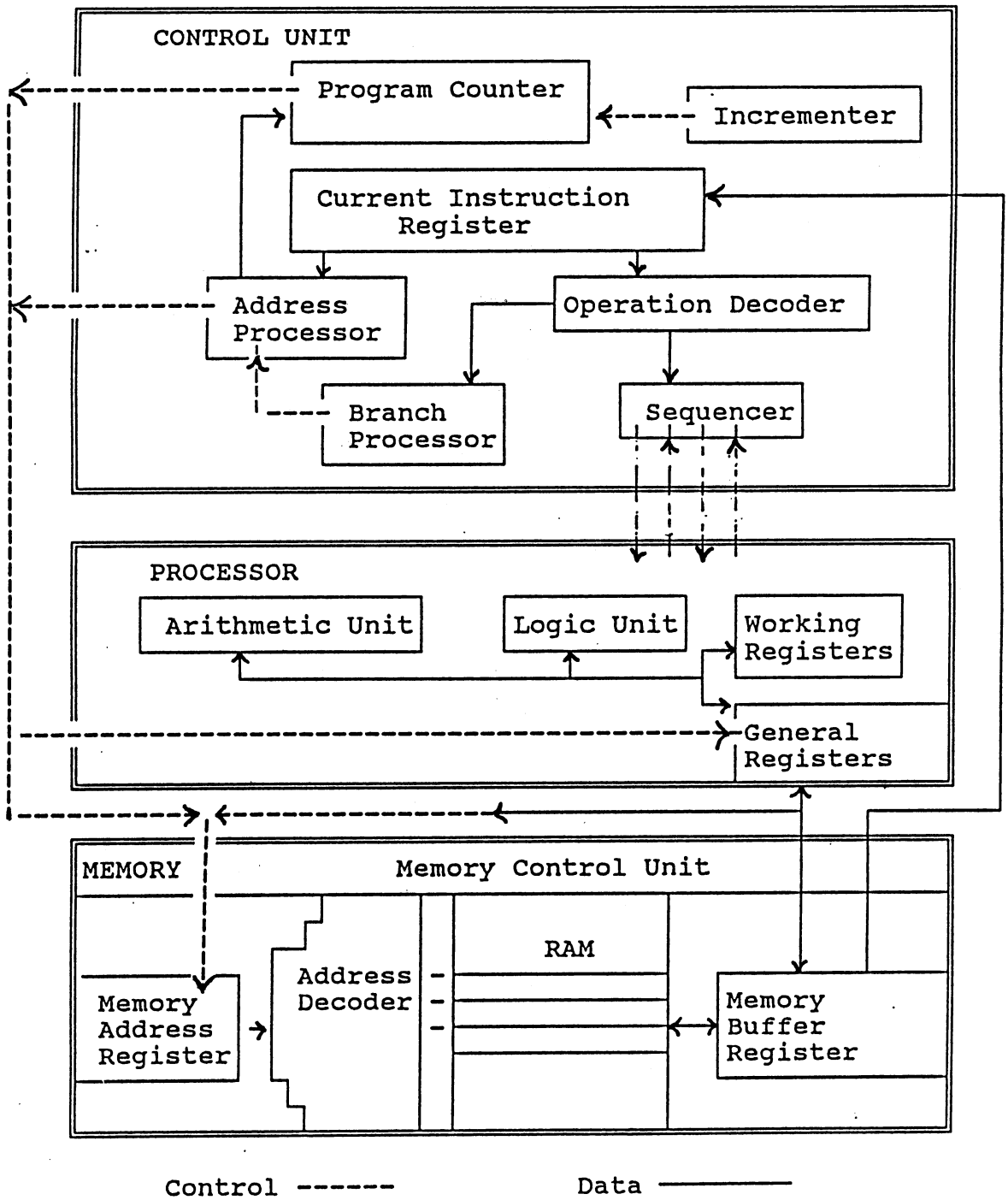


Figure 1. Organization of the von Neumann Computer

computing speeds increased. Today most computers are synchronous machines. That is, each event takes place under the synchronization of a clock, whose signals are distributed throughout the machine.

An instruction cycle on a von Neumann machine has six steps. Initially, the program counter (PC) contains the address of the next instruction to be executed. The steps are the following:

1) Instruction Fetch. The address in the PC is sent through the memory control unit and stored in the Memory Address Register (MAR). The address is decoded and the instruction is passed from memory into the Memory Buffer Register (MBR) and through the memory control unit to the Current Instruction Register (CIR).

2) Program Counter Increment. The program counter is incremented by the length of the current instruction so that it points to the next sequential instruction in memory. Should an abnormal termination occur during the execution of the current instruction, the PC contains the address of the next instruction to be executed, not the one causing the termination.

3) Address Calculation. The address portion of the current instruction is sent to the address processor. The address processor translates the address field values into target addresses.

The mode of addressing indicated for the instruction determines the use of the target address. If the mode of

addressing is that of immediate addressing, the target address is used as an operand. If the addressing mode is direct or indirect and the operand is in memory, an operand fetch is initiated.

4) Operand Fetch. If the operands reside in memory, each target address is passed from the address processor through the memory control unit into the MAR. The address is decoded and the memory value is copied from the appropriate memory bytes into the MBR. If a direct addressing mode is indicated in the instruction, the MBR value is the operand value and is routed to the processor. If the mode is indirect, then the MBR holds the indirect address; this value is routed back around to the MAR and undergoes address decode. The bytes identified by the second address decode procedure are copied into the MBR; the MBR value is the operand value. This value is routed to the processor. When the operand value arrives in the processor, it is placed into some appropriate register within the processor.

5) Opcode Decode and Execution. The operation code for the instruction is passed from the CIR to the operation decoder where the bit pattern of the field is converted into electrical signals that drive the processor.

If the opcode indicates a jump or branch (nonsequential execution) then the branch processor is signaled to determine whether a branch should occur. If

the branch processor determines that a branch is required and a direct addressing mode was indicated, then the target address calculated by the address processor is passed into the program counter instead of the MAR as in the case of an operand fetch from memory. If indirect addressing was indicated, then the target address is used to cycle memory for the indirect address and the indirect address is copied from the MBR to the PC.

6) Result Store. If the opcode indicates a write back to memory, the result generated in the processor is passed from a processor register through the memory control unit into the MBR. Concurrently, the target address is passed from the address processor through the memory control unit and into the MAR. The address is decoded and the value in the MBR is written to the bytes specified by the MAR.

This concludes one instruction cycle. The program counter contains the address of the next instruction to be executed and the next instruction cycle begins [55, p. 281].

From this discussion, two important characteristics of the von Neumann architecture should be clear:

1) It has a global addressable memory to hold both data and program instructions. The instructions frequently update the data cells as the program executes. These shared data cells are the means by which data is passed from one instruction to the next.

2) Sequencing of instructions is determined by a program counter. The program has complete control over instruction execution sequencing based on the original order in which the instructions were loaded into sequential memory. The flow of control is implicitly sequential. One instruction may execute at a time.

2.2 Summary and Preview

This chapter briefly reviews the historical beginnings of modern computing, focusing on the historical source of what is known as the von Neumann architecture.

The essence of the von Neumann architecture as it is understood today is reviewed.

The need for faster computations, shorter turnaround times, and greater system throughput has generated a great deal of activity directed toward creating von Neumann machines which operate faster. Increased speedup has been accomplished through new advances in underlying technology. However, the architecture now appears to be bounded by the speed of light itself. Since man has little hope for changing the basic laws of nature, computer designers are now searching for alternate approaches to computer design which will speedup computer processing. A primary approach to the problem of increasing the computer's operational speed has been to design systems which allow multiple operations to occur concurrently whenever possible within an algorithm; this

is exploitation of parallelism. The next chapters are devoted to introducing the reader to the principal parallel systems in use today.

CHAPTER III

ARRAY PROCESSORS: THE ILLIAC IV

3.0 Introduction to Array Systems

This chapter presents array systems. Array systems are parallel computer systems that allow multiple data items to be processed in exactly the same way at the same time. This form of parallelism is termed data parallelism. Such machines are single-instruction-stream multiple-data-stream computers, or SIMD, as described by Michael Flynn in his computer architecture classification [31].

The basic components of array systems and their general strategy of operation are presented first. Later, the operation of the ILLIAC IV array processor is reviewed. The ILLIAC was an array processor developed in the late 1960's and the predecessor of modern array systems.

3.1 Basic Ingredients of an Array System

This section identifies the basic elements of an array system. It demonstrates how they are organized and controlled, and how data is transmitted within the system.

The manner in which some systems can alter or reconfigure their arithmetic/logic units, allowing them to operate on different size data words is reviewed. Lastly, the section distinguishes between array processors and associative processors, the two subclasses of array systems.

3.1.1 Configuration of Array Systems: ILLIAC and BSP

Array systems generally are understood to have the following basic elements:

1) P processing elements (PEs), or arithmetic logic units with attached registers, and

2) M memory modules (PEMs) for the storage of operands to be processed by the PEs, and

3) a single control unit (CU) with its own memory for program and scalar storage.

The CU fetches instructions from its own memory. Scalar and control operations are performed in the CU's local registers. Vector operations are broadcast to the PEs (single instruction stream) where each of the P processing elements fetches operands from one of the M memory modules (multiple data stream). The PEs then execute the same instruction synchronously. The array system achieves spacial parallelism through the duplicate lockstep actions of the PEs.

Array systems usually have another general-purpose computer that acts as a front-end for the system. This general-purpose machine acts as a host to the array system. The host interacts with the outside world, oversees all I/O functions and manages the various resources of the overall system.

Within an array system there must be a communication network which links the processing elements, PEs, so that data may be passed from one PE to another. There are two basic configurations. The first is termed the ILLIAC type configuration because it was implemented on the ILLIAC IV array processor. Within this configuration the number of processing elements, P, is equal to the number of memory modules, M. Each PE is attached directly to its own memory module, or PEM, and directly accesses its operands from that PEM. The PEs are linked by an interconnection network [Figure 2]. The second basic configuration is termed the BSP type configuration since it was used in the Burroughs Scientific Processor. Here an alignment network (see APPENDIX A) is used. The alignment network is positioned between the memories and processing elements. The memories act as a shared resource for the PEs; a PE may fetch its operands from any one of the memory modules. The number of PEs, P, may differ from the number of memories, M; they have, in some cases, been chosen to be relatively prime [Figure 3].

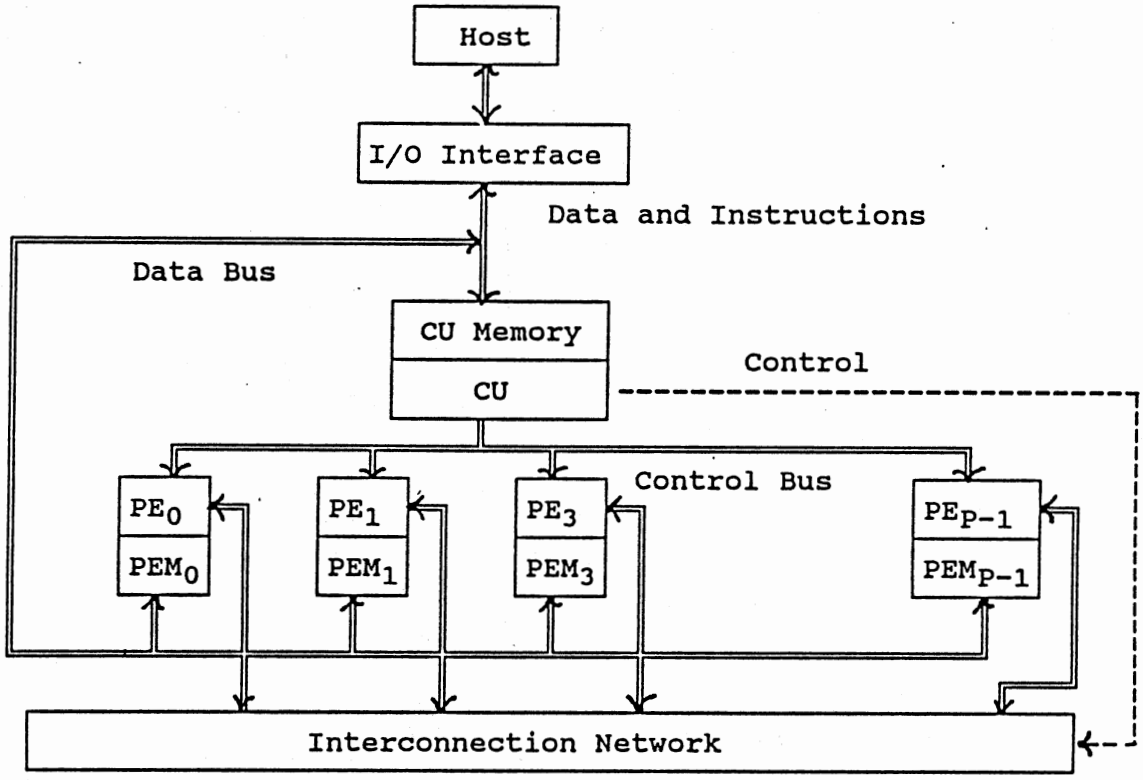


Figure 2. Illiac Type Array Configuration
[41, p. 326]

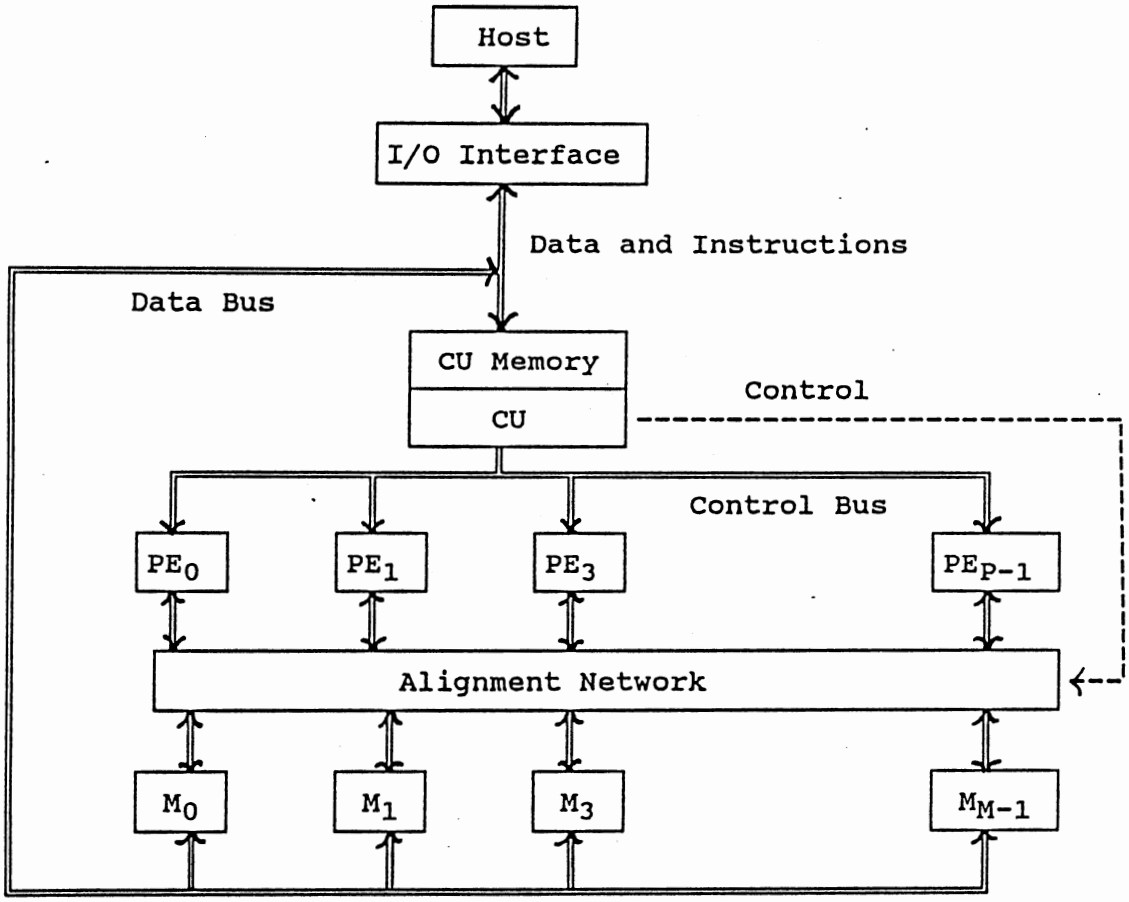


Figure 3. BSP Type Array Configuration
[41, p. 326]

3.1.2 Processing Element Enablement

The CU broadcasts the instructions to the PEs, and the PEs all execute the instructions together. However, on certain occasions all PEs may not be required to execute an instruction. In such a case, masking schemes are employed to control the execution or non-execution of an instruction by a specific PE. Under the masking scheme, a PE may be enabled or disabled. Only enabled PEs will execute a broadcast instruction. In general, each PE has an enable/disable bit. If the bit is 1, the PE is enabled; if the bit is 0, the PE is disabled. Within the CU there is a mask register (MR) containing one bit for each PE. The bit pattern of the MR is set by control operations within the CU. When the enablement of the PEs is to be established, each bit in the masking register, MR_i, $i = 0,1,2,\ldots,P-1$, is exchanged with its corresponding PE_i enable/disable bit. Thus, the programmer can control which PEs are executing at a given time by setting the CU's mask register bit pattern.

3.1.3 Interconnection networks

In an array system there must be a way for data to move from PE to PE. This is accomplished via a network. The ILLIAC type configuration interconnection network seems to be the most frequently discussed in the literature and it is the focus here. An interconnection

network can be described by a set of interconnection functions, where each interconnection function is a one-to-one and onto mapping, or bijection, on the set of PE addresses. When an interconnection function f is applied to PE_i, PE_i sends the contents of a data transfer register to that of $PE_{f(i)}$. This occurs for each i = 0, 1, 2,..., P-1 and PE_i enabled. This implies that each enabled PE sends data to exactly one PE; and each PE receiving data receives it from only one PE. Generally, a disabled PE cannot send data, but may receive it. To pass information from one PE to another, a programmed sequence of one or more interconnection functions must be executed. Data may be transferred directly by one function execution or may move through a series of PEs by executing a series of functional instructions. Since an array processor is SIMD, all enabled PEs must execute the same interconnection function at the same time. Several different interconnection functions have been defined for SIMD systems. Some of the common ones are known as shuffle-exchange, barrel shifter (see APPENDIX A), and ILLIAC network functions [41, p. 333]. Section 3.2.1.6 presents the specific attributes of the ILLIAC network function when it examines the ILLIAC IV parallel array system.

3.1.4 Reconfigurability

An attribute possessed by some array systems is reconfigurability. The term is sometimes used to refer to the capability of disabling certain PEs as presented above. However, the term is also applied to identify the capability of a machine to rearrange each PE and PEM into several smaller size processors and memory modules, or vice versa, under software control. For example, a 64 bit word PE and PEM may be able to be reconfigured into two 32 bit word PEs and PEMs. Thus a reconfigurable array system may increase or decrease the number of data items processed in parallel by changing the processor's size.

3.1.5 Array Processors and Associative Processors

Array systems frequently are classified into two subgroups. The first is that of array processors. Array processors access standard random access memory modules. They were developed to do parallel computations on matrices. Many algorithms including matrix operations of addition, multiplication, transposition and inversion, summation and Fast Fourier transformations, and partial differential equation solutions have been developed for array processors. The ILLIAC IV and Burroughs Scientific Processor are array processors. The second subgroup is that of associative processors. Associative processors

access content addressable memories. These systems are a special class of array or SIMD computers. As such they are applied to specific specialized problems, usually related to fast information retrieval and data base retrieval. Examples of associative processors are the Burroughs' Parallel Element Processing Ensemble, PEPE, and Goodyear Aerospace STARAN. The PEPE accomplishes real-time radar tracking of antiballistic missiles, and the STARAN computer performs image processing.

3.2 The ILLIAC IV - The Computer and Its Beginnings

This section discusses the work of some early researchers in the area of array systems and the initial steps to implement the first such computer system.

The concepts of array processors had their beginnings early in the history of digital computers. In 1958, S.H. Unger proposed a two dimensional array of PEs operating in lockstep under a common control unit [41, p. 394]. In 1962, Daniel L. Slotnick, et al., proposed the SOLOMON computer [9]. The SOLOMON introduced a high degree of parallelism. This parallelism may be outlined by four principle features:

1) A single control unit broadcasts a single instruction (single instruction stream) to a large array of arithmetic units, each processing distinct data elements (multiple data stream) in lockstep fashion.

2) In addition to instructions, the control unit also broadcasts memory addresses and global data values.

3) Local enable/disable flip-flops allowed individual arithmetic units to execute only selected instructions.

4) Processing elements had nearest-neighbor connections to provide direct communication. These communication channels operated simultaneously [9].

Studies of these features indicated that such a parallel approach was feasible by the late sixties due to the advent of LSI circuitry. The work to create a machine based on the SOLOMON description was initiated by the Department of Computer Science of the University of Illinois in the late 1960's. The Illinois Array Computer, better known as the ILLIAC, was originally designed to contain 256 processing elements arranged in four reconfigurable SOLOMON-like arrays of 64 processors each. Each array of 64 processors, or quadrant, was to be directed by its own control unit. The four control units were to be capable of independent processing. Thus a multiple-single-instruction stream - multiple-data stream or MSIMD parallelism was to be implemented [Figure 4]. However, due to cost escalation and schedule delays the system was ultimately limited to one set of the 64 processors and one control unit. Although the ILLIAC IV is no longer operational, it is of interest as it was the

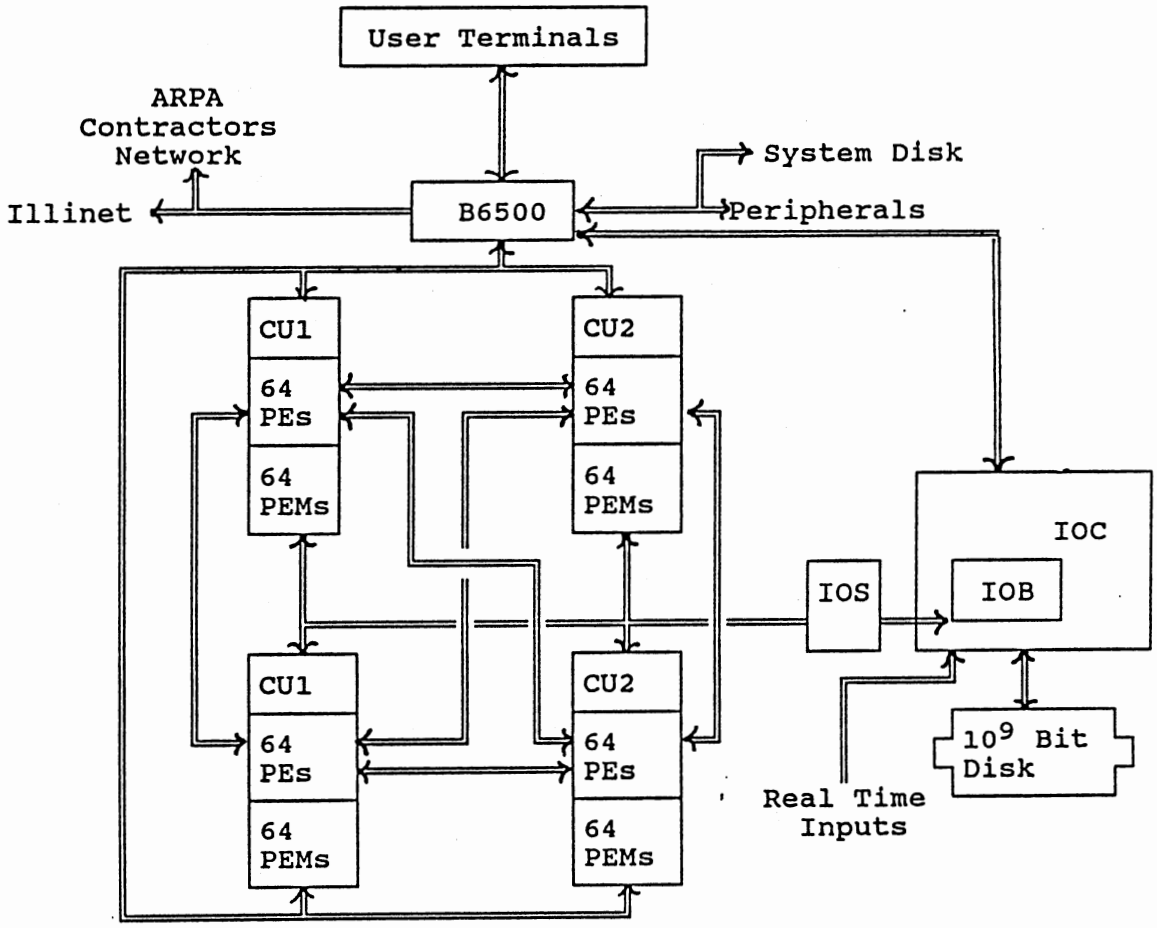


Figure 4. The Original 4-Quadrant ILLIAC IV System with Host [54]

first major array supercomputer developed and is a direct predecessor of the Burroughs Scientific Processor and the Phoenix project of Feierbach and Stevenson [41, p. 394]. Further, the ILLIAC demonstrates the basic concepts of an array system in a simple straight forward manner; its concepts may be extrapolated to more complex array systems such as the Connection Machine built by Danny Hillis and the Massively Parallel Processor (MPP) from Goodyear Aerospace [59].

3.2.1 The Components of the ILLIAC IV

This section details the structure of the components of the ILLIAC array processor and the organization of those components within the system. Section 3.2.2 shows how the ILLIAC memory, PEs, and CU work together to implement processing of matrices.

The basic structure of the ILLIAC IV computer is shown as it was originally conceived in Figure 4 and as it was finally built in Figure 2.

3.2.1.1 The ILLIAC Host Computer. The ILLIAC had a Burroughs B6500 that acted as a front-end for the system [Figures 2 and 4]. The B6500 was timeshared by ILLIAC IV, its highest priority user, and several other terminal users, ARPA and ILLINET networks. A high speed 10^9 - bit head-per-track parallel access Burroughs disk system was directly attached to the array. When a user was ready to

run, he would request space on this disk for his programs and data files. ILLIAC's control unit program memory and the PEMs data memory would be loaded from the disk and all output from the user program would be written to the disk. As the host computer, the B6500 held and executed the ILLIAC operating system. It was to the B6500 that the user issued his request for space on the ILLIAC I/O disk and for execution time on the ILLIAC system. The host administered batch mode job scheduling on the ILLIAC. It oversaw all array-disk I/O and the loading of programs and data into the array processor system.

3.2.1.2 ILLIAC Memory and Operand Access. In the ILLIAC System, each PE connected directly to one and only one PEM. Each PEM was composed of 2048 64-bit random access words. While each PE referenced only its own PEM, the CU accessed the entire combined PEM system. Both data and instructions were stored in the PEMs. Data to be processed by an individual PE was loaded in its associated PEM.

Each address used by a PE to access an operand within its PEM, a local operand, contained three components:

1) a fixed value contained in the instruction (analogous to the displacement value in an IBM-370 instruction);

2) a CU base value added by the CU from a CU accumulating register;

3) a local PE index value added by the PE from a PE register prior to PEM access.

Thus, when an instruction was broadcast from the control unit, each PEM access could be tailored to the specific operand load pattern characteristic to that PE - PEM organization. Global values, operands to be processed by all PEs together, were fetched and stored by the control unit and broadcast to the PEs through the instruction involving the value. Not only did this have the benefit of eliminating the need for duplicate copies in each PEM, but it also allowed for a degree of parallelism in that these global values could be fetched by the control unit while the PEs were executing.

3.2.1.3 ILLIAC Processing Elements. Each PE performed local indexing for operand fetches and executed the data computations dictated by the CU.

Each PE was composed of the following units [Figure 5]:

1) For holding operands and results, there were four 64-bit registers:

i) register A was the accumulator,

ii) register B held the operand to be processed with the accumulated value,

iii) register R held the multiplicand and was used for routing data between PEs,

iv) register S was a general purpose storage area,

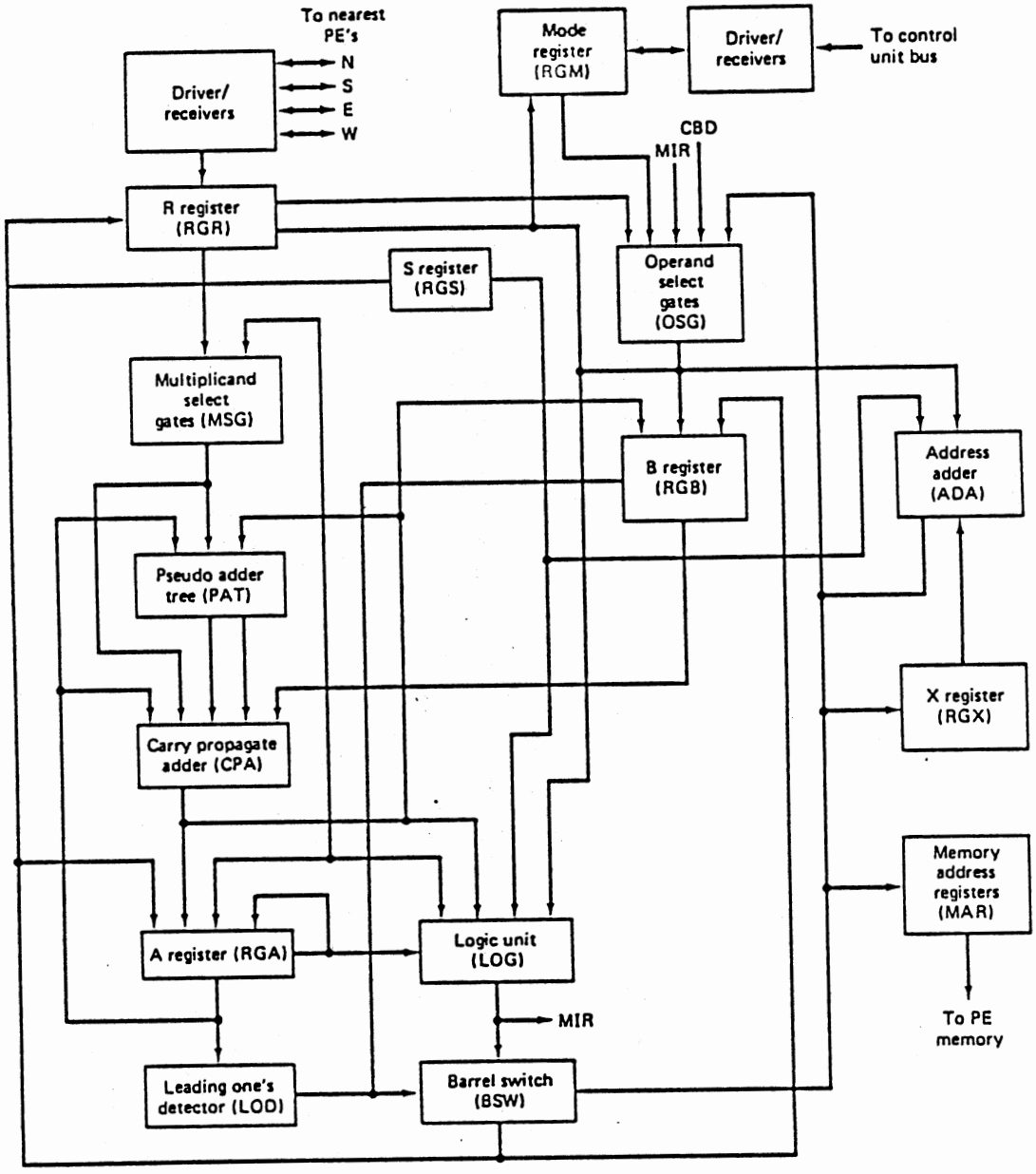


Figure 5. ILLIAC IV Processing Element Block Diagram [48, 9]

2) There were modules for performing

i) arithmetic operations--the adder/multiplier units, the multiplicand select gate, pseudo adder tree, and carry propagate adder,

ii) Boolean operations--the logic unit,

iii) shifting operations--the barrel switch,

3) Memory addresses were calculated by the address adder. It added the contents of the local index register to the address broadcast with instruction by the control unit. The index register (RGX) was a 16-bit register. The result of this calculation was sent to the memory address registers (MAR) for PEM access.

4) results of tests were held in an 8-bit mode register.

3.2.1.4 PE Reconfigurability and Enablement. A processing element could be reconfigured into either a floating point 64-bit word processor, or two floating point 32-bit word subprocessors, or eight 8-bit binary word subprocessors. By utilizing these data formats, the array of 64 PEs could process 64, 128, or 512 data items at a time.

Each PEM could be either enabled or disabled. Two bits of the mode register controlled the enablement of the PE. When the PE was configured to a 64-bit mode only one of the bits was monitored. When the PE was configured to two 32-bit subprocessors both bits were monitored, one for

each subprocessor. If the PE were configured to eight 8-bit subprocessors, the individual subprocessors did not have separate enable/disable modes.

3.2.1.5 Fault Detection. Additional bits in the mode register established masking information; other bits were set by arithmetic faults such as overflow and underflow. Fault bits were monitored continuously by the CU to detect a fault condition and to begin a CU trap.

3.2.1.6 The ILLIAC Interconnection Network. The PEs were linked together so that data could be transferred from PE to PE. As mentioned earlier in the discussion of general array systems, this was implemented by an interconnection network. This network established a 64-bit wide routing path from each PE to four of its nearest neighbors. The interconnection functions applied were

$$f(i) = i \pm 1 \pmod{64}$$

or $$f(i) = i \pm 8 \pmod{64}$$

where i = 0, 1,...,63 identified the address of each PE.

For example, if PE_{57} were enabled, it could transfer data to one and only one of the following: PE_{56}, PE_{58}, PE_{49}, or PE_{1}. Similarly, PE_{0} could receive data from one of the following: PE_{63}, PE_{1}, PE_{8}, or PE_{56}. Thus, when data was to be routed, all enabled PEs might transfer the contents of their routing register, R, to their neighbor PE + 8 positions away. All enabled PEs must execute the same transfer operation under control of the CU.

Logically, the arrays could be considered to be

positioned in an 8x8 array with nearest neighbor connections and wraparound end connections [Figure 6]. The maximum number of data transfers required to shift data from any PE to any other would be 7. However, transfers numbering more than 2 steps were rare [9].

3.2.1.7 The ILLIAC Control Unit. The ILLIAC instruction set was composed of two distinct types: those which were executed by the CU (branching, operating on common global values) and those which were executed by the PEs. Instructions were fetched from the combined memory [Figure 7 and Figure 8] and flowed into the control unit's instruction buffer on the control unit bus. The instructions were loaded into the 64 word instruction buffer in blocks of 8 words (each instruction was 32-bits in length and each word was 64 bits, giving 16 instructions per block). The von Neumann style program counter maintained standard sequentiality in program execution via a mapping process facilitated by a content addressable memory. As control advanced, each instruction was copied into the instruction register and sent to the advanced instruction station (ADVAST). In ADVAST, the instruction was decoded. If it was a CU instruction, it was executed in ADVAST; otherwise, ADVAST processed address or operand values, as necessary, and stacked the results into the final queue to await broadcasting to the

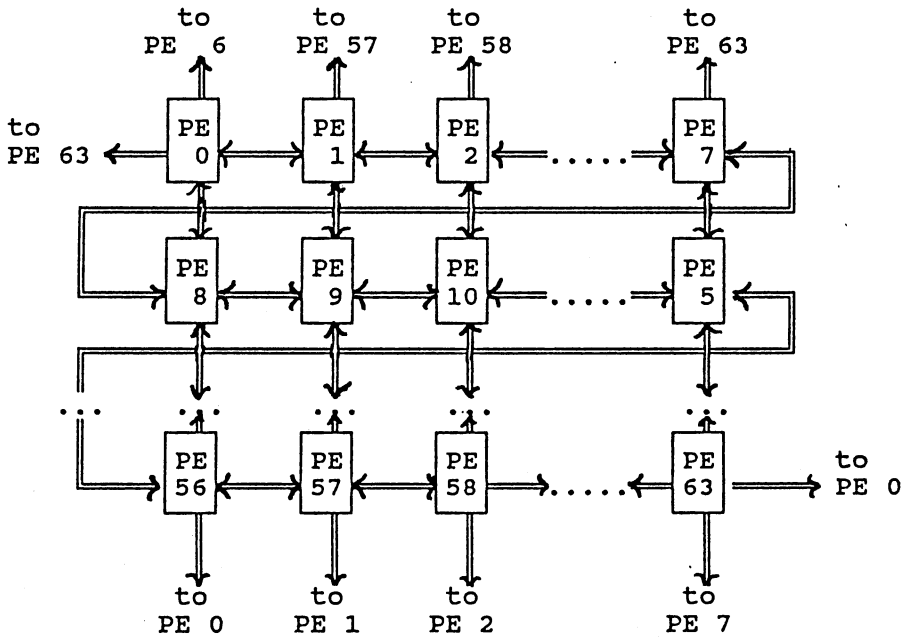


Figure 6. ILLIAC IV Nearest Neighbor Communication Network [48]

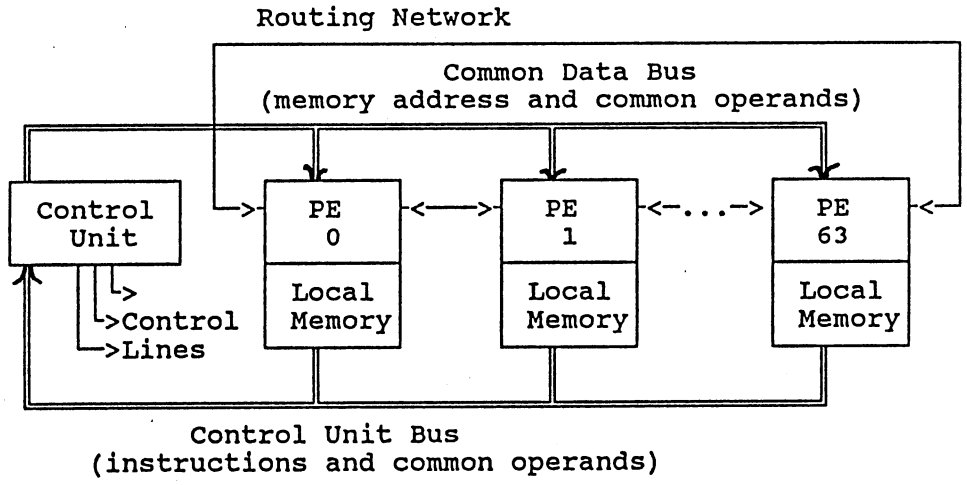


Figure 7. ILLIAC IV array organization with common bus [9, 48]

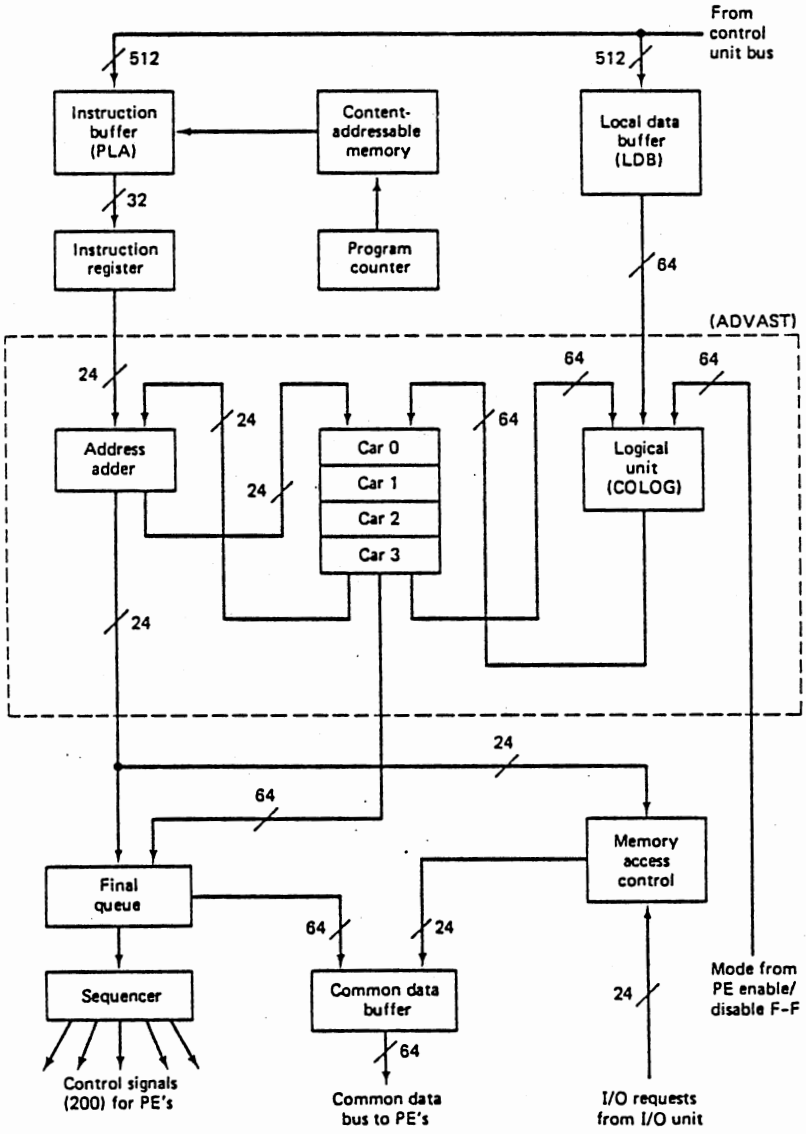


Figure 8. ILLIAC IV Control Unit Block Diagram [9, 48]

PEs. The PE instructions were extracted in sequence from the final queue, taken to the final instruction station sequencer, and transmitted via control pulses to all the PEs. Scalar values were passed from the final queue to the common data buffer and onto the common data bus to the PEs.

3.2.1.8 Inherent Parallelism within the Control Unit. The PE instruction Final Queue allowed for a degree of instruction parallelism. The execution of CU instructions in ADVAST could be overlapped with execution of PE instructions in the processing elements.

CU's instruction buffer held 64 words or 128 instructions. This size was believed to be ample to hold a loop structure of average size. When the instruction or program counter had progressed to the eighth instruction in a block of 16 instructions, fetch of the next block was initiated. The possibility of a branch operation was ignored. If the next block to be executed was already present in the instruction buffer, then the fetch operation was immediately aborted. If the block was not present in the buffer then the next block was fetched from the combined array memory. Thus a loop of a size small enough to fit in the instruction buffer could execute until exit without accessing the relatively slow array memory. Fetch of a new block to the instruction buffer from array memory required approximately the same amount of time as executing 8 instructions. Thus, if execution

continued straight line in the old block, the new block would be in place in the instruction buffer by the time execution of the old block was complete. In this way, an additional element of parallelism was introduced into the ILLIAC processing.

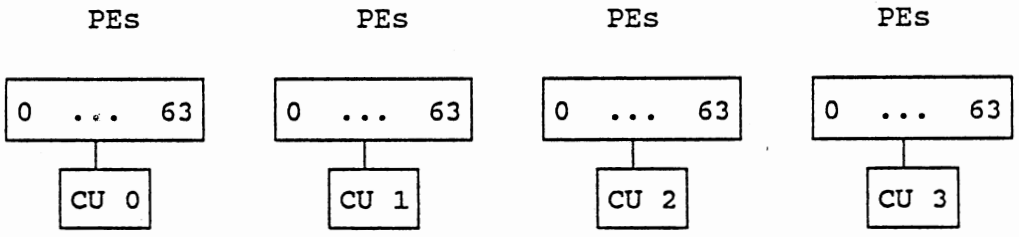
All these time saving strategies needed to be known to the programmer in order for the most efficient use of the hardware to be made. This resulted in increased programming time and costs. Also, it required considerable expenditure of effort in developing optimizing compilers for the system.

3.2.1.9 Proposed Reconfigurability for the ILLIAC IV. The original MSIMD design of the ILLIAC was as indicated in Figure 4. Under this original plan the Burroughs B6500 host computer was to have the capability of reconfiguring the 256 processing elements into the following 3 distinct configurations [Figure 9]:

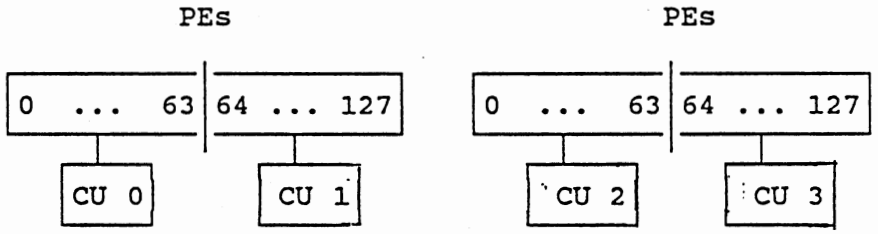
1) Four arrays of 64 PEs under control of the four control units; each CU executing its own unique program, fetched from its own array memory. Under this configuration the interconnection network or routing scheme functions were as given earlier, $f(i) = i \pm 1 \pmod{64}$ or $f(i) = i \pm 8 \pmod{64}$.

2) Two arrays of 128 PEs. Each 128 PE array is controlled by two CUs.

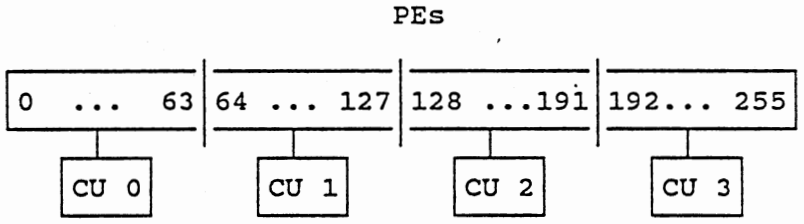
3) One array of 256 PEs under control of four control units.



Configuration 1: Four Arrays of 64 PEs. Each Array Under the Control of One Control Unit. Addresses of PEs Range from 0 to 63.



Configuration 2: Two Arrays of 128 PEs. Each Array Under the Control of Two Control Units. Addresses of PEs Range from 0 to 127.



Configuration 3: One Array of 256 PEs Under the Control of One Control Unit. Addresses of PEs Range from 0 to 255.

Figure 9. Possible Configurations of ILLIAC IV

In configurations 2) and 3), the control units controlling a common array were to fetch their instructions from a common instruction stream. Such reconfiguration techniques would have allowed the number of distinct instruction streams to be 1, 2, or 4; and would have allowed considerable latitude in dealing with data sets of various dimensions. The multiple control units controlling one array could execute asynchronously except when fetching new instruction blocks, routing data between PEs, implementing branch instructions, and changing configurations. Configurations 2) and 3) above required the routing paths to be restructured so that the interconnection functions could be described as

$$f(i) = i \pm 1 \pmod{N}$$

or $$f(i) = i \pm 8 \pmod{N}$$

where N was the number of PEs in the array and i was the address of each PE relative to the new array size.

3.2.2 Processing Dimensional Structures, an Example.

The ILLIAC, like other array processors, was developed primarily for processing dimensional data sets. To get a brief feel for how the array processor was used, consider an array of 3 enabled PEs designated to process a 3x3 matrix A. By loading the matrix A into the PEM memories in a skewed fashion, each row and each column may be accessed [Figure 10]. Suppose the PEs are to multiply

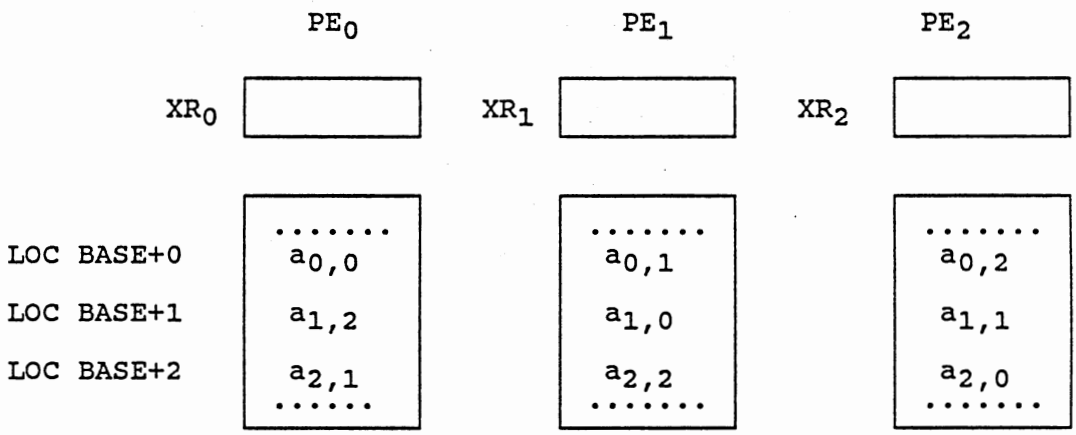


Figure 10. A Two by Two Array Loaded Skewed Fashion into PEMs 0, 1, and 2. XR_i is the Index Register of PE_i.

row 2 by the scalar b. The Index Registers, XR_i, i = 0,1,2, would be set to 2. Then the CU would broadcast the scalar b, the base address and the control pulses to multiply. Each PE_i would add its own index value, $c(XR_i)$ = 2, to the base address, fetch from location base + $c(XR_i)$ (= base + 2) and multiply the value by b. Thus each element of row 2 would be processed simultaneously.

Alternatively, suppose the PEs are to multiply a column by a scalar b. Each Index Register, XR_i would be set to ((i - j) (mod 3)), where j is the column number.

If j = 1, then XR_0 = ((0 - 1) (mod 3)) = 2;

XR_1 = ((1 - 1) (mod 3)) = 0;

XR_2 = ((2 - 1) (mod 3)) = 1.

The process would then proceed as before, each PE adding its index register value to the base address to access the operand to be multiplied. Thus all elements in column 1 would be processed in lockstep. Any other row or column could be accessed in a manner similar to that just described.

The size of the matrix could be extended from 1x1 to MxN, where M < 2048 and N $\leq$ 64; (each of the 64 PEMs contained 2048 64 bit words. Some of these words were used to hold instructions and global data). These figures for M and N would apply for a 64 bit element format. Larger arrays could be handled; however, reconfiguration of the PEs for fewer bits per element and/or alternate mappings of the matrix onto memory would be required.

3.3 Summary

This chapter presents the basic elements of an array processor. It demonstrates the method by which multiple data elements may be processed at one time using the lockstep action of multiple processing elements under the control of a single control unit. In this single-instruction-stream multiple-data-stream environment, data parallelism is established.

This chapter reviews the ILLIAC IV array processor and demonstrates how an array processor may be utilized to process matrices.

CHAPTER IV

PIPELINED COMPUTERS: THE HEP

4.0 Introduction to Pipelining

This section introduces pipelining and its fundamental concepts and elements. Also, it introduces pipeline configurations and classifications that are used to either describe or identify various forms of pipelines which may be found.

Pipelining is another technique frequently used to implement parallelism in a computer architecture. The parallelism introduced by pipelining is quite distinct from that of array systems. In an array system, a basic function such as that performed by a PE is replicated many times and each replica performs the same function at the same time. Pipelining, on the other hand, takes the same function and partitions it into many autonomous but interconnected subfunctions. Input flows from subfunction to subfunction much as fluid in a physical pipeline; or, as products may move from station to station on an assembly line. Each subfunction may be performed during the same time span but on different input. Throughput through the pipe is directly dependent on the rate at which input enters the pipe; once the pipe is full, enter

rate is the same as exit rate. In general, if some function with a straightforward design takes T time units to complete, then a full pipeline designed to perform the same function but divided into N subfunctions may produce a result every T/N time units [Figure 11]. Such pipelining can deliver an N-fold increase in performance. Pipelining can increase the parallelism of a computer system and can deliver dramatic performance gains.

The hardware (combinational circuits) required to perform each subfunction is called a stage. Thus input flows from stage to stage until processing is complete. In order for input to pass in orderly clocked intervals from stage to stage, each stage should perform its associated subfunction in the same amount of time. Frequently, this is not possible precisely; but, it is the ideal. When each stage executes its function in the same amount of time, the stages can operate synchronously with full resource utilization. When the delays are unequal, the stages must be timed for the slowest stage. The slowest stage becomes the bottleneck in the process flow. To facilitate the passing of input from stage to stage, data is buffered between stages in fast registers, termed latches. These registers are so named because they are frequently implemented with the hardware module referred to as a latch. A latch hardware module is a limited form of clocked flip-flop that is activated by a positive, or

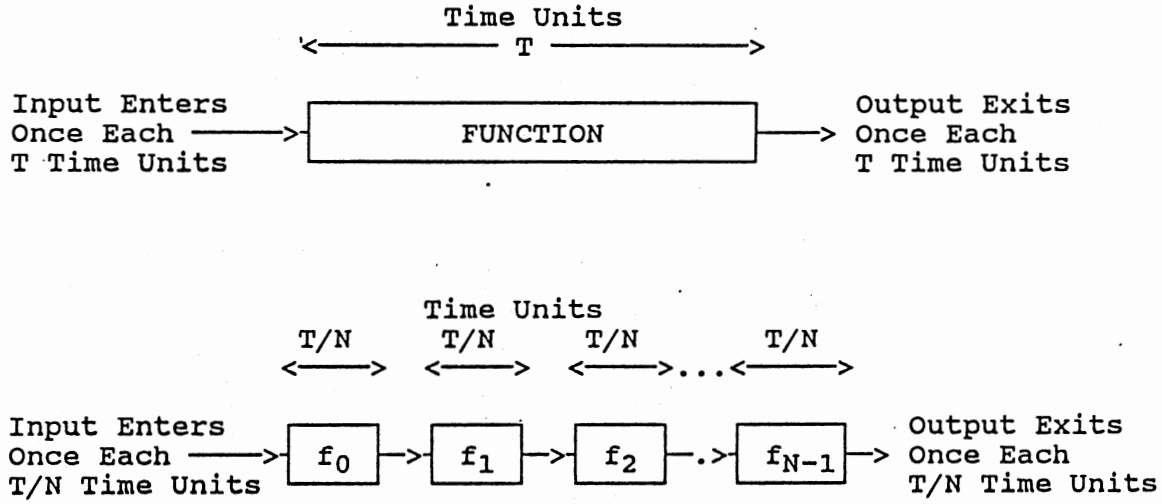


Figure 11. Execution of FUNCTION Equivalent to Execution of $f_0f_1f_2...f_{N-1}$.

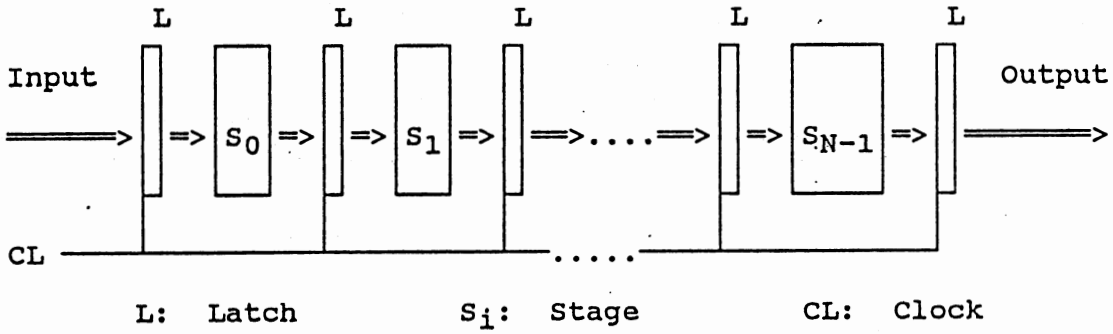


Figure 12. Pipeline with Latches between Stages [52]

high, level on the clock input. D-type latches which change their state to match their input are especially appropriate for this type register. These registers may also be called staging platforms or reservation stations. The latches then act as holding areas for retaining semiprocessed input between unequal delay time stages [Figure 12].

4.0.1 Pipeline Configurations

There are many distinct pipeline configurations. These configurations may be categorized as linear and non-linear. The simplest of these is termed a linear pipeline. The pipeline in Figure 12 is a linear pipeline. A linear pipeline is characterized by the fact that each stage, S_j, receives its input only from stage S_i, where $j = i + 1$ [Figure 13.a].

In addition to the simple linear configuration, pipelines may also be expanded to more general configurations in which a stage may receive input from some stage several steps backward or forward in the subfunction sequence. More precisely stated, a pipeline may contain feed forward connections such that some stage, S_j, receives input from another stage S_i, where $j > i + 1$ [Figure 13.b]. Also, a pipeline may contain feedback connections in which some stage, S_i, receives input from some stage, S_j, such that $j \geq i$ [Figure 13.c].

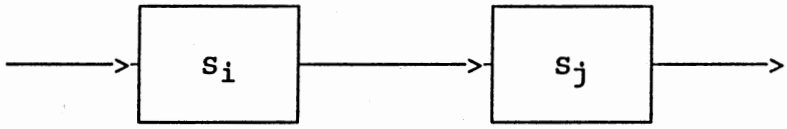


Figure 13.a. Linear Pipeline Connection.
S_j only receives input
from S_i where j = i+1

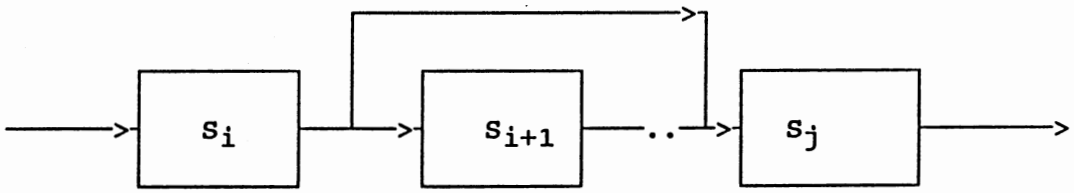
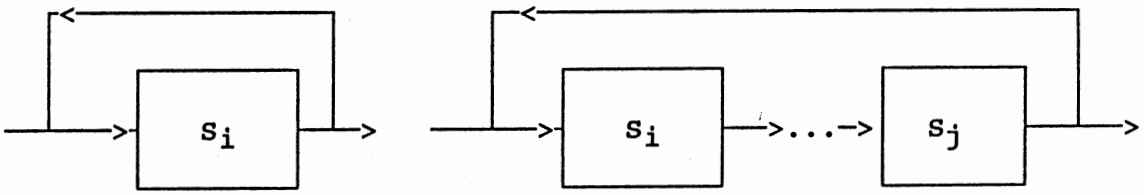


Figure 13.b. Feed Forward Connection.
S_j may receive input
from S_i where j > i+1



j = i j > i

Figure 13.c. Feed Backward Connection.
S_i receives input from
S_j where $j \geq i$

4.0.2 Classifications of Pipelines.

Based on the functional configurations of a pipeline, and the control strategies used to implement it, certain terms can be used to classify a pipeline.

4.0.2.1 Unifunctional vs. Multifunctional. A pipeline may be termed unifunctional or multifunctional. A unifunctional pipeline can evaluate or perform one and only one function. A multifunctional pipeline can perform a set of functions where each function has its own peculiar stage sequence or configuration.

4.0.2.2 Static vs. Dynamic Multifunctional Pipelines. The manner in which a pipeline configuration is controlled to implement the performance of multifunctions is indicated by terming them either static or dynamic. A static pipeline is one such that at any instant in time only one configuration is active and only one function is under evaluation. Clearly, a unifunctional pipeline is always static. A static multifunctional pipeline implies that only one of the possible functions of the pipe will be performed over some period of time so that a sequence of inputs may be streamed into it. Thus, inputs which require the same functional processing are grouped together and streamed one after the other into the pipe. When performance of a different function is required then the pipe must be

reconfigured. This implies that incoming input which require the new functional processing must be delayed until the pipe empties and the stage connections are altered appropriately. When the stage sequence has been reconfigured then inputs may again stream into the pipe.

Dynamic multifunctional pipes permit pipelining among several active configurations at the same time. Thus, several functions may be under evaluation at the same time. Each distinct set of inputs clocked into the pipeline will follow a functional path distinct from other inputs requiring alternate functional processing. Such pipelines obviously require elaborate control and sequencing techniques.

4.1 Pipeline Input Sequencing

This section discusses how to determine when inputs may be allowed to enter a non-linear pipeline. It discusses the use of special tools, the reservation table and collision vector, which may be used to make this determination.

When considering a non-linear pipeline, an important concept that must be dealt with is that of sequencing. That is, controlling the time of entry for each input value. If an evaluation of a function is initiated at a time t_i and a second initiation of the function is made at time t_j, where $j > i$, it may be that both functional evaluations will require the use of the same stage at the

same time. Such a condition is termed a collision. In the determination of viable function initiation sequences a common tool is a reservation table. From the reservation table, a collision vector can be created. The collision vector will indicate proper sequencing of the input values for the pipeline. These concepts will be investigated in greater detail in the following subsections.

4.1.1 A Function's Computational Sequence

Each function is defined by its computational sequence; that is, the sequence of stages through which inputs are piped in order to produce the required functional output. This computational sequence will determine the allowable time table for inputs to the pipeline.

As an example, consider the non-linear pipeline of Figure 14. The crosses refer to data multiplexors. Each multiplexor is used to select among multiple connection paths in evaluating different functions. Thus, this pipeline is multifunctional, and for the purposes of this example, static.

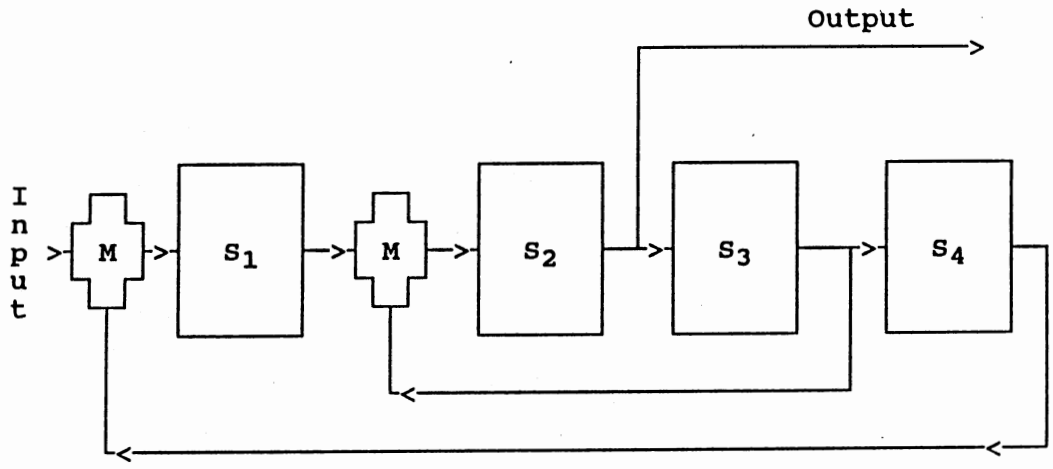


Figure 14. Example Static Non-Linear Pipeline. Each Stage Requires an Equal Time to Execute

Assuming each stage requires an equal time unit to perform its subfunction, one function's definition may require the inputs to traverse the stages in the following computational sequence:

at time t_0, input passes to S_1;

at time t_1, input passes to S_2;

at time t_2, input passes to S_3;

at time t_3, input passes to S_4;

at time t_4, input passes to S_1;

at time t_5, input passes to S_2;

at time t_6, input passes to S_3;

at time t_7, input passes to S_2;

and from S_2 out of the pipeline.

4.1.2 Reservation Tables

The computational sequence of a function can be indicated clearly and graphically in a reservation table. In a reservation table, row i corresponds to stage S_i, and column j corresponds to time t_j. A mark in a square (i,j) indicates use of stage S_i at time t_j. Multiple marks in a column indicate concurrent use of two or more stages while multiple marks in a row indicate reuse of the same stage in overall functional evaluation performed in the pipeline.

The computational sequence described in section 4.1.1 above would lead to the reservation table shown in Figure 15.

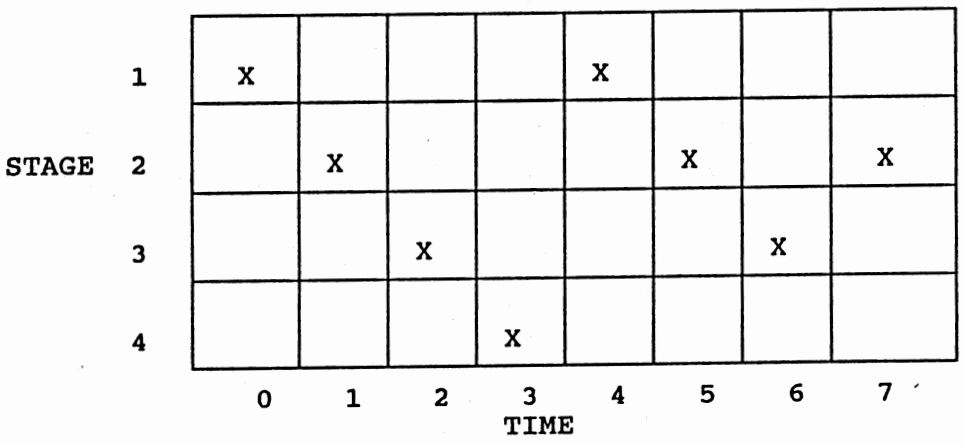


Figure 15. Reservation Table for Pipeline of Figure 14. An Entry in Row i and Column j Indicates the Use of Stage i at Time j

4.1.3 Forbidden Latency and the Collision Vector

A scheduling strategy may be developed for the pipeline based on the computational sequence and reservation table of a function. The strategy should schedule inputs into the pipe in such a way as to prevent collisions and to maximize the throughput of the pipe. The technique for implementing such a strategy is based upon the concepts of forbidden latencies and collision vectors. Simple latency is the time between successive initiations of the pipeline.

If stage S_i is in use at times t_m and t_n, then the difference $|t_m - t_n|$ is called a forbidden latency. If two initiations of the pipeline occur $|t_m - t_n|$ time units apart, a collision will be generated. In the example of Figures 14 and 15, the forbidden latency for stage S_1 is 4; for stage S_2, it is 2 and 4; and, for stage S_3, it is 4. There is no forbidden latency associated with stage S_4. Zero is always a forbidden latency (two inputs cannot begin at the same time). The set of forbidden latencies for all stages establishes a forbidden list. The forbidden list in our example in section 4.1.1 is {0, 2, 4}.

The collision vector is constructed from the forbidden list. The collision vector has d elements,

where d is the number of time units required to traverse the pipeline (compute time).

If $C = (c_0, c_1, c_2, \ldots, c_{d-1})$ is the collision vector and i is an element of the forbidden list, then $c_i = 1$; otherwise, $c_i = 0$. For the pipeline of our example, d = 8 and C = (1, 0, 1, 0, 1, 0, 0, 0). By use of the collision vector, a simple control mechanism can be used to prevent collisions. Before initiating a new computation, the collision vector can be tested. If for each previous initiation of the pipeline, the difference between the previous initiation time and a new initiation time is i and $c_i = 0$, then the new initiation is allowed; if $c_i = 1$, the initiation is delayed. A control strategy which minimizes the immediate delay time and allows initiation of the pipeline as soon as the control vector allows is called a greedy strategy. Performance analysis of various control strategies indicate that a greedy strategy may not insure the maximum throughput for the pipe while a more patient one may increase throughput [52, p. 80].

4.2 Pipeline Applications

There are two functional areas where pipelining is employed most often. One is the instruction fetch-execute process of the control unit, instruction pipelining. The other is computation of the arithmetic/logic unit, arithmetic pipelines. The following sections, discuss these two important areas.

4.2.1. Instruction Pipelining

This section presents instruction pipelining. It details some problems (or hazards) which are inherent in this technique for speeding up a control unit's operation. Some possible techniques for resolving these problems are discussed. Finally, it reviews the ILLIAC's instruction cycle in the context of pipelining.

Instruction execution by the control unit may be partitioned into several distinct subfunctional steps: instruction fetch, program counter update, operation code decode, compute addresses of operands, operand fetch, execute, operand store, and housekeeping. These steps may be accomplished by a series of stages to establish an instruction pipeline. Thus one instruction may be fetched while another is decoded, another has its operand addresses calculated, etc.

In a non-pipelined control unit, total execution of one instruction is completed before initiation of the next is begun. In a non-pipelined computer, the order of execution matches the logical order of the program. In a pipelined design, one instruction is begun before its predecessor is completed. This difference can cause problems if not adequately dealt with during the design phase of the pipeline.

4.2.1.1 Hazards and Their Classifications. An instruction which depends on the preceding instruction's results may enter the pipe and begin the execution sequence before the preceding instruction has completed the sequence. Such critical dependencies between instructions generate hazards. A data hazard occurs when two separate instructions access or update the same storage location while their execution is overlapped within the instruction pipeline. These hazards must be detected by the computer and resolved so that the final product of the instruction sequence is that expected by the programmer. Such resolution may prevent the pipeline from accepting inputs at the maximum rate.

These types of hazards are possible especially when the control structure of the pipe is such that instructions may exit the pipe in an order other than that in which they entered. This may occur in pipes with multiple execution stages. That is, when an instruction has progressed through the pipe to the execute stage, it may be routed to one of several parallel execute stages. An ADD instruction going to one stage while a COMPARE would pass into another, etc. Each distinct execute stage may require a different amount of time for completion, allowing one instruction to exit the pipe before another which preceded it into the pipeline.

Hazards may be grouped into several different classifications. Some simple examples demonstrate those

classifications. For the purpose of discussion, two instructions I1 and I2 are ordered by having I1 precede I2 into the pipe. I1 and I2 are in different stages within the instruction pipeline. Three primary classes of hazards exist:

1) The Read after Write hazard exists when I1 updates a data element which I2 reads. For example, consider the 2-address, IBM-370 instruction sequence

```
I1   ST   1,DATA    /* ST = STORE */
I2   A    2,DATA    /* A = ADD    */
```

If the ADD is in the operand fetch stage while the STORE is still in the execute stage, the contents of register 1 may not reside in DATA when it is fetched for ADD. Some previous value of DATA may be added to register 2.

2) The Write after Read hazard exists when I1 reads a data item which is to be updated by I2. For example, consider the following IBM-370 instruction sequence.

```
I1   A   2,DATA    /* A = ADD     */
I2   LR  2,3       /* LR = LOAD REGISTER */
```

The LR instruction may pass over the operand fetch stage since both of its operands are in registers and into the execute stage while the ADD instruction is still completing its operand fetch. Thus register 2 may have been updated by the LR instruction before the ADD has an opportunity to act on it. The value in register 2 that is actually added to DATA may be "too new."

3) A Write after Write hazard exists whenever I1 and I2 both attempt to update the same location but I2 completes before I1. For example, consider the following IBM-370 instruction sequence.

```
I1    STM    14,12,SAVE    /* STM = STORE MULTIPLE
                                 REGISTERS */
I2    ST     14,Save+56    /* ST = STORE */
```

Both instructions update a location 14 full words down from location SAVE, but, the STM will take longer to update the location as it must first store in the 13 full words preceding SAVE + 56. Although the ST will enter its execute stage after the STM, it will complete execution while the STM continues its execution stage; and finally as a last activity places the contents of register 12 into SAVE + 56.... overwriting the value placed there by the ST.

An additional interesting hazard may exist. This hazard is a result of self-modifying code. In this situation I1 may alter I2 itself. Thus a Read after Write hazard is established between the write action of I1 and the instruction fetch action of the instruction pipeline.

4.2.1.2 Hazard Detection and Resolution. The detection and resolution of hazards is a major consideration in the design of an instruction pipeline. There are two common approaches in hazard detection. Both approaches imply the maintenance of a set of facts which

characterize each instruction in the pipeline. Each characterizing set includes an indication of all locations (registers, memory, etc.) whose contents are updated by the execution of the corresponding instruction. In the first hazard detection approach, the characteristics of an instruction in the pipeline instruction fetch stage are compared with all those already in the pipeline. A hazard is detected if there is any intersection between the sets of instruction characteristics. The second approach is similar but more complex in its implementation. An instruction is allowed to flow through the pipe in the usual way until any element in its characterizing set is required. At that point, the control unit checks the intersection between the instruction in question's characterizing set and those of all other instructions in the pipe. If any non-null intersection exists, then a hazard is recognized.

Resolution of a detected hazard may be handled in one of two ways. If I1 and I2 are instructions and I1 has preceded I2 into the pipe, then one method of resolution is as follows. If I2 is found to generate a hazard condition with I1, then I2 and all succeeding instructions are halted and prevented from progressing further into the pipeline, while I1 and all other instructions preceding I2 continue through the pipeline. When I1 has passed all stages which could effect I2, then I2 and those instructions succeeding it are allowed to continue through

the pipeline. Although this is a relatively simple technique, it degrades the performance of the pipeline because all stages are not kept busy. It may actually imply a complete emptying of the pipeline. In such a case, output from the pipe cannot resume until the pipe has been filled.

An alternative to this that may be employed, is to halt the progress of I2 when its hazard condition with I1 is detected, but, to allow those instructions logically behind I2 and which have no hazard relation to I2 or any other instruction in the pipeline to stream into and through the pipe. When I1 has passed all the stages which could effect I2, then I2 is allowed to proceed. In this way, some instructions which logically follow I2 may enter the pipe and complete execution before I2. This is perfectly acceptable since their execution and results are independent of I2. Such hazard resolution has value since it keeps the pipeline filled most of the time and thus more productive. But, it is more complex to implement and requires more hardware design overhead.

4.2.1.3 Branching in an Instruction Pipeline. Even worse than the instruction dependencies just discussed, branching and interrupt handling can diminish greatly the performance of an instruction pipeline. Branching alters the program counter and implements nonsequential access to program memory. The instruction fetch stage cannot

continue to fetch its instructions from memory in the usual sequential fashion. However, the address to which the program counter will be updated is not available until the effective address of the branch is evaluated in the later address calculation stage; and in the case of a conditional branch, the conditions of the branch may not be known until some instruction ahead in the pipe completes the execution stage. Resolution of the dilemma has been accomplished in several ways. Two of these ways will be discussed here. For unconditional branches, a simple technique used is to include enough logic in the instruction fetch stage to recognize or decode a branch, calculate its specified effective address and update the program counter appropriately, then continue the fetch function. For conditional branches, an extension of this technique has been used called "guess and correct." Here a "guess" is made as to the likelihood of the branch actually being implemented. The program counter is adjusted according to the indication of the prediction and the fetch function continues in the usual way. When the branch instruction has progressed far enough in the pipe for the correctness of the guess to be ascertained, a check is made to determine if a correct guess was made. If so, the instructions continue to stream through the pipe; if not, all instructions behind the branch in the pipe are aborted, the pipe is flushed, the program counter is updated to the correct branch value and the fetch

function restarts. This technique is viable because statistically over 50% of all conditional branches are taken and certain types of branches (eg. branch on count) are nearly always taken [52, p. 243]. The better the "predictive technique," the better the performance of the pipe. Apparently, if the programmer in such a case knows the guess algorithm, he can write more efficient algorithms. This is a good example of software-hardware interdependencies.

A second technique employed for conditional branches is to have a secondary program counter which is used when a branch is decoded in the fetch stage (as described above). The program counter and the secondary one are updated, one with an address from one side of the branch option, the second with the address from the other side; then tagged instructions from both sides of the branch are fetched into the pipe according to the two program counters. When the branch instruction has progressed far enough into the pipe for the proper pathway to be established, the instructions in the pipe tagged from the wrong side of the branch are aborted. This method involves more instruction fetches and thus can contend with operand fetches from a common memory, but it has the advantage of keeping the pipeline full and the execute stage busy a greater proportion of the time than did the "guess and correct" method.

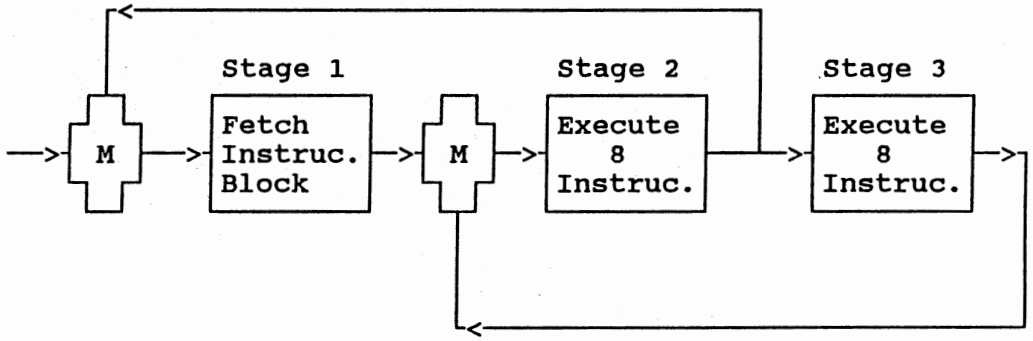
4.2.1.4 Interrupt Handling in an Instruction Pipeline.

Interrupts disrupt the sequential instruction fetch in much the same way that branches do. When an instruction generates an interrupt condition, the interrupt should be handled before the instructions behind it in the pipe are executed. Since interrupts are unpredictable, no technique such as "guess and correct" or "fetch from both sides" is viable. The IBM 360/91 implemented a technique of interrupt handling which has proven successful [66]. In the IBM 360/91, interrupts are categorized as precise and imprecise [52, p. 269]. Precise interrupts are ones that can be detected early in the pipe stage sequence (eg. illegal operation code is detected in decode stage, immediately after fetching). In the case of a precise interrupt, fetch of new instructions is halted. All instructions behind the interrupt generating instruction are aborted while those ahead in the pipe are allowed to flow on through the pipe in the usual way. Imprecise interrupts are generated in stages internal to the pipe (eg. operand fetch or execute stages). In such cases, the pipe contains multiple instructions behind the offending one which have already undergone various stages of processing. To flush them would be counterproductive. When an imprecise interrupt occurs, fetches of new instructions are halted but all instructions which have already entered the pipe are allowed to stream on through to completion.

Whether precise or imprecise, the program counter can be initialized with the interrupt handler address while the pipe is being emptied. Debugging of imprecise interrupts may be difficult due to the nonsequentiality of the offending instruction and the action of the interrupt handler. However, since some interrupts such as I/O interrupts are unrelated to the instructions within the pipeline, such techniques are clearly advisable.

4.2.1.5 A Review of the ILLIAC's Instruction Cycle and Pipelining. The ILLIAC IV instruction fetch-execute function employs an overlap instruction fetch and instruction execute technique whereby a 16 instruction block is fetched while an 8 instruction subblock is executed. This establishes a sequence of subfunctions that act in a nearly pipelined fashion. But, instructions are passed in blocks and processed in blocks; the input to individual stages is not individual instructions as found in modern pipelined systems. One configuration of the stages might be as given in Figure 16 and the related reservation table in Figure 17.

The reservation table indicates a forbidden latency of $|t_2 - t_0|$ for stage 1; but, according to the reservation table, stage 1 should be able to receive inputs at time t_1. In a true pipeline, stage 1 would be active with a new set of inputs at time t_1. But, the ILLIAC does no fetch during execution of the first 8



M: Multiplexor

Figure 16. Possible Stage Sequence for ILLIAC IV Instruction Fetch/Execute Function

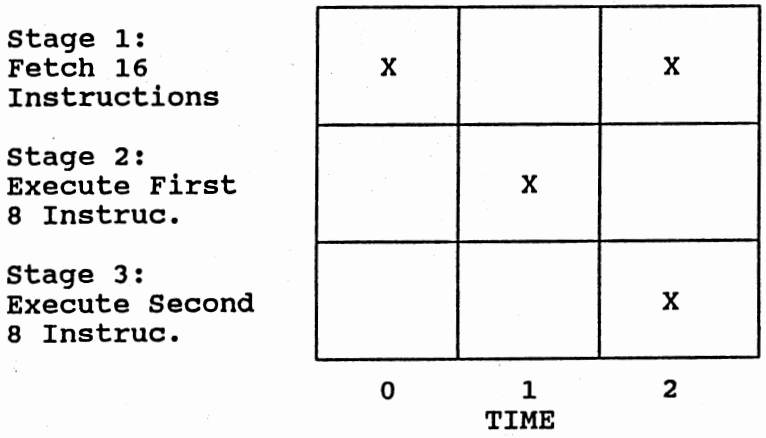


Figure 17. Possible Reservation Table for ILLIAC IV Instruction Fetch/Execute Function. In a True Pipeline Stage 1 Would Be Active at Time t_1 with a New Set of Inputs

instructions in the block. Such a system can only be characterized as overlapped or asynchronous. Asynchronous or overlapped systems have at least one of the following characteristics: 1) dependencies between evaluations; 2) each evaluation may require a different configuration of subfunctions; 3) subfunctions are not closely related; and 4) the time required by each stage is not constant [52, p. 5]. The ILLIAC meets overlap specifications 1), 3) and 4).

4.2.2. Arithmetic Pipelines

This section examines arithmetic pipelines and how they relate to computers termed vector processors.

Arithmetic functions constitute another major application for pipelining. An arithmetic pipeline is like any other pipeline; in this case, the function to be performed is simply the calculation of some arithmetic value. The most common example of an arithmetic pipe is that of a floating point adder where the addition of two floating point values may be broken into a series of subfunction/stages as shown in Figure 18. When a large number of floating point number pairs require addition, the pairs can be streamed through the pipe producing output of one floating point sum for each pair input.

Arithmetic pipelines have been built to perform a wide variety of arithmetic functions such as floating point addition, subtraction, multiplication, division, and

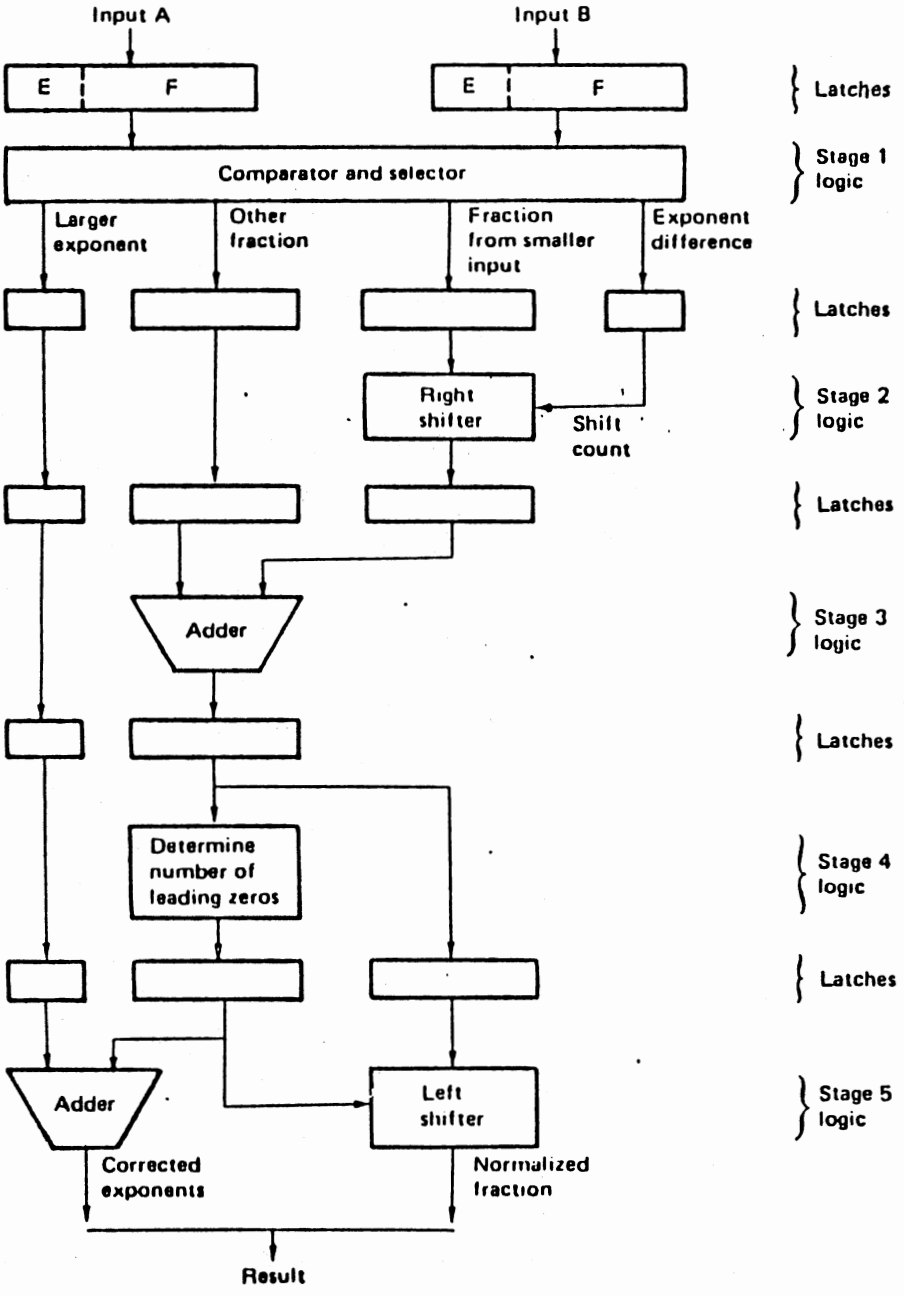


Figure 18. Arithmetic Pipeline to Add Two Floating Point Values [52]

square root functions. Frequently an arithmetic pipe is multifunctional, capable of performing several functions such as those just listed above. If the reconfiguration of the multifunction arithmetic pipe may be controlled by the user programmer at the machine instruction level, the computer architecture is called a vector processor. In a vector processor, a single machine instruction specifies an operation and the location of a set of arithmetic values which are located according to some linear mapping function (vector elements which are stored contiguously or are separated by some stride distance). The pipeline is configured to execute the operation specified and the vector elements are streamed through the pipe. After all the values have been streamed through, the next instruction can request a distinct operation and the arithmetic pipe can again be reconfigured to its specifications. Vector processors were designed especially for processing vectors just as array processors were also developed for that purpose, but each has its own unique architecture.

4.2.3 Pipelining Embedded in Other Parallel Architectures

The parallelism made available through instruction and arithmetic pipelining can be embedded within many architectural environments. For example, an array processor's control unit could employ an instruction

pipeline; the overlapped system of the ILLIAC could be replaced with one. Further, the PEs of an array system could have arithmetic pipelines allowing each PE to process a stream of values concurrently.

In Chapter 5 of this treatise, the Alliant FX/8 multiprocessor is surveyed. The Alliant has both instruction and arithmetic pipelines embedded within each of its multiple CPUs. As is discussed in Chapter 5, it is the multiple CPUs that give the Alliant its multiprocessor standing, but the use of pipelining within each processor extends its exploitation of parallelism.

4.3 The Heterogeneous Element Processor, HEP

In the late 1970's, The Heterogeneous Element Processor or HEP computer was initiated by Denelcor, Inc., under contract to the U.S. Army Ballistics Research Laboratory. By 1981, it was commercially available from Denelcor, Inc. It is a highly pipelined computer capable of implementing a multiple-instruction stream multiple-data stream (MIMD) architecture as described by Flynn [31]. It is capable of executing 10 million instructions per second (MIPS). Because of its ability as an MIMD machine to perform concurrent processes and to establish such a high degree of parallelism, the HEP has generated a great deal of interest.

The HEP computer consists of one or more Process Execution Modules (PEMs) with a common data memory base [Figure 19]. The number of PEMs in a system makes no difference in the way processes are created and managed or in the way they communicate. Only the number of instructions executing at a time is affected; each PEM can have about 12 instructions in some stage of execution at a time. Each PEM consists of an Instruction Processing Unit (IPU) and 3 distinct memory entities. The PEM's memory entities are program, register and constant memory. In addition to the PEM's internal memories, each PEM has an attached local data memory module. Furthermore, all PEMs may access one or more global memory modules through a packet switched network.

This section examines in detail the HEP computer and its intensive use of the pipeline concept. Sections 4.3.1 and 4.3.2 describe in detail the organization of the HEP memories and instruction processing units and the way HEP has of resolving instruction pipeline hazards. Section 4.3.3 describes further use of pipelining made in the HEP data transfer system which simplifies interprocess communication.

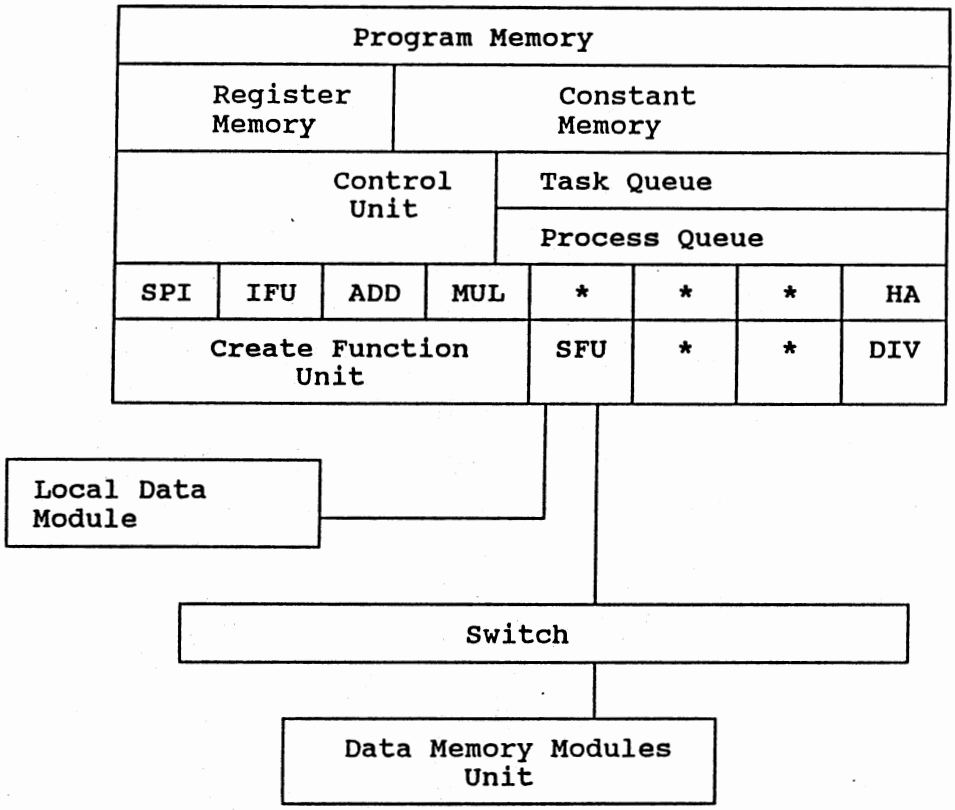


Figure 19. HEP System Showing the Process Execution Module, Switch, and Data Modules Accessible by the PEM.
SFU - Accesses Local Data Synchronously. Accesses Data Memory Asynchronously through the Switch.
DIV - Consists of 8 Distinct Asynchronous Floating Point Divider Modules.
ADD - Performs Synchronously all Floating Point Addition and Subtraction.
MUL - Performs Synchronously all Floating Point Multiplication.
IFU - Performs Synchronously Integer Operations and Logical, Shift, Compares, and Type Conversions.
HA - Hardware Access Unit Reads and Writes Program Memory and Performs all Bit Encode and Decode Operations Synchronously.
SPI - System Performance Instrument Collects Data for Measurement of Performance Synchronously.
Create Funct. Unit - Performs all Operations Affecting PSWs Synchronously.
* - Undefined Units [20, 21]

4.3.1 The HEP Memory System

4.3.1.1 Program Memory of the PEM. Program memory is expandable in 1 megabyte units to 8 megabytes. It is execute-only memory for non-privileged users. One instruction can be fetched every 100 nanoseconds. This rate is important as it makes the program memory consistent with the requirements of the Instruction Processing Unit (IPU).

4.3.1.2 Register Memory of the PEM. Each PEM's register memory consists of 2048 64-bit registers for storing operands and operational results. A process executes at its fastest possible rate when utilizing these registers.

4.3.1.3 Constant Memory of the PEM. Constant memory is a read-only data area for non-privileged users. This area can be loaded during the same time period as the program load for a process and can be accessed during execution to facilitate fast constant retrieval. It consists of 4048 64-bit registers.

4.3.1.4 Data Memory Modules. There exists a fourth memory element in a HEP system. This is called the data memory module. There can be as many as 128 of these in a HEP system. Each module can be from 1 to 8 megabytes in size. One data memory module may be local to a given PEM.

All other data memory modules can only be accessed by a PEM through a high speed packet switching network. It is used for storing most of the data of the system and for communication between processes executing on separate PEMs. Interaction between data memory and the PEM is much slower than that between the PEM and its internal memories (program, register, and constant memories).

4.3.1.5 HEP Hardware Memory Management. The HEP utilizes a dynamic relocatable partitioned memory management system. Each program or job step constitutes a task; each task is assigned a region in program, register, constant, and data memory. The first byte address and last byte address of each region is recorded as base and limit values along with other status information in a Task Status Word (TSW). Effective addresses within each program are assembled as though the program will be loaded at location zero. Then as the program executes, effective addresses from instructions are added to the base value in the Task Status Word to determine a real address for access. Memory is protected by comparing the real address calculated with the limit value. If the real address is larger than the limit value, a memory protection exception is generated. Constant memory is different; it has no limit value, only a base.

4.3.2 The HEP Instruction Processing Unit

The Instruction Processing Unit (IPU) consists of a control unit and function units. The function units are of two varieties, synchronous and asynchronous. The function units are identified in Figure 19.

4.3.2.1 The IPU Pipelines. All synchronous function units are pipelined in eight stages, each with a delay time of 100 nanoseconds. The control unit is also pipelined, performing the instruction fetch, decode, operand address calculation, and operand fetch. Then it passes its results to the appropriate function unit. The control unit can fetch an instruction from program memory to the function units once every 100 nanoseconds. Thus when fully utilized, synchronous function units can produce a result every 100 nanoseconds, giving the 10 MIPS result of which the HEP is capable.

4.3.2.2 The IPU's Task Status Words. A Task Status Word (TSW) is assigned each task, as discussed earlier in the context of memory management. Each task's TSW is maintained in the IPU. The IPU holds a maximum of 16 Task Status Words in a hardware queue. Half of these are allocated for user tasks, the other half for supervisor tasks.

4.3.2.3 The IPU's Process Status Words. The execution of a program constitutes a process. To identify the position of program execution for a process, a Process Status Word (PSW) is maintained for each process. The Process Status Word acts as the program counter for each process. Normally, when a task is ready for execution, it is assigned one PSW, identifying the initial instruction for execution of the task. A task may be modularized by the programmer into a series of subprograms which, if executed in parallel would minimize the time requirements for task execution. By use of a CREATE instruction, the programmer can require additional PSWs to be created for his task, one for each subprogram to be run concurrently. By doing this he is initiating parallel execution of his subprograms, creating concurrent processes.

4.3.2.4 The Task Queue and the Process Queue. The PSWs are maintained in a process queue. Each PSW in the queue is identified by a Process Tag (PT); that is, each Process Tag is a pointer to a unique PSW in the process queue. When a task is loaded, it is assigned a Task Queue as well as a Task Status Word. The PT for each Process Status Word initiated for a task is maintained in this hardware Task Queue. The process queue can hold a maximum of 128 PSWs. Sixty four are allocated for supervisor use. This leaves a total of 64 PSWs available for user use. These are divided in a first-requested, first-allocated

manner among the 8 possible user tasks. If one task can CREATE requests sooner than the other tasks, it may utilize all 64 slots in the process queue. Thus one task could generate 64 concurrent processes, each process working towards the completion of the given task.

When the Control Unit of the IPU fetches an instruction, it accesses Task i's FIFO Task Queue for a PT and logically removes it from the Task Queue. The PT directs the Control Unit to a PSW in the process queue which in turn addresses the correct program memory word containing the instruction to be fetched. The instruction is piped into the Control Unit pipeline where the Task Status Word will be consulted for real operand address calculation, etc., and the PSW is updated. Beginning on the next 100 nanosecond period, the Control Unit accesses Task i + 1 (mod 16)'s Task Queue for the PT pointer to the PSW pointer to the next instruction to be fetched. The next instruction fetched for execution will be from a process distinct from that of the previously fetched instruction.

The PT is not returned logically to the Task Queue until an 800 nanosecond delay has transpired. This is the time required for the instruction to flow through one of the synchronous Function Units (ie. complete execution). After the delay, the PT is returned to its original Task Queue and becomes available once again for selection by

the Control Unit as it makes its round-robin poll of the Task Queues.

4.3.2.5 The Beauty of the HEP Pipelines. This then is what makes the HEP instruction pipeline distinct from a conventional one; each instruction in the pipe is from a distinct different process, a unique instruction stream. There are no instructional dependencies within the pipe! There are no data hazards, no read after write, no write after read hazards! There is no hazard detection and resolution, and no "guess and correct" branching schemes! Each instruction stream is handled as though it were executing on a nonpipelined control unit, one instruction executing at a time.

Additionally, the HEP utilizes its pipeline to obtain its multiple-instruction-stream categorization. Although only one instruction is fetched from program memory each 100 nanoseconds, during an instruction's total execution period, at least 8 instructions will be fetched and each from a different process stream.

4.3.3 Interprocess Communication

Because the HEP was designed to implement concurrent processing, it has built into its register and data memory, hardware access states to facilitate communication between cooperating processes. Data memory access states can be "full" or "empty". A LOAD instruction can be made

to wait if its designated location is "empty" and wait until it is set "full" by a concurrently executing STORE. This setting of states occurs in one machine cycle. Register memory has similar states. An instruction executing on register memory may require both operands to be full and the destination empty before executing and then mark the destination as "reserved" while it is in the pipeline. The programmer can designate when the states should be tested. Thus, the HEP implements in hardware some significant LOCK and UNLOCK, P and V type activities [20].

4.3.3.1 Asynchronous Function Units. Synchronous function units all compute their results in eight 100 nanosecond cycles and access register and constant memory for their operands. There are two asynchronous function units, the Scheduler Function Unit (SFU) and the Divider Function Unit. The Divider contains 8 individual 64-bit floating point divider modules. It can complete a divide instruction in 1700 nanoseconds. The Divider utilizes the reserved state of register memory to prevent a synchronous function unit from utilizing a destination register before it is filled. This acts to prevent a read after write hazard for an instruction which would follow a DIVIDE in a process instruction stream.

The Scheduler Function Unit (SFU) is both synchronous and asynchronous. It executes all instructions involving

data transfers to or from Data Memory. Most Data Memory Modules are connected to a PEM by the packet switching network; but, one module may be local to a PEM. The SFU executes transfers through the switch asynchronously, while those to a local Data Memory are executed synchronously. The SFU is pipelined and can receive a new data transfer request once each machine cycle, 100 nanoseconds. When a Data Memory transfer instruction is piped to the SFU, the PT associated with that instruction fetch is not returned to the Task Queue after the usual delay. The SFU contains 16 queues analogous to the Task Queues of the IPU. The PT is placed into one of these corresponding queues of the SFU instead. The SFU also contains a queue analogous to the process queue of the IPU. In this queue the SFU maintains SFU Status Words (SSW). Each SSW contains enough information about the conditions of the Data Memory transfer to restart it as many times as necessary. If a location is accessed by the SFU, but its access state "full"/"empty" is not that prescribed by the programmer, the SFU aborts and tries again later when the SSW for that transfer comes up again in a round-robin poll of the SSW Queue. This process continues until all the conditions of access are met and the data transfer is completed. Then the PT for the completed instruction is returned to its IPU Task Queue and removed from the SFU. The data has now been transferred as requested and the Control Unit is now able

to access the PT again to fetch the next instruction in that process. Thus, once again, any hazard which could have existed due to the unequal compute time of the SFU with that of the other synchronous Function Units has been averted. Additionally, by virtue of being a pipelined data transfer function, the SFU is in the process of transferring multiple data elements during a given period of time, thereby qualifying it as a Multiple Data Stream computer.

4.3.4 Conclusion on HEP

The designers of HEP made excellent use of the hardware technology available and the reduced cost of RAM. These elements were combined with existing procedures and some new ideas to create a very exciting machine.

4.4 Summary

This chapter presents pipelines. Pipelines allow multiple inputs to be in various stages of processing at any given time. Their use in implementing machine instruction cycles allow the execution of multiple instructions to be under way at any given time. When the instructions piped into the pipeline are from the same process, then hazards may occur. These must be detected and resolved. When the instructions are from different processes, as in the HEP, no hazards exist and execution is from multiple instruction streams. The use of

arithmetic pipelines allow multiple data elements to be operated on within the pipe at the same time allowing a form of data parallelism. A data fetch pipeline such as the Scheduler Function Unit of the HEP allows the processing of multiple data streams.

Furthermore, each type of pipeline can be implemented within the same machine so that each of the multiple instructions streams in the pipe can be executing on its own stream of data. Thereby, pipelines can be used to establish instructional parallelism or data parallelism or both.

CHAPTER V

MULTIPROCESSORS: THE ALLIANT FX/8 AND THE COSMIC CUBE

5.0 Introduction to Multiprocessors

Chapter 3 presents the way parallelism can be introduced into a system by maintaining one control unit and many arithmetic/logic units. The use of such an array system allows multiple data elements to be processed simultaneously, providing data parallelism. Such a computer is a single-instruction-stream multiple-data-stream (SIMD) computer. Chapter 4 introduced pipelining. An instruction pipeline allows multiple instructions to be in various stages of evaluation at the same time; thus, affording instructional parallelism as is done in the Heterogeneous Element Processor (HEP). Further, an arithmetic pipeline can be employed to provide data parallelism as is done in vector processors. In the HEP computer, instruction pipelining in the Instruction Processing Unit (IPU) and data transfer pipelining through the Scheduler Function Unit (SFU), provides both instructional and data parallelism. These considerations show that the HEP is a multiple-instruction-stream multiple-data-stream (MIMD) computer.

Another computer architecture that affords an MIMD system is the multiprocessor. A computer system that is composed of more than one CPU is a multiprocessor. Unfortunately, this simple definition may be applied not only to multiprocessors, but to distributed systems and computer networks as well.

5.0.1 What a Multiprocessor is Not

What are distributed systems and network systems and how they are different from a multiprocessor? A distributed system is a computer system composed of multiple stand alone computers that communicate via telephone lines or a high speed bus. The user of such a system logs onto the system as a whole and is unaware of which computer is giving him service. The system hides the hardware from the user at logon and routes his service request to a particular computer unit based on availability. The interface with which the user interacts runs on each computer unit; thus, the system appears the same to the user regardless of his logon location.

A computer network implies the existence of a collection of interconnected autonomous computers, similar to a distributed system. Each of the computers is capable of supplying service to the user; but, the user specifies at logon the computer to be utilized. Networks with geographically widespread computers are connected via leased communication lines or satellite links while those

with computers located in close proximity may be connected by radio, coaxial cable, fiber optics, etc.

Once a user has been assigned a computer as in a distributed system or has specified one as in a network, the execution of that user's job proceeds on the one computer. Thus, although there are multiple processes active in distributed and networked systems, each job is serviced by one and only one individual computer at a time. Exploitation of parallelism in the individual job is that afforded by the one computer to which the user's job is mapped.

5.0.2 What a Multiprocessor Is

This chapter presents the class of MIMD systems termed multiprocessors. A multiprocessor system is one in which more than one processor, or CPU, is combined to form one computer and each processor contributes to the solution of a single problem or task. In a multiprocessor, the user's job is partitioned into separate subtasks (or subroutines) and these subtasks are mapped onto the set of CPUs. Thus, different portions of the user's code is executed simultaneously on different processors, each processor working on its own data; this is the significant difference between a multiprocessor system and systems termed distributed or networked.

As a simple example of the application and problems of a multiprocessor, consider the task of multiplying two

N x N matrices, A and B. Recall that for each of the N^2 elements in the result, this matrix multiplication implies the multiplication of the elements of a row vector of A by the corresponding elements of a column vector of B and then the summation of these products. That is to say, N^2 inner products must be computed. On a uniprocessor this means the total time required for the multiplication will be that required for N^2 inner product computations. If it were possible to divide the inner product computations evenly between two processors then the time required would be that required for the computation of $N^2/2$ inner products. It should be noted that exactly how this may be done is not necessarily clear; there are many design issues that must be considered. In this example, one of many issues is where should the array values be stored? If they are all in one large global memory, stored in column major order as is common on many uniprocessor systems, then how can the data elements be accessed by the concurrently executing processors? The two processors easily could attempt to access the same element of A or B at the same time; that is, they could clearly contend for memory access, resulting in poor turnaround.

5.1 Issues in the Design of a Multiprocessor

Variations in multiprocessor architecture are many. There are certain fundamental points which one should consider when examining a given multiprocessor architecture. Some of the most significant are the following:

1) How is the memory, or memories, attached to the processors?

2) How do processes executing concurrently on separate processors communicate? How do the processes synchronize their activity?

3) On which processor(s) is the operating system executing?

4) How are computations partitioned to exploit parallelism? How is the job divided into subtasks?

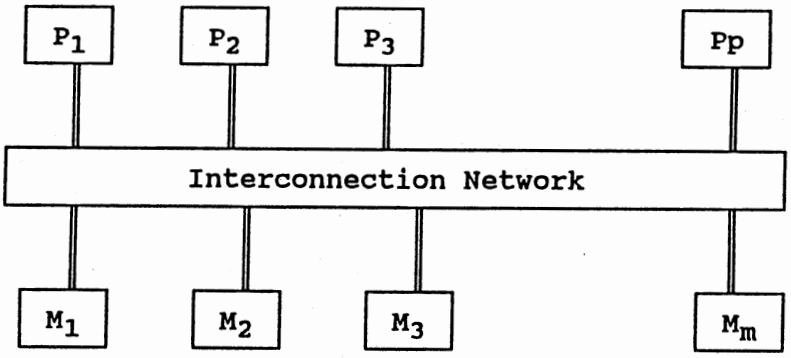
This section gives some general answers to these questions. Subsequent sections present two very different multiprocessors, the Alliant FX/8 and the Cosmic Cube, that demonstrate some contrasting solutions to these questions.

5.1.1 How is the Memory or Memories Attached to the Processors?

Initially, most multiprocessors were designed to share access to main storage [16, p. 108] [55, p. 131]. Most multiprocessor systems now fall into one of two categories of memory-processor organizations: (1) all processors accesss a global memory, and (2) each processor has access only to its own local memory.

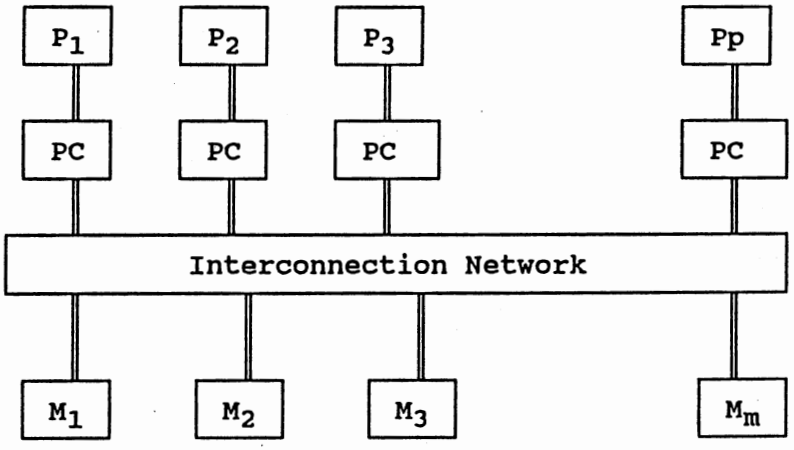
5.1.1.1 A Global Memory. In the first category, a collection of processors, usually eight or fewer [59], are connected to a bank of memory modules via an interconnection network designed from complete crossbar switches [30] [Figure 20]. Such multiprocessors are frequently termed tightly coupled multiprocessors. Under such an arrangement as this, any processor can address any memory unit and, thereby, read from or write to any memory unit [41, p. 460].

A factor which may limit severely the speed of a tightly coupled multiprocessor is that of memory contention. This problem was mentioned earlier in the example of the matrix multiplication. Memory contention occurs when more than one processor attempts to access the same memory unit at the same time. This problem can be reduced by interleaving multiple memory modules, but it cannot be eliminated altogether [41, p. 460].



P: Processor M: Memory Module

Figure 20. Multiprocessor with Global Memory



P: Processor M: Memory Module
PC: Private Cache

Figure 21. Multiprocessor with Global Memory and Private Caches

The use of private caches is another method used to reduce memory contention between tightly coupled processors [Figure 21] [41, p. 470,517]. A cache is a small RAM which has a high speed access time that matches the processor speed. Recently used words and others spatially local to the used words are held in the cache in anticipation of their use in the near future [55, p. 420]. Frequently, the next word required from memory by the processor will be in the cache, thus the processor will not need to access a global memory module. Since memory module access is reduced, so is memory contention [34].

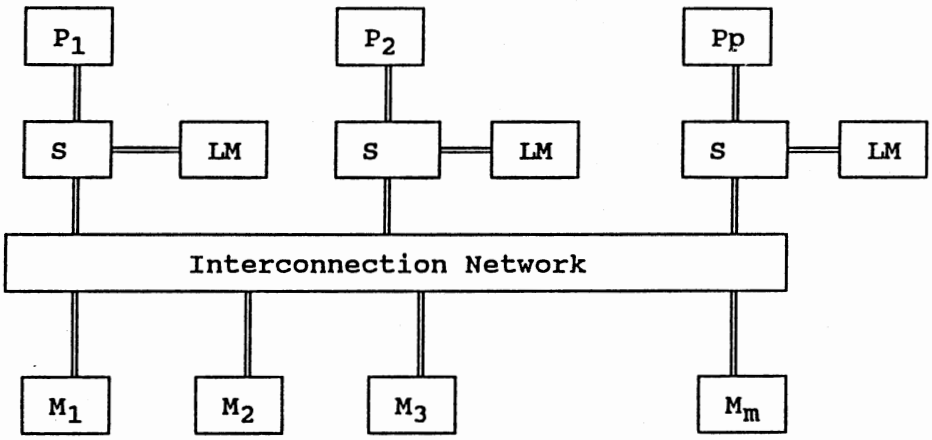
As usual, an apparent solution initiates additional problems. Suppose processor i and processor j , $i \neq j$, each have a copy of location x in their respective caches. If processor i writes to location x in its cache and if processor j reads from location x in its cache, then processor j has an old copy of the data to be processed. Naturally, this leads to erroneous results. This is referred to as the cache coherence problem. Two common ways of solving the problem of cache coherence are now given.

The first and simplest method of solving cache coherence problems is the static coherence check. In this technique, code and data are divided into two categories: (1) read only information such as instructions (cacheable) and (2) read or write information such as updatable data (non-cacheable). Non-cacheable information is restricted

from the processors' caches. Since a large amount of data and code is cacheable this technique does eliminate many shared memory accesses while preventing non-coherence of cached data.

Dynamic coherence checking is the second way to insure positive cache coherence. As the word dynamic implies, this technique is activated during run time. In this scheme, multiple copies of read or write information are allowed in the processors' caches. However, each time a processor writes to a location x, it "cross interrogates" the other processors, via a high speed bus or other communication line, to determine if they also have a copy of location x in their caches. If so, the processors whose caches need updating are signaled to mark their copies as out or not present. Then the updating processor "writes-through" or updates shared memory as well as its own cache. The next time the other processors need location x's data, they will refresh their caches from the shared memory.

There is an additional technique used to lower the frequency of memory contention in some tightly coupled computer systems. It is to provide each processor with a local memory as well as the global memory. The local memory is used to hold operating system and processor status information pertinent to the particular processor [Figure 22]. A switch is employed to map each specified



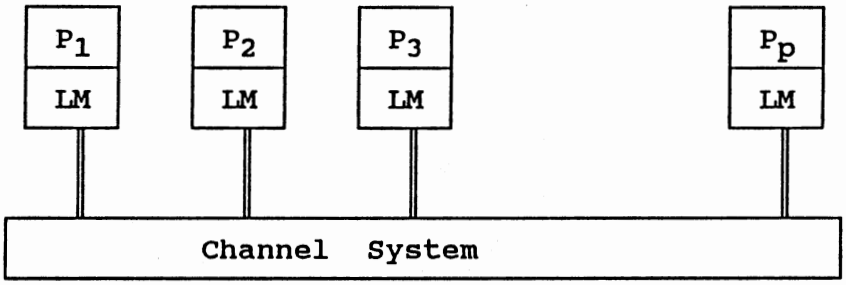
P: Processor
S: Switch
M: Memory Module
LM: Local Memory

Figure 22. Multiprocessor with Global Memory and Local Memories

address onto either the local or global memory. The local memory serves to lower the frequency with which the global memory must be accessed.

Another problem with tightly coupled multiprocessors is that as the number of processors and memory modules hung on the interconnection network increases, so also does the complexity of the network. If there are p processors and m memory modules, then the number of 2 by 2 crossbar switches is on the order of p*m. Thus the complexity, cost, and delay time for data transmission increases rapidly as either the number of processors or the number of memory units increases.

5.1.1.2 Each Processor Has Access Only to Its Own Local Memory. Multiprocessors of the second category contain a collection of processors, normally a large number, from 64 to 65,536 [59], each with its own local memory where it accesses its own instructions and data. Each individual processor with its local memory and I/O devices may be referred to as a computer module. The computer modules are connected by channels that link the modules together according to some designer determined pattern [Figure 23]. A computer system such as this often is termed a loosely coupled computer system. With the local memory approach, memory contention is no longer a problem and there is no costly interconnection network.



P: Processor LM: Local Memory

Figure 23. Multiprocessor with Only Local Memories

There are a number of different channel link patterns used in multiprocessors. Some common configurations are the (1) linear array, (2) tree, (3) star, (4) near-neighbor mesh, (5) ring, and (6) hypercube [Figure 24-29]. The problem to which the multiprocessor will be applied determines the merit of a given configuration. This issue is discussed more in the immediately following section.

5.1.2 How do Processes Executing Concurrently on Separate Processors Communicate? How do the Processes Synchronize their Activities?

The answers to the questions, "How do processes executing concurrently on separate processors communicate? How do the processes synchronize their activities?" depends primarily on how the memories are configured.

5.1.2.1 Tightly Coupled Multiprocessors.

Multiprocessors that have a global memory as in Figure 20 communicate by writing to and reading from common memory locations. That is one reason why the issue of cache coherence mentioned above is so very significant. Processes executing concurrently and sharing common data on a multiprocessor system face many of the same problems dealt with by concurrent processes running on a uniprocessor. Issues such as critical sections, mutual

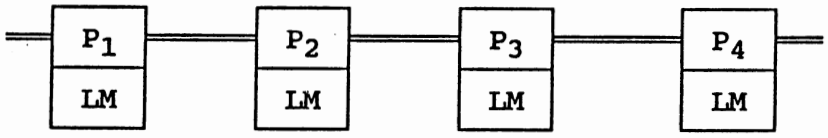


Figure 24. Linear Array of Four Processors

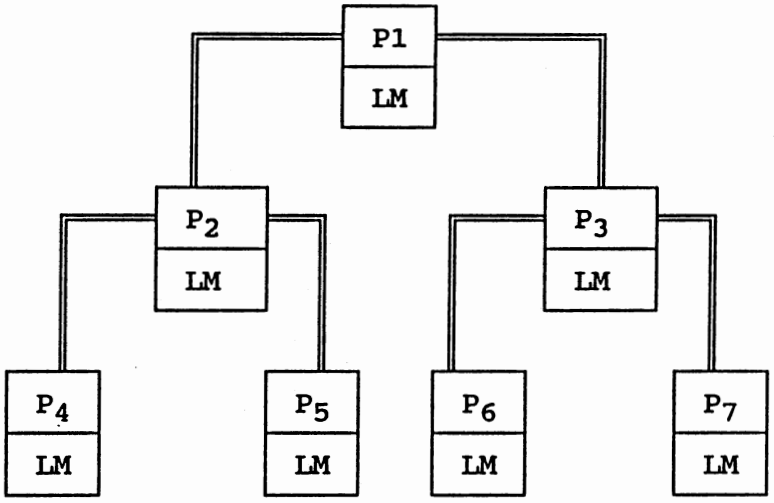


Figure 25. Tree Configuration of Seven Processors

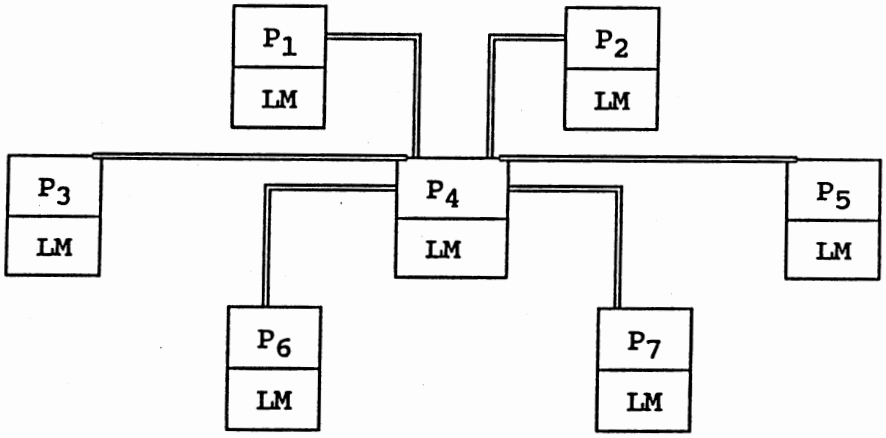


Figure 26. Star Configuration of Seven Processors

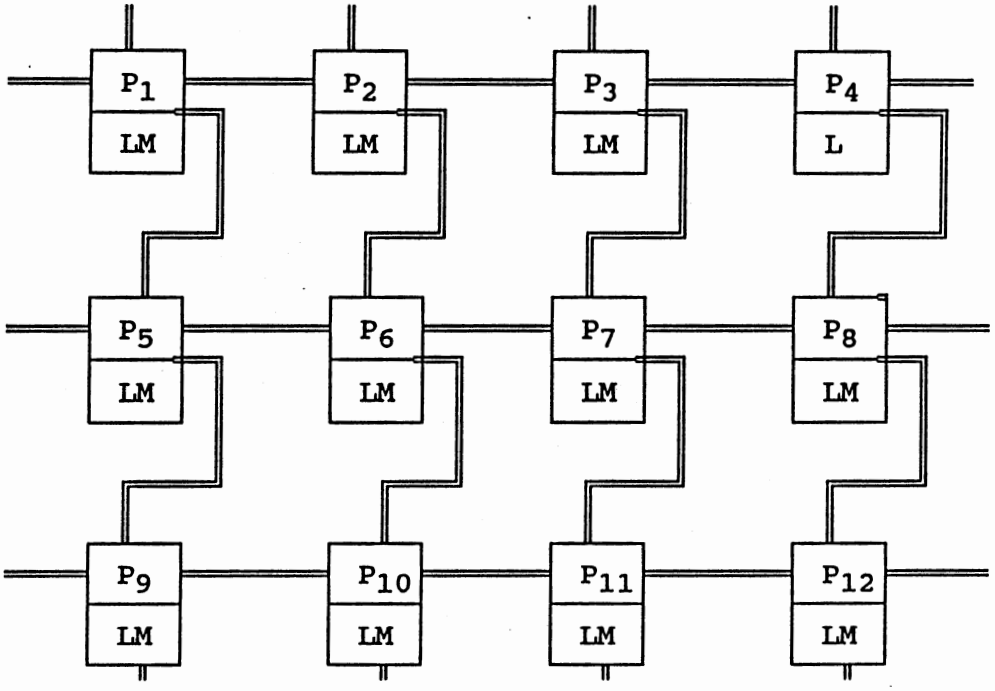


Figure 27. Near-Neighbor Mesh Configuration with Twelve Processors

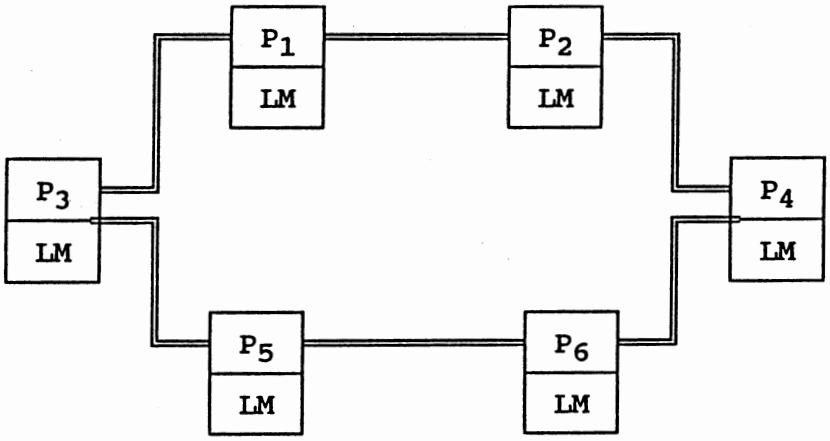
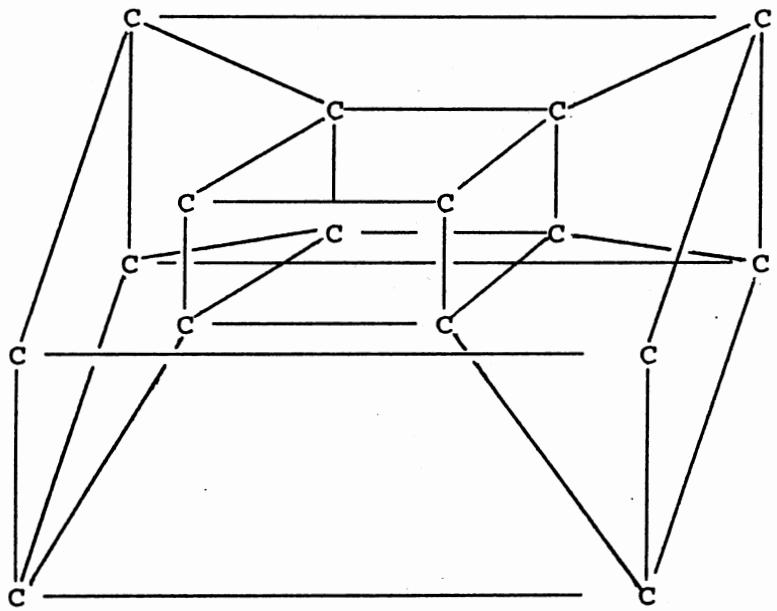


Figure 28. Ring Configuration of Six Processors



C: Computer Module -
Processor and Local Memory

Figure 29. Four-Dimensional Hypercube Configuration

exclusion, deadlock, and use of semaphores are handled similarly to the uniprocessor conventions. For further study in these areas, the reader is referred to Calingaert (1982), Chapter 4. The primary complication which multiprocessors add to these issues is that of simultaneous access to semaphores by processes executing concurrently on distinct processors. On a uniprocessor, access to a semaphore by one and only one process is implemented by disabling interrupts on the lone processor during the time period in which the semaphore is processed. Interrupt disabling by a process prevents any other process from gaining access to the CPU. However, disabling interrupts on one processor does not effect processes running on other processors in a multiprocessor environment. The common solution to this problem is the inclusion of indivisible read-write instructions such as test-and-set into the machine's instruction set. Such instructions are used to force processes on separate processors into executing loops, busy waiting, until processing of the semaphore by the current process is complete. Figures 30 and 31 demonstrate the distinction between P and V operations for a uniprocessor and those for a global memory based multiprocessor system. The test-and-set instruction, TS(S.Mutex) [Figure 31], assigns the variable named Permission the value read from S.Mutex and sets S.Mutex FALSE in one non-interruptable machine instruction cycle. Processes executing the procedures P

```
procedure P(S)
   record S (integer Count, pointer Ptr);
   process P;
   begin
      disable interrupts;
      S.Count := S.Count - 1;
      if S.Count < 0 then
             begin
                insert calling process on list
                    pointed to by S.Ptr;
                P := some ready process;
                dispatch P with
                        interrupts enabled
             end
      else  enable interrupts
   end;


procedure V(S)
   record S (integer Count, pointer Ptr);
   process P;
   begin
      disable interrupts;
      S.Count := S.Count + 1;
      if S.Count ≤ 0 then
             begin
                 P := remove some process from
                   the list pointed to by S.Ptr;
                 WAKE UP P
             end;
      enable interrupts
   end;
```

Figure 30. Uniprocessor Implementation of P and V [16, p. 99]

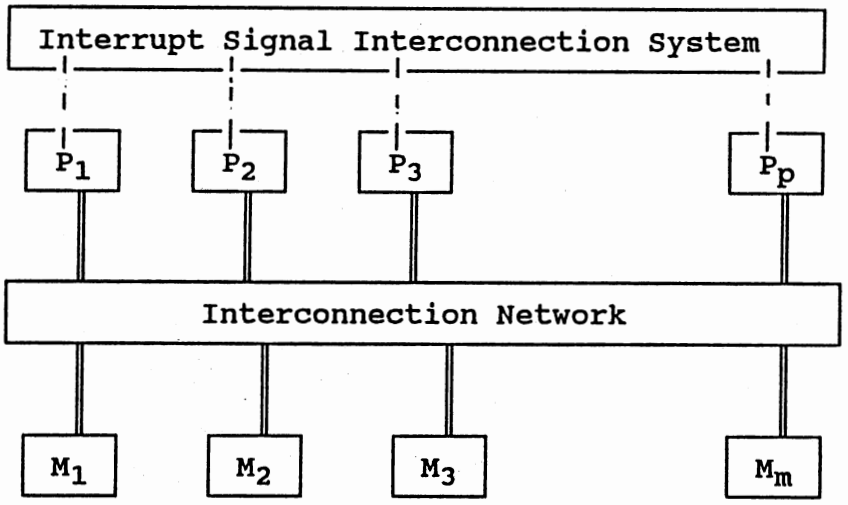
```
procedure P(S)
  record S (integer Count, pointer Ptr, boolean Mutex);
  process P;
  boolean Permission;
  begin
      disable interrupts;
      repeat permission := TS(S.Mutex)
      until permission = TRUE;
      S.Count := S.Count - 1;
      if S.Count < 0 then
             begin
                insert calling process on list
                  pointed to by S.Ptr;
                P := some ready process;
                S.Mutex := TRUE;
                dispatch P with interrupts enabled
             end
      else  begin
               S.Mutex := TRUE;
               enable interrupts
            end
  end;


procedure V(S)
  record S (integer Count, pointer Ptr, boolean Mutex);
  process P;
  boolean Permission;
  begin
      disable interrupts;
      repeat Permission := TS(S.Mutex)
      until permission = TRUE;
      S.Count := S.Count + 1;
      if S.Count ≤ 0 then
             begin
                P := remove some process from
                  the list pointed to by S.Ptr;
                wake up P;
             end;
       S.Mutex := TRUE;
       enable interrupts
   end;
```

Figure 31. Multiprocessor Implementation of P and V [16, p. 210]

or V on other processors execute the repeat/until loop until the process accessing the semaphore member S.Count resets S.Mutex to TRUE and exits the appropriate procedure. Thereby, one and only one process is allowed access to the semaphore member S.Count, during the execution of either a P or a V procedure.

Additional communication between tightly coupled multiprocessors can be established via an interrupt system [Figure 32]. The interrupt system allows interprocessor interrupts; that is, any processor may interrupt any other processor. The interrupt signal interconnection system



P: Processor M: Memory Module

Figure 32. Multiprocessor with Global Memory and Interrupt Signal Interconnection System

may be a simple time shared bus or a complex crossbar switch. The bus is lower in cost but slower to use due to the additional logic needed to make the appropriate processor to processor connection (arbitration logic). Use of the interrupt system may allow one process to signal another of its desire to synchronize. One multiprocessor called the HYDRA uses such an interrupt system to provide mutual exclusion (access by one and only one process) during operations on queues [16, p. 209].

5.1.2.2 Loosely Coupled Multiprocessors.

Multiprocessors that have local memories and are connected by a pattern of channel links communicate via message passing. Data and/or synchronization signals are passed as message packets via the channels to neighbor computer modules. The neighbor module may reroute the message to another module as necessary until the message reaches the appropriate processor. When synchronization is required between processes, the processor to which the message is addressed can be programmed to halt execution at a certain point in its performance until an expected message packet arrives at one of its ports.

A message packet is generally a block of bytes containing such information as the destination processor address and destination process id, source processor address and source process id, count of bytes to be transmitted, data, and control information [Figure 33].

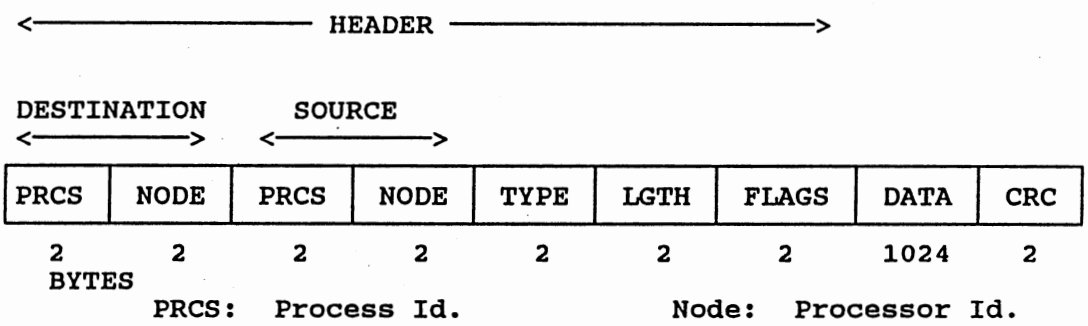


Figure 33. Message Packet Format for Interprocessor Communication [44]

The message packet is compiled by operating system routines. The communicating process calls the operating system routines and passes them the information needed to compile the packet. The operating system may split the message/data into multiple packets depending on the amount of data to be sent. Then, the operating system routines route the message over the appropriate channel from the sending processor onto the channel opened or specified by the process. Rather than passing straight through each computer module node on its way to the destination processor, the packet is stored temporarily in each computer in a buffer or queue area. Then the packet is forwarded to the next node in its journey to the destination by routines of the locally executing operating system. This store-and-forward packet switching allows efficient use of the channels which network the computer modules together.

The use of message packets allows large units of data to be transferred from one processor to another without seizing the channel and blocking out messages that need to be sent over the same path or intersecting path by other concurrent processes. In the tree configuration of Figure 25, if a process on processor P_1 sends a message to a process running on P_7, and a process on P_2 sends a message to another running on P_3, then the channel from P_1 to P_3 is required by two concurrent communications. If one communication is excessively lengthy, it can block the

other out for an extended period, negatively effecting the execution of the blocked process. Similarly, a communication over a long path can block out many short path communications. By using packet switched store-and-forward message passing each packet has its own virtual circuit through the system.

The problem to which a multiprocessor is applied determines the merit of a given processor configuration, ie. linear array, ring, etc. If a given problem may be solved by the operation of a sequence of subroutines, S_i, i=1,2,..,n, where the result of subroutine S_i is passed as input to subroutine S_{i+1}, if this computation needs to be made for a number of different initial input values, and if n computer modules are available, the processors can be configured as a linear array, with S_i executing on processor P_i. The result of each subroutine can be passed as a message packet to the subsequent subroutine on the next processor down; and the initial input values can be pipelined through the multiprocessor linear array, allowing the completion of all computations in about 1/n-th of the time to do the computations sequentially on a uniprocessor. This speedup is only approximate after the pipe is full.

One reason that the speedup anticipated in the above example is not attained is the high overhead inherent in message passing; work done by the operating system routines to implement the message passing can be time

consuming. Systems which utilize message packet communications usually display a high degree of efficiency as long as the amount of message passing required is maintained at a low level; otherwise, the overhead inherent in transferring messages from one module to another may deteriorate performance significantly [41, p. 468].

5.1.3 On Which Processor(s) is the Operating System Executing?

Operating Systems for multiprocessors are very similar conceptually to those that run on uniprocessors utilizing multiprogramming. The reader is referred to Calingaert (1982) [16] and other such texts which discuss the fundamentals of operating system design. The need to support multiple processors executing asynchronous tasks is the factor which increases the complexity of multiprocessor operating systems. Additional intricacy is involved in the support of graceful degradation. One advantage of a multiprocessor system is the potential of keeping the system up and running in the advent of a hardware fault in one of the multiple processors. Graceful degradation implies the capability of reconfiguring the system to omit the faulty unit and continue running. Failure to support graceful degradation mars the positive features of a multiprocessor.

An operating system manages the specific facilities for which it is designed. Of course, that implies that the operating system's internal design depends directly on the organization of the hardware. Each multiprocessor operating system works differently. The processor-memory organization and the method of interprocess communication provided by the hardware clearly influences operating system design. This section outlines three basic operating system configurations which have been used in existing multiprocessors: (1) master-slave processor configuration, (2) each processor with its own separate supervisor, and (3) floating supervisor which may be in any processor at a given time.

5.1.3.1 Master-slave Processor Configuration. In a master-slave processor configured operating system, one processor is designated as the master processor. The operating system is executed by this one processor alone. It maintains the status of each processor in the system and allocates tasks to the other processors, or slaves, according to some rule. The slaves are treated as schedulable resources. This implies that the master should be able to assign tasks to the slaves as fast as they can do them. Should the master not be able to match the speed of slave processor service then the slaves must wait; clearly, this condition implies poor use of facilities. If the master fails then the multiprocessor

fails; under such circumstance, it is impossible to degrade gracefully. Since the operating system always executes on the one processor, a slave processor communicates with the master through an interrupt signal interconnection system. The slave either generates a trap or executes a supervisor call instruction. The master processor operating system's appropriate interrupt handler acknowledges the request and performs the required service. The advantage of a master-slave arrangement is that it is relatively easy to implement as an extension of a multiprogramming uniprocessor operating system. Since only one processor is executing the operating system code, and in behalf of only one user at a time, the code need not be reentrant [41, p. 527]. The code need not be reentrant in the sense that no separate data areas need be established for separate instances of execution. Naturally, the machine instructions should not modify themselves. Master-slave operating systems work well in environments with special applications such that the tasks are clearly specified. Also, it works well on multiprocessors that have only two or three processors, as the slaves are not so likely to contend for service from the master.

5.1.3.2 Each Processor with Its Own Separate Supervisor. Each processor may have its own copy of the supervisor system to execute, then each processor provides

for its own management requirements. Multiprocessors with local memories as in Figures 22 and 23 utilize this operating system configuration, although it may also be used in a totally global memory system such as Figure 20. Although the processors take care of their own needs, they must interact with each other. In a message based system, this implies that each operating system possesses the routines needed to implement the store-forward data packet message passing discussed earlier.

In a multiprocessor with a global memory, supervisor code is replicated for each processor. In order for the processors to interact, it is necessary for some of the data structures such as job tables and the state of shared resources such as file structures to be held in the global memory and shared by the whole system. Shared tables create access problems. The prevention of simultaneous access of the tables by multiple processors may be implemented using test-and-set instructions and P and V procedures as discussed in the previous section on synchronization. Any shared code must be reentrant.

The separate supervisor for each processor configuration provides more graceful degradation than the master-slave system; since each processor is providing its own primary needs, then when one processor fails the others can continue.

5.1.3.3 Floating Supervisor which may be in Any Processor at a Given Time. Under a floating-supervisor operating system, any one processor may be executing the supervisor at a given time. Further, several of the processors may be executing supervisor service routines at the same time. All the processors and other resources are treated equally. Code and tables are maintained in a global memory and any processor may access the code or tables for use. Thus, most code must be reentrant and table access conflicts cannot be prevented, but can be handled as mentioned earlier with test-and-set instructions and P and V procedures. This operating system mode is considered to be the most difficult plan of operation and the most adaptable. If a processor fails the other processors simply pick up its load and continue. This provides graceful degradation.

These three operating system configurations are generalizations of the systems found in practice. Actual systems fit somewhere in the continuum between the simple master-slave approach and the sophisticated floating-supervisor mode.

5.1.4 How are Computations Partitioned to Exploit Parallelism?

A program written for a multiprocessor must exploit the parallelism of the algorithm in order for there to be any speedup in the program execution over that found in a

uniprocessor. How the parallelism is detected and the computation partitioned so that different processors may work on separate portions of the job is the question addressed here.

Parallelism may exist at different levels. The chapters on array processors and pipelining demonstrate how parallelism can be exploited at the data and instruction levels. The chapters on data flow and reduction present additional methods of exploiting parallelism at the instruction level. In the MIMD environment of a multiprocessor, it is the parallelism that may exist between blocks of code that is of concern. The idea is to identify those blocks of code that are self-contained units of the computation to be performed and can be executed during the same time period on different processors. This does not imply that they must begin and end execution together, but, rather that the operation of one may begin before the termination of the other. In this section, focus is on both this issue and on how this information is conveyed to the multiprocessor.

5.1.4.1 Data Dependency. The primary issue which delimits parallelism between blocks of code is data dependency. Consider the statements of Figure 34.

```
s1      X = A + B
s2      Y = M + N
s3      Z = X + Y
```

Figure 34. Statements for Parallel Evaluation

The statement s3 is dependent on the results of statements s1 and s2; there exist data dependencies between s3 and both statements s1 and s2. On the other hand, the computation of Y is independent of the computation of X, and vice versa. The computation of Z is independent of the order in which X and Y are computed. Therefore, statements s1 and s2 may be interchanged, or commuted, and still produce the same result in Z.

In general, when two blocks of code demonstrate this condition of commutativity, as s1 and s2 do here, then there are no data dependencies between them and they can be executed in parallel. More precisely, the Bernstein condition must be satisfied before sequentially organized processes can be executed in parallel [63, p.310].

If the following definitions are made, Bernstein's condition may be identified. B_1 and B_2 define blocks of code. R_i defines the set of all memory locations such that the first access of the location by B_i is a read

operation. W_i defines the set of all memory locations to which B_i performs a write operation.

Bernstein's condition [10] may be stated as follows. B_1 and B_2 may be executed in parallel if they fulfill the following requirements:

1) $R_1 \cap W_2 = \emptyset$

2) $R_2 \cap W_1 = \emptyset$

3) $W_1 \cap W_2 = \emptyset$

Blocks of code that meet the Bernstein condition have no data dependencies and are appropriate candidates for parallel execution on distinct processors.

In the example of Figure 34, if $s_i = B_i$, then

$R_1 = \{A, B\}$, $W_1 = \{X\}$,

$R_2 = \{M, N\}$, $W_2 = \{Y\}$.

and $\{A,B\} \cap \{Y\} = \{M,N\} \cap \{X\} = \{X\} \cap \{Y\} = \emptyset$

showing that s_1 and s_2 may be executed in parallel. Further,

$R_3 = \{X, Y\}$, $W_3 = \{Z\}$, and

$R_3 \cap W_1 = \{X, Y\} \cap \{X\} \neq \emptyset$, also

$R_3 \cap W_2 = \{X, Y\} \cap \{Y\} \neq \emptyset$.

The non-empty intersections indicate s_3 may not be executed in parallel with either s_1 or s_2.

Although a pair of blocks may demonstrate commutativity, they are not necessarily appropriate for parallel execution. Consider the process called Fast Fourier transform, or FFT. This process produces its output in bit reversed order. As a result, a complete FFT

computation implies that either (1) the input of the FFT first be bit reversed or (2) the output from the FFT be bit reversed. The processes of FFT and bit reversal exhibit commutativity. But, either (1) the output of the bit reversal, W_1, serves as the input for the FFT, R_2, and $W_1 \cap R_2 \neq \emptyset$, or (2) the output of the FFT, W_2, serves as the input for the bit reversal, R_1, and $W_2 \cap R_1 \neq \emptyset$. Clearly, the procedure blocks fail Bernstein's condition and they are not suitable for parallel execution. Commutativity is a necessary but not sufficient condition to insure valid parallel execution [41, p. 542].

It has been determined that the Bernstein condition is necessary and sufficient for blocks of code to be executed in parallel. How are the results of a data dependencies analysis conveyed to the multiprocessor; or, how does the multiprocessor "know" which blocks may be executed on different processors? There are two approaches; one strategy is to employ implicit concurrency, the other is to apply explicit concurrency.

5.1.4.2 Implicit Concurrency. Implicit concurrency indicates that the compiler performs a data dependency analysis of the source program. Based on the above described conditions, appropriately designed compilers can determine potential parallelism in high-level language programs automatically. Most existing parallelizing compilers examine loops for consideration as parallel

blocks, where each iteration of the loop, if executed sequentially, is considered as a single process. A subsequent section of this chapter examines the Alliant FX/8 multiprocessor which is bundled with FX/FORTRAN, a parallelizing compiler. That section surveys this subject further.

5.1.4.3 Explicit Concurrency. Explicit concurrency indicates that the programmer considers his code and divides it into logical units, considers the data dependencies between the units, and then specifies the blocks or program units which may be executed in parallel using certain language constructs. Some of those constructs are surveyed in the following paragraphs.

FORK and JOIN are two statements that allow explicit specification of parallelism or concurrency. These two statements are not totally standardized; thus, they may be defined differently in different settings. This discussion attempts to convey the basic concepts of FORK and JOIN [55, p. 182] [41, p. 533-534] [63, p. 310-314]. The FORK and JOIN statements function as system primitives; they are indivisible, or uninterruptable, procedures.

FORK is used to spawn a new process from code beginning at a specified address; FORK also continues the current process in which it is expressed. Execution of

the FORK function expressed in the format

FORK S1

initiates execution of code beginning at statement S1 as well as allowing execution of the code following the FORK to continue. Execution of the FORK function expressed in the format

FORK S1,J,N

initializes a counter J to the value N; then initiates execution of code beginning at statement S1 and continues execution of the code following the FORK, as in FORK S1 above.

JOIN is used to end all but one of a set of concurrent processes. JOIN has the format

JOIN J

Execution of this statement results in decrementing counter J. If the value of counter J is not zero after decrementing then the process executing the JOIN terminates. A process that performs the JOIN and sets the counter to zero continues to execute.

The value N from the FORK S1,J,N statement specifies the number of concurrent processes that are to be funneled together and joined into one process. The counter J is decremented from N down to zero by JOIN J as each concurrent process executes the JOIN. This implies that the longest executing process of the process set will not be terminated by the JOIN.

In the code of Figure 34, statements s1 and s2 may be executed in parallel. This parallel execution may be implemented by the code in Figure 35. The FORK S2,J,2 sets J to two and begins execution of the code at S2; S1 also begins execution concurrently with S2. If the process of S2 reaches the JOIN J first, then the J will be decremented to one which is not zero, so that process will terminate; then the process from S1 will GO TO the JOIN, decrement the counter J from one to zero and continue by executing statement S3.

```
      FORK S2,J,2
S1    X = A + B
      GO TO S4
S2    Y = M + N
S4    JOIN J
S3    Z = X + Y
```

Figure 35. Parallel Implementation of Figure 34 Using Fork and Join

A more complicated example is given in the code of Figure 36.

```
        DO S2 I = 0,N-1
S1          A(I+1) = 2*I + 1
S2          READ B(I+1)
        DO S7 I = 1,N
S3          C(I) = B(I)**2
S4          WRITE C(I)
S5          D(I) = C(I)**2 +A(I)
S6          WRITE D(I)
S7          WRITE A(I)
```

Figure 36. Sample Code

Since the programmer is to express his concurrency explicitly and wishes to maximize concurrency, it is appropriate for him to analyze the code in an effort to break it down into self-contained blocks with minimal data dependencies between blocks. After considering what activities may be done independently of the others, he could rewrite the code as shown in Figure 37.

```
      DO S1 I = 0,N-1
S1       A(I+1) = 2*I + 1
      DO S2 I = 1,N
S2       READ B(I)
      DO S3 I = 1,N
S3       C(I) = B(I)**2
      DO S4 I = 1,N
S4       WRITE C(I)
      DO S5 I = 1,N
S5       D(I) = C(I)**2 + A(I)
      DO S6 I = 1,N
S6       WRITE D(I)
      DO S7 I = 1,N
S7       WRITE A(I)
```

Figure 37. Code of Minimally Dependent Blocks. Assume Distinct I's in Each Loop

A data dependence analysis of the blocked code would show the data relationships presented in the diagram of Figure 38. Computation of the vector D depends on the previous computations of vectors A and C; computation of C depends on reading B; writing of vectors A, C, or D may not proceed until the vector elements are computed. Vector A can be computed while B is read and then C computed. Vectors A, C, and D can be written at the same time, assuming adequate output device resources are available. Vector A could also be written while B is read and C computed since those actions are independent of A.

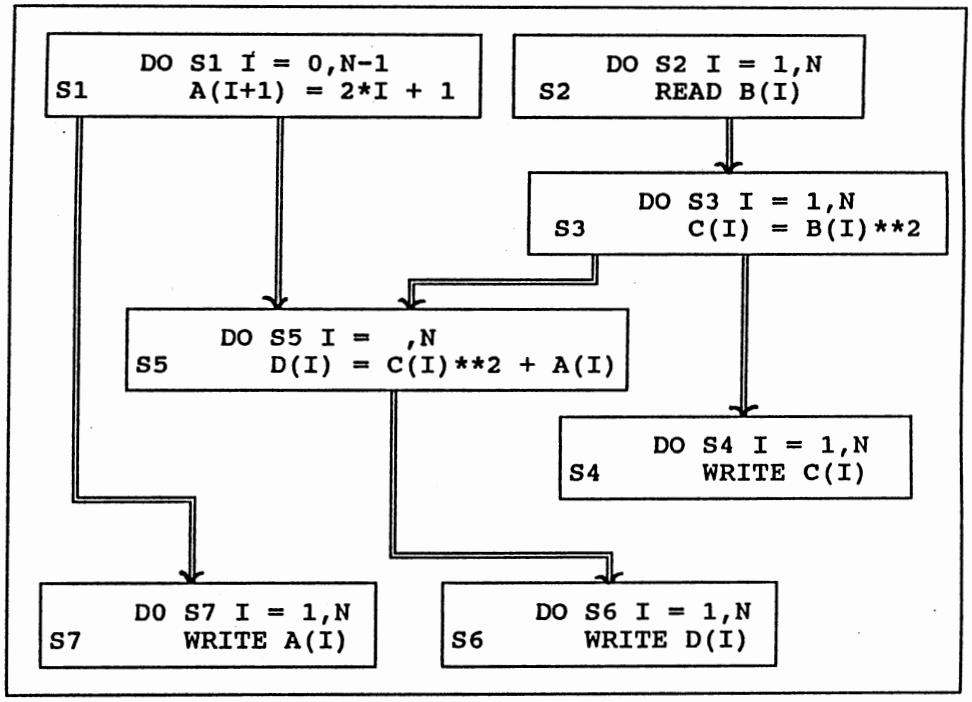
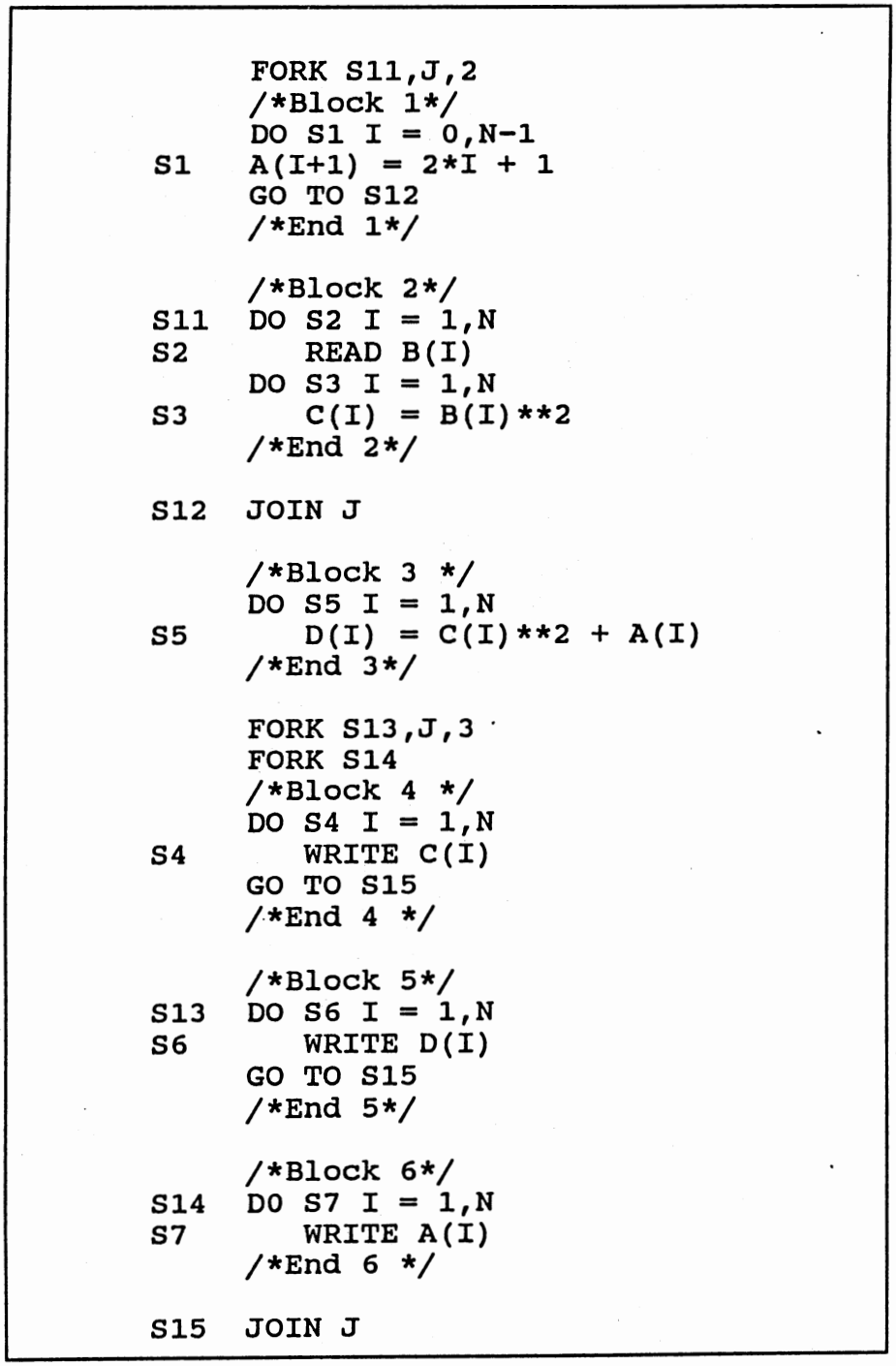


Figure 38. Data Dependence of Code in Figure 37

Vector A could be written while D is computed, since the operations do not contradict Bernstein's condition; but, they should not be done concurrently since both processes would access vector A and contend for memory access.

Thus, the code could be written with FORKs and JOINs as indicated in Figure 39. Figure 39 can be represented pictorially as in Figure 40. Thus, the FORK S11,J,2 in the initial process indicates J is set to two, the execution of the current process into Block 1 is continued, and a new process in the execution of Block 2, beginning at S11, is initiated. Blocks 1 and 2 may then be executed concurrently on separate processors. The JOIN J at statement S12 indicates synchronization, in that the process to reach statement S12 first decrements J to one and ends; the second process to reach S12 decrements J to zero and continues to execute Block 3. FORK S13,J,3 sets J to three and initiates execution beginning at statement S13, as well as continuing in line execution. FORK S14 initiates execution of code beginning at line S14 as well as continuing in line execution. Thus, Blocks 4, 5, and 6 will execute concurrently. The first to complete will execute JOIN J, decrement J to two, and quit; the second will execute JOIN J, decrement J to one, and quit; and, the third to complete will execute JOIN J, decrement J to zero, and continue execution.



```
      FORK S11,J,2
      /*Block 1*/
      DO S1 I = 0,N-1
S1    A(I+1) = 2*I + 1
      GO TO S12
      /*End 1*/

      /*Block 2*/
S11   DO S2 I = 1,N
S2       READ B(I)
      DO S3 I = 1,N
S3       C(I) = B(I)**2
      /*End 2*/

S12   JOIN J

      /*Block 3 */
      DO S5 I = 1,N
S5       D(I) = C(I)**2 + A(I)
      /*End 3*/

      FORK S13,J,3
      FORK S14
      /*Block 4 */
      DO S4 I = 1,N
S4       WRITE C(I)
      GO TO S15
      /*End 4 */

      /*Block 5*/
S13   DO S6 I = 1,N
S6       WRITE D(I)
      GO TO S15
      /*End 5*/

      /*Block 6*/
S14   DO S7 I = 1,N
S7       WRITE A(I)
      /*End 6 */

S15   JOIN J
```

Figure 39. Parallel Implementation of Figure 36 Using Fork and Join

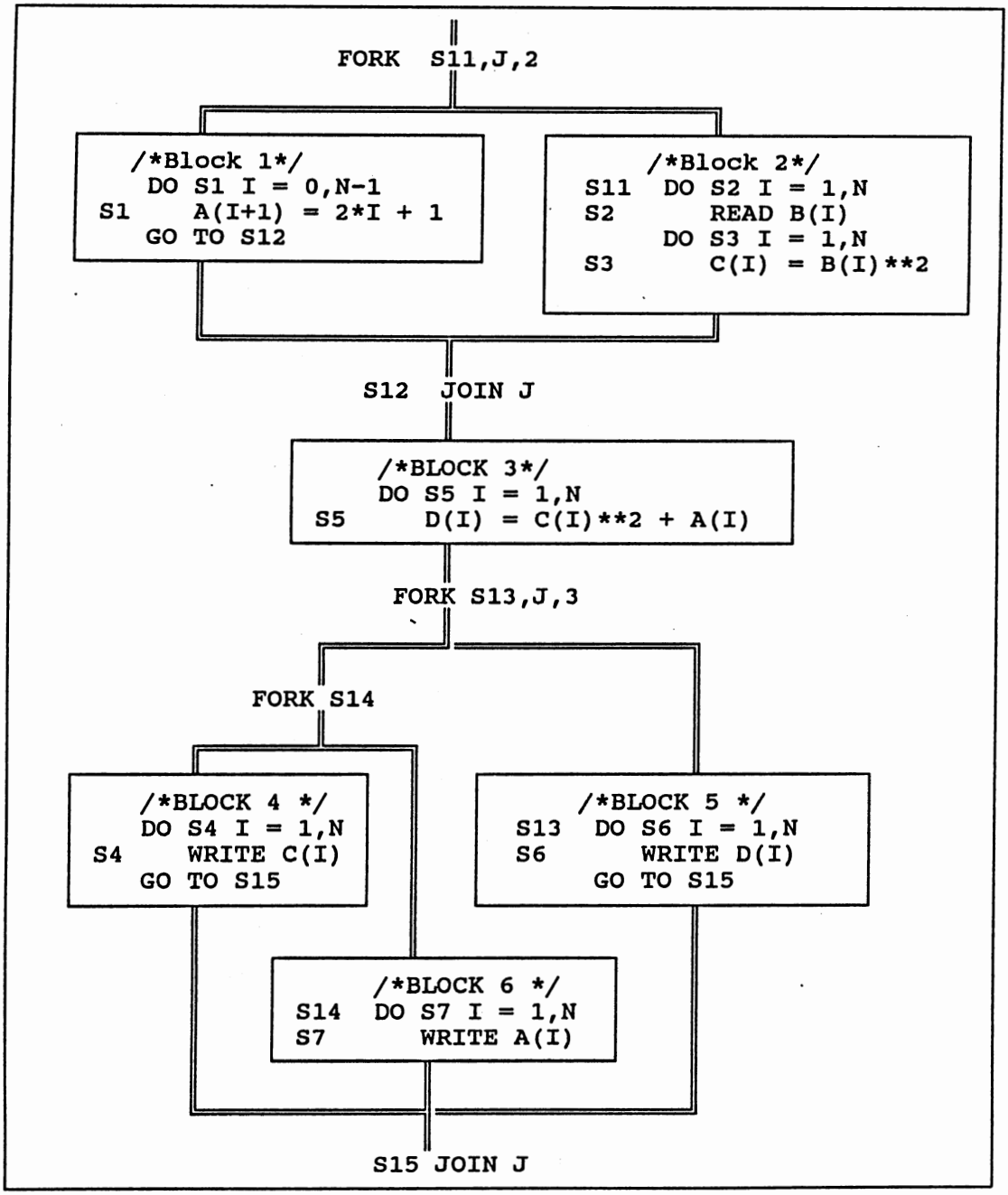


Figure 40. Flow of Control Graph of Code of Figure 39

FORK is analogous to the use of the GO TO construct in its effect on the point of execution. Thus, FORK/JOIN have fallen into disfavor as structured programming has gained approval. Further, use of FORK/JOIN blurs the distinction between statements executed sequentially and those that may be executed concurrently. In the example of Figure 39, it is not immediately obvious that vector D is computed alone while vectors A, C, and D are written concurrently.

Some structured constructs in use are PARBEGIN/PAREND (or, COBEGIN/COEND) and PARFOR. PARBEGIN and PAREND delimit disjoint blocks of code; all of the code blocks set aside by the PARBEGIN/PAREND construct may be executed concurrently. The disjointness of the blocks implies that a variable X written by one block may not be read by another block, although all concurrent blocks may reference the same variable. Using the PARBEGIN/PAREND construct, the example program using FORK and JOIN may be rewritten as Figure 41.

The PARFOR construct is analogous in construction to the Pascal FOR statement. Its basic construct is

```
PARFOR I = 1 UNTIL N DO
     BEGIN
     ....statements...
     END
```

```
BEGIN
   PARBEGIN
      BEGIN /*Block 1*/
            DO S1 I = 0,N-1
         S1    A(I+1) = 2*I + 1
      END
      BEGIN /* Block 2*/
            DO S2 I = 1,N
         S2    READ B(I)
            DO S3 I = 1,N
         S3    C(I) = B(I)**2
      END
   PAREND

     /*BLOCK 3*/
       DO S5 I = 1,N
    S5    D(I) = C(I)**2 + A(I)

   PARBEGIN
      BEGIN /* Block 4*/
            DO S4 I = 1,N
         S4    WRITE C(I)
      END
      BEGIN /* Block 5*/
            DO S6 I = 1,N
         S6    WRITE D(I)
      END
      BEGIN /*BLOCK 6*/
            DO S7 I = 1,N
         S7    WRITE A(I)
      END
   PAREND
END
```

Figure 41. Parbegin and Parend Construction Equivalent to the Fork and Join Code of Figure 39

The PARFOR construct implies that N concurrent processes will be initiated, one for each value of variable I. Each process is generated from one iteration of the loop body.

An example is the multiplication of an NxN matrix, A, by an Nx1 vector, B, giving an Nx1 vector, C [41, pp. 538]. This multiplication requires the computation of N inner products. The computation may be divided between multiple processors by spawning multiple processes for scheduling on the processors. If there are P processors, such that P divides N evenly, and S = N/P then the code of Figure 42 will generate P processes. Each process may be scheduled on a different processor for parallel execution.

In Figure 42, if there be two processors, P = 2, and if N = 4, then S = 4/2 = 2. There will be two parallel processes generated. The first process, for I = 1, is the execution of the code indicated in Figure 43. This first process computes C(1) and C(2). The second process, for I = 2, is the execution of the code indicated in Figure 44. This second process computes C(3) and C(4).

These examples demonstrate how the FORK/JOIN, PARBEGIN/PAREND and PARFOR constructs may be used to allow a programmer to exploit explicitly the potential parallelism between disjoint code blocks. Of course, not all processes a programmer wishes to execute in parallel are disjoint. Many processes need to share common data

```
PARFOR I = 1 UNTIL P DO
     BEGIN
          FOR J = (I - 1) * S + 1 UNTIL S * I DO
               BEGIN
                    C(J) = 0;
                    FOR K = 1 UNTIL N DO
                         C(J) = C(J) + A(J,K) * B(K);
               END
     END
```

Figure 42. Parfor for Generating P Parallel Processes to Calculate N Inner Products where S = N/P

```
BEGIN
     FOR J = 1 UNTIL 2 DO /* (I-1)*S+1 = 1 */
                          /* S*I=2 */
          BEGIN
               C(J) = 0;
               FOR K = 1 UNTIL N DO
                    C(J) = C(J) + A(J,K) * B(K);
          END
END
```

Figure 43. Process Code for Calculation of C(1) and C(2) from Figure 42

```
BEGIN
     FOR J = 3 UNTIL 4 DO  /* (I-1)*S+1=3 */
                           /* S*I=4 */
          BEGIN
               C(J) = 0;
               FOR K = 1 UNTIL N DO
                    C(J) = C(J) + A(J,K) * B(K);
          END
END
```

Figure 44. Process Code for Calculation of C(3) and C(4) from Figure 42

bases. A common example of such sharing are producer/consumer process pairs. Here one process writes what another process reads. Clearly, the consumer should not read a location prior to the writing of the location by the producer and the producer should not write over data that has not yet been read by the consumer. Different languages and architectures handle this problem differently. In HEP's data memory, a solution to this problem is to flag memory as 'full' or 'empty'.

Code that accesses a variable or data base or other resource that is common to two or more concurrent processes is called a critical section. A critical section must be executed only by one process at a time. If the concurrent processes are running on separate processors then a critical section must be executed by only one processor at a time. This requires that one processor must 'lock out' all other processors from access to the shared resource while it is accessing the common resource. Such an operation is called synchronization. Means of synchronization are discussed in section 5.1.2. Synchronization between two parallel processes accessing a common resource can be implemented using the P and V operations of Figure 31 as shown in Figure 45. In Figure 45, P1 and P2 could execute in parallel except when they attempt to access the common resource simultaneously. For further reading on this topic the reader is referred to Calingaert (1982) [16], Chapters 4 and 8, and to Hwang and

Briggs (1984) [41], pages 539-541 and Chapter 8 of Hwang and Briggs.

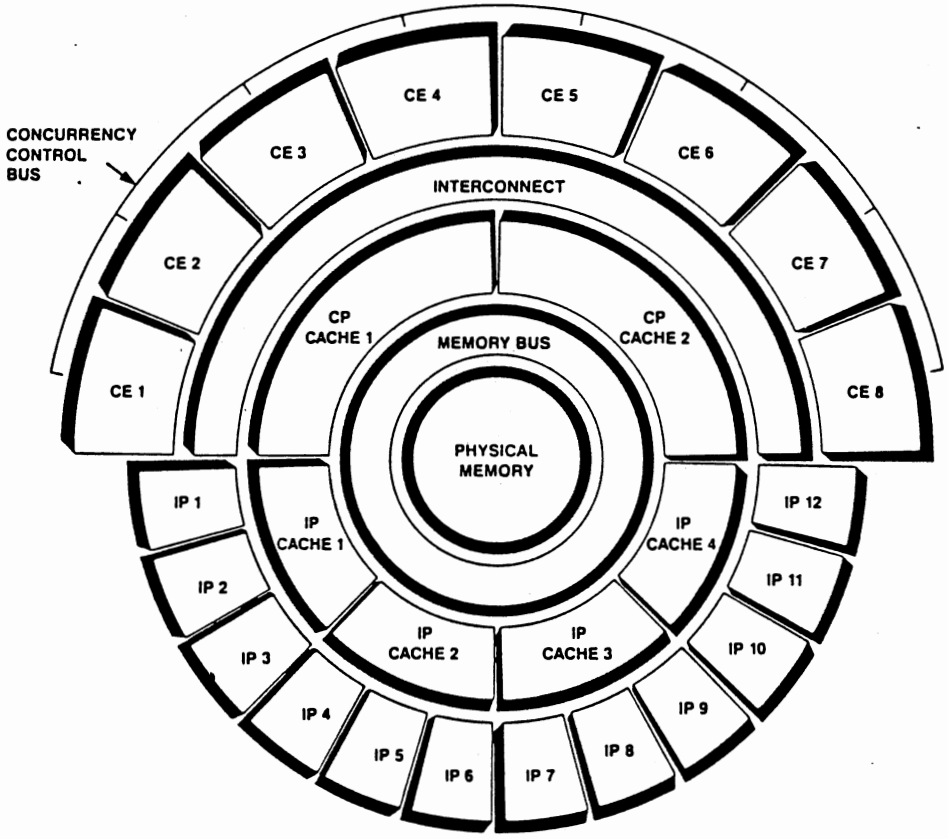
```
RECORD S(INTEGER COUNT; POINTER PTR; BOOLEAN MUTEX)
S.MUTEX = TRUE        /*NO PROCESS EXECUTING P OR V*/
                       /* IS ACCESSING S.COUNT */
S.COUNT = 1           /*NO PROCESS HAS EXECUTED P OR V*/
S.PTR = NULL          /*LIST OF PROCESSES BLOCKED ON */
                       /* S IS EMPTY INITIALLY*/
PARBEGIN
   P1: BEGIN
          P(S)
          CRITICAL SECTION./*ACCESS COMMON RESOURCE*/
          V(S)
       END
   P2: BEGIN
          P(S)
          CRITICAL SECTION./*ACCESS COMMON RESOURCE*/
          V(S)
       END
PAREND
```

Figure 45. Use of Multiprocessor P and V Operations to Synchronize Execution of a Critical Section by Parallel Processes P1 and P2

5.2 The Alliant FX/8 Multiprocessor

The Alliant is the first of two multiprocessors surveyed in this chapter. The Alliant FX/Series is a multiprocessor architecture which combines up to eight processors in a parallel design. The FX/Series architecture has gained the interest of many due to its high performance/cost ratio. In performance comparisons of different computers running LINPACK [28] software in a FORTRAN environment the Alliant has shown itself to be a consistently high performer. Using the Cray-1S computer as a standard of 1, the Alliant FX/8 has produced equivalent results in 1.6 times as many time units as that of the Cray-1S. This may be compared with the DEC VAX 8600 which produced its results in 32 times as many time units as that of the Cray-1S [28].

The full Alliant FX/8 configuration may be pictured as shown in Figure 46. It is composed of eight FX/1's. Each FX/1 is composed of one Computational Element (CE) and one or two Interactive Processors (IP), eight megabytes of physical memory, one cache for the Interactive Processors, and one cache for the Computational Element. The user may upgrade his Alliant System by purchasing additional FX/1's as needed, up to a maximum of eight. Thus the full Alliant FX/8 architecture as pictured [Figure 46] has eight CEs, twelve IPs, sixty



CE: Computational Element
IP: Interactive Processor

Figure 46. The Alliant FX/8 Multiprocessor System [3]

four megabytes of physical memory (it may be increased to eighty megabytes), two computational processor caches for the CEs and four caches for the IPs. The CEs are connected to the computational caches via a crossbar interconnection network. Each CE is capable of 11.8 million floating point operations per second (MFLOPS), allowing more than 94 MFLOPS on the FX/8 system when operating at peak performance. The CEs as a group are referred to as the computational complex.

The Alliant FX architecture uses the IPs to run interactive user jobs; all I/O is done through the IPs. Concentrix, the Berkeley 4.2 Unix operating system runs in parallel on the IPs. The computational elements are scheduled by the operating system as a single resource. When scheduled and utilized, the CEs reduce time-to-solution for a single application.

The Alliant FX/8 implements parallelism at several levels including:

1) Instruction pipelining in the CEs and IPs.

2) Vector processing. Each CE contains a floating point pipeline for the implementation of floating point array calculations. Integer and logical operations are also allowed.

3) Concurrent processing of distinct jobs. The IPs are used to service individual user jobs as in a distributed system.

4) Concurrent execution of distinct instruction streams for the same job. The computational complex (the CEs) is scheduled by the operating system as a single resource to be applied to the parallel execution of portions of a single user's program.

It is the parallelism provided by 4) above which makes the Alliant FX/8 a true multiprocessor as defined in this chapter. This section examines the computational element and Alliant cache and memory system to determine how it implements some of these parallel techniques. The Alliant FX/FORTRAN parallelizing compiler is also examined and its concurrency applications are surveyed.

5.2.1 The Computational Element

The computational elements are the heart of the Alliant multiprocessor system. Each CE is a microprogrammed computer with pipelined data and control paths. The basic CE instruction set modes include the following:

1) concurrency instructions such as test-and-set and wait-and-start (stops and starts the CEs).

2) vector processing instructions which allow logical, integer, and floating point operations on vector registers which may hold up to 32 elements each. The vector operations include register-to-memory operations, comparisons and logical operations on operands in vector registers, reduction functions such as summing the

elements of a vector, and vector-vector, scalar-vector arithmetic operations.

3) IEEE floating point instructions.

4) scalar instructions.

The CE supported data types are 32 and 64 bit floating point; 8, 16,and 32 bit integer; BCD; and bit.

The CE has four main functional systems [Figure 47]. The first is a pipelined instruction unit. The stages of the instruction pipeline are as follows:

1) The current or logical instruction cache holds the current instruction stream. When an instruction fetch 'misses' this immediate cache, the cache controller initiates a load from the computational cache through the interconnection network and CE switch. Each instruction in the stream is piped in sequence from the logical instruction cache to the control section.

2) The control section consists of an instruction parser, microsequencer, and RAM-based control store. The instruction parser receives the opcodes from the data path and decodes them to generate control store microaddresses. The parser also stores the instruction fields of the opcodes that are in the pipe and checks for dependencies between instructions that are in various stages of execution within the pipe. Having thus checked for hazards, it prevents a new instruction from starting when a hazard exists. The parser also contains a branch prediction unit that anticipates the most likely flow of

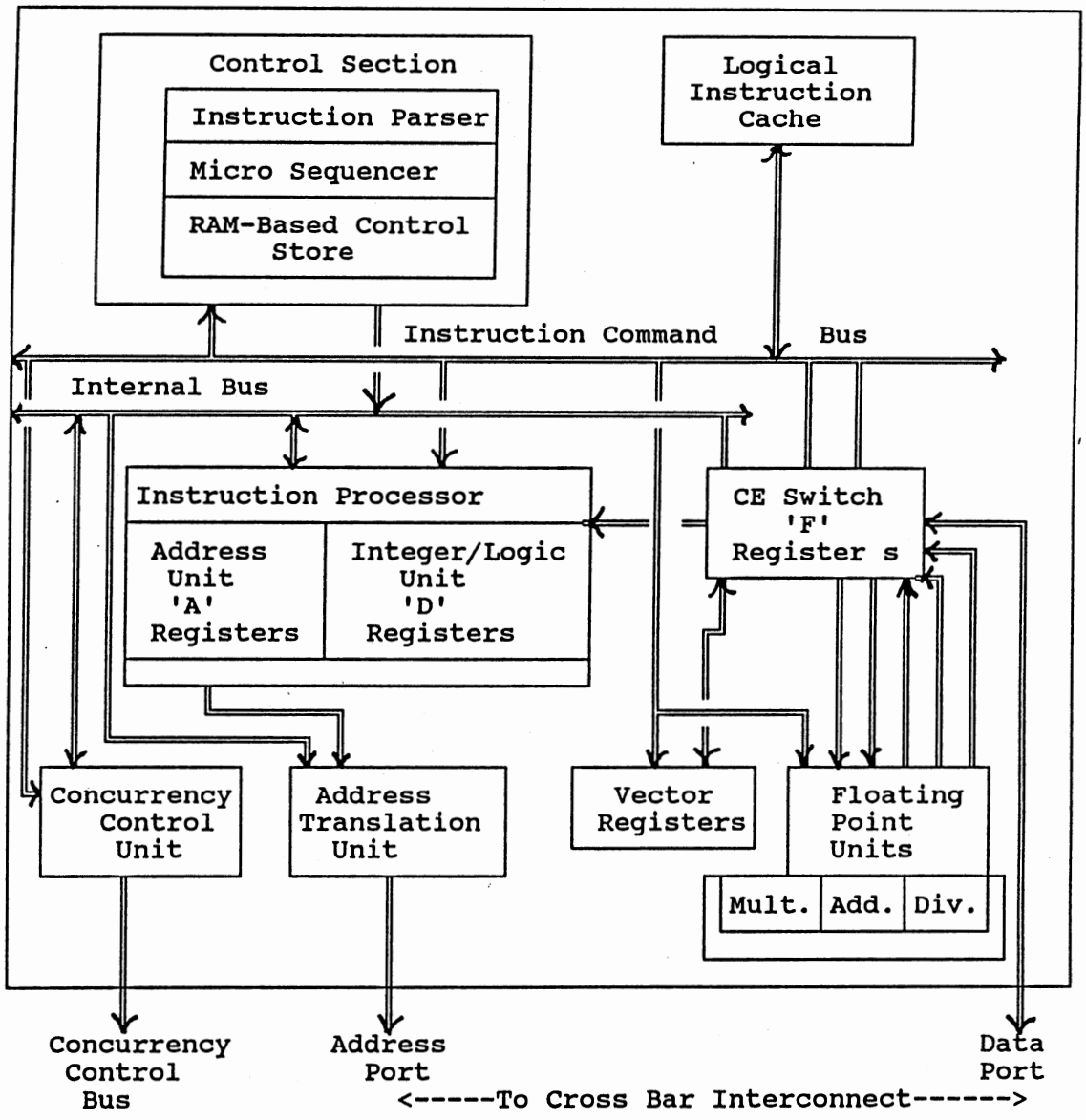


Figure 47. Alliant Computational Element Block Diagram

control and prefetches instructions from the predicted side of the branch. As is observed in the study of pipelines in Chapter 4, this is a common technique used to help maintain a full pipeline and the highest possible performance. The microsequencer and control store are responsible for issuing the control words to drive the system. The next stage of the pipe is the instruction processor.

3) The instruction processor consists of the address unit and the integer/logic unit.

The address unit contains the instruction buffer, or current instruction register. The instruction buffer latches the output of the instruction cache. Immediate operands, immediate addresses, and displacements are accessed from the instruction buffer. This unit contains the circuitry and registers for implementing the various addressing modes. The program counter also resides here.

The integer/logic unit contains an Arithmetic/Logic Unit, full barrel shifter, eight data registers and four temporary registers. Simple integer scalar operations and shifts are executed here.

When an instruction requires a memory access, the address computed by the address unit is passed to the address translation unit.

4) The address translation unit performs logical-to-physical address translation on memory addresses passed to it from the address unit. The address translation unit

includes a translation cache which stores recently computed address translations; thus, when the same address is used repeatedly in close proximity in the instruction stream, recalculation is not necessary. The translated address is passed out through the address port to the interconnection network and the computational caches.

5) Data fetched from the computational caches is routed to the CE switch. There it is routed by the CE to the appropriate functional unit for execution. The executing functional units available are the integer/logic unit of the instruction processor and the pipelined vector and floating point unit.

The second main functional system of the CE is the pipelined vector and floating point unit. It is here that floating point data and vectors are processed. Each CE contains several register sets to handle floating point and vector operations. Use of these registers minimize cache and memory references. There are eight 32/64-bit (single or double precision) floating point registers. Additionally, there are eight vector registers; each contains thirty-two 64-bit wide components. Each component in the vector register may hold a single or double precision floating point number or a 32-bit integer number. When the microsequencer indicates, the values in the components are pipelined through the CE switch and through the floating point or integer arithmetic/logic unit based on the data type held in the vector register.

Three additional registers aid in the processing of the data in the vector registers. Data register 4 (d4), the length register, holds a value from 1 to 32 (or, 0 to 31) which indicates the length of the current vector stored in a given vector register. Vectors of lengths longer than 32 are processed iteratively in a programming loop, 32 elements at a time. For example, a vector of length 67 would be processed by completely loading a vector register and piping it through with register d4 holding 32. This action would be repeated a second time. And finally, the vector register would be loaded with 3 elements, register d4 with the value three, and the last three elements of the vector would be piped through the pipe.

Data register 5 (d5), the increment register, may be used to specify the stride between vector elements. For example a value of two in d5 will allow the processing of every other element in a vector.

Data register 6 (d6), the mask register, allows the specifying of any pattern of elements for processing in a vector register.

The third main functional system in the CE is the CE switch. This module operates in the instruction unit and the pipelined vector and floating point unit. The CE switch acts as a data interface between the main memory and computational caches and the various modules of the CE which require data or instructions.

The fourth main functional system in the CE is the concurrency control unit. This functional system is a very important part of the CE as it relates to the multiprocessor environment. Each concurrency unit is connected to the other CEs in the multiprocessor via a concurrency control bus [Figures 46 and 47]. The concurrency instructions of the Alliant instruction set are executed here in the concurrency control unit. One CE may signal another to start, wait, resume, or suspend execution. These communications are carried over the concurrency control bus independent of program data and instruction paths. This hardware concurrency control allows from one to eight CEs to execute on a single program, and provides for the allocation and synchronization of the CEs.

5.2.2 Alliant Cache and Memory Systems

Having examined in some detail the various elements of the CE, this section presents the relationship of the CEs and the computational caches and the global memory. The Alliant cache and memory system service the multiple CEs and IPs of the multiprocessor. There is one large global interleaved memory which services all of the processors. The FX/8 has eight 8-megabyte modules and each module is four-way interleaved.

The main memory is accessed via a high speed, synchronous bus that consists of two 72-bit-wide

bidirectional data paths, a 28-bit address bus, and a control bus. The data bus is driven by memory modules, computational caches, and the IP caches. The memory bus has an 85 nanosecond cycle time and can sustain a bandwidth of 188 megabytes per second when performing sequential reads from memory. It can sustain 150 megabytes per second when performing sequential writes. The address bus has twice the bandwidth needed to maintain the data buses at top utilization. The memory bus supplies the computational and interactive processor caches with their required data. The CEs, computational caches, and memory bus are configured as in Figure 48.

The FX/8 computational cache is composed of two cache modules, totaling 128 kilobytes. Each module is a two-way interleaved cache, thus the full cache functions as a four-way interleaved cache. This computational processor cache is connected to the computational complex via a crossbar interconnect that dynamically connects the eight CEs with the four cache ports. The cache and interconnection network provide a peak bandwidth to and from the computational complex of 376 megabytes per second.

All processors in the Alliant system, CEs and IPs, share a common view of global memory regardless of the cache from which the processor is reading. For example,

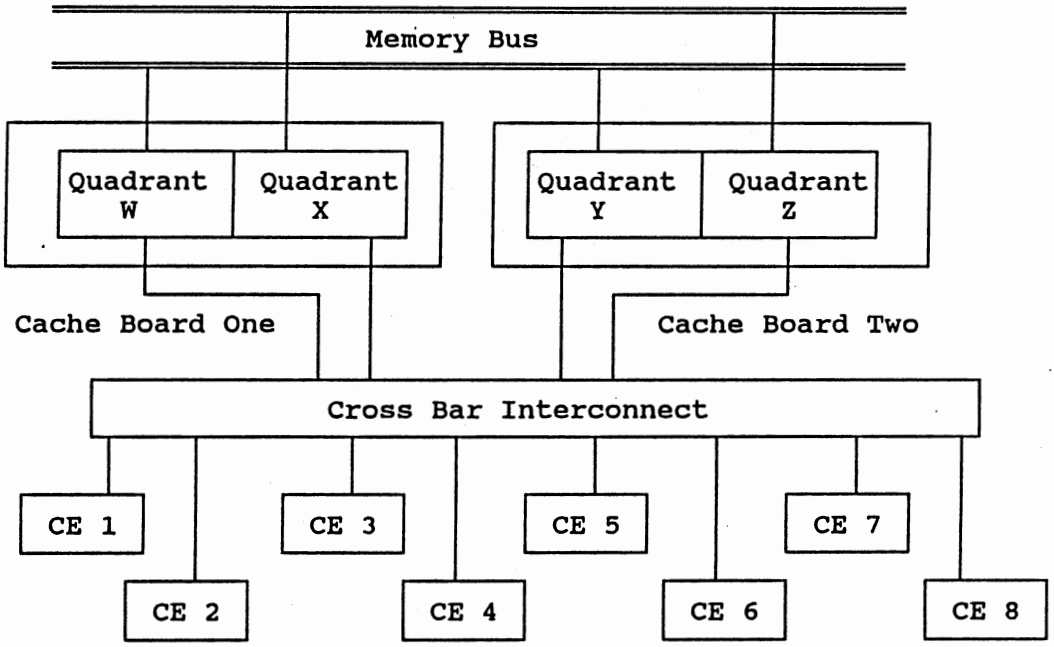


Figure 48. Alliant Cross Bar Interconnect with Memory Bus, Caches, and Computational Elements

suppose the computational complex has been scheduled to execute concurrently the iterations of a given FORTRAN DO-loop. Having completed execution of the loop, the process is 'stopped' on the computational complex and 'resumed' on an IP for execution of the nonparallel portion of the process. Alliant cache coherency guarantees that the IP will have access to the most up-to-date copy of the data when the process is resumed. The cache coherency, common view of global memory, and a minimization of memory bus traffic is maintained by a hardware implementation of a memory-to-cache paging policy termed a write-back policy. On a uniprocessor with only one logical cache, a write-back policy implies waiting until a page has to be replaced in the cache before writing the page back to main memory [7, p. 314, 319]. However, in a multicache system such as that of the Alliant, this may not be adequate to maintain coherency. In the example above, suppose the results of the DO-loop are in the computational processor cache when the process is stopped. If its page is not replaced, the current results of the process will not be in memory for loading into the IP cache. The Alliant overcomes this problem by implementing an additional strategy to that of simple write-back. In each Alliant cache there is a hardware monitor. Each cache monitors the memory address bus; when a cache monitor detects that a request is being made by a second cache for an up-dated page held by the first, then the first cache intercepts

the request and transmits a copy of the page over the memory bus to both the requesting cache and main memory [Figure 49]. This hardware implementation of the write-back policy reduces traffic on the memory bus by minimizing the number of cache to memory writes. Further, the writes are in blocks or pages, allowing full utilization of the 150 megabyte per second sequential write access available in the memory modules.

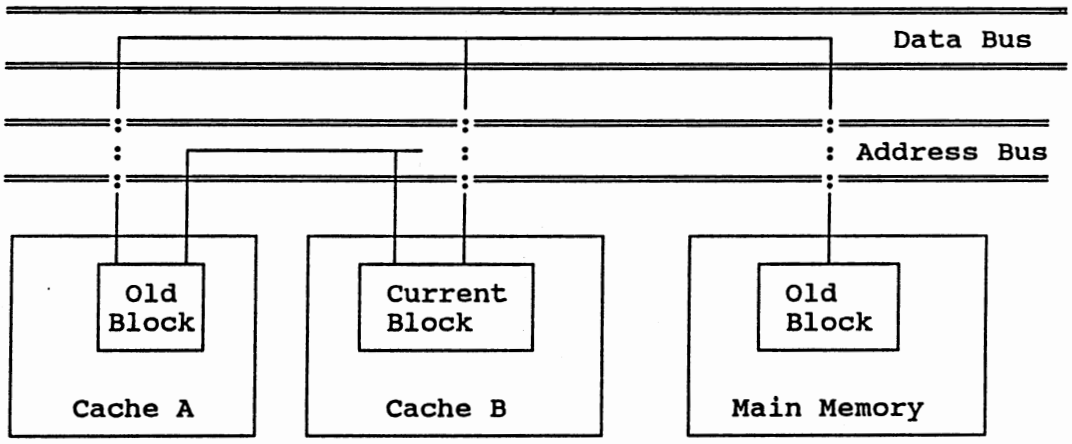


Figure 49. Cache Coherency Maintained by Hardware. Cache A's Request for a Current Copy of Old Block is Intercepted by Cache B. Cache B Writes Back to Both the Requesting Cache and to Main Memory

As discussed in earlier sections, multiprocessors with global memories must deal with memory contention problems. This problem is a primary cause of inefficiency in many parallel systems. In the study of this subject in Section 5.1.1 of this text, the possibility of increasing the number of interleaved memories is identified as one aid in decreasing the frequency of memory contention. However, it is observed that the increase in memory modules raises the cost and lowers the efficiency of the interconnection network. Another technique is to provide private caches with each processor. However, the issues of cache coherence imply some data cannot be kept in the cache or else data modified in one cache must be passed through global memory to update common data held in another cache. Either practice can increase the time for completing access of the required data. The Alliant memory system designers moved the interconnection network so that it interfaces between the CEs and the computational processor cache; the cache is connected to the global memory via the bus.

The purpose of this arrangement is to keep speed performance as high as possible while keeping cost down. By maintaining a large sized computational processor cache with limited ports, four in this case, the interconnection network complexity and cost can be limited. With only a four-by-eight interconnection network, cache bank contention will occur, but the cycle time for cache memory

is much less than that for main memory, thus conflicts arising from more than one processor attempting to read from the same cache will be resolved relatively quickly. Since all CEs have access to the same code and data in the computational cache, the problem of coherence between private caches is eliminated for the computational complex; traffic on the memory bus is reduced. When the computational complex is busy multiprocessing the iterations of a loop, the data generated by one CE from one iteration is immediately available for processing in the next iteration by another CE; again, traffic on the memory bus is minimized.

5.2.3 Concurrency and the Alliant FX/FORTRAN Compiler

This section discusses the general philosophy of the Alliant concurrency, and how it is applied in the Alliant system. The ways that the system supplies parallelism to the programmer is discussed as some FX/FORTRAN programming constructs are presented; and how the constructs are parallelized by the FX/FORTRAN compiler is reviewed.

The Alliant philosophy is based on the premise that a very small percentage of a program generally accounts for most of its running time. That small percentage of a program, the Alliant designers decided, is the execution of loops and advanced array operations, such as A = B,

where A and B are arrays. The idea determined by these designers was to develop a compiler that could recognize loops and array operations, analyze the data dependencies that exist within the constructs, and ascertain what code blocks could be executed in parallel. Additionally, they determined to build a fast multiprocessor in which these types of operations could be executed in parallel under the control of instructions placed in the compiled code by the compiler itself.

Thus, Alliant concurrency uses the program loop as the instruction stream to be executed in parallel. During compilation the FX/FORTRAN compiler identifies those sections of code which may be vectorized and generates vector instructions for them. It determines the loops that may be executed concurrently on multiple CEs and generates the start, wait, resume or suspend instructions in the code to initiate execution and implement any needed synchronization. Remember that these instructions are executed in the concurrency control unit of the CEs and transmitted on the concurrency control bus linking all of the CEs.

As an example of the parallelizing of a loop, consider the following example.

```
      N = 6
      F(1) = 10.0
      DO 12 I = 1 , N
      X1 = A(I)
      X2 = 10.0 + X1 * 2.3
   12 F(I+1) = F(I) + X2
```

This do-loop contains a common data dependency between iterations of a loop. A value computed in the current iteration is used in the next iteration. F(2) = F(1) + X2 and F(3) = F(2) + X2 of iterations 1 and 2, respectively, may not be executed concurrently as the second statement depends on the result of the first. The FX/FORTRAN can detect these statements' dependency and generate wait and resume instructions which will synchronize these statements appropriately so that no F(I+1) computation will be attempted until the corresponding F(I) has been computed and stored in the computational cache. Suppose the loop is to be executed in parallel by three CEs, then the concurrent execution of the iterations of the loop may be illustrated as in Figure 50.

TIME	CE1	CE2	CE3
1	N = 6		
2	F(1) = 10.0		
3	DO 12 I = 1 , N		
4	X1=A(1)	X1=A(2)	X1=A(3)
5	X2=10.0+X1*2.3	X2=10.0+X1*2.3	X2=10.0+X1*2.3
6	F(1+1)=F(1)+X2		
7	X1=A(4)	F(2+1)=F(2)+X2	
8	X2=10.0+X1*2.3	X1=A(5)	F(3+1)=F(3)+X2
9	F(4+1)=F(4)+X2	X2=10.0+X1*2.3	X1=A(6)
10		F(5+1)=F(5)+X2	X2=10.0+X1*2.3
11			F(6+1)=F(6)+X2

Figure 50. Distribution of DO Loop Iterations over Three Computational Elements

In this example, processors CE1, CE2, and CE3 each begin concurrent execution of iterations 1, 2, and 3. As CE2 and CE3 each reach their data dependent statement, they wait until the value they require has been stored. As soon as their awaited value has been stored by the generating processor, then they resume execution. From this point forward, each processor may begin the next appropriate iteration as soon as it has finished the previous one. The process continues until all iterations are complete. In this example, with six iterations, it requires eight time steps to compute the six iterations, while on a uniprocessor, it would have required at least eighteen time steps. As the number of iterations increase, the time expended during the first few iterations to synchronize the loops will be negligible.

The FX/FORTRAN compiler is an extended ANSI standard Fortran-77 compiler that also contains most of the VAX/VMS Fortran extensions. It also has language extensions to allow assignment and other operations on full arrays. The FX/FORTRAN in conjunction with the Alliant FX/8 is designed to perform five modes of execution. They are the following:

1) scalar. Operations are performed serially. If the instruction parser of the CE detects no data dependencies from one instruction to the next, the instructions are pipelined through the processor. The FX/FORTRAN compiler orders instructions in the object

code to take advantage of this processor aspect whenever possible.

The following are scalar operations.

```
A(1) = 5

X = X + 1.0
```

2) vector. The FX/FORTRAN compiler generates vector instructions to utilize the 32 64-bit element registers in the CE whenever simple assignments or operations are made to an array.

The following equivalent constructs could result in the generation of vector instructions to execute upon the 32 element registers in the vector unit of one CE.

```
      DO 12 I = 1,32     OR     A(1:32) = B(1:32)
12    A(I) = B(I)
```

3) scalar concurrent. This mode implies that scalar operations are performed by two or more CEs concurrently. The example of Figure 50 is an example of scalar concurrent. Each reference to an individual array element is handled as a scalar operation as also are the references to non-dimensional variables X1 and X2.

4) vector concurrent. Vector concurrent implies that vector instructions are generated and these are executed concurrently on more than one CE at a time.

An example of code which could be optimized as vector concurrent are the following two equivalent constructs. Elements 1 through 32 of the array could be processed in

vector mode on CE1, elements 33-64 on CE2, elements 65-96 on CE3, and elements 97-100 on CE4.

```
      DO 12 I = 1,100
12     A(I) = A(I) + 5
```

OR

```
A(1:100) = A(1:100) + 5
```

5) concurrent-outer-vector-inner. Where the loops are nested, FX/FORTRAN will attempt to run the outermost loop concurrently while vectorizing the innermost loop.

In the following equivalent constructs, the operations on the elements of each column of matrix A would be vectorized while the processing of each distinct column would be distributed for concurrent execution over the available CEs.

```
   DO 12 J = 1,8
      DO 11 I = 1,100
11      A(I,J) = A(I,J) + 5
12 CONTINUE
```

OR

```
   DO 11 J = 1,8
11   A(1:100,J) = A(1:100,J)
```

OR

```
A(1:100,1:8) = A(1:100,1:8) + 5
```

Within a given loop or array operation, the FX/FORTRAN compiler supplies the programmer with the full scope of optimization available on the Alliant FX/8

architecture. Thus, the programmer may purchase a multiprocessor and run Fortran programs with the problems of data dependence, synchronization, and execution scheduling determined by the compiler and operating system. Here is a very nice turn-key multiprocessor which delivers high performance with very little start-up time required from the user.

The next section discusses a very different multiprocessor, based on both a message passing interprocessor communication system and multiple processors, each with its own local memory.

5.3 The Cosmic Cube

This section is a study of the Cosmic Cube multiprocessor. It is a multiprocessor in which each processor is a computer module with its own local memory. The computer modules are connected by a pattern of channels over which message packets are transferred. These messages provide the interprocessor communication of the system. These concepts are discussed in general in sections 5.1.1 and 5.1.2.

The Cosmic Cube's communication links are configured in a manner equivalent topologically to a multidimensional cube - a hypercube [Figure 29]. This processor connection topology has gained a great deal of popularity in the design of parallel systems [59]. The next section of this text is a study of the hypercube topology. After

considering the hypercube topology, the Cosmic Cube's application of that topology is presented. Finally, the Intel Personal Supercomputer, iPSC, the offspring of the Cosmic Cube is surveyed very briefly.

5.3.1 Hypercube Topology

The hypercube is a binary n-cube, also referred to as a binary hypercube or boolean hypercube. A binary n-cube may be described in one of several ways.

Intuitively, the 3-cube or 3 dimensional cube is the familiar cube. Higher-dimensioned cubes are built from this basic structure. The "dimension" in these higher-dimensions is n, where $2^n = N$ and N is the number of vertices in the cube. Thus, a cube with 16 vertices would be a 4 dimensional cube or 4-cube since 2^4 is equal to 16; and a 5-cube would have $2^5 = 32$ vertices. Each vertex in a cube may be referred to as a node. In a multiprocessor with a hypercube message-transfer system, each node represents a computer module. Each node in an n-cube is attached to its n nearest neighbors.

Also, a hypercube may be described recursively; the n-cube that is used to connect $2^n = N$-nodes is assembled from two (n-1)-cubes, with corresponding nodes connected by an additional channel [95]. This property clearly identifies one of the benefits of the hypercube; it offers users of such systems an option to expand to larger systems as need arises. Other architectures, especially

tightly coupled ones are limited in the extent to which they can be expanded.

A final hypercube description will be based on graph theory. This description is helpful in understanding some relevant topological properties of the hypercube.

Let $N = 2^n$ and let $\langle N \rangle = \{0,1,2,\ldots,N-1\}$. An n-cube with N nodes may be described as a graph G = (V,E), where $V = \langle N \rangle = \{0,1,2,\ldots,N-1\}$ in binary; and E is the set of edges incident upon vertices that differ by exactly one bit in their labels [Figure 51].

For example, if N = 4, then the vertices of the cube may be labelled 00, 01, 10, 11 (base 2) [Figure 51]. The edges of the graph are (00,01), (01,11), (11,10), and (10,00).

A number of standard topologies such as linear arrays, rings, and 2 dimensional mesh can be embedded into the binary n-cube.

For example, Figure 52 shows the embedding of an 8 node linear array in a 3-cube. Figure 53 shows the embedding of a 16 node ring into a 4-cube. Figure 54 shows the embedding of a 16 node near neighbor mesh on a 4-cube.

Dimension	Nodes	Channels	Topology
0	1	0	
1	2	1	
2	4	4	
3	8	12	
4	16	32	

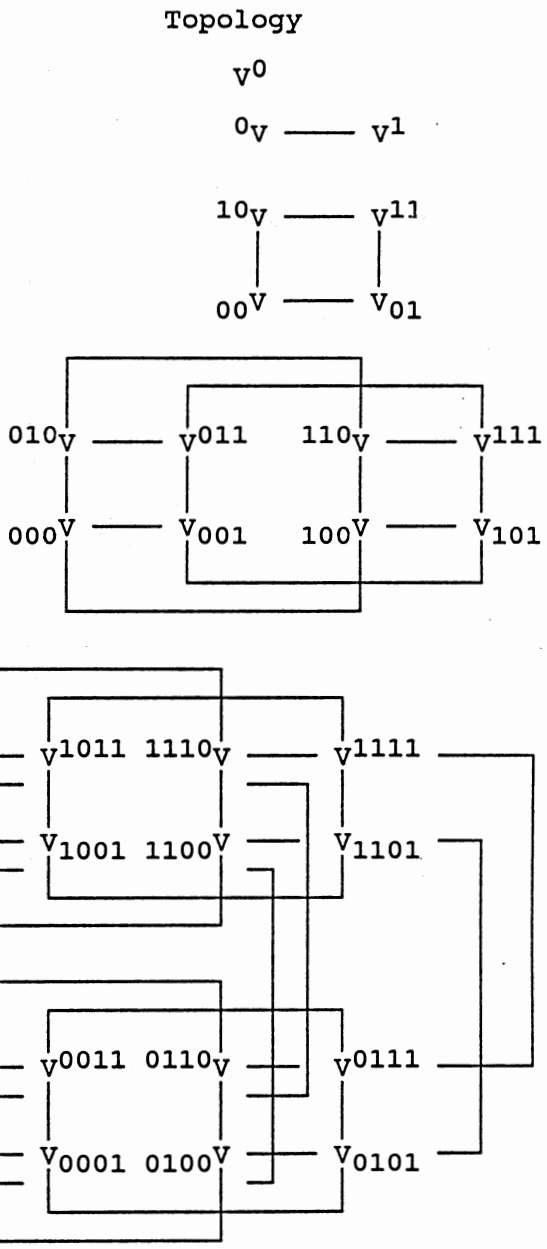


Figure 51. Correlation of Hypercube Dimension, Node Count, Channels, and Topology. V's are Vertices.

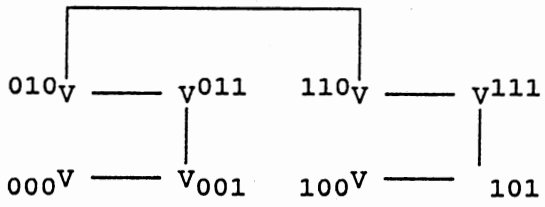


Figure 52. Eight Node Linear Array Topology on a 3-Cube

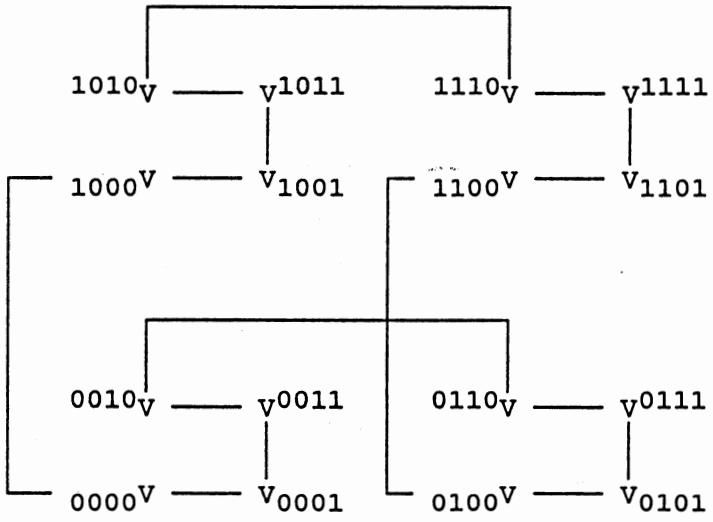


Figure 53. Sixteen node Ring Topology on a 4-Cube

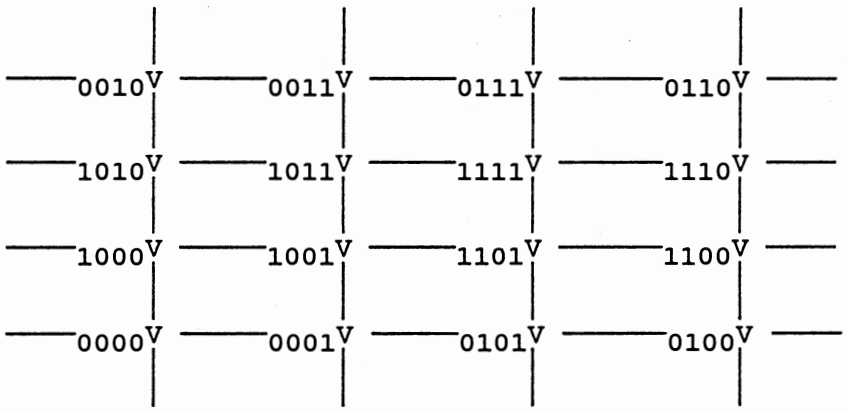


Figure 54. Sixteen Node Near Neighbor Mesh on a 4-Cube. Note the Node Addresses are Reordered to Facilitate Viewing

A large amount of research activity in concurrent architectures has centered on interconnect structures. Various structures such as linear arrays, rings, meshes, and others have been considered [30]. The hypercube topology is flexible enough to simulate these structures. Researchers interested in studying the properties of these different interconnect structures may find a computer based on the hypercube topology helpful in their studies.

A specific implementation of a hypercube based multiprocessor is presented in the next section.

5.3.2 The Cosmic Cube Multiprocessor and its Offspring, the iPSC

Two well known machines have been built using the hypercube topology as the basis for their interconnect system. The first was built at the California Institute of Technology, under the primary direction of C. L. Seitz and G. C. Fox [70][95]. The California Institute of Technology machine is known by several names, including the Nearest Neighbor Concurrent Processor (NNCP), Hypercube, Homogeneous machine, and the Cosmic Cube [95].

The second is the Intel Personal Supercomputer or iPSC. It is based on the hypercube interconnection scheme developed by Seitz and Fox at California Institute of Technology. After licensing the concept from California Institute of Technology, Intel developed the iPSC.

Consequently the fundamental architectural attributes of the Cosmic Cube and the iPSC are very similar [22].

5.3.2.1 The Cosmic Cube. The Cosmic Cube is based upon a 64 node 6-cube hypercube topology. It has a computer module at each of its hypercube vertices. Each computer module is composed of an Intel 8086/8087 microprocessor chip, 128 kilobytes of dynamic RAM with parity checking, and 8 kilobytes of ROM for initialization, bootstrap loader, RAM refresh, and dynamic testing programs. Additionally, each computer module has 7 channels total; one for each connection to an adjacent node in the hypercube (there are 6), and one additional channel for connection to a cube manager or intermediate host. Each channel is asynchronous, full-duplex, and includes queue storage for a 64-bit hardware packet. The queue is present in each direction in order to decouple the sending and receiving program executions. The channel to the intermediate host is for program and data loading and communication with the "outside world" [70][95].

Each node or computer module executes its own local copy of the operating system. This operating system allows multiprogramming and timeslicing in a round robin fashion. Thus, in any given time period, each node may be context switching between the operating system and one or more user programs.

The Cosmic Cube architecture is said to be a proven one for computationally intensive problems from the natural sciences which enjoy the property of physical or logical partitionability [95]. Whatever the application, it is the programmer's responsibility to formulate and express an algorithm or job explicitly in terms of a collection of communicating subprogram executions. The programmer is also required to determine and control the appropriate assignments of each subprogram to a computer module or node in order to achieve the desired concurrency and load balancing. This allows a lot of freedom and control over the program's activities but it also allows a lot of room for programmer error.

Compilers for the languages FORTRAN, Pascal, and C exist for the system. These languages have been extended with external procedures which implement the sending and receiving of messages. The programs are compiled on other computers such as a VAX host which is attached to the intermediate host or cube manager mentioned earlier. The job's subprograms, as binary code, data, and stack segments, are routed from the VAX to the intermediate host and from the intermediate host to each node as determined appropriate by the programmer. Each subprogram assigned to a node runs independently of the subprograms running in other nodes except for the receiving and sending of messages over the channels which comprise the message transfer system.

The external message-send/receive procedures which extend the languages of the system are devised so that they must be called by the programmer when data or control information is to pass from one executing subprogram to another. For an executing subprogram to send a message, the "SEND" library routine is called and passed the necessary parameters; these include the IDs of both the sending and the receiving executing subprograms (each one is assigned an ID by the local resident operating system), the addresses of the nodes to and from which the message is being sent (these addresses would be analogous to the binary numbers discussed earlier in the context of the hypercube topology), and the data itself. This information is packaged in a "packet" along with some control information and transmitted out along a channel. If there are intervening nodes, the local operating systems of those intervening nodes retransmit the packet. The node addresses are used by the operating systems to determine the best channel over which to transmit. To receive a message, the subprogram which awaits the packet must invoke a "RECV" procedure with the appropriate parameters. When the packet arrives, the operating system picks it up from the channel queue and passes it to the named subprogram which executed the "RECV".

5.3.2.2 The iPSC. Intel's iPSC is similar to the Cosmic cube in many ways. The technology upon which it is

built is more advanced. Each computer module in the hypercube is based on the 80286/80287 microprocessor chip. The RAM memory has been boosted from the 128 kilobytes of the Cosmic Cube to 512 kilobytes. Each node has 8 channels; each channel is controlled by an 82586 LAN coprocessor. This allows considerable leeway in the dimension of the hypercube. The Intel iPSC may be configured as a 5, 6, or 7-cube, depending on the needs and financial standing of the purchaser [44]. The 8-th channel is a global Ethernet channel which provides direct access to and from the cube manager (a system 80286/310) for program loading, data I/O, and diagnostics [44][59].

In Intel's latest configuration, the iPSC-VX, each node has a vector coprocessor that occupies the slot adjacent to the processor in the system. A private iLBX bus connects the two boards in a tightly coupled, shared-memory interface that maximizes system efficiency but is transparent to the user. It is reported that this brings the peak performance of the iPSC-VX/d4, a 16 node 4-cube, to approximately 106 MFLOPS [59].

The Cosmic Cube and iPSC offer the user a multiprocessor architecture which provides a great degree of concurrency and high rate of performance. However, the programmer must perform his own program partitioning and synchronization steps while attempting to minimize the path length for message passing; and, to keep 64 processors (or up to 128 on the iPSC) coordinated and

working at a high rate of efficiency or utilization may well require quite a lot of ingenuity on the part of the programmer. Nevertheless, if the programmer is up to the challenge, computationally intensive problems can receive rapid service from such a system.

5.4. Summary

This chapter introduces multiprocessors. A multiprocessor affords a programmer with a computer system that allows him to exploit parallelism between blocks of program code which contain no data dependencies. It reduces the time to solve a single application. The multiprocessor accomplishes the endeavor by the distribution of the independent code blocks of the single job over multiple CPUs. Each processor executes a different code block; each instruction set executing on its own data. In this way, the multiprocessor provides an MIMD architecture and improved turnaround time for the user.

Various issues of multiprocessor design are discussed including those of memory, interprocess communication, operating systems, and exploitation of parallelism.

Finally, this chapter presents two different types of multiprocessors. The tightly coupled, global memory, Alliant FX/8 and the loosely coupled, local memory, Cosmic Cube are examined. Also, the Intel iPSC, the offspring of the Cosmic Cube is reviewed briefly.

CHAPTER VI

DATA FLOW COMPUTERS: THE DENNIS STATIC DATA FLOW MACHINE AND THE MANCHESTER DYNAMIC DATA FLOW MACHINE

6.0 An Introduction to Data Flow

The need for faster computations, shorter turnaround times, and greater system throughput has generated a great deal of activity directed toward creating von Neumann machines which operate faster. In the preceding chapters, architectures that extend the von Neumann architecture to allow the exploitation of parallelism in various ways are presented. This chapter introduces an architectural approach that is totally different from any of the ones studied in the prior chapters. It is that of data flow computers, a non-von Neumann architecture. After exploring the general aspects of the data flow machine, two significant computers that implement the data flow architecture are reviewed. The first is the Static Data Flow Machine built by Jack Dennis and his associates at Massachusetts Institute of Technology. The second is the Manchester Data Flow Computer built by researchers at the

University of Manchester, England; the principal workers being John R. Gurd, C.C. Kirkham, and Ian Watson.

The von Neumann architecture implies the program is loaded sequentially into main memory and program execution is under the control of a program counter. The von Neumann architecture is that of control flow. The data flow concept of computer operation is that an instruction executes as soon as all of its operands are available [80].

In a multiprocessor with global memory, it is possible for the processors to have race conditions while updating a memory cell. In such a situation, two processors may try to write to the same location, a write-write race. A similar problem is that of the read-before-write race. When producer and consumer processes share data cells and execute concurrently, the consumer may read the shared location before the producer has written to it. Synchronization must be accomplished by test-and-set, semaphores, or message-based primitives. Such synchronization can incur considerable overhead that degrades the overall benefits of the parallel approach [5].

The race problems described are inherent to the shared data cell concept; it is inherent to functions that have call-by-reference parameter passing, that is, the function has the address of its parameter, not its value. Two functions that pass parameters by reference and share

common updatable parameters (by one function or the other) cannot execute concurrent processes successfully without synchronization. On the other hand, consider a function, F, with no globally defined variables that employs call-by-value parameter passing, that is, it has its own individual copy of its input parameters. If F returns a distinct value to any function that requires its returned value, then function F can execute concurrently with any other such function that does not pass it a parameter and to whom it does not return a value. Function F cannot have a race condition with any such functions as there are no data dependencies between them.

These observations may lead to an understanding of the data flow architecture. At the machine level, one can think of each individual machine instruction as a small function. Under the von Neumann architectural approach, each machine instruction's operand (parameter) is established by the address of the data cell where the actual value resides. Thus, the von Neumann instruction is a small function with pass by reference parameters. It is desired to establish a parallel computer architecture with multiple processing units that allows concurrent execution of these small functions. Machine level synchronization techniques such as FORK and JOIN may be used to specify explicitly single instructions to be executed concurrently. However, the number of these functions (instructions) which could execute concurrently

are limited by data dependencies and the resulting race conditions.

However, if an architecture is developed that allows each function (instruction) to pass parameters (operands) by value, each instruction will have its own individual copies of the operands. Since each function will then be free from any synchronization constraints, it can be assigned a processing unit as soon as its parameters are available. A copy of its result, or returned value, is awaited by the functions (instructions) that use that value. They, in turn, begin execution as soon as they have received all their parameters (operands). This implies that the instructions can execute asynchronously, without the control of a program counter; and concurrently with any instructions that do not supply their operand values and do not await their returned results. Ordering of instructions is based on data dependencies within a program. This computer architecture is currently implemented by several groups of computer designers around the world; it is referred to as a data flow or data-driven architecture since the availability of data values determines the execution sequencing of the instructions.

6.1 An Introduction to the Data Flow Graph

Data flow concepts first emerged in the 1960's. Writing compilers for standard serial programs, compiler writers used data flow graphs to do performance optimization. A data flow graph is a directed graph in which the vertices represent primitive functions such as addition or subtraction and the edges represent data dependencies between functions. By the 1970's, it was realized that if such a graph could be executed directly by a computer architecture, then the parallelism in a given algorithm would be exploited greatly. In a data flow architectural environment, data flow programs are represented by such directed graphs. Each primitive function is represented by an activity template [Figure 55]. Each such template is very closely related to the actual machine instructions used in prototype data flow computers. Each activity template is understood to contain fields for holding the operands' values, or tokens, when they arrive. This is call-by-value parameter passing. Additionally, each instruction contains the addresses, or destinations, of the instructions awaiting copies of the result value returned by the current instruction's execution. The edges on the graph indicate the logical paths along which the result will be forwarded. Thus, tokens move along the directed edges of

the graph. As it moves along such an edge, each token carries not only the operand value but also the "name" of its destination instruction; it may carry other information as well as will be discussed later in the context of dynamic tagged systems. Each template may execute, or fire, according to its firing rules. The fundamental firing rule of any data flow system is that each template may only fire when all its operands are present. A template which meets its firing conditions is said to be enabled.

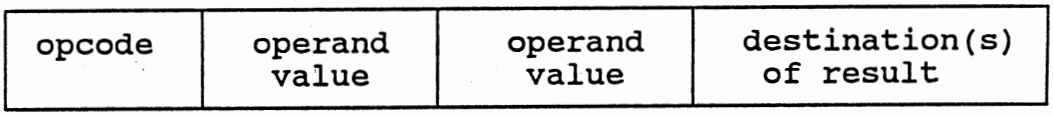


Figure 55. Activity Template

For example, a program may have the following computations.

$$z = v * (x + y) - x * (u + w)$$

This program segment is represented by the data flow graph of Figure 56. If the additions received their operand values during the same time interval, each addition would be enabled. Each could be assigned a processing unit and

executed, or fired. Thus, their execution could occur during the same time period.

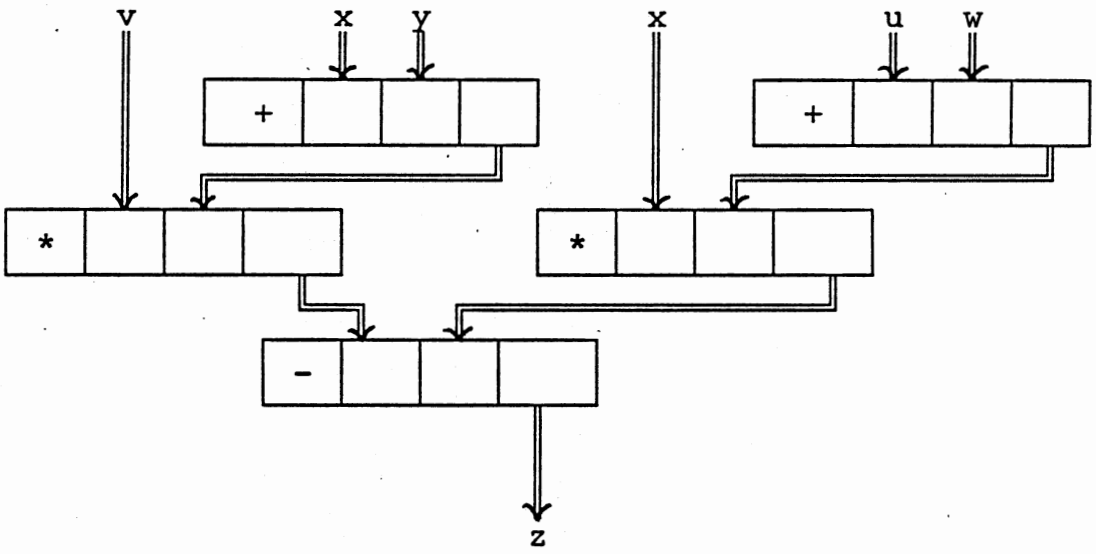


Figure 56. Data flow graph for
z = v * (x + y) - x * (u + w)

Any two enabled operations can be fired in any order or concurrently. In Figure 56, when the sum of x and y is available, it can be stored in the multiply instruction template where the value of v had been copied; similarly, the sum of u and w can be stored in its prescribed activity template with the value of x. Each multiply can

begin execution on a free processor as soon as its operand values are in place and the multiplies can proceed concurrently. Each multiply instruction result is routed to the subtraction template, triggering upon their arrival the execution of the subtraction operation upon a free processor. The final result is routed to any instruction requiring the z value or to output.

If the computation of z is done for a series of distinct u_i, v_i, w_i, x_i, and y_i, i = 1,2,3,..,n, values then the values can be pipelined through the data flow graph. Thus, as soon as $x_1 + y_1$ and $u_1 + w_1$ are computed, their results are passed to the multiplies. The add operations can begin again when x_2 and y_2 arrive at their add, and similarly for u_2 and w_2. This allows multiple levels of parallel exploitations. Figure 57 demonstrates an alternative pictorial representation of Figure 56 in which square nodes represent the activity templates. Figure 57 demonstrates the same graph with pipelined computation. The darkened squares represent the tokens with their values written beside them as they flow through the graph.

The lack of data dependency between the addition operators, and also between the multiply operators, in the example graph is sometimes called horizontal, or spatial, concurrency. This contrasts with the temporal concurrency of the pipeline [19].

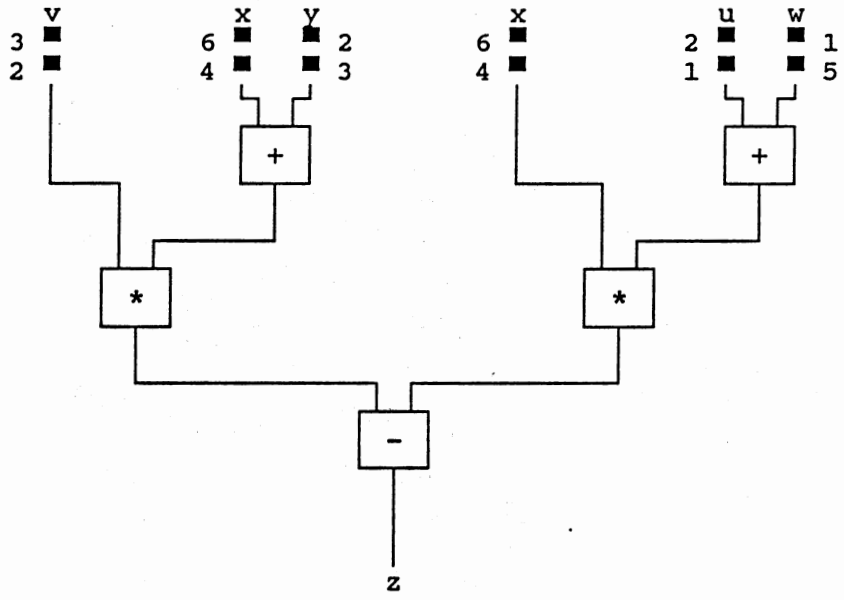


Figure 57.a. Values ready to enter the graph for computation. First expression evaluated is 2*(3 + 4) - 4*(1 + 5). Second expression is 3*(2 + 6) - 6*(2 + 1).

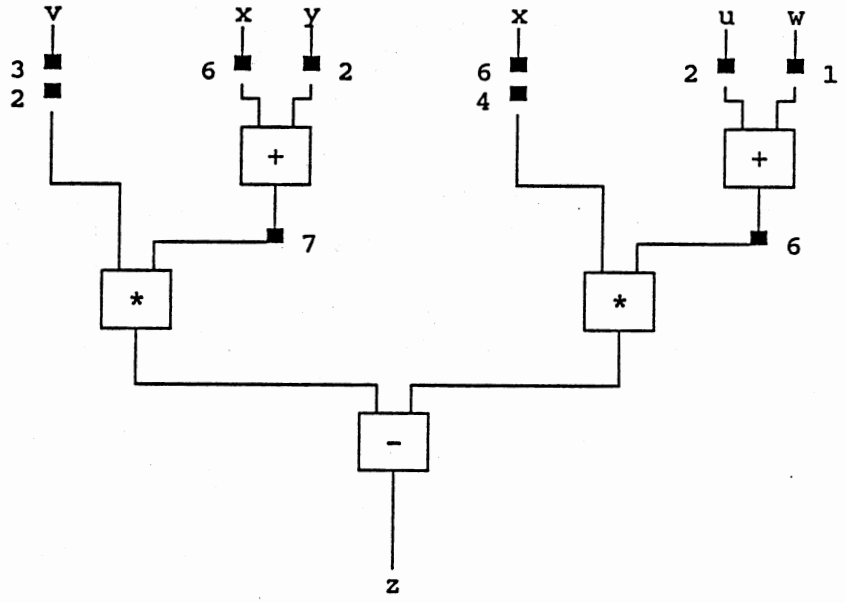


Figure 57.b. Addition template fires with values from first expression.

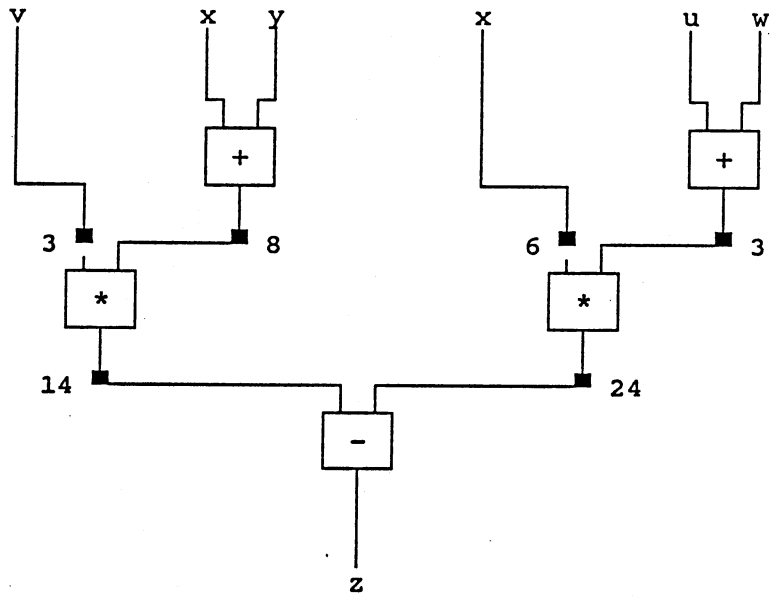


Figure 57.c. Multiply templates fire with values from first expression. Add templates fire with values from second expression.

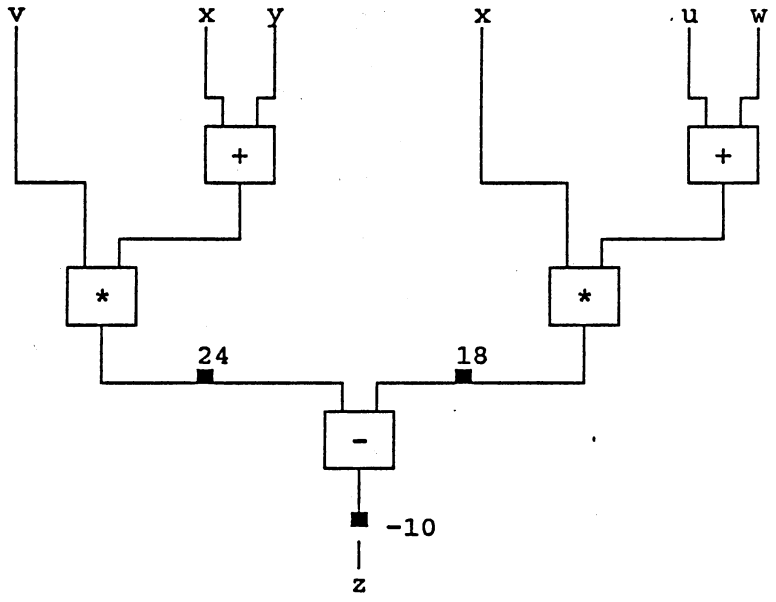


Figure 57.d. Subtraction template fires with result first expression giving (-10). Multiply templates fire with values from second expression.

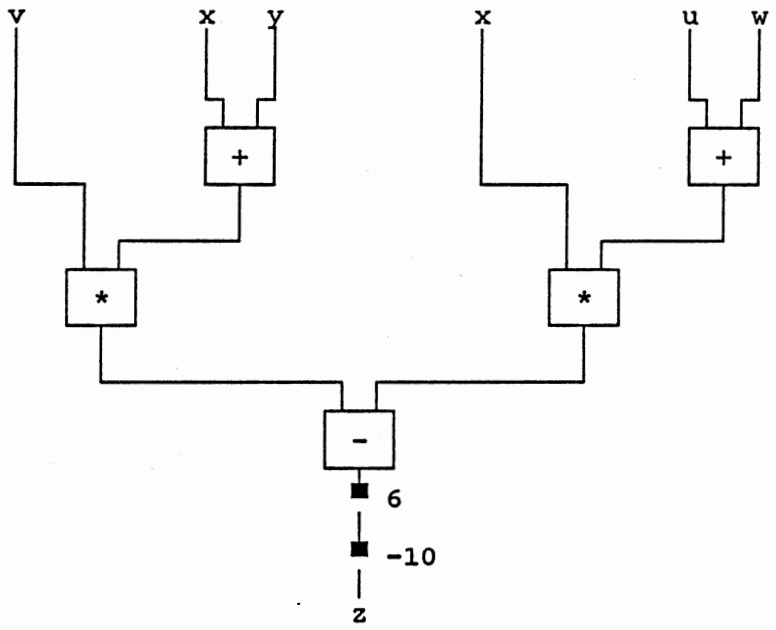


Figure 57.e. Subtraction template fires with values from second expression giving a result of 6.

6.2 The Static Data Flow Approach and the Dynamic Data Flow Approach to Activity Template Firing and Program Graphs

As values are piped into the data flow graph pipeline there is a problem of matching the rate of a producer template to that of a consumer template. For example, in the graph of Figure 57, the time required to produce the sum of x_2 and y_2 may be less than that of computing the product of v_1 and $(x_1 + y_1)$. Thus, the addition template may be ready to fire (execute) based on its operand availability before the multiply has completed processing its current operand values and is ready to receive a new operand. The destination of the addition template's output token would not be ready to accept a new value. Control of the values or tokens passing through the data flow graph pipeline is a design issue which has been handled in various ways.

The two most common techniques are termed static and dynamic. This section investigates the fundamental concepts of these two approaches. In the static approach, use of the data flow graph is limited by allowing only one token to reside on each edge at any time. The firing rule is rewritten so that an operation is enabled only when:

1) its input tokens or operands are available and

2) no tokens exist on its output arc. That is, its last output value has been processed by its awaiting operation. The destination instruction is ready to receive a new operand value.

This implies sequential pipelined use of the data flow graph. Such pipelining is implemented by use of acknowledge signals that are returned to the producer templates by the consumer templates when they are ready to receive a new token. No template can fire unless it has received its acknowledge or control token. These acknowledge signals effectively double the number of edges in the data flow graph. Additionally, as in any such pipeline, the speed of the slowest stage determines the overall throughput for the pipeline. Thus the slowest executing template would determine the output rate of the graph.

In the dynamic approach, each operation may fire when all input tokens are available and multiple tokens may appear on output arcs. Thus, the dynamic approach maintains the fundamental firing rule. However, tokens carry with them a tag. The tag may also be called a label or color. These tags identify the order of the tokens on the input arc to the consuming template. The tokens are consumed according to the order implied by the tags. No control or acknowledge tokens are required. Instead, additional time and hardware is needed to attach labels to

tokens, and match like tagged tokens for consumption [41, p. 755].

Both of these approaches have been investigated and implemented by researchers at various locations. Dennis's static machine and the Manchester machine are implementations of these two approaches and are discussed later in this paper.

6.3 Looping with a Data Flow Graph

The data flow graph example of Figure 57 described a straight line computation. However, few programs of interest can be written without conditional looping. Looping strategies, or iteration, can be represented through cyclic data flow graphs. Additional activity templates other than the simple arithmetic ones must be included in the implied instruction set in order to achieve selective routing of data tokens among the operations. Such templates of primary usage are displayed pictorially in Figure 58. An important distinction between these operations and the simple arithmetic operations discussed earlier is that they include as their operands not only data values but also controlling boolean values. They may be described as [19, 41, p. 742]:

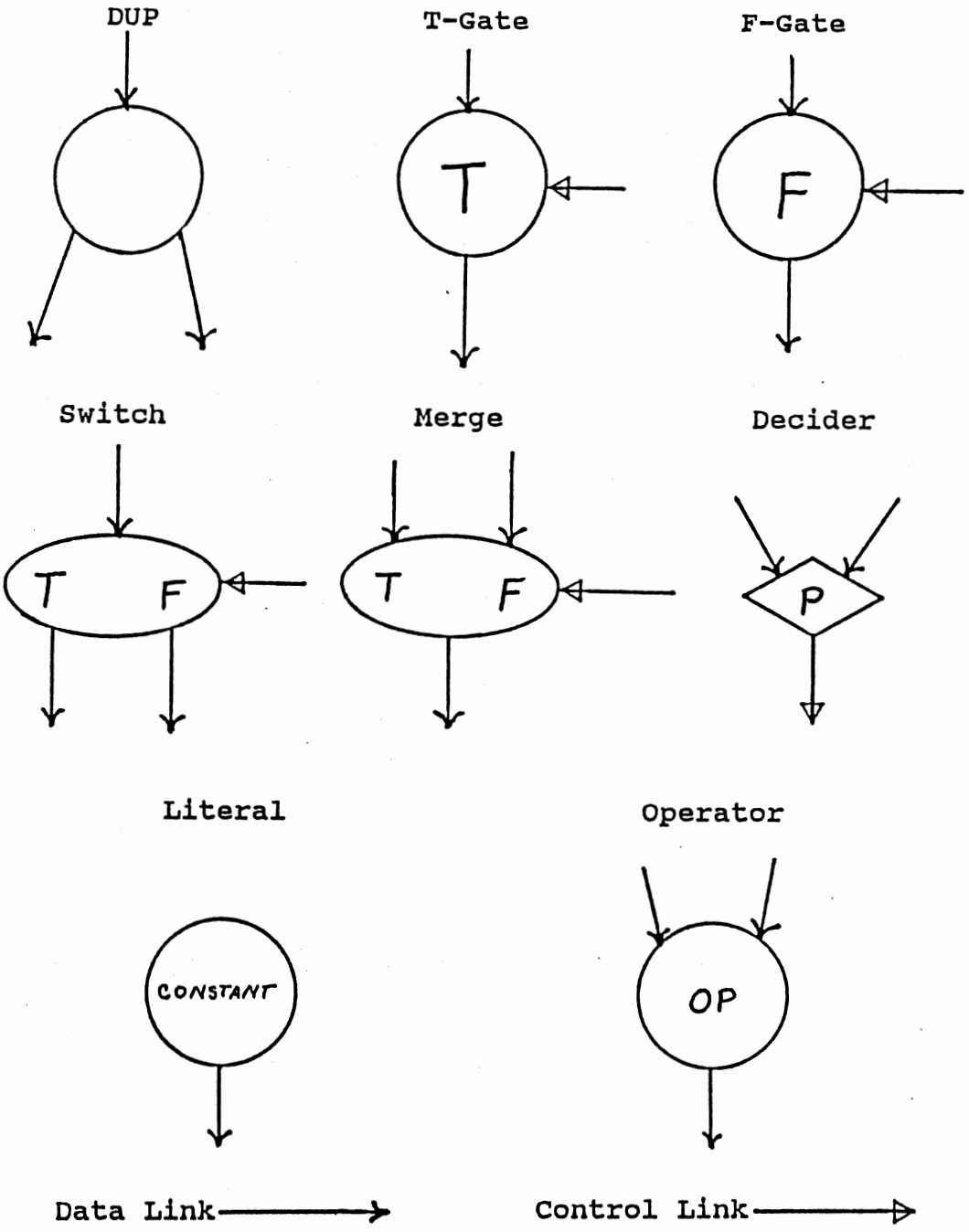


Figure 58. Operators or Nodes for a Data Flow Graph

1) DUP. When a token is required as input in more than one instruction, its value is replicated and placed on multiple output paths. There are DUP operations defined for both data and boolean tokens.

2) T and F gates. These gates are designed to pass or block data tokens along a designated pathway. Each operation has a data token input path and a boolean token input path; each has one output path. T gates route their data tokens onto the output arc only when their boolean token is true. No value is sent out if the boolean token is false. F gates have the opposite action.

3) SWITCH. The purpose of this operation is to direct a data token down one of two possible output paths, a "true" path and a "false" path. It has as its input a data token and a boolean token. If the boolean token is true, the data token is routed onto the output "true" path; a false boolean value sends the data value down the "false" path.

The SWITCH operation could be implemented as a DUP followed by a T and an F gate applied to each of its output arcs with the identical control signals delivered to each gate.

4) MERGE. This operation may be thought of as a selector function. It selects a token from one of two possible input paths and places the selected value onto its one output path. There are two data token input paths, recognized as "true" and "false" paths.

Additionally, there is a boolean token input path. If the boolean token is true, the token on the "true" input path is routed onto the one output path; otherwise, the one on the "false" path is selected.

5) DECIDER. Decider operations are used to implement conditional strategies. It has as its input two (or more) data tokens. The operation has associated with it some defined predicate which when applied to the input values can be determined as true or false. The decider has one output path which carries the boolean token determined by the predicate.

6) LITERAL. The LITERAL operation makes the constants or literal of an expression available to the proper instruction. It has no input path, only a literal data token output path. Such a node regenerates its constant value as often as it is needed by nodes to which it's value is input. As soon as its constant token is removed from its output arc, it fires again [19]

7) OPERATOR. This final pictorial template corresponds to the templates in the example of Figure 56. The input to the operation are one are more data tokens. There is one output path that carries a data token.

These templates can be combined to represent iterative computations. The following algorithm computes the integer power $z = x^n$.

```
input x,n;
y = 1;
i = n;
WHILE i > 0 do
     begin
          y =y * x;
          i = i - 1
     end;
z = y;
output z;
```

This computation can be represented by the data flow graph of Figure 59 [41, p. 743]. Snapshots of the computation are pictured as the tokens flow through their successive steps. The computation is performed with x = 3 and n = 1. The darkened circles on the arcs represent data tokens with their values written to the side, the squares represent boolean tokens with their values. The time for each operational step is assumed to be one. The algorithm is initiated with "false" values on the boolean input arcs for the MERGE operations.

Thus, looping can be implemented in a data flow environment using the operators outlined in Figure 58.

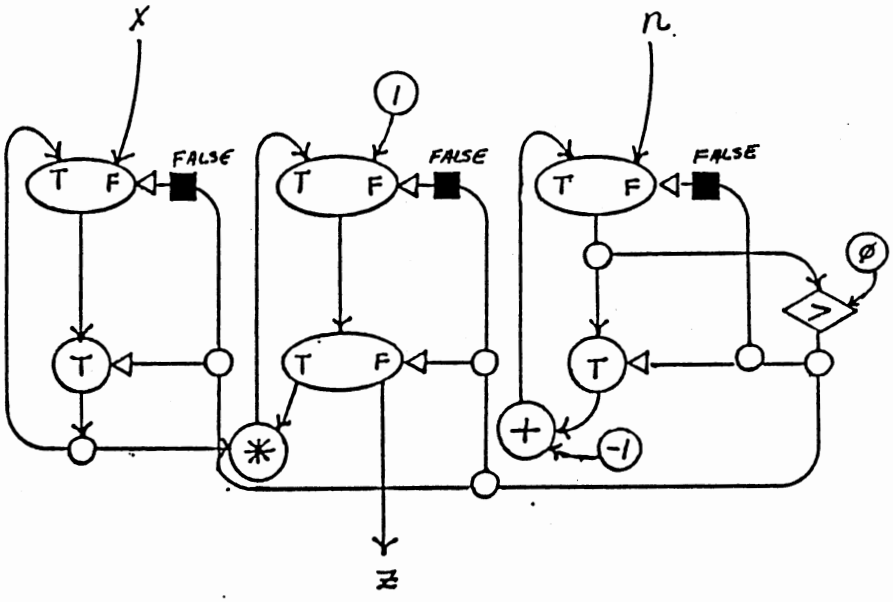


Figure 59.a. Data Flow Graph Corresponding to $z = x^n$ at time t_0 [41, p. 743]

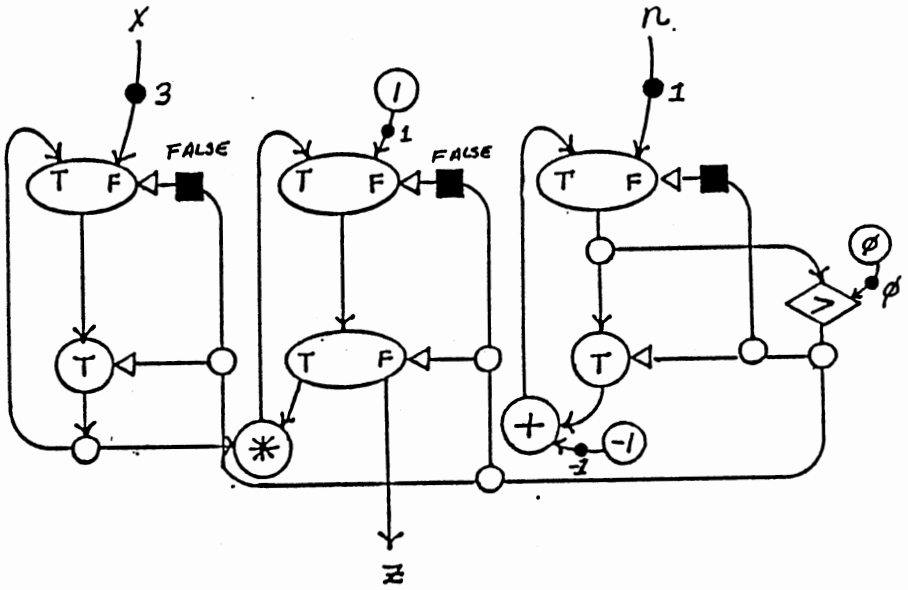


Figure 59.b. Tokens at time t_1

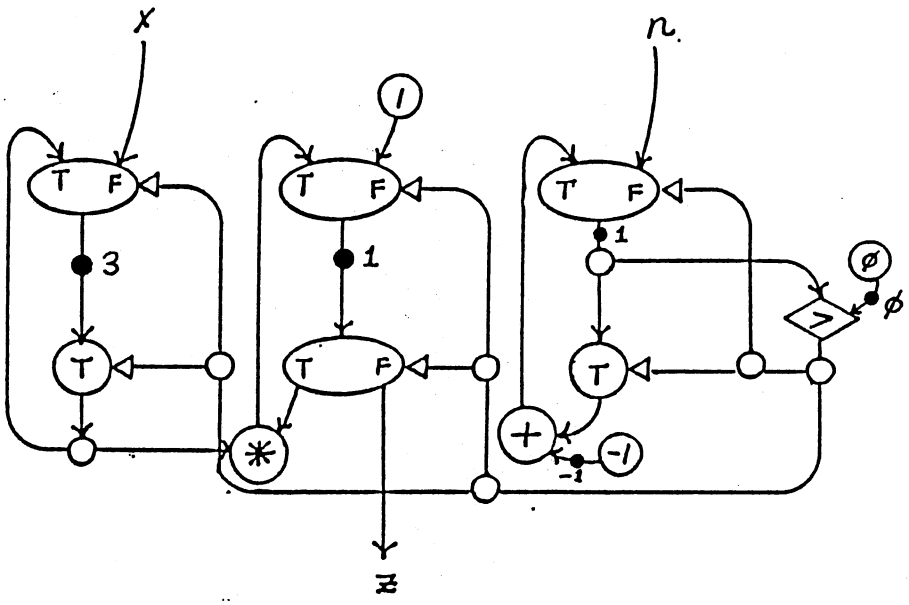


Figure 59.c. Tokens at time t_2

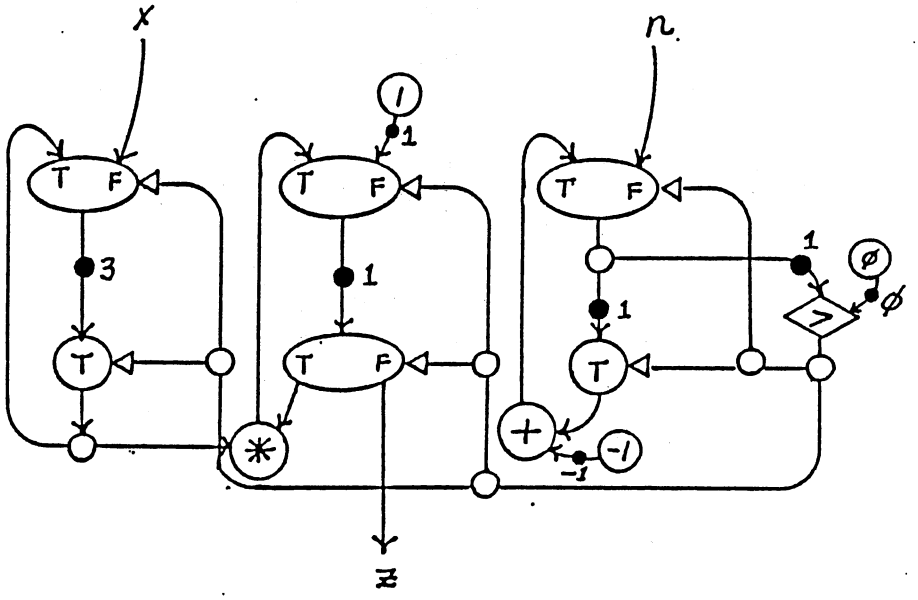


Figure 59.d. Tokens at time t_3

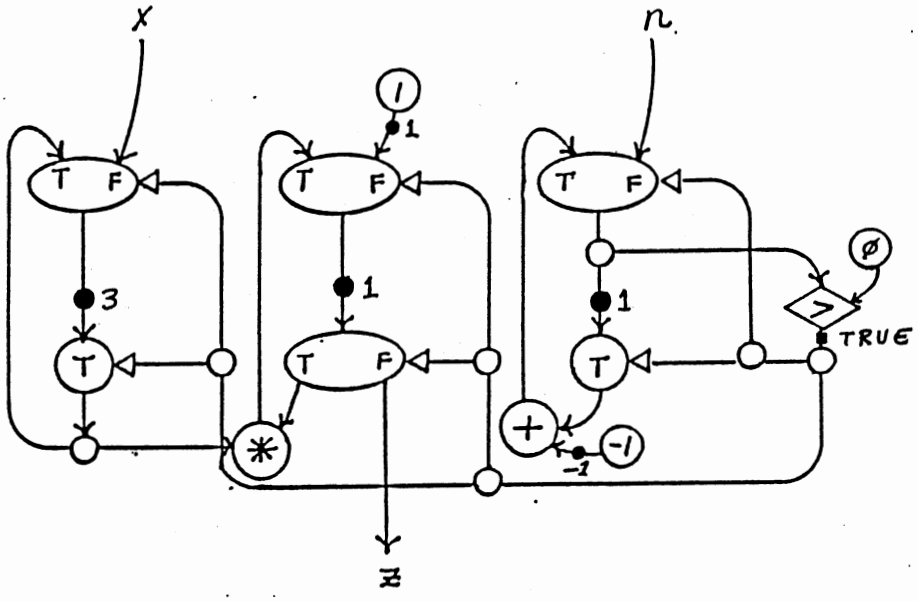


Figure 59.e. Tokens at time t_4

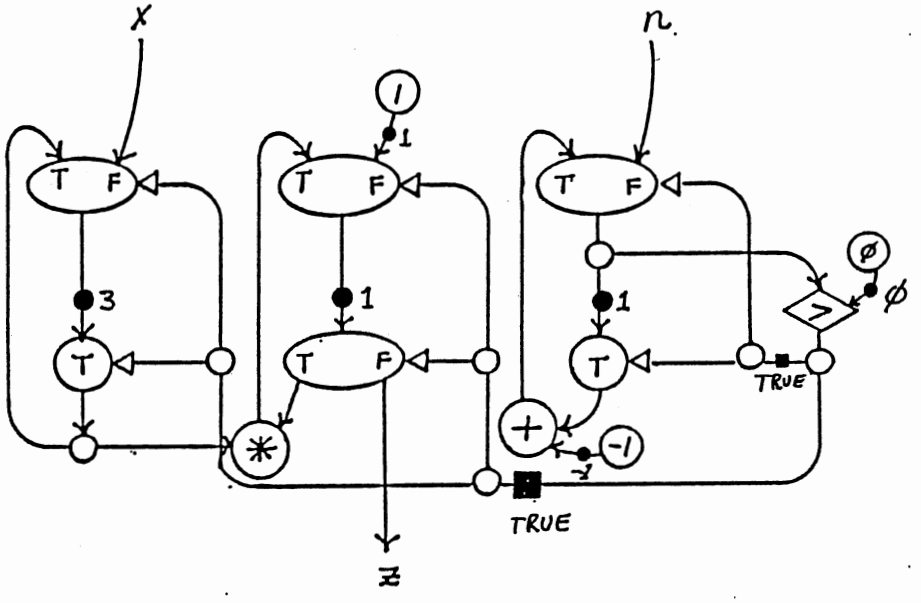


Figure 59.f. Tokens at time t_5

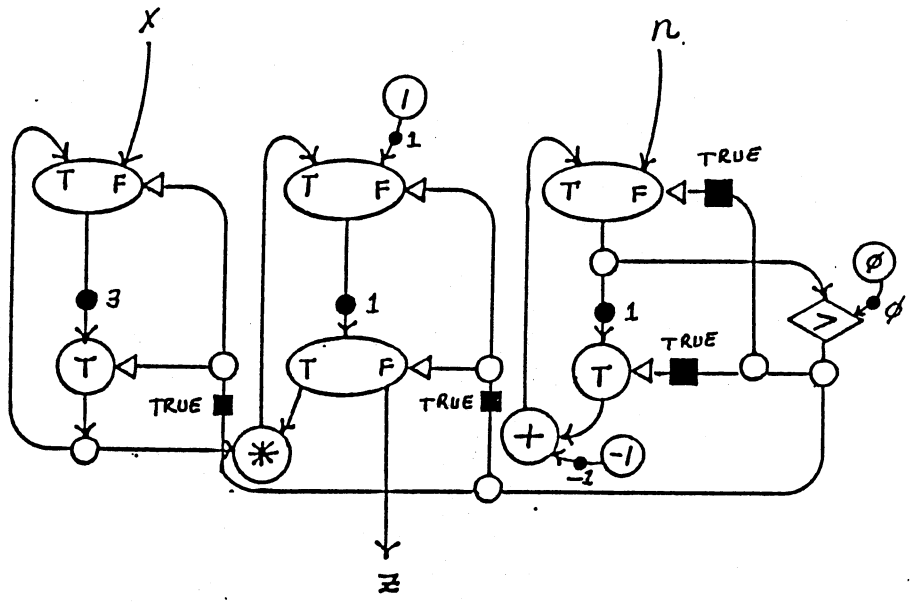


Figure 59.g. Tokens at time t_6

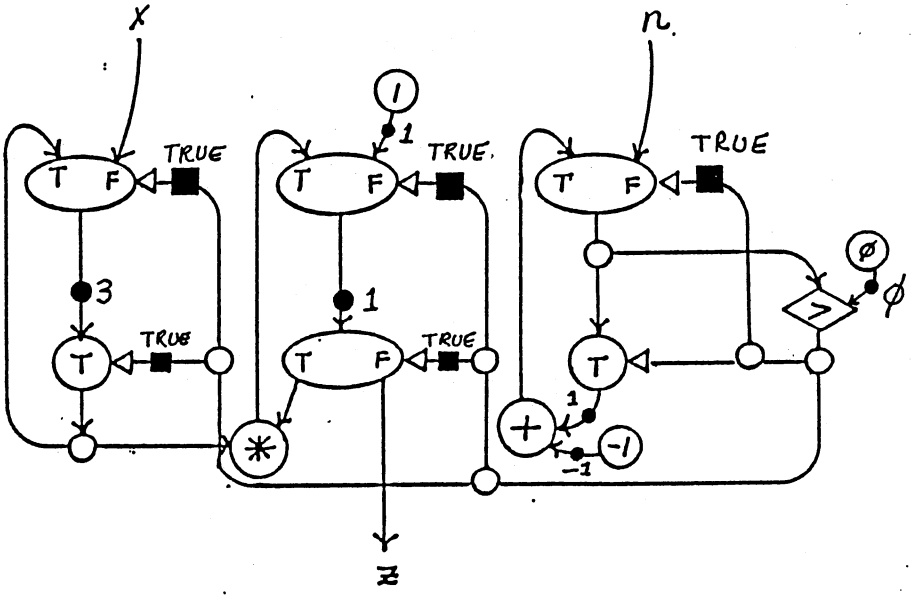


Figure 59.h. Tokens at time t_7

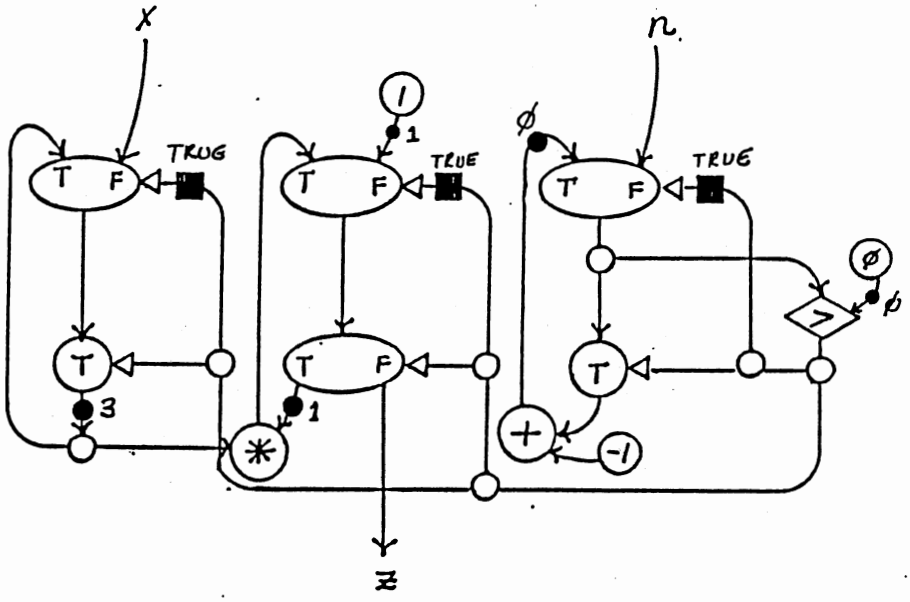


Figure 59.i. Tokens at time t_8

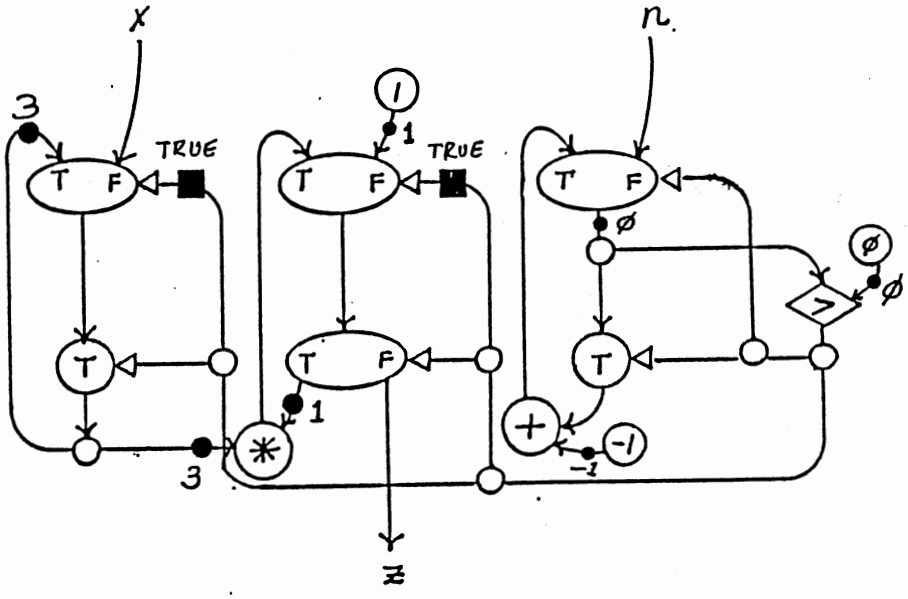


Figure 59.j. Tokens at time t_9

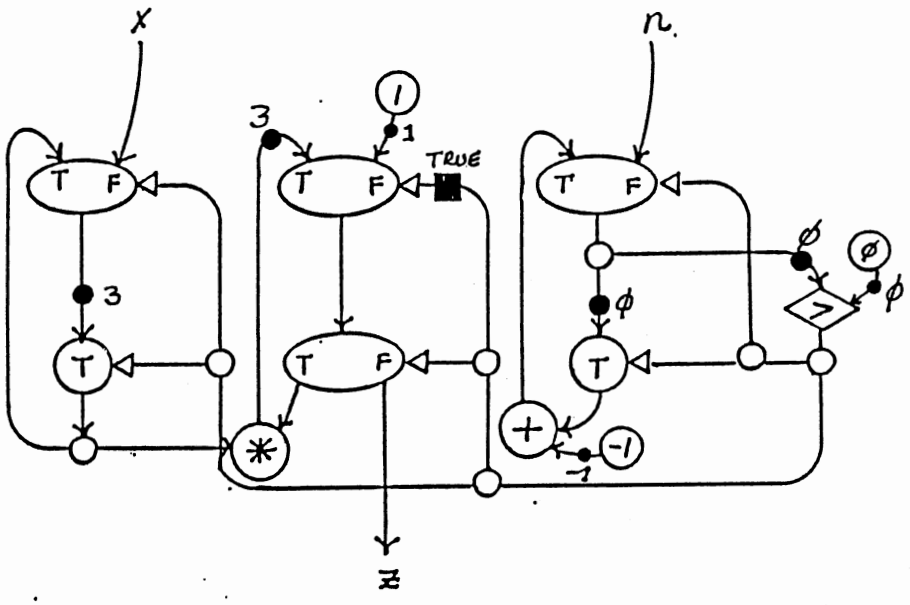


Figure 59.k. Tokens at time t_{10}

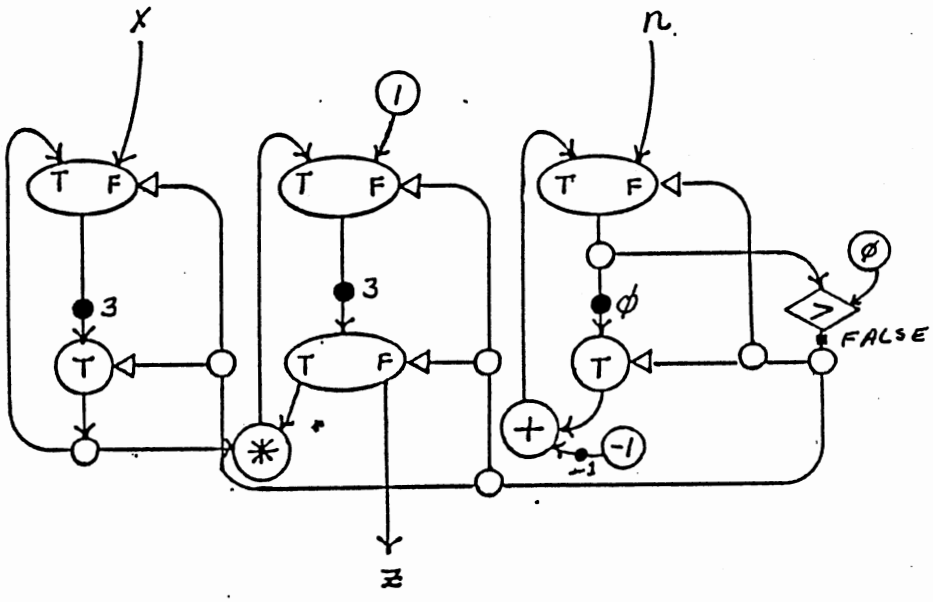


Figure 59.l. Tokens at time t_{11}

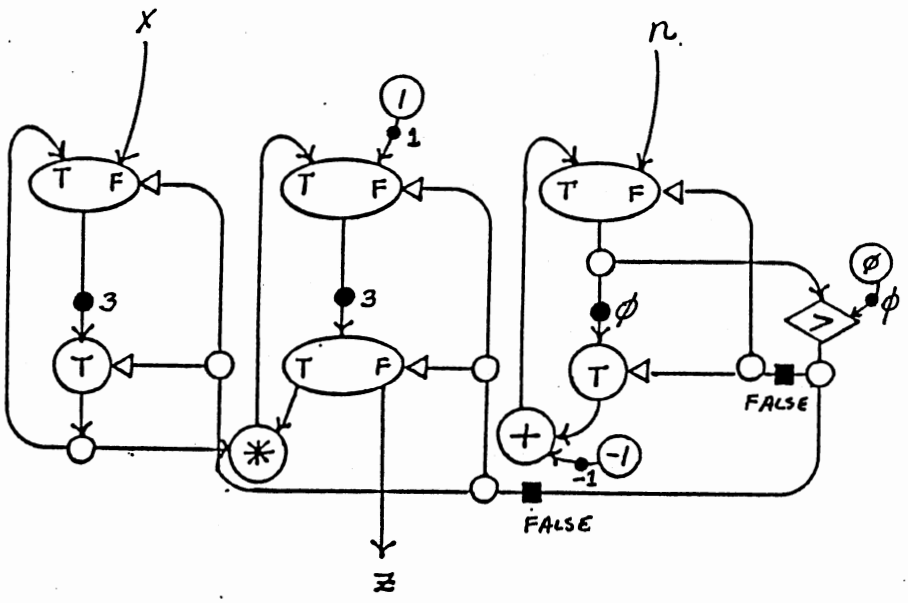


Figure 59.m. Tokens at time t_{12}

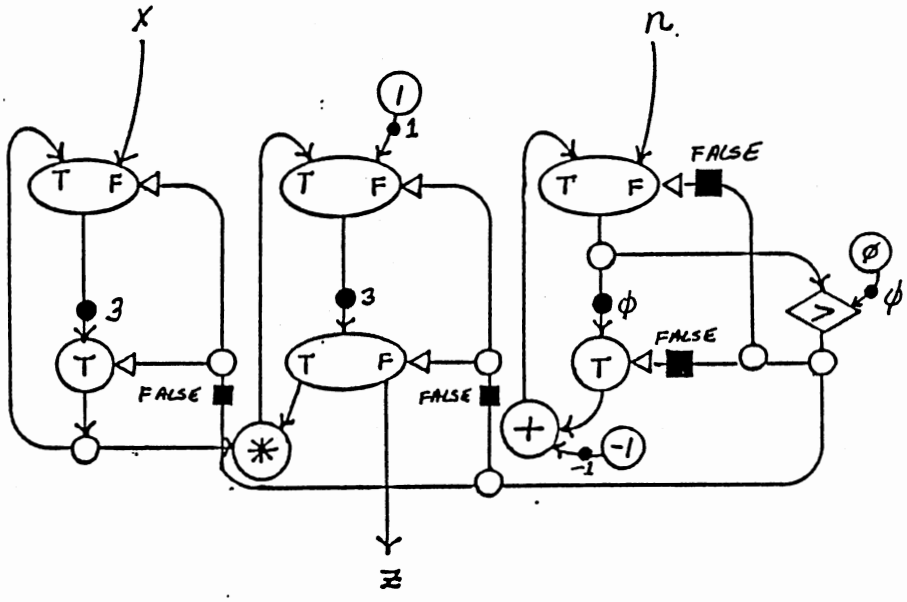


Figure 59.n. Tokens at time t_{13}

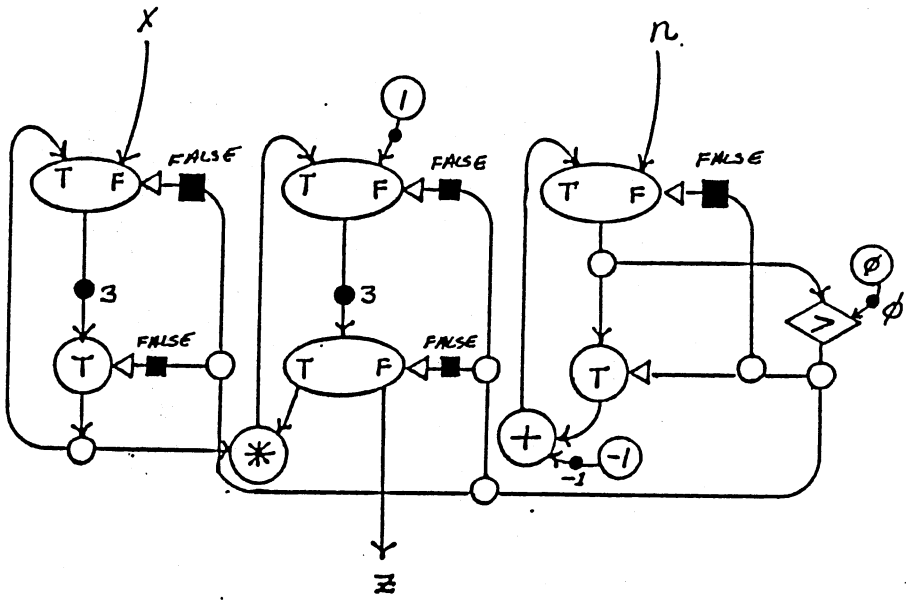


Figure 59.o. Tokens at time t_{14}

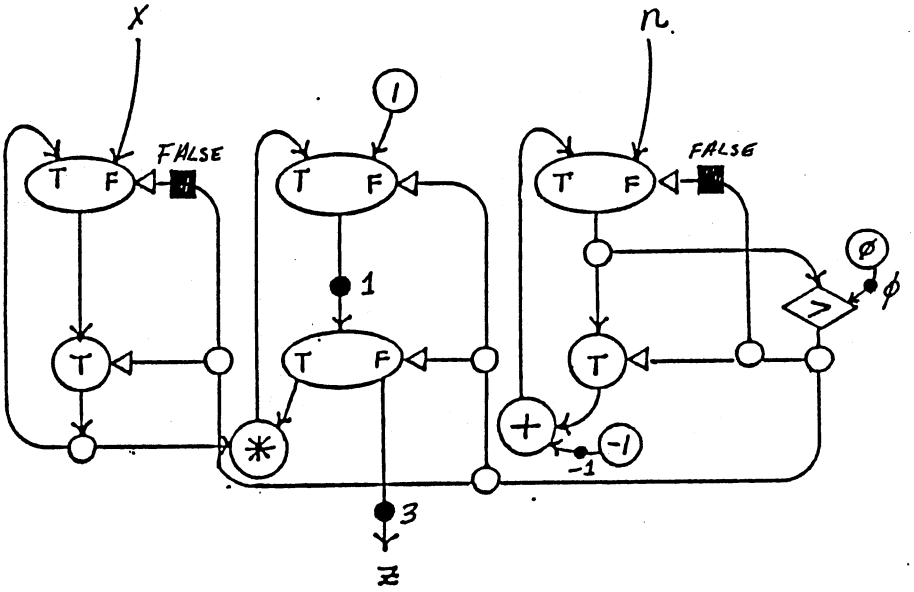


Figure 59.p. Tokens at time t_{15}

6.4 Recursion, Tagging, and Maintaining Temporal Concurrency in the Iterative Data Flow Graph

In an iterative computation, such as that in the example of Figure 59, very little pipelining can take place. In the example, use of the MERGE functions at the source input arcs limits access to the rest of the graph. The control tokens to the MERGE functions are always true until the iterative computation is complete, precluding entry of further values into the pipe. The horizontal concurrency is maintained but the temporal concurrency is lost.

Two different solutions to this problem have been established. One approach to "unfolding" iterations, or allowing distinct evaluations to take place as separate data sets pipe through a cyclic graph, is to apply the dynamic data flow approach discussed earlier. Two different groups of researchers, Arvind and Gostelow in their development of the U-interpreter [4] and a group at the University of Manchester in England [36], arrived at the idea of explicitly labeling computational activities for parallel execution. Tokens are assigned activation names as tags, or labels, upon each occurrence of reentry to a graph. Token activation names must match at a particular node in order for the node to be enabled for a token set. This allows concurrent executions of the same

procedure to share one version of its data flow graph, or instruction code. Additionally, tokens can be tagged with their iteration level. Iteration level tags indicate the tokens specific sequence step through the loop. Tokens must then also match according to their iteration level in order to enable their operation to receive them. This allows each iteration of the loop to proceed at its own speed, several different iteration values can be active within a loop at one time. These techniques allow temporal concurrency to be maintained [4, 36].

A second approach is one used in static environments. In order to maintain temporal concurrency, looping is eliminated in favor of a form of recursion. This method is based upon the observation that any iterative procedure can be expressed recursively. Under this strategy, each iterate subgraph is encapsulated into a macrofunction. This macrofunction replaces the iterate subgraph within the overall whole. When a token reaches the macrofunction during execution time, the subgraph is spliced into the whole. Since the macrofunction can be generated at run-time for different generations of input tokens, the graph permits pipelined concurrency [19]. Consider the example of Figure 59. It could be represented recursively by the graph in Figure 60. If x = 3 and n = 2, the run time code generated for those parameters would appear as in Figure 61. This method has the obvious drawback of utilizing

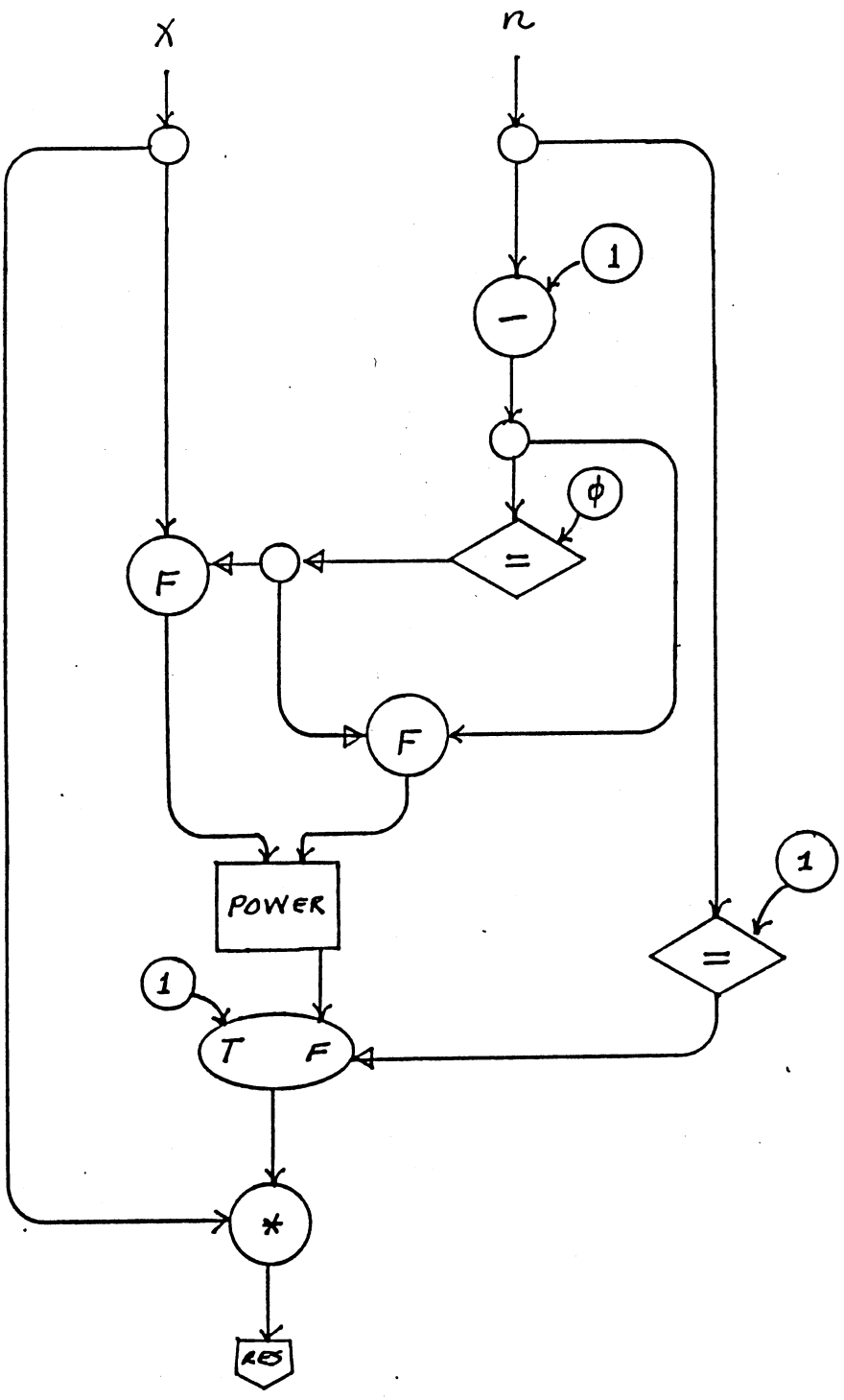


Figure 60. Recursive Graph for POWER Macro Function. POWER Computes $z = x^n$

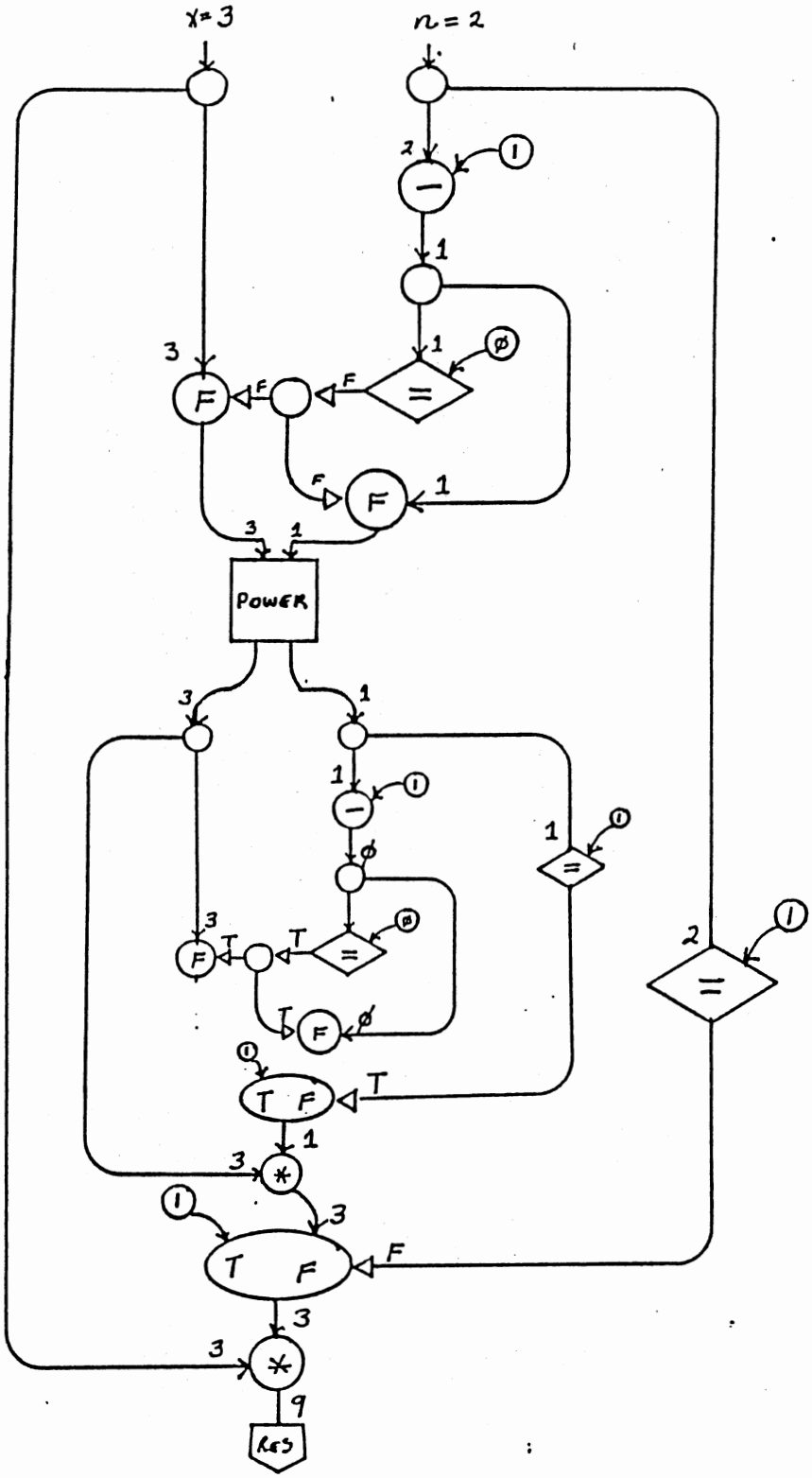


Figure 61. POWER Expansion Resulting from Input Values x = 3 and n = 2

a great deal of computer memory when the depth of recursion is large.

True recursion, using reentrant code, can be implemented directly in the dynamic tagged environment. This is done by attaching the activation name within the context of concurrent executions of the same procedure. In the recursive context, a new activation name is attached to input tokens on each successive invocation of the recursive function. Again, tokens are matched according to their activation name prior to instruction enablement. This allows for reentry to the one version of the graph for recursive execution [36].

6.5 Data Structures in the Data Flow Environment

A major issue of discussion among data flow researchers and data flow detractors has been that of how to handle data structures such as vectors, matrices, trees, and linked lists. If tokens are allowed to carry structures such as these, the result is a large data transmission and storage overhead. Furthermore, the size of the object might not be known until it arrives at a given node where it should be stored. Frequently, within an aggregate data structure, only one or a few elements from the structure are altered or used, yet the entire structure would require copying from one node to another.

Two proposed solutions are the following:

1) Arvind and Thomas proposed I-structure storage [6], and

2) Dennis proposed use of finite directed acyclic graphs to represent structures in memory [33].

The goal inherent in these approaches is to preserve the requirements of the data flow environment, the maintenance of the by value parameter passing mechanism and enablement only when all tokens are present, as well as to circumvent structured token recopying from one instruction to the next.

6.5.1 I-Structure Storage

The I-structure storage concept of Arvind and Thomas [5, 6] is designed to prevent read before write races. Within the memory hardware are presence bits associated with each memory word. Their function is analogous to that of the semaphores used to synchronize concurrent processes. These presence bits are very similar to the status bit used in the Denelcor HEP multiprocessor to coordinate cooperating concurrent processes.

In the data flow environment, the presence bits are used to coordinate access of producer/consumer instructions to a single copy of a structure. In the I-structure storage, the presence bits have three implied values. 'A' implies absent or not written. 'P' implies present or value written. 'W' implies waiting; a read

request to this location has been made but not yet satisfied and the read is waiting in a list of deferred reads. The presence bits are tested by the memory controller when a read request for the contents of a given word arrives. If the presence bits indicate that the word has been written, the contents are retrieved and forwarded to the requesting instruction. If a read request arrives and if its word's presence bits indicate 'absent' or 'waiting,' then the read is deferred until the data arrives; the read is linked with other reads awaiting the same datum on a deferred read list. Further, when such a read is placed on the deferred read list the presence bits are set/reset to 'waiting.' When a write request arrives for a given word, the presence bits for the word are tested. If the presence bits are 'absent,' then the data is written and the presence bits are set to 'present.' Or, if they are 'waiting,' then the memory module writes the data to the word, forwards the data to all the reads linked on the deferred list for that location, and sets the presents bits to 'present.' To avoid excessive data transmission of whole structures in a token, the I-structure storage can be used to hold the structures while the token carries the address of the structure. Using such storage, a structure's storage can be allocated, all its words' presence bits set to 'absent', and its token address started down the data flow graph (program). If only a certain element of the structure is to be read and

written, then the token would also carry such indicators as was needed to identify the specific element, for example, an index to a vector. An instruction node on whose input arc the token arrives, reads the element indicated by the token from the structure in the I-structure storage. If the token exits the instruction node on an output arc, the instruction writes to the structure.

Unfortunately, checking for deferred reads on every write degrades the write process. Also, it appears to be a move back in the von Neumann direction since tokens are now carrying addresses rather than values. Still, the data flow rule is maintained since no read can proceed and no node can fire until the data item is written to its proper location.

6.5.2 Finite Directed Acyclic Graphs

Jack Dennis is credited with describing a technique for implementing data structures using finite directed acyclic graphs [33]. Arrays for example, are stored as trees; each individual array element is stored as a leaf. A three by three array is represented by a ternary, or three-ary, tree as shown in Figure 62. This tree is maintained in a structure storage memory.

Similar to the I-structure storage discussed earlier, instead of a token carrying an actual structure, it carries an address. In this case, the token carries the

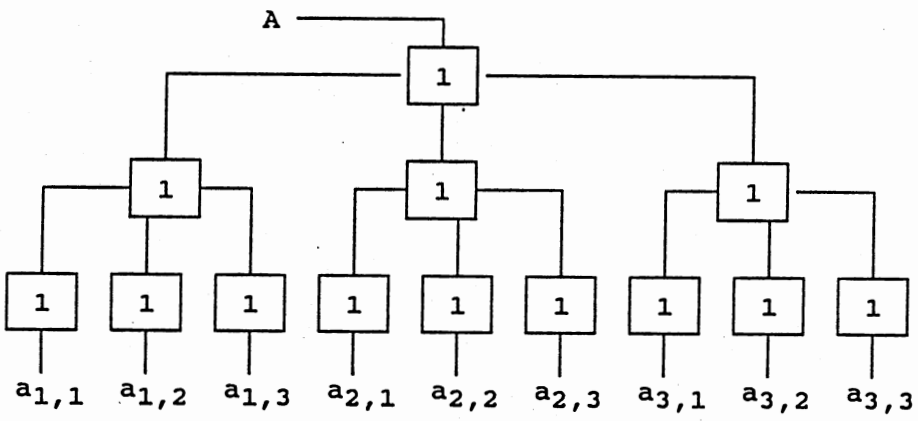


Figure 62. Storage Scheme for a Three by Three array Identified by Token A. The $a_{i,j}$ Represent the Element Values

address of the tree's root node along with whatever information is needed to identify individual leaves to be read or written. Each node in the tree contains a reference count indicating the number of directed edges arriving at the node.

The data flow graph of Figure 63 indicates a series of actions to be performed on a three by three array. A DUPlicate operation [Figure 58] on the array results in the root node being referenced by two separate tokens, A and B, one for each output token on the data flow graph. The reference count for the root is then two [Figure 64]. This circumvents the need for duplicating the data but yields separate tokens.

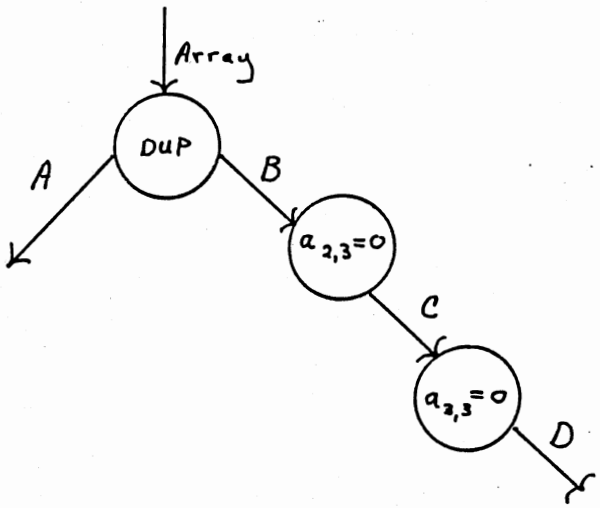


Figure 63. A Data Flow Graph that Duplicates an Array and Serially Assigns New Values to Two of its Elements

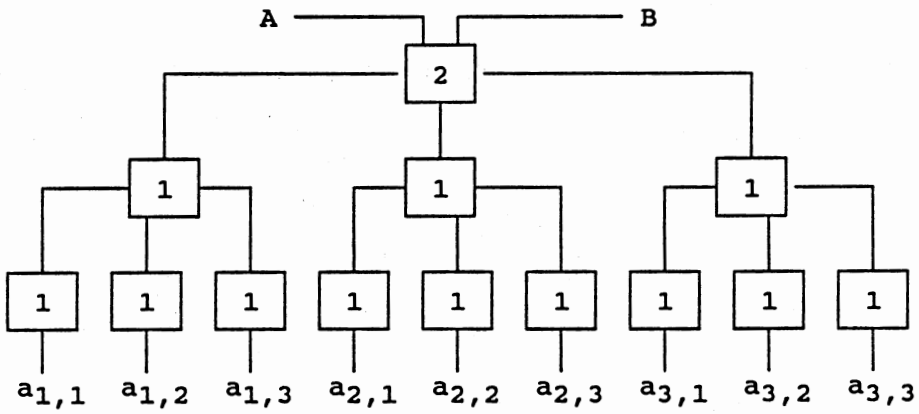


Figure 64. B is a Duplication of Token A. Root Reference Count, the Number of incoming edges, is Incremented, but no Nodes are Copied

Setting the element in row two and column three to zero on input token B results in the generation of the tree pointed to by output token C [Figure 65]. Setting the element in row three and column three to zero on input token C would results in the generation of the tree pointed to by token D [Figure 66]. This is sequential processing of the two elements in the array.

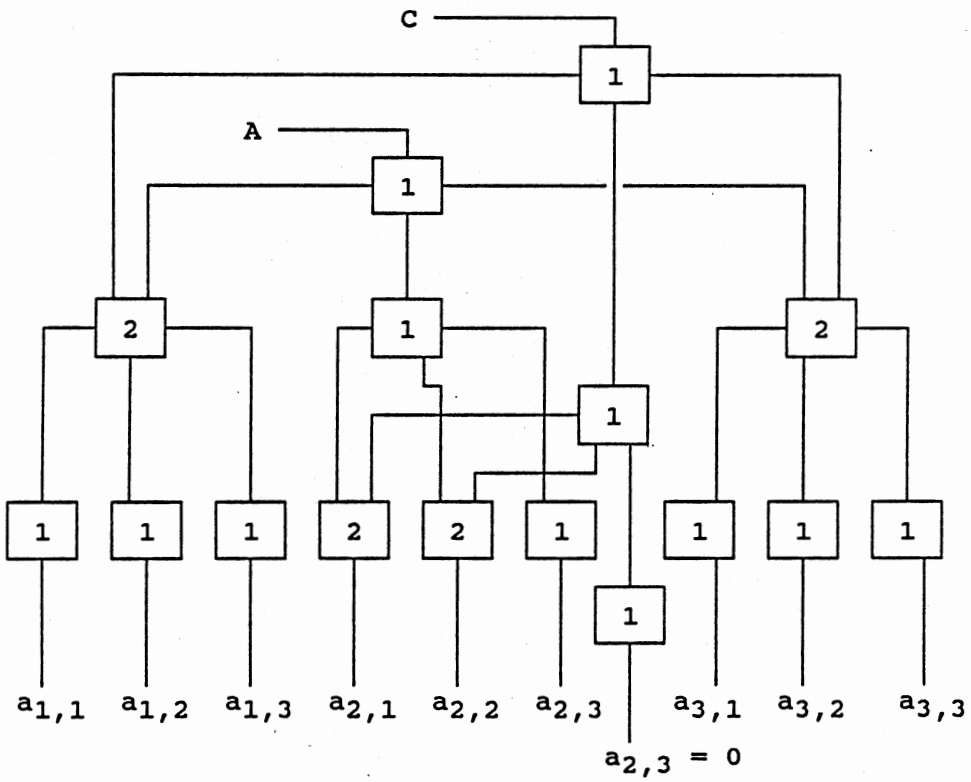


Figure 65. Generation of New Token C from B by Setting the Element in Row Two and Column Three to Zero. Token B is Consumed

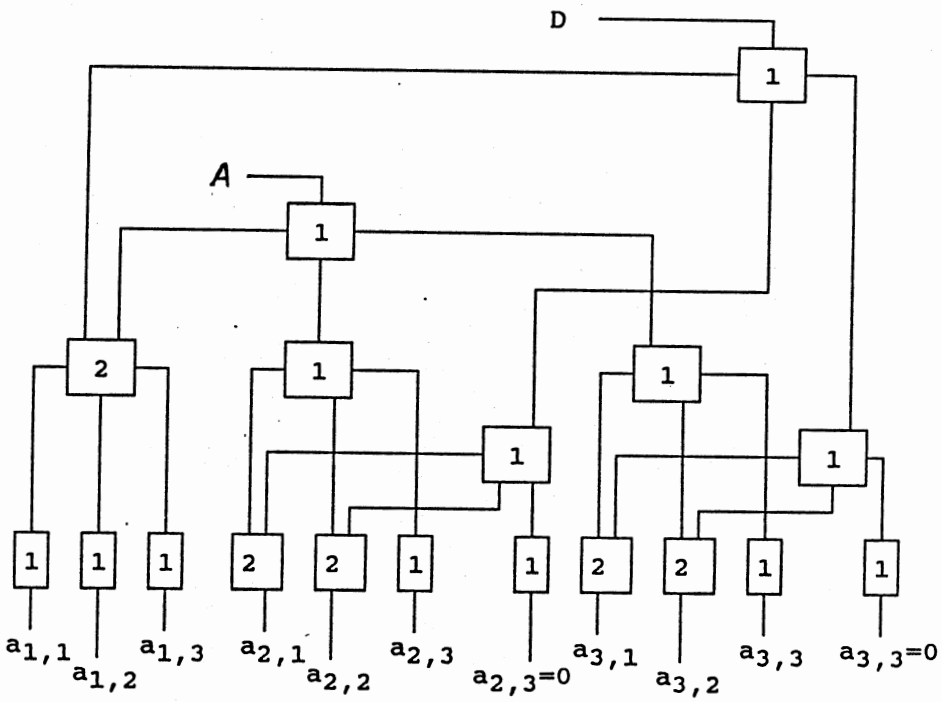


Figure 66. Generation of New Token D from C by Setting the Element in Row Three Column Three to Zero. Token C is Consumed

Concurrent execution on the array elements is shown in the data flow graph of Figure 67. Setting both the element in row two and column three and the element in row three and column three to zero concurrently results in output tokens C and D [Figure 68]. The final result of concurrent execution on the array elements would be quite different from that of sequential execution. Thus such activities as setting the elements of a column to zero would require sequential execution.

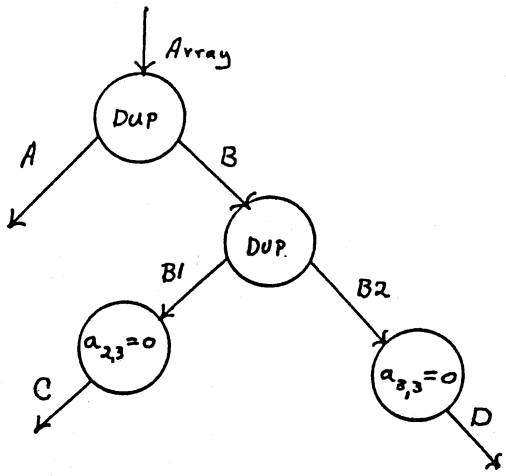


Figure 67. A Data Flow Graph that Duplicates an Array and Concurrently Assigns New Values to Two of its Elements

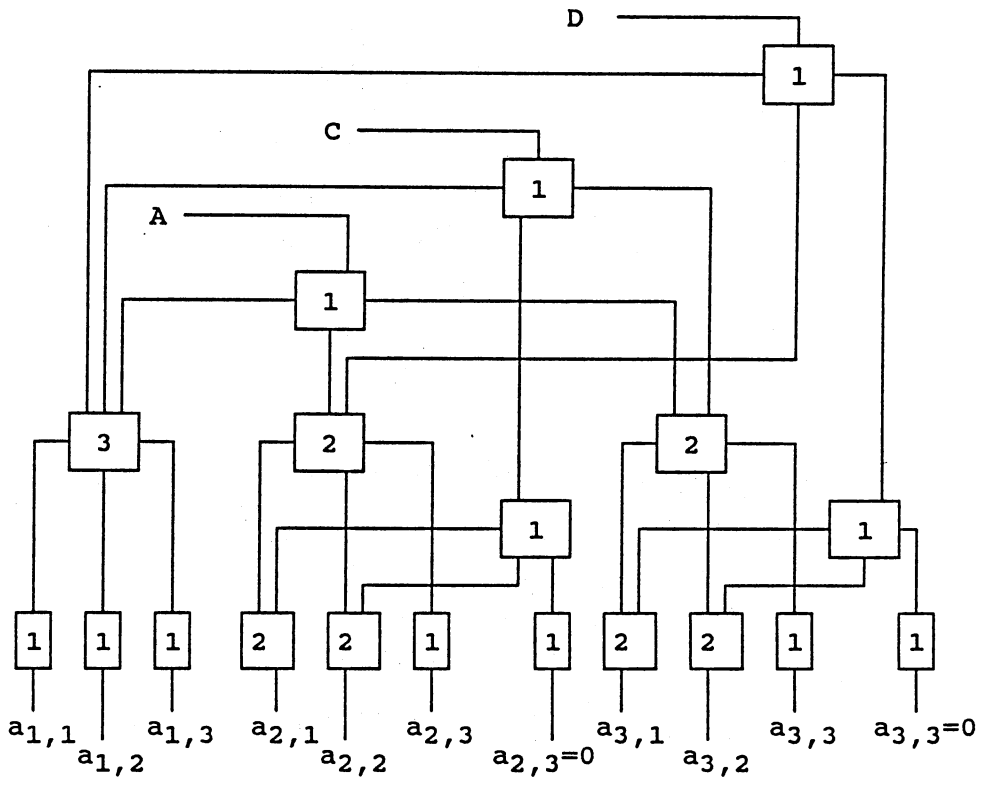


Figure 68. Concurrent Generation of New Tokens C and D by Concurrently Setting the Element in Row Two and Column Three to Zero and the Element in Row Three and Column Three to Zero

Furthermore, the updating of reference counts and the depth of the tree increase the memory references required to access elements in the represented structure. Clearly, this lowers the performance level of the machine.

Research on the problem associated with data structures in a data flow environment continues among data flow researchers.

6.6 Implementations of the Data Flow Graph, the Data Flow Computer

Data flow computers are computers whose architecture allows them to execute the abstract graphical model of the data flow graph. The data flow graph is the method used to present data flow programs. The nodes or activity templates represent machine instructions. The tokens represent the values processed by the machine.

Many computer systems which are designed to minimize execution time by exploiting data-driven parallelism exist. They include the Dennis machine and the Arvind machine, both at the Massachusetts Institute of Technology, the Distributed Data Processor designed by the Texas Instruments Company, the Data-Driven machine at the University of Utah, the LAU machine at the CERT Laboratory in Toulouse, France, the Newcastle Data-Control Flow Computer at the University of Newcastle upon Tyne, England, the EDDY (Experimental system for Data Driven processor arraY) machine of Japan, and the Manchester

machine at the University of Manchester, England [85, 41, p. 748-768]. Each of these machines has its own distinctive elements. However, to examine each machine is beyond the scope of this treatise. Instead, two machines generally representative of their basic types are examined. One is the Dennis machine of Massachusetts Institute of Technology, as described in Dennis's papers [23, 24]; it is designed to execute a static data flow graph. The other is the Manchester machine of the University of Manchester, as described in the papers of Gurd, Kirkham, and Watson [36, 94]; it is implemented for execution of dynamic graphs.

6.6.1 The Dennis Static Data Flow Machine

The Dennis machine of Massachusetts Institute of Technology is designed to exploit the parallelism represented by static data flow graphs. It has the organization displayed in Figure 69. It consists of five major units connected by channels through which information packets are passed according to an asynchronous transmission protocol. The five units are the following:

1) the Memory Section, partitioned into addressable Instruction cells. Instruction cells hold individual instructions and their operands.

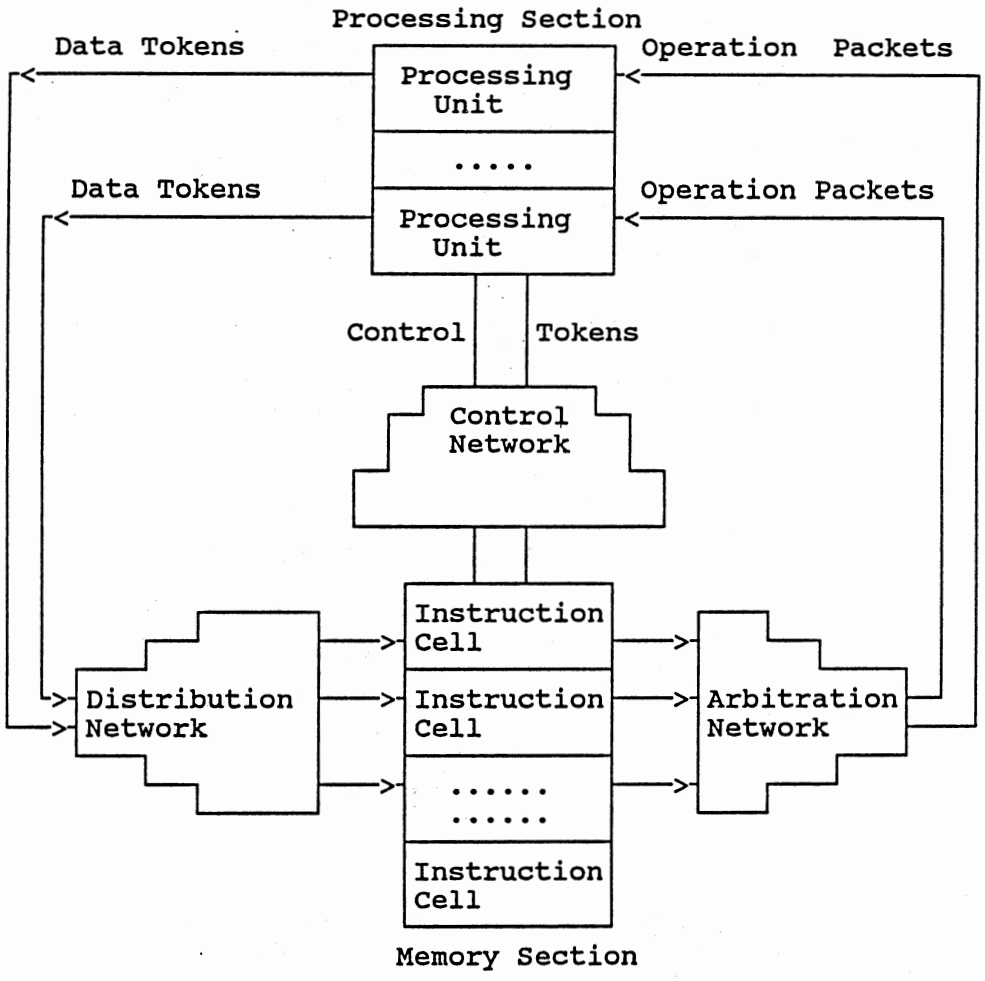


Figure 69. Machine Organization for the Dennis Static Data Flow Machine [41, p. 749, 23, 24]

2) the Processing Section, consisting of processing units which perform specialized functional operations on data tokens.

3) the Arbitration Network, routing executable instruction packets from the Memory Section to the Processing Section.

4) the Control Network, routing control tokens from the Processing Section to the Memory Section.

5) the Distribution Network, routing data tokens from the Processing Section to the Memory Section.

Instructions are held in the uniquely addressable Instruction cells (representing the activity templates) of the Memory Section. When loaded, each instruction cell holds an instruction operation code of the data flow program/graph. The Instruction Cell maintains several locations for holding result destination addresses; these implement the output arcs of the data flow graph. Additionally, the Instruction Cell contains three registers which will hold the operand values received as data tokens over the Distribution Network. When all the operands required by the operation code have arrived in the instruction cell and the appropriate control/acknowledge signals have arrived from the Control Network, the instruction represented in the cell is said to be enabled.

Upon enablement, the operation code, destination addresses, and operands are grouped together logically in

operation packets and routed through the Arbitration network to the Processing Section. As the operation packets are routed through the Arbitration network, the opcode is decoded partially. This process allows the packets to arrive at the proper functional unit for execution.

Processing results are paired with the destination addresses specified in the processed operation packet, and sent through the Distribution and/or Control networks to the Memory Section Instruction cells. The results are stored in the Instruction cells whose addresses were specified as destinations. These results may be of two possible types,

1) Acknowledge signals and boolean values generated by operations such as DECIDERs.

2) Integer or other data values. These are Data tokens and are routed over the Distribution network.

Acknowledge signals are directed back to the instruction cell that produced the result that was just consumed by the currently executed instruction. Acknowledge signals indicate that a node has utilized the token and is ready for another. The acknowledge signals are used to implement the firing rule for program graphs. They are Control tokens and are routed over the Control network.

When all the result packets, data and control, required by a receiving instruction cell have arrived, it

becomes enabled and begins its passage through the Arbitration network. The requirement that acknowledge signals arrive before an instruction is enabled maintains the static data flow firing rule: an instruction may fire when all its operands are available and there is no token on its output arc.

In order to maintain equal accessability of instruction cells, and to minimize the number of devices and interconnections required to connect the great number of instruction cells found in such a system, the above described architecture was refined slightly. The Instruction cells are grouped into blocks and each block realized as a single device. Each instruction cell block is accessed via a single input port and single output port. The resulting structure is shown in Figure 70. Under this arrangement, cell blocks are grouped together. A given cell block group is serviced by an arbitration network which transmits operation packets to a specific set of functional units. This allows simplification of the arbitration network. Further, each cell block is addressable through the distribution network; and the distribution network has fewer ports to contact.

The mechanism of the cell block itself is as shown in Figure 71. The grouped instruction cells compose the activity store. Result packets arrive over the distribution network at the update unit. The update unit writes the operand tokens into the instruction cell

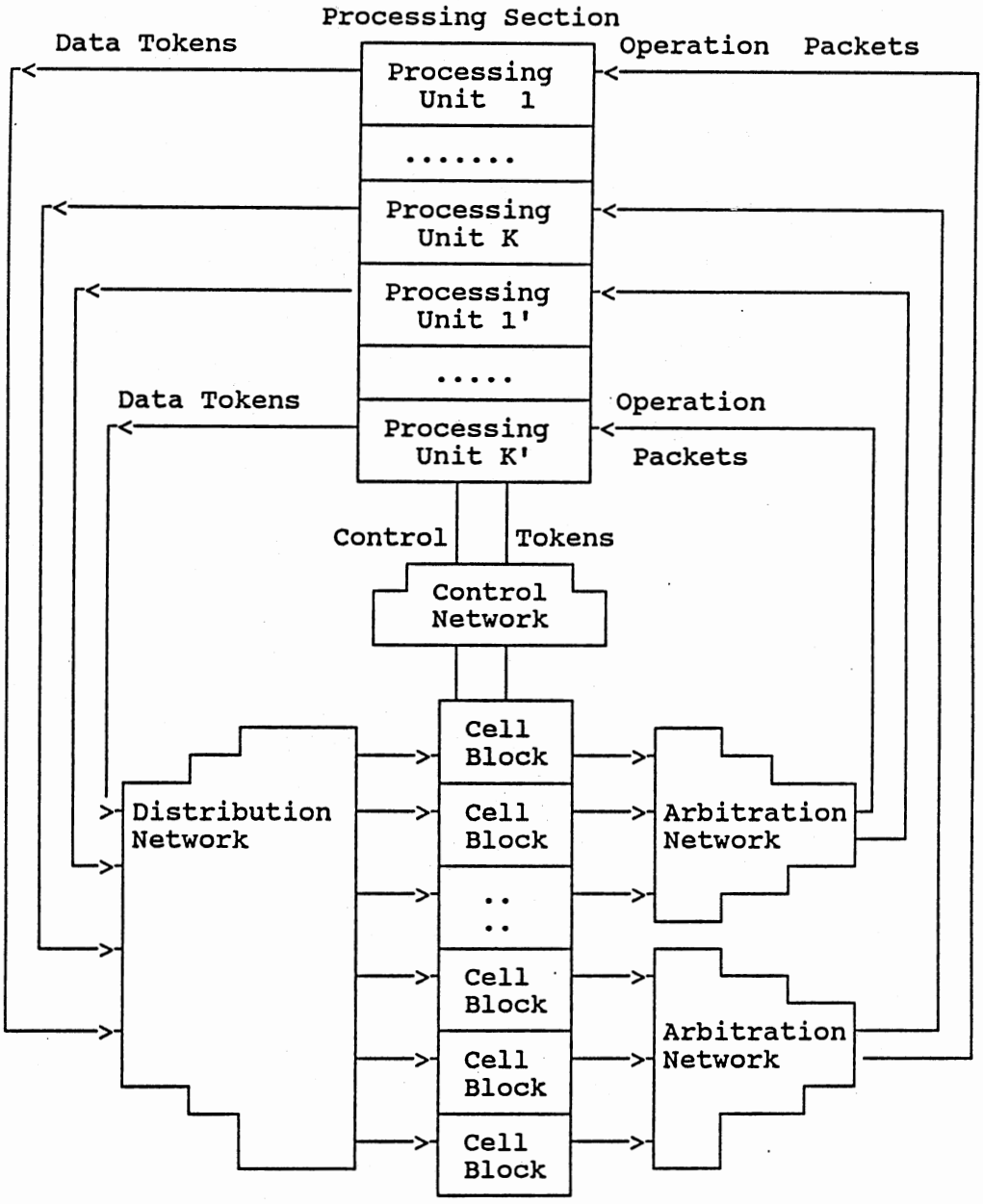


Figure 70. Cell Block Architecture. Each Cell Block Has its Own Input and Output Port. Distribution Network Can Access Each Cell Block [23]

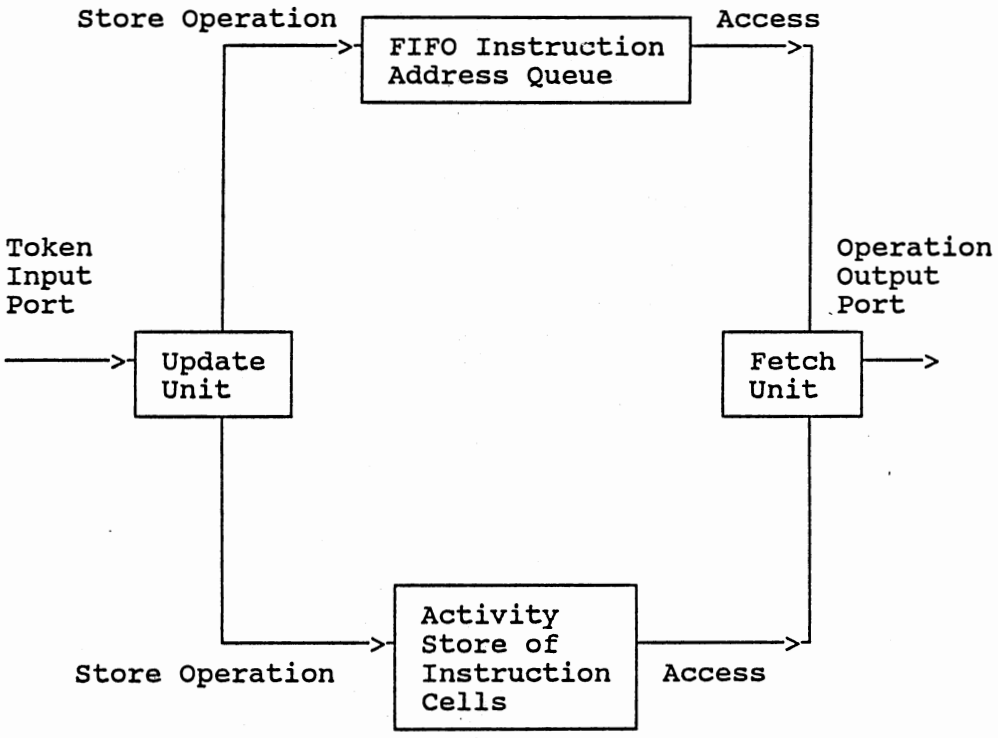


Figure 71. Cell Block Implementation of Dennis Machine [23]

registers and tests whether all control and data tokens have arrived at the instruction cell currently being updated. If they have, the update unit enters the address of the instruction into the FIFO instruction address queue unit. Meanwhile, executing asynchronously, the fetch unit removes an address from the FIFO instruction queue and reads the corresponding instruction cell from the activity store. The fetch unit forms it into an instruction packet and puts it out onto the arbitration network where it is routed to the appropriate processor as before.

Because of the way the cell blocks are accessed from the processing units through the distribution network, communication of a result packet from any instruction cell to another requires the same amount of time. During program execution the number of instructions addressed in the instruction address queues of the cell blocks gives a measure of the degree of concurrency present in the program. The concurrent activities possible are built in at the hardware level [23, 24, 85].

6.6.2 The Manchester Machine

The Manchester machine of the University of Manchester, England, is designed to execute a tagged token dynamic data flow graph. The block diagram of the prototype Manchester system is shown in Figure 72.

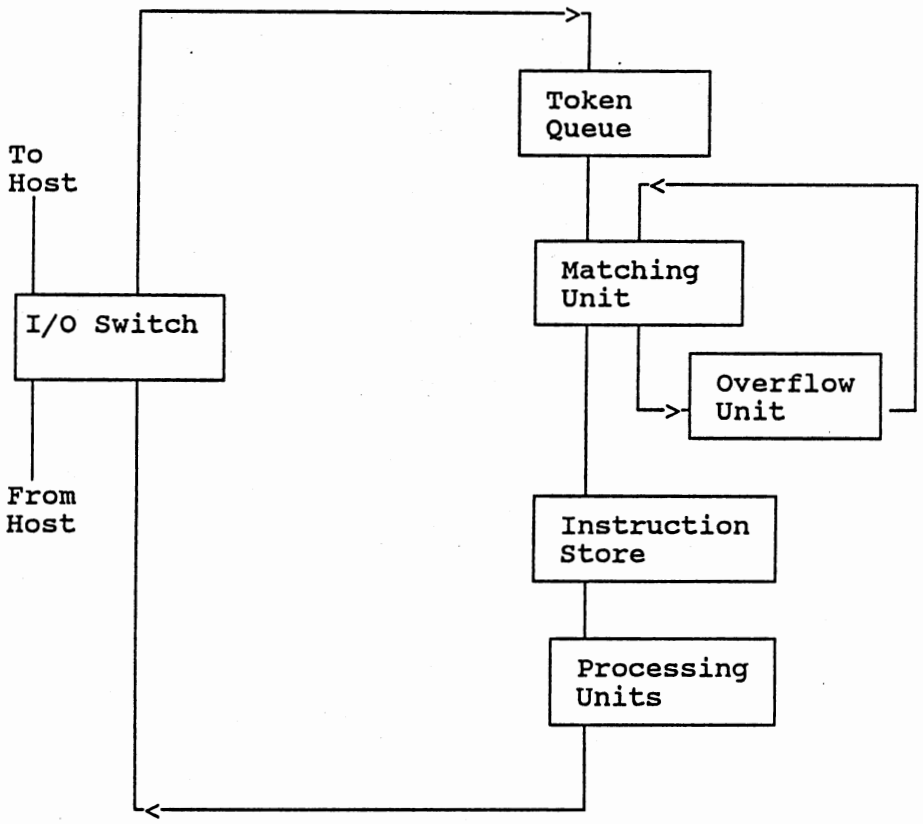


Figure 72. Manchester Data Flow System Organization, Based on Tagged Tokens and Dynamic Graphs [36]

A host system is attached via an I/O switch module to the basic ring structure of four modules. The modules are the following [94, 85, 36]:

1) the Token Queue, consisting of a 32K-word circular FIFO store with three surrounding buffer registers. Tagged tokens on the output arcs of the executing data flow graph are queued here to await further processing.

2) the Matching Unit is a pseudoassociative memory with 6 pipelined registers and two buffers interfacing it with the Overflow Unit. Tokens whose destination instructions are unary operations pass directly through the Matching Unit. Otherwise, tokens are stored in a parallel hash table (the pseudoassociative memory) until another token arrives with a matching "name". The "name" used for pairing tokens is a combination of the tokens' tag and their destination instruction address.

2.a) the Overflow Unit handles tokens that cannot be loaded into the parallel hash table because all table entries are full. Overflow tokens are stored in the Unit as linked lists. When space is available in the hash table, overflow tokens are bussed back to the Matching Unit and restarted through it. The asynchronous nature of the data flow model allows tokens to be matched in any order without effecting the computation. Token pairs matched on their "name" are passed out of the Matching Unit to the Instruction Store.

3) the Instruction Store consists of a Random Access Memory and an input and an output register. The instructions identified by the destination fields of the tokens are selected from the RAM and coalesced with the tokens into an operation packet containing an opcode, data values, operation result destination fields for the instruction now enabled, tags and a marker bit. All instructions now enabled by the presence of all their tokens are routed to the Processing Unit.

4) the Processing Unit contains a preprocessor which executes a few instructions, but most are passed on to one of several homogeneous microcoded function units via a distribution bus. The instruction packet is processed in its assigned function unit. An output token is produced from the execution, composed of tag, operand value, instruction destination addresses, and marker bit. The token is then passed out of the Processing Unit to the I/O Switch.

At the I/O switch, the marker bit is decoded to determine if the token should be routed out of the data flow system to the host machine or passed back around to the Token Queue to initiate further computations.

By use of its tagging mechanism, the Manchester machine is capable of concurrent executions of reentrant programs; thus, recursion and pipelined iterative loops are allowed. The machine is operational, running

reasonably large programs at maximum rates of between 1 and 2 million instructions per second (MIPS) [36].

6.7 Data Flow Languages

Closely related to the subjects of the data flow graph and the data flow computer is that of the data flow language. When problems become complex, direct coding of data flow graphs into a format appropriate for the internal workings of the hardware becomes difficult to say the least. High level languages are needed.

Many data flow languages have been proposed and compilers for a considerable number of them have been written. Many data flow research groups have defined a language for their system. The Dennis group has developed VAL and VIMVAL [25, 58, 77]. The Manchester group has SISAL [36], while the Arvind group has defined ID [62]. The number of these languages is too large for this subject to be dealt with in detail at this time.

However, they display certain common characteristics worth mentioning [2]:

1) Freedom from side effects, based on functional programming. They operate by application of functions on values.

2) Locality of effect. Instructions do not have far reaching data dependencies. Names are limited in their scope.

3) Equivalence of instructional scheduling constraints with data dependencies. All of the information needed to execute a program is contained in its data flow graph, which can be generated directly from the high level language.

4) A single assignment convention. Each name may appear only once within the area of the program in which it is active, or, more stringently, only once within a program. Thus, the definition of each name is clear.

5) Unfolding of iterative computations into parallel constructs. (Related to the discussion earlier on "unfolding" iterative loops.)

6) A lack of "history sensitivity" in procedural calls. Names of values are manipulated so that each function begins execution with new values and is not influenced by past values.

Most data flow languages are functional languages, as identified in item 1 above. Functional languages are discussed in greater depth in section 7.1.

Of course, there are exceptions. The Texas Instruments Distributed Data Processor is an interesting case in point. It has been operational since 1978. This computer is programmed largely in extended FORTRAN 66. A cross compiler, based on the optimizing FORTRAN compiler of the highly pipelined Texas Instruments Advanced Scientific Computer, separately translates FORTRAN subprograms into directed graph representations. The

directed graphs are then processed by a linkage editor into an executable program [85].

6.8 Summary

Computer architects are searching constantly for new approaches to designing high-performance computing machines. Data flow offers a totally different approach to computing than that of the von Neumann architecture. It promises to be an exceptional mode for exploiting the fine grain parallelism embedded in most programs. It also offers an opportunity to realize the enormous potential of VLSI technology.

This chapter introduces the data flow graph, and its firing rules. It identifies the two possible tactics for firing and program graph interpretation, namely, static and dynamic rules. Looping and recursion are discussed in the light of these two strategies.

The problems associated with data structures in the data flow environment are identified and two possible alternative solutions are presented, I-structures and directed acyclic graphs.

Two data flow computers are reviewed. The Dennis Data Flow Machine is presented as an example of an architecture implementing static data flow graph firing

rules. The Manchester Data Flow Machine is surveyed as an example of a tagged token, dynamic graph firing rule implementation.

The common qualities associated with most data flow languages are identified.

CHAPTER VII

REDUCTION MACHINES

7.0 Introduction to Reduction

Chapter six investigates a computer architecture that is non-von Neumann in nature; that system model is termed a data flow system. In a data flow system, the control of program execution is based on the availability of the data; when a function or operator has all its required arguments, it will be evaluated. Thus, the system is said to be data driven. This chapter presents another non-von Neumann computer architecture, reduction machines. A large amount of the work done on reduction machines has been based on the work of data flow researchers [18, 17]. However, reduction machines have a different form of program control. In reduction machines, functions are evaluated or reduced when their result is needed, or demanded, for the evaluation of some other required function. Thus, these machines are often said to be demand driven.

In a data flow system, some computations may be performed simply because their operands have arrived although their results will never be needed. This allows the processors to do non-productive work that in some

situations can saturate the system and prevent productive evaluations from taking place. A good compiler can reduce the number of these non-productive operations, but the potential for non-productive activities is present in the data flow architecture. The idea behind a demand driven system is to allow only the evaluation of those functions whose value is demanded or needed for the completion of the assigned task.

In order to understand the notion of a reduction machine, one needs to first understand a little bit about functional languages. This is because reduction machines are closely linked with such languages. In many cases, machines have been expressly designed for the execution of programs written in some given functional language [92]. The sections of this chapter introduce some of the primary aspects of functional languages and the concepts inherent in a reduction system. A specific implementation of a reduction machine, ALICE, or the Applicative Language Idealized Computing Engine, is reviewed. The ALICE machine is the product of a group headed by John Darlington, in close association with Mike Reeve, working at Imperial College of Science and Technology, London, England [65, 18, 17].

Compilers that compile programs written in Prolog, Parlog (parallel Prolog), LISP, and HOPE have been written for ALICE. The functional language HOPE is intended to be the primary language for use on ALICE. HOPE was designed

at Edinburgh University, England, by Burstall, McQueen, and Sannella [15, 65]. It is an experimental language as not all required production features have been incorporated. In this discussion of functional languages and related reduction concepts, some HOPE programs are used as examples.

7.1 An Introduction to Functional Languages

This section defines functional languages and examines an example of the functional language HOPE. The program flow of control implied by such a language is discussed.

7.1.1 Procedural Languages and Contrasting them to Functional Languages

Current computer languages fall into several general classifications based on the way in which they allow the programmer to communicate with the machine.

The "old" languages such as FORTRAN and COBOL and the newer ones such as Pascal and Ada are called procedural languages. In a procedural language, the programmer is allowed to specify a set of imperative statements that are to be performed in a particular sequence. The procedural language concept is a direct extension, or "high level version," of the von Neumann computer model. One

instruction is executed, then the next instruction is executed, as specified by the program counter. Each instruction addresses operands at locations in memory, and since multiple instructions may access the same locations the order of execution is important. The execution of one instruction alters the environment of the other instructions. This environment may be referred to as the present state of the machine; it includes the program counter, register values, values of all data stored in memory, the run time stack, etc. There are some identifiable disadvantages in thinking of program execution in terms of the present state and its manipulation. Disadvantages identified by functional language proponents are the following [92]:

1) Two widely separated pieces of code may reference a common global variable and thus produce an unexpected result. Also, programmers must be concerned about aliasing, that is, which names are bound to a location. Such issues increase program complexity.

2) The programmer is forced to focus on data manipulation rather than on the crucial elements of the algorithm.

3) Program proof of correctness and program updates are difficult in a procedural language as the imperative style does not lend itself to mathematical analysis. Based on the context in which certain variable names are

used, alteration of code in one area can cause side effects which undermine other program blocks.

4) It is difficult to implement parallel execution of a program when the asynchronous parallel execution of its subroutines have side effects on one another. This last disadvantage is significant in the study of parallel architectures.

Functional languages contain no notion of a present state. The program is a function in the true mathematical sense of the word. The program execution consists of a function evaluation in which the input data is used as arguments to the function; the value returned by the function is the program output. Within the body of the program, additional required values are attained by invoking additional functions. In a functional language, the only activity permitted is the definition, application, and combination of functions. Because of this, a functional language may also be referred to as an applicative language.

The essential notion of a pure functional language is referential transparency; the value of an expression depends only on its immediate textual context, rather than on computational history [92]. Data dependencies exist only as a result of functional application; that is, the value of a function is determined completely by its arguments. More specifically, a strict functional language does not allow the use of variables or assignment

to variables, and the only control structure permitted is recursion. No data is stored, all data is passed as arguments to functions and returned as results from functions [65].

LISP is a well-known example of an applicative language, although it is not always implemented as a pure one. Most modern versions of LISP allow assignment using SET and SETQ statements and iterative loops. The languages VAL and ID identified earlier in the context of data flow languages are functional languages but allow the binding of an expression value to a name; each name may receive only one assigned value. Thus, these data flow languages are termed single-assignment languages. HOPE, the language linked with ALICE, is a functional language. It is strongly typed, which means that it has data types which must be declared by the programmer and is checked by the compiler as in Pascal. It is a pure functional language in that it does not allow assignment and each functional evaluation produces no side effects. And, it is a higher-order functional language which means that functions may be passed as arguments to other functions or they may be returned as results [65, 18, 17].

Functional languages are interesting because they do not have the disadvantages inherent in procedural languages identified earlier. Because they are based on mathematical functions, the programmer may address the problem to be solved at a higher level, with no emphasis

on data manipulation. The problem may be approached in a more logical fashion allowing for proofs of correctness based on the well-understood concepts of the function. Since functional languages do not allow assignment and are free from side effects it is easier to produce and maintain correct code. The absence of side effects makes each part of a functional program independent of every other part implying that the parts can be executed in parallel, in any order, without effecting the final outcome of the evaluation [92].

7.1.2 Hope, an Example of a Functional Language

This section examines an example of the functional language HOPE and the program flow of control implied by the program.

The following HOPE program calculates the factorial of a positive integer.

```
dec Fact : num -> num ;
--- Fact(n) <= Factb(0,n) ;
dec Factb : num x num -> num ;
--- Factb( i,i) <= i ;
--- Factb( i,i+1)) <= i+1 ;
--- Factb( i,j) <=
        Factb( i, (i+j)/2) * Factb( (i+j)/2, j) ;
```

The program consists of two declared functions. The first function is Fact. It maps a value of type num (ie. a non-negative integer) onto another value of type num. The second function is Factb. It maps the cross product of type num values onto a type num value.

One other function is implied; it is the Succ function. The Succ function or successor function returns the next larger value in the sequence of whole integers. The successor function is called a constructor function. Specifically, Succ is used to construct the elements of the data type num. For example, use of the digit 3 is a shorthand for the expression Succ(Succ(Succ((0)))). Each data type has a constructor function for values of that type.

The notation --- marks the definitions, or rewrite rules, of each function. The symbol <= is not an assignment operator. It implies that an occurrence of the function meeting the template form of the definition found on the left hand side of the rule may be rewritten or reduced to the form on the right hand side. The identifiers i,j,n are not variables; they are formal parameters. They refer to the value passed to the function at runtime, and not to any specific memory location.

Based on the given program, evaluation of the function for a given value, Fact(5), can be described by a graph [Figure 73]. In the graph, a function is linked

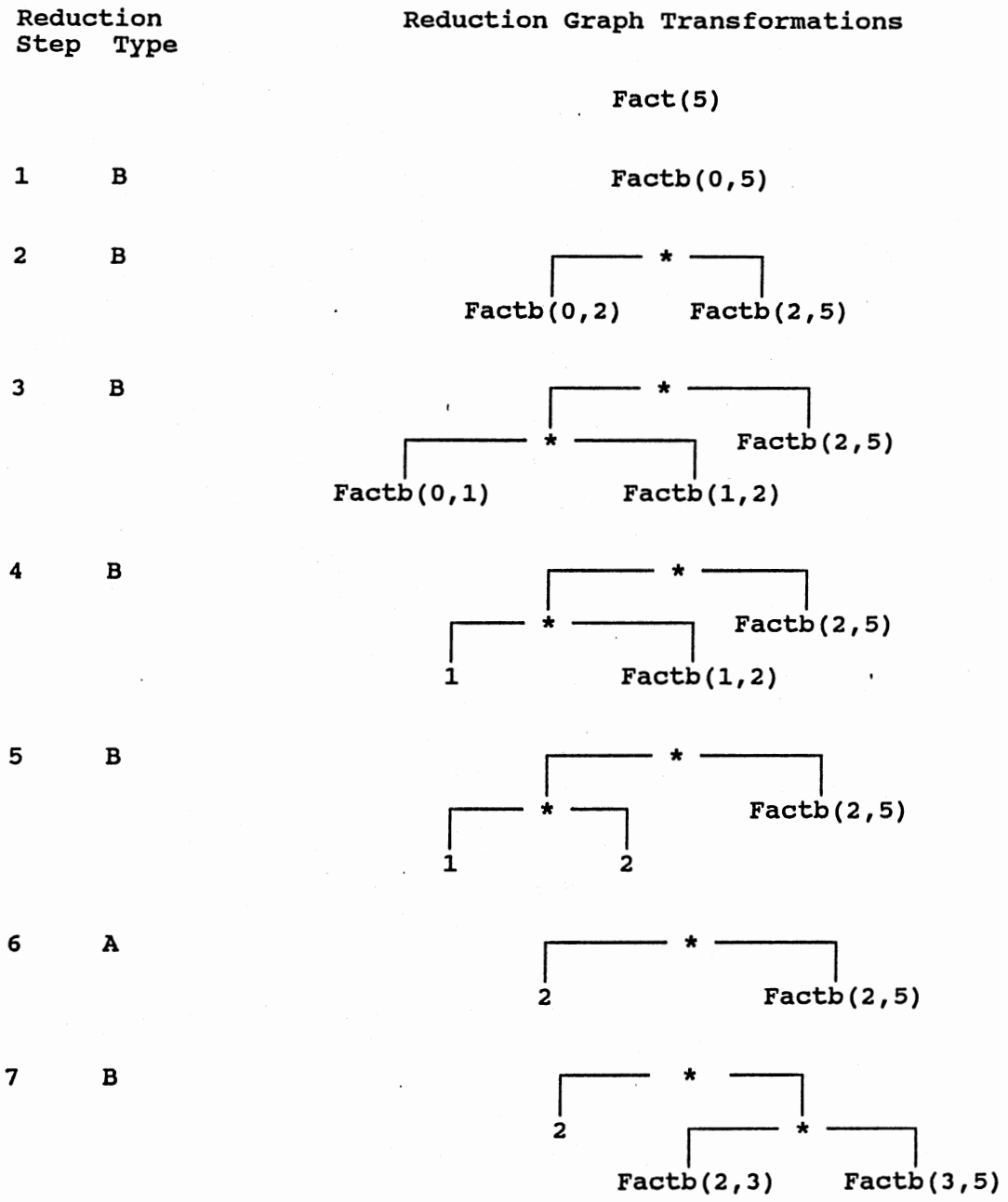


Figure 73.a. Steps One Through Seven in the Sequential Reduction of Fact(5)

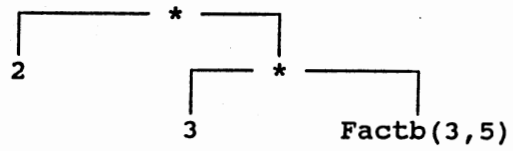
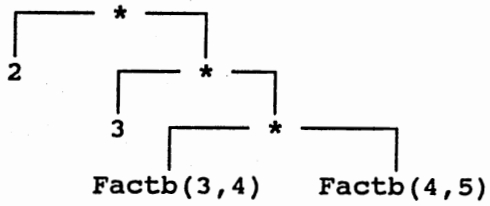
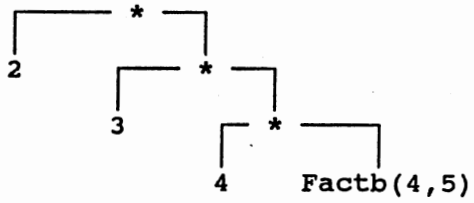
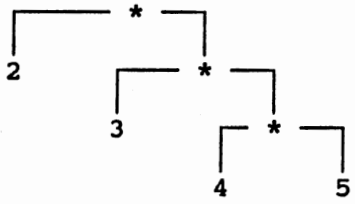
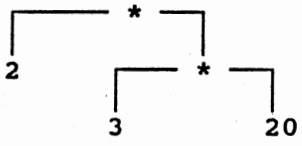
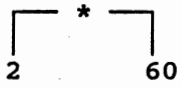
Reduction Step	Type	Reduction Graph Transformations
8	B	
9	B	
10	B	
11	B	
12	A	
13	A	
14	A	120

Figure 73.b. Steps Eight Through Fourteen in the Sequential Reduction of Fact(5)

to its arguments. The graph representing the state of the execution is transformed repeatedly. Each transformation is the result of one of the following two operations:

A) A primitive function such as add, subtract, multiply, or divide has all of its arguments furnished as constructor functions (eg. Succ) that produce constant values. The function with its arguments is replaced by the result of the operation on the constants.

B) One of the rewrite rules is applied to the computation. That is, when a given instance of a function matches the argument form of some left hand side of a definition, it is replaced by an instance of the corresponding right hand side.

Each of the above transformations is called a reduction. When an initial instance of a function is replaced based on one of these transformations it is said to have been reduced. In order for a type B reduction to take place, the function must be one for which rewrite rules exists, rather than a constructor function such as Succ. When functional language programs are interpreted on a von Neumann machine they are reduced one step at a time, sequentially, as indicated in Figure 73. However, since each functional value is independent of another, any function instances ready for reduction at a particular time could be reduced simultaneously, or in overlapped time; functions may be reduced in parallel asynchronously. Figure 74 shows the same function evaluation as indicated

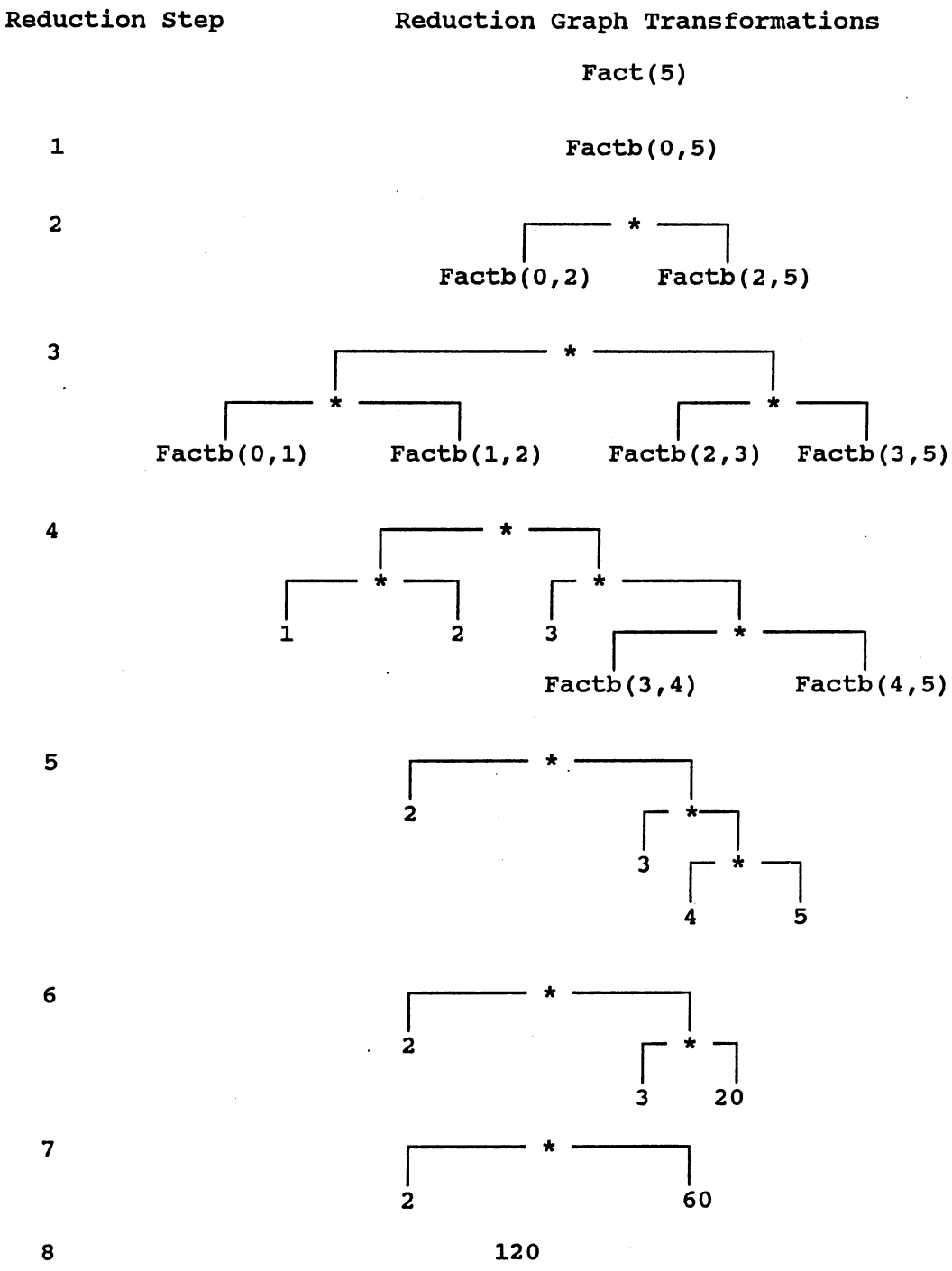


Figure 74. Eight Steps in the Parallel Graph Reduction of Fact(5)

in Figure 73; but, at each step any function ready for reduction is reduced. Machines which implement this strategy are termed reduction machines.

Performing the reductions sequentially required fourteen steps [Figure 73], but, parallel evaluation required only eight [Figure 74]. A machine capable of physically realizing such parallel evaluations offers a significant increase in performance over that of the conventional sequential implementation.

7.2 Implementing the Functional Model and ALICE

This section describes the basic scheme used by the implementors of the reduction machine, ALICE. It introduces the concepts of graph reduction, eager, constrained, and lazy evaluation modes. Finally, it reviews ALICE's architectural approach to reduction.

7.2.1 The Basic Schemes - Graph Reduction and Eager Evaluation

Graph reduction is a form of reduction. Its basis is that each instruction that accesses a particular definition will manipulate references to the definition. That is, graph manipulation is based on the sharing of arguments using pointers. When a function with a specific parameter value is demanded, the function is traversed in order to reduce the definition and return with the actual

value. Any subsequent references to the function with that specific parameter will immediately receive the functional value [85]. True reduction machines use the graph reduction approach [85, 92].

The basic scheme the designers of ALICE employ to implement graph reduction is to represent the execution graph of a function by a collection of packets. Each packet represents one node of the graph and the arcs extending downward from that node. Each packet may be formatted as shown in Figure 75. The primary fields presented at this time are the following:

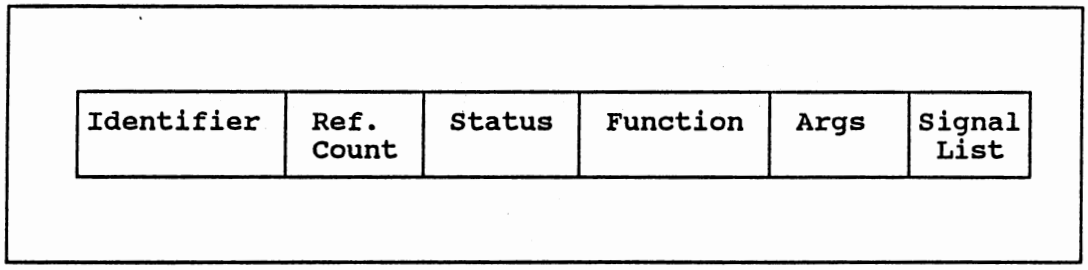


Figure 75. Software Packet

1) the Identifier field; it holds a value that uniquely identifies the packet.

2) the Function field; it specifies the function of the node this packet represents in the graph.

3) the Args field; it contains the identifiers of the packets representing the arguments of the function.

4) the Ref field; it contains the number of packets which reference the current packet. For example, this packet is the argument of a certain number of other packets; that number is recorded in the Ref field.

Figure 76 shows an expression graph and its packet representation. The constant arguments of the functions are represented in their successor constructor function form. Figure 77 shows the same packet collection with the shorthand notation [N]. The notation [N] is used to designate the identifier of the root node of the subgraph resulting from the Succ(Succ(...Succ(0)...)) construction of the constant N. Additionally, a packet with no function or argument field, and only with an integer constant designates the subgraph resulting from the Succ(Succ(...Succ(0)...)) construction of the constant. This notation will be used in future examples.

A collection of packets represents the graph resulting from each reduction step. Figure 78 shows the packet sets that would result from the evaluation of Fact(3). At each type B reduction, the packet of the function being reduced is replaced by a new group of

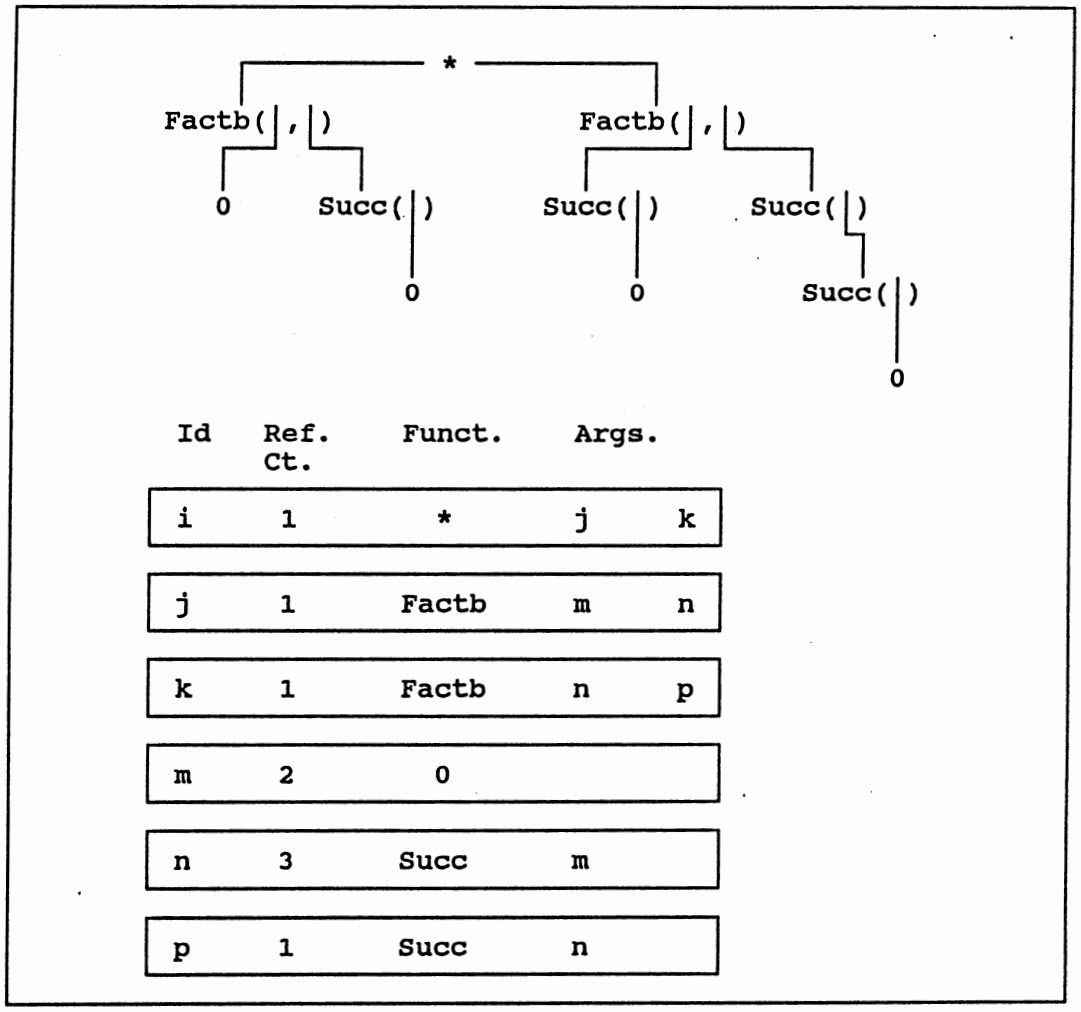


Figure 76. Graph of Factb(0,1) * Factb(1,2) and Equivalent Packet Representation

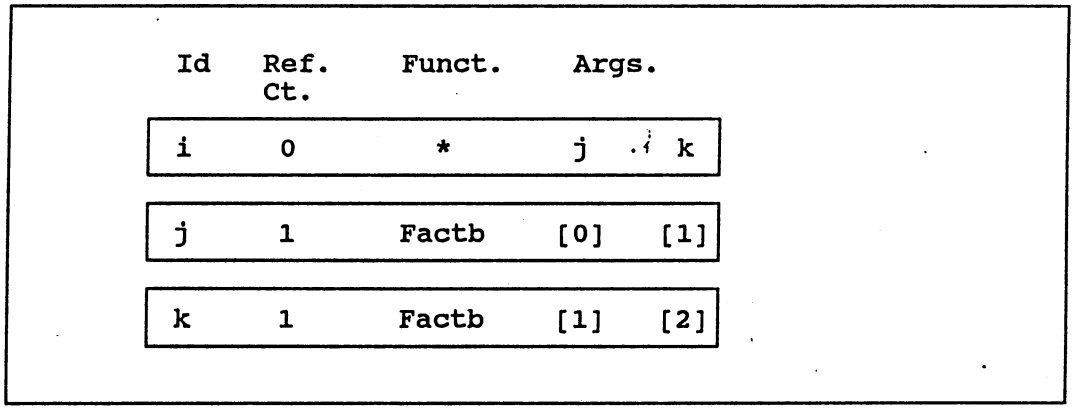


Figure 77. Shorthand Notation for Packets in Figure 76

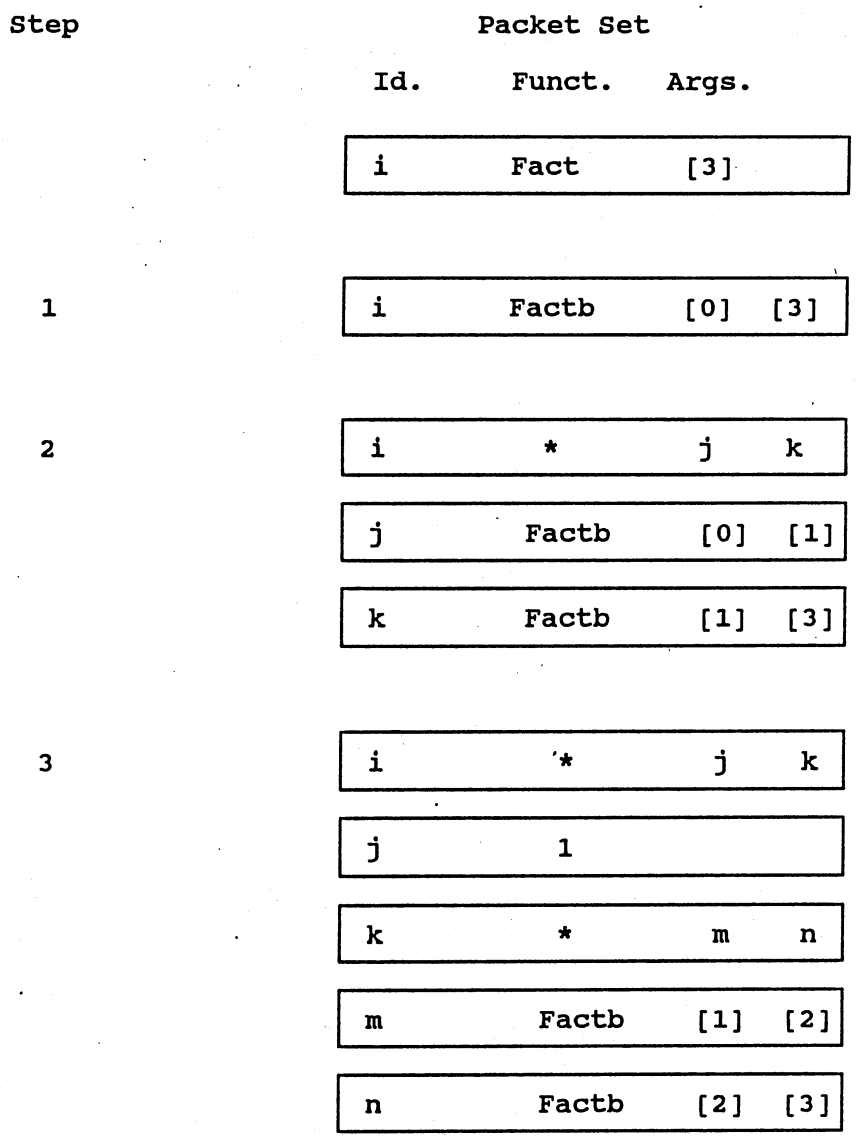


Figure 78.a. Steps One Through Three in the Packet Reduction of Fact(3)

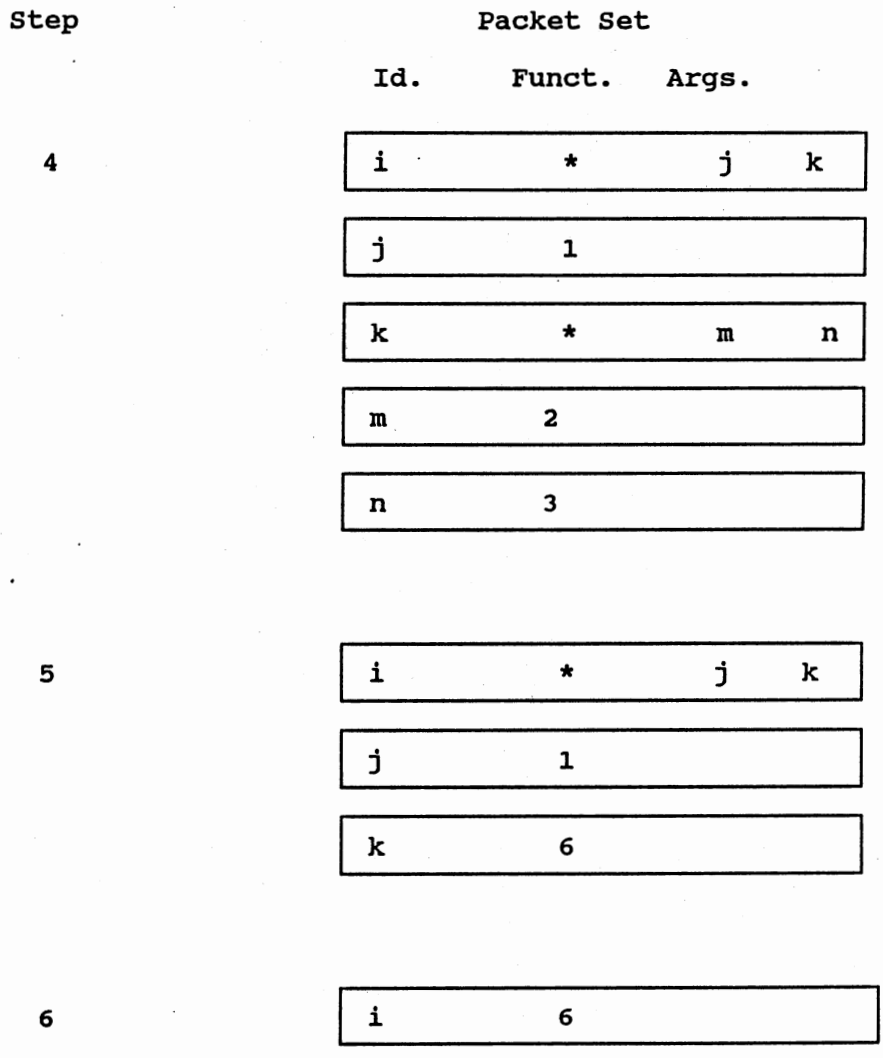


Figure 78.b. Steps Four Through Six in the Reduction of Fact(3)

packets representing the application of one of the rewrite rules. The identifier of the reduced function's packet is associated with the topmost packet of the replacing packet group. The topmost packet represents the function of lowest precedence in the replacing expression, this is referred to as the outermost function. Also, a type A reduction places the constructor result in the same identifier packet.

When a function requires a constructor function as an argument in order to make a type A reduction, it must wait until all its arguments become of the correct form. While it is waiting for constructor arguments, it need not be considered for reduction and can be "put to sleep". Then when its arguments become of the correct form, they can signal the sleeping function packet to "awaken". The "awakened" function is again available for reduction. This process is implemented by the following two fields [Figure 75]:

1) the Status field; it holds the number of arguments which are not yet of the required constructor form. A value of zero indicates the packet is awake.

2) the Signal field; it holds the identifier of the packet which needs to be signaled when the current packet becomes a constructor function packet.

Figure 79 demonstrates this process. Packet i is reduced to the primitive function multiply; it requires the constructor function Succ for its two arguments. When

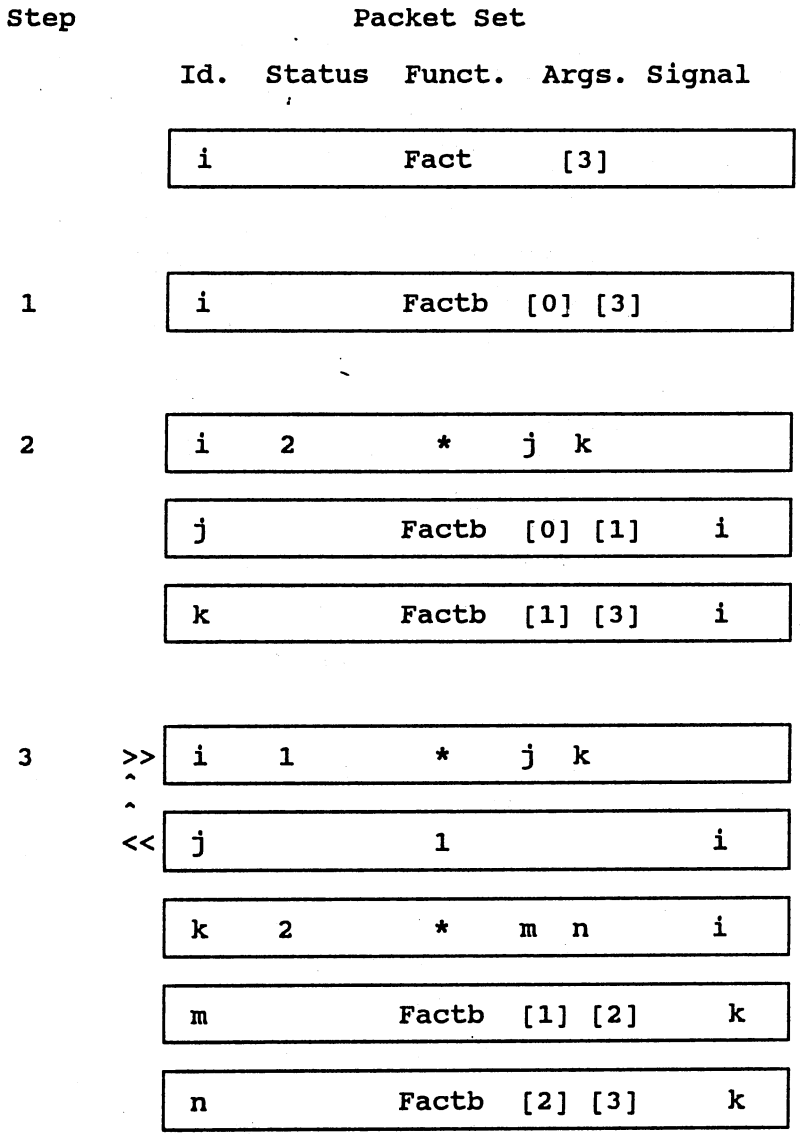


Figure 79.a. Steps One Through Three in the Reduction of Fact(3) with Packet Signaling

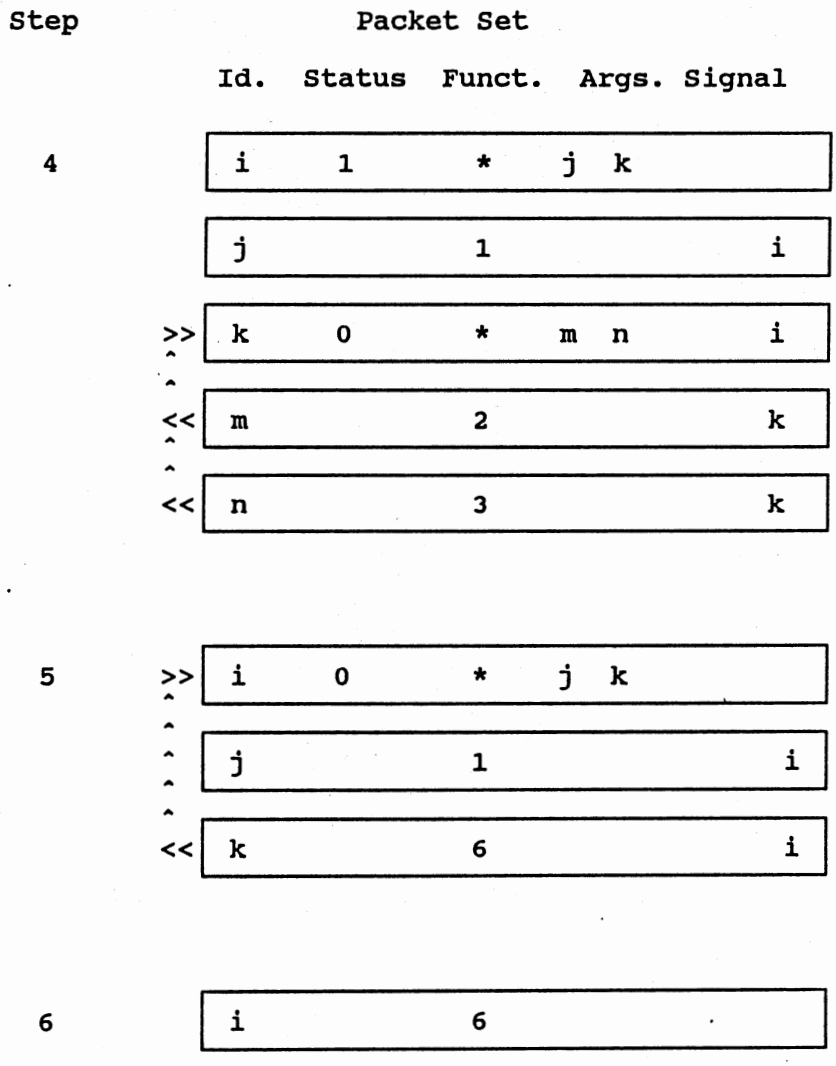


Figure 79.b. Steps Four Through Six in the Reduction of Fact(3)

packet i is created, its status field is set to 2 indicating it should sleep until two signals arrive from constructor function packets. Each of the two argument packets, j and k, has identifier i written in its Signal field. When packets j and k are reduced to Succ constructor functions for the indicated values, each signals packet i that they are of the correct form. Each signal decrements the Status field of packet i. In Figure 79, the signal is indicated with >>. When the Status field of packet i equals zero, the multiply packet wakes and becomes available for type A reduction.

This scheme is referred to as eager evaluation; each reducible function is reduced as soon as possible.

7.2.2 Constrained parallelism

This section considers constrained evaluation which is a technique to prevent some reducible functions from being reduced even though they are ready. Constrained evaluation prevents their reduction in parallel with other functions.

Suppose a function named Reciprocal were defined with the following rewrite rule.

```
--- Reciprocal(x) <= 0 if x = 0 else 1/x;
```

The right hand side of the rewrite rule is a conditional expression which may be written in a more general fashion as Cond(P,Q,R); where the function Cond returns Q when P is true; otherwise, R is returned. In an eager

evaluation, arguments P, Q, and R will be evaluated in parallel; when each argument has reduced to constructor functions, Cond will be reduced. As can be seen in the Reciprocal example this is not always expedient. R may be undefined; in Reciprocal's case, R = 1/x. Or, only one of Q or R may be required and thus the reduction of one of them will be non-productive, possibly utilizing resources which could be applied elsewhere. Thus, in some situations, it is beneficial to constrain the potential parallelism existing between P, Q, and R. The usual approach is to suspend reduction of Q and R while allowing P to reduce until it has returned either a TRUE or FALSE constructor; then, the appropriate function is awakened and its eager reduction begins. When the selected function reduces to a constructor, Cond will reduce, or return the value of the selected function.

The process indicated above may be implemented by a "sleeping/wake up" scheme similar to that discussed earlier. The programmer may indicate in the source code when constrained evaluation is required. As a result, when the packets associated with Cond(P, Q, R) are generated, those representing Q and R have their Status field marked as being asleep. When P reduces to a constructor function generating a TRUE or FALSE value, it signals its constructor status to Cond. This triggers Cond to send a wake up signal to the selected function's packet. Once awakened, the selected function reduces to a

constructor and signals Cond. Finally, Cond has its condition argument, P, and the selected argument, either Q or R, supplied as constructors and reduces.

This scheme allows parallel evaluation be performed in the reduction of the selected function, but it prevents evaluation of an undefined function or of a lengthy and unnecessary function.

7.2.3 Lazy evaluation

This section presents another form of evaluation, lazy evaluation. It prevents function packets from reducing indefinitely when called in an infinitely recursive fashion. This allows the definition of infinite data structures; but, only those elements which are needed are ever generated.

The following HOPE program builds the infinite list of counting numbers.

```
dec Numbers : -> list(num) ;
--- Numbers <= 1::IncrementByOne(Numbers) ;
dec IncrementByOne : list(num) -> list(num) ;
--- IncrementByOne(n::L) <= (n+1)::IncrementByOne(L) ;
```

The notation :: is a list constructor function; n::L means the list whose head is n and whose tail is L. For example, the notation (n::L) matches the list [2, 3, 4], where n is 2 and L is [3, 4]. The first four stages of the graph representation of the evaluation of Numbers is given in Figure 80. Numbers references itself as an

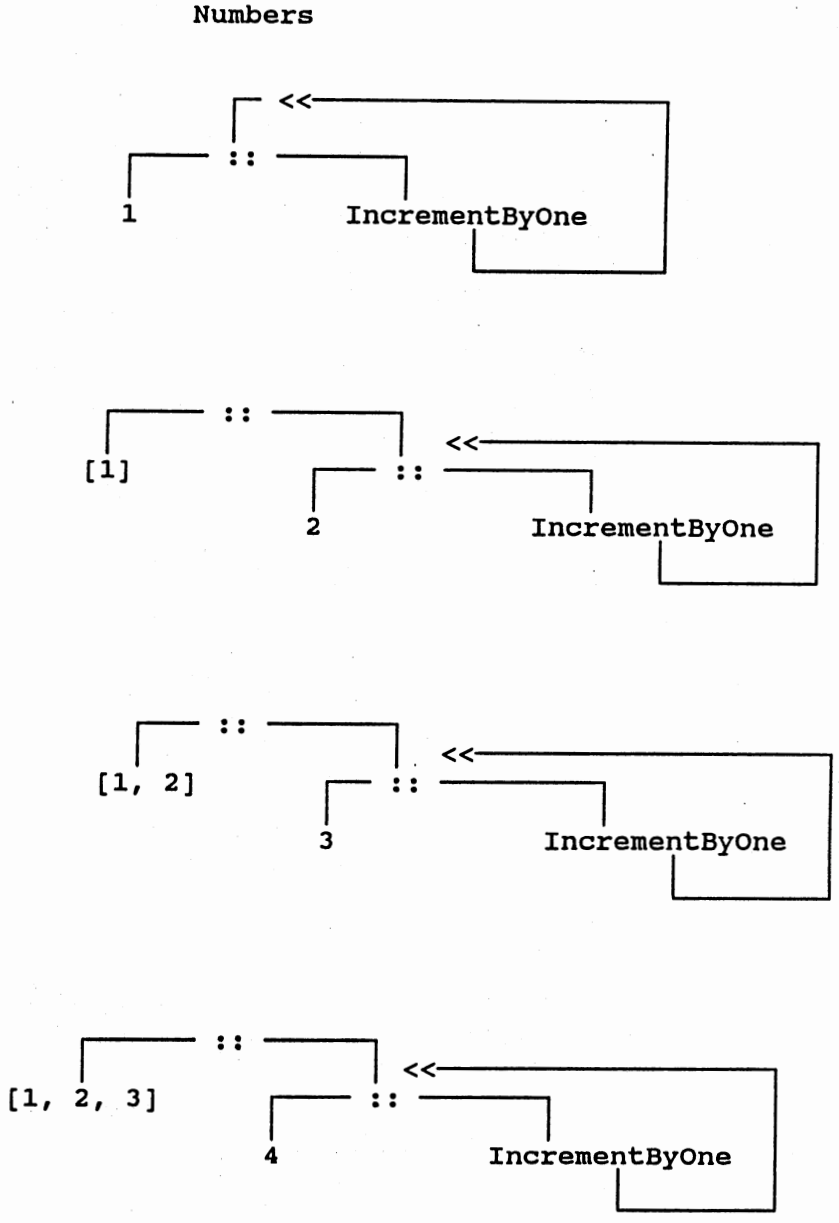


Figure 80. Reduction Graph for the Function Numbers

argument for IncrementByOne; thus, establishing the cyclic arc from IncrementByOne back up into the graph.

Eager evaluation of this program would generate elements of the list indefinitely. However, in most cases, only some finite segment of such a list is needed. As a result, the processing resources may become thoroughly involved in the computation of values which currently are unneeded or which may never be needed. This may be overcome by lazy evaluation of Numbers. As with constrained evaluation, code that is to be evaluated lazily can be flagged either by the programmer or by the compiler. The compiler flags the computation when it cannot determine that the computation will terminate [92]. Lazy evaluation implies that reduction of a function is postponed until it explicitly is requested to reduce to a constructor function by its parent node. In the Numbers example, an instance of the IncrementByOne function will not reduce to the :: constructor function on the right hand side of its rewrite rule until it is requested to do so by its parent :: list constructor [Figure 80].

Lazy evaluation may be implemented at the packet level by use of two subfields in the Status field. The outermost function's packet, for example, the Numbers packet, would be allowed to reduce eagerly but each reducible packet generated thereafter would be marked as "lazy" and "not-yet-required" in the Status subfields. Subsequent reducible packets resulting from the reduction

of one of these packets would themselves be marked as "lazy" and "not-yet-required", thus extending the lazy feature. Packets marked "not-yet-required" are not considered for reduction. When an additional reduction is required, the parent packet signals the "lazy" child packet to reduce by initiating a change in the child's "not-yet-required" subfield; the packet to be signaled is identified through the argument identifier list in the Signal field.

Thus, through use of lazy evaluation, infinite structures can be defined functionally although only some finite subset is actually to be returned, and results may be generated sequentially when needed.

7.2.4 ALICE - an Architecture for Implementing Reduction

This section investigates ALICE, the Applicative Language Idealized Computing Engine. ALICE implements the direct evaluation of functional, or applicative, languages. It is considered to be a true reduction machine because it utilizes the packet system to represent each node in the computation and thereby satisfies the requirements of graph reduction [92]. The ALICE architecture is that of a shared-memory multiprocessor connected by a crossbar interconnection network and pairs of rings [42]. Functionally, it is composed of processing agents, packet pool segments, an interconnection network,

and a distribution system [65]. Abstractly, ALICE is simply a collection of processing agents and a packet pool [18, 17].

Each processing agent in the abstract model follows the following sequence of actions.

1. Remove a non-sleeping packet of a reducible function from the packet pool.
2. If the packet represents a type A reducible function then using the Args field to locate the function's arguments, determine if the arguments required to be constructor functions are indeed constructors.

   If all arguments are of the correct form, then
   a. alter the function and argument fields to represent the constructor function for the result.
   b. decrement the Ref field of unneeded argument packets, indicating they are no longer needed by the current packet.
   c. jump to step 1.

   If any arguments are not of the correct form, then
   a. write the identifier of the current packet in the Signal field of each non-constructor argument packet.

b. write the count of the failing argument packets in the Status field of the current packet, thus marking the packet "asleep".

c. replace the packet in the packet pool.

d. jump to step 1 above for retrieval of another packet.

3. Match the current packet's function and its argument packets with the correct left hand side of some rewrite rule.

4. Implement the type B reduction of the current packet by the following actions.

a. Use the current packet to represent the outermost function of the right hand side of the rewrite rule. Maintain the same identifier.

b. For each argument of the outermost function, obtain an unused identifier. Generate a packet for each argument.

c. Record the obtained identifiers in the Args field of the outermost function's packet.

d. If the current packet was the subject of lazy evaluation, then mark each of the generated packets as "lazy" and "not-yet-required".

e. set the Ref field of each newly generated argument packet to reflect that it is referenced by the outermost function.

f. Deposit all of the packets into the packet pool.

5. Jump to step 1. above and retrieve another packet.

Continuing the abstract description, the packet pool must provide 3 major aspects. The first is passive in nature. The packet pool must provide read/write access to any packet based on its identifier; it should provide simultaneous read access but private write access. The second and third aspects of the packet pool are active in nature. The packet pool supplies the processing agents with non-sleeping packets of reducible functions. The packet pool supplies the processing agents with unused identifiers for type B reductions.

In implementing this abstract model, the developers of ALICE have utilized a special VLSI chip called a transputer. Briefly, a transputer is a von Neumann computer. A processor, 4K bytes local memory, four link interfaces for interfacing to other transputers, interfaces for accessing other devices, and system services such as reset and the clock are all packed onto a single chip. The transputers are programmed in a language called Occam. Each transputer in a system executes its own Occam program using its own local memory [93].

The abstract machine is fulfilled by the organization represented in Figure 81. The agents are implemented by pairs of transputers. Similarly, the packet pool segment is implemented via two transputers and standard RAM memory of 256K bytes [93, 65, 42].

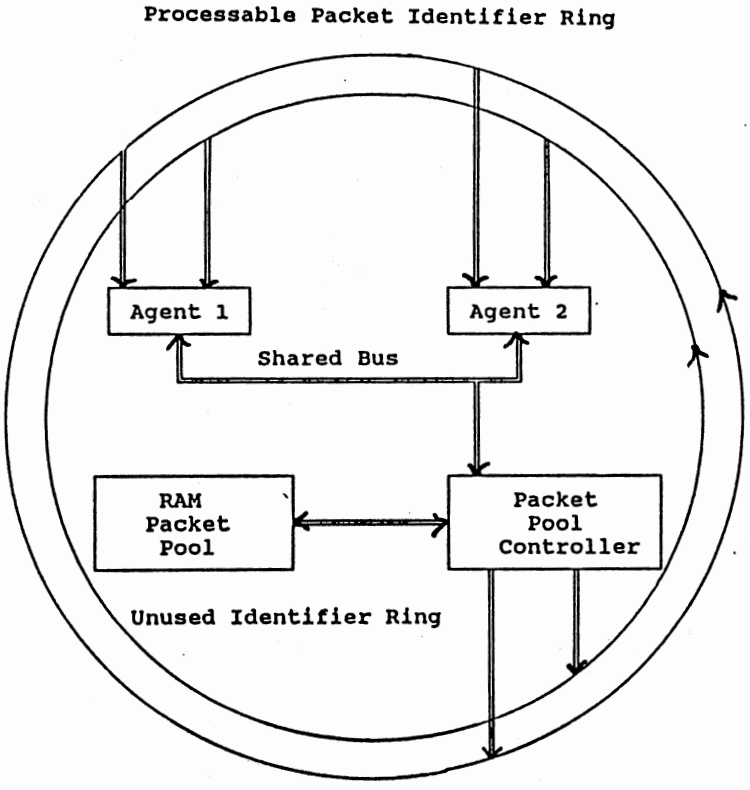


Figure 81. Single Module of ALICE

The transputers of the packet pool act as a highly intelligent memory controller. The passive aspect of the packet pool is fulfilled in the addressable RAM. Packets are stored in the RAM. The identifier field of each packet is dropped and instead, each packet is identified by its unique address in memory. The active aspect of the packet pool segment is implemented by the transputers. The transputers identify the reducible not-sleeping packets in the RAM and supply them to the agents. Further, those packets for which the Ref field value has fallen to zero are recognized by the transputers as empty or unused. As new unused identifiers are required for type B reductions, the memory transputers furnish the addresses of these packets to the agents. Thus, garbage collection is performed concurrently with program evaluation.

The specific processing to be performed by a given transputer is determined by Occam programs loaded into the transputers when the system is initialized. This not only allows the application of the transputer to such distinct tasks as agent and packet pool controller, but, also allows certain agents to specialize in the execution of specific functions such as Input/Output.

The processable packets and unused identifiers are made available to the agents by the packet pool controller over two distinct slotted rings [79, p. 312]. Each agent has its own slot window on the constantly circulating

rings. As empty slots pass by on the packet ring [Figure 81], the packet pool controller writes the identifiers, or addresses, of processable packets into them; and as its slot passes, each free agent picks off the address of its next packet from the ring. Similarly, the unused identifier ring carries unused identifiers from the packet pool controller. Each unused identifier, or free address, is written onto the Identifier ring only once. Any agent needing an unique unused identifier simply picks one off the Identifier ring.

A shared bus connects the agents with the packet pool controller. When an agent has seized an address from the Packet ring, it accesses the RAM by way of the controller for a copy of the processable packet. Argument packets and rewrite rules are also accessed in this fashion. This allows each agent to read from the same memory location (but, not simultaneously). When an agent has performed a type B reduction, the addresses seized from the Identifier ring are employed to rewrite new packets in the RAM via the bus. Other types of memory rewrites such as signalling and changing the required status of lazy packets is also done via the bus. Thus, only one agent may write to a given location since only one agent possesses a given address.

The bus is the bottle neck for the system. It is estimated that each packet pool access takes about 1 nanosecond and the processing of each packet requires

approximately six packet pool accesses with an average of 128 nanoseconds required for the processing of a packet. Figures indicate that twenty agents would be required to utilize the bus fully [18, 17]. At this time, only two transputer pairs are mounted on a board, thus excessive bus contention does not appear to be a problem.

The organization of Figure 81 is a fundamental module of the ALICE system. A single-user workstation is composed of a single module. The modules can be combined to form a multiuser mainframe as shown in Figure 82. In the larger system, the basic single-user modules are connected together by a Delta network built from four-by-four crossbar switches, implemented as a custom chip in ECL (emitter-coupled logic) [65]. In the extended environment, the packet pool is distributed throughout the system in the 256K-byte segments of each module. Each packet pool segment is addressable from any module over the Delta network. The Delta network provides for the movement of packets and rules at a rate of two hundred megabits per second.

In order to improve load balancing, intelligent links are positioned between the rings of adjacent modules. A link monitors the load on functionally equivalent adjacent rings. It transfers identifiers from heavily loaded rings to lightly loaded ones. Thus, if each slot were full on the Packet ring of module two and the Packet ring of module one was near empty, the link would begin filling

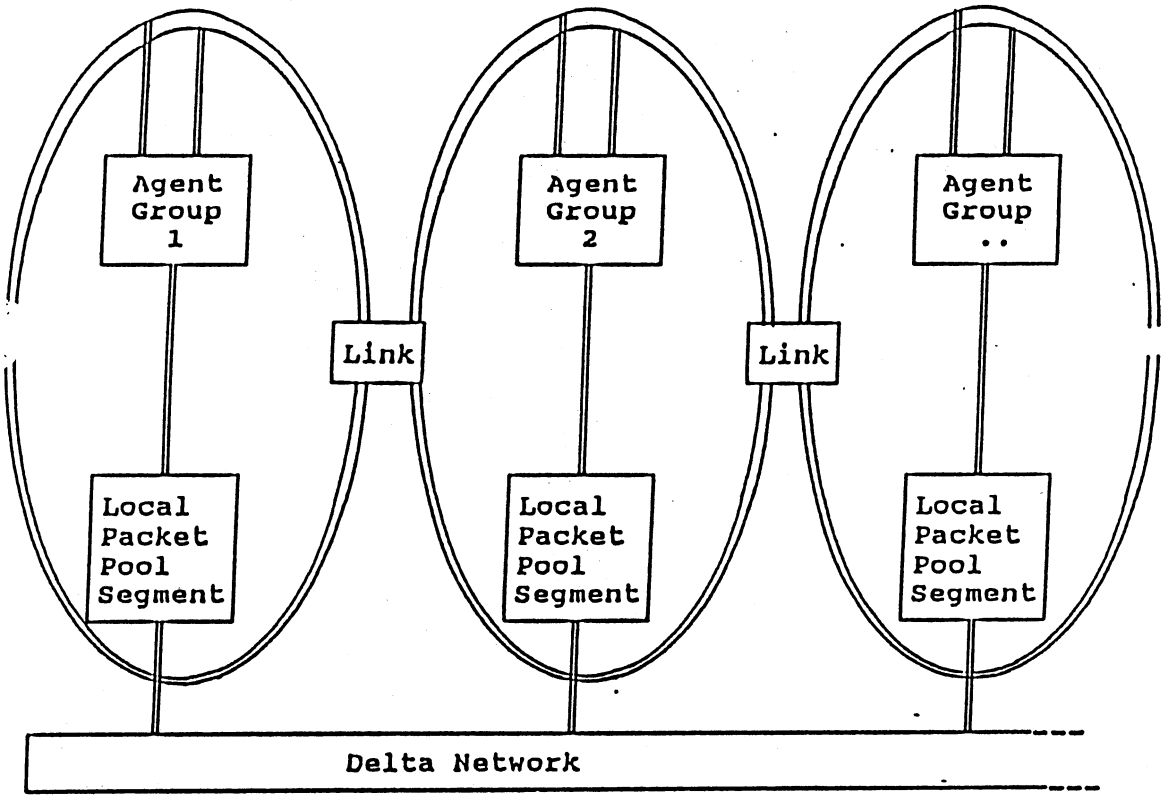


Figure 82. Multi-module ALICE System

module one's empty Packet ring slots with identifiers from module two's Packet ring. Such a link also exists for the Identifier Rings. Thus work and storage are distributed between modules, and identifiers from each ring can migrate through out the system. The rings and the links compose ALICE's distribution system.

The processing rates of the ALICE system are very positive. Estimates indicate a single-module desk-top system will process in the neighborhood of 150,000 packets per second. A multi-module system of 4096 nodes can process in excess of 150 million packets per second [18, 17].

The packet pool and processing agents work together to implement a parallel reduction system based on the packet representation of graph reduction. The distribution system and delta network function with the agents and packet pools to implement a shared memory distributed multiprocessor system.

7.3 Summary

This chapter introduces the concepts of functional languages and reviews the distinctions between functional and procedural languages.

The chapter shows that a machine capable of function evaluation, based on application of rewrite rules, implements a demand driven system; functions are evaluated when their results are demanded. Demanded functions may

be evaluated in parallel without effecting the outcomes of other functional evaluations.

The basic scheme of utilizing packets to implement graph reduction and to represent the nodes and arcs in a reduction graph is introduced and the concepts of eager, constrained, and lazy evaluation are reviewed.

The architecture of the Applicative Language Idealized Computing Engine, ALICE, a shared-memory multiprocessor, reduction machine is surveyed.

CHAPTER VIII

SUMMARY AND POSSIBLE EXTENSIONS

8.0 Summary

This treatise reviews the von Neumann computer architecture and presents the fundamental elements of five classes of parallel computer architecture. Further, it provides example architectures from each of the parallel classes. The architectures and examples presented are the following:

1) Array processors and the ILLIAC IV. Array processors allow the simultaneous identical processing of multiple streams of data and are termed single-instruction-stream multiple-data-stream processors (SIMD);

2) Pipelined computers and the HEP. Pipelined computers allow functions such as instruction fetch/execute and floating point arithmetic operations to be broken down into subfunction stages and input to be sequenced through the subfunctions to produce a final result for each input value. The rate of result production is the same as the rate of input entry as long as the pipeline of subfunctions is full. This approach allows parallel processing of instructions, or data, or both;

3) Multiprocessors, the Alliant FX/8, and the Cosmic Cube. Multiprocessors allow the application of multiple CPUs to the solution of a single problem using a multiple-instruction-stream multiple-data-stream (MIMD) scheme and thus reduce the time to solution of a single problem;

4) Data flow machines, the Dennis Static Data Flow Computer, and the Manchester Data Flow Computer. Data flow computers are non-von Neumann in nature. They base their execution on data flow graphs where each node in the graph fires when all its inputs are available. The structure of a data flow graph is based on the instructional data dependencies inherent in the program to be executed. The control of data flow machines is based on operand value availability rather than program instruction sequencing.

5) Reduction machines and the ALICE. Reduction machines are also non-von Neumann in nature. Reduction machines base their execution on the demand for data and graph reduction. A function is evaluated only when its result is needed. Thus, they are termed demand driven.

8.1 Proposed Additions to the Text

This treatise can be used as a class text in computer architecture. Complete utilization suggests certain additions should be made to the work. The additions are the following:

1) appendices on several topics should be added. One appendix should investigate networks. Crossbar switches, interconnection, alignment, and Delta networks should be discussed, focusing on the function, similarities and distinctions of the networks. Also, an appendix on transputers should be incorporated.

2) an index should be provided,

3) a set of problem oriented questions with solutions, and a set of discussion oriented questions with suggested answers or references to other sources for further study should be included with each chapter,

4) an annotated bibliography should be appended to each chapter to aid the student interested in further investigation on a given topic

8.2 Text Readability

This treatise is designed for undergraduate students. In order to confirm that the readability of the text is appropriate for undergraduates, the text was submitted to a readability analysis based on the Fry Readability Scale[1]. The analysis data is shown in Table II. Based on the Fry Readability Scale, the reading grade level of the text is eleven. This is satisfactorily low to be read by undergraduates.

---

[1]Fry, Edward B., Fry Readability Scale. Jamestown Publishers, 1978.

TABLE II

ANALYSIS DATA OF TEXT READABILITY
FOR THE FRY READABILITY
SCALE

Text Page	Beginning Line Number	Syllable Count	Sentence Count
17	1	168	5.8
39	2	156	3.0
46	12	160	6.1
58	12	162	5.7
72	28	168	6.9
99	1	155	6.4
148	13	155	6.7
159	1	173	6.6
177	3	173	5.5
216	11	188	5.8
248	9	154	5.7
255	3	165	6.8
		---	---
	Average ---->	164	5.9

8.3 Final Statement

These chapters have been created with computer Science undergraduate students in mind. The discussions are designed to lead them to a better understanding of the structure and organization of parallel computing systems and to open their imaginations to the exciting computing possibilities made available by parallel computer architectures.

BIBLIOGRAPHY

1. Abe, S., Hiraoka, R., Fukunaga, et al. Preliminary Evaluation of data flow computers. Proceedings of the 24-th IEEE Computer Conference (San Francisco, Calif., Feb.). IEEE, New York, 1982, 87-90.

2. Ackerman, W.B. Data flow languages. IEEE Computer (Feb. 1982), 15-25.

3. Alliant Computer Systems Corporation. Alliant FX/Series Product Summary. October 1986.

4. Arvind and Gostelow, K.P. The U-interpreter. IEEE Computer (Feb. 1982), 42-49.

5. Arvind, and Iannucci, R.A. A critique of multiprocessing von Neumann style. Proceedings of the 10-th Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 13-17). ACM, New York, 1983, 426-436.

6. Arvind and Thomas,R.E. I-structures: An efficient data type for functional languages. Rep. LCS/TM-178, Lab. for Computer Science. Massachusetts Institute of Technology, June 1980.

7. Baer, J. Computer systems architecture. Computer Science Press, Inc., Rockville, Md., 1980.

8. Baer, J. Computer Architecture. IEEE Computer (October 1984), 77-87.

9. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., and Stokes, R.A. The ILLIAC IV Computer. IEEE Trans. on Computers (Aug. 1968), 746-757.

10. Bernstein, A.J. Program analysis for parallel processing. IEEE Transactions on Electronic Computers (Oct. 1966), 757-762.

11. Bic, L. Data-driven logic: a basic model. Proceedings of the Conference on High Level Language Computer Architecture (Los Angeles, Calif., May). 1984.

12. Bril, R.J. On cacheability of lock variables in tightly coupled multiprocessor systems. Computer Architecture News (June 1987), 25-32.

13. Burns, D.M., and Rome, D.L. Computer architecture alters concept of parallel processing. Research and Development (April 1986), 70-74.

14. Bursky, D. Advanced ECL family boosts performance threefold. Electronic Design (July 23, 1987), 41-46.

15. Burstall, R.M., McQueen, D.B., and Sannella D.T. HOPE: an experimental applicative language. Internal Report, Dept. of Computer Science, University of Edinburgh, 1980.

16. Calingaert, P. Operating System Elements: A User Perspective. Inc., Englewood Cliffs, New Jersey, 1982.

17. Darlington, J., and Reeve, M. ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages. Proceedings ACM Conf. Functional Programming Languages Computer Architecture, 1981, 65-75.

18. Darlington, J., and Reeve, M. ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages. Proceedings of the International Symposium on Functional Programming Language Computers and Architecture (Goteborg, Sweden, June). 1981, 32-63.

19. Davis, A.L., and Keller, R.M. Data flow program graphs. IEEE Computer (Feb. 1982), 26-41.

20. Denelcor, Inc. Heterogeneous Element Processor: Principles of Operation, April 1981.

21. Denelcor, Inc. Techdatasheet, Sept. 1982.

22. Denning, P.J. Parallel computing and its evolution. Communications of the ACM (Dec. 1986), 1163-1167.

23. Dennis, J.B. Data flow supercomputers. IEEE Computer (Nov. 1980), 48-56.

24. Dennis, J.B. The varieties of data flow computers. IEEE Proceedings of the Conference on Distributed Computing, 1979, 430-439.

25. Dennis, J.B., Stoy, J., and Guharoy,B. Vim: an experimental multi-user system supporting functional programming. Proceedings of the Conference on High Level Language Computer Architecture (Los Angeles, Calif., May). 1984.

26. Dhas, C.R. A ring-based data-flow multiprocessor. Proceedings of the 24-th IEEE Computer Conference (Francisco, Calif., Feb.). IEEE, New York, 1982, 87-90.

27. Dias, D.M., and Jump, J.R. Packet switching interconnection networks for modular systems. IEEE Computer (Dec. 1981), 43-53.

28. Dongarra, J.J. Performance of various computers using standard linear equations software in a Fortran environment. Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 23 (July 29, 1986).

29. Dubois, M., Scheurich, C. and Briggs, F.A. Synchronization, coherence, and event ordering in multiprocessors. IEEE Computer (Feb. 1988), 9-21.

30. Feng, T. A survey of interconnection networks. IEEE Transactions on Computers (Sept 1980), 109-124.

31. Flynn, M.J. Very high-speed computing systems. Proceedings of the IEEE 54 (1966), 1901-1909.

32. Frenkel, K.A. Evaluating two massively parallel machines. Communications of the ACM (August 1986), 752-758.

33. Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H. A second opinion on data flow machines and languages. IEEE Computer (Aug. 1982), 58-69.

34. Gajski, D.D., and Peir, J. Essential issues in multiprocessor systems. IEEE Computer (June 1985), 9-27.

35. Gerdts, A., and Kowalewski, D.L. First experiences with an emulation of a system of cooperating reduction machines. Proceedings of the 10-th Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 13-17). ACM, New York, 1983, 8.8-8.15.

36. Gurd, J.R., Kirkham, C.C., and Watson, I. The Manchester prototype data flow computer. Communications of the ACM 28, 1, (Jan. 1985), 34-52.

37. Gutzmann, K.M. Optimal dimension of hypercubes for sorting. Computer Architecture News (March 1987), 68-72.

38. Handler, W. The impact of classification schemes on computer architecture. Proc. 1977 Int. Conf. on Parallel Processing, 7-15.

39. Hankin, C., Till, D., and Glaser, H. Linking data flow and functional languages. Byte (May 1986), 123-134.

40. Hughes, J.L. Implementing control-flow structures in data flow programs. Proceedings of the 24-th IEEE Computer Conference (San Francisco, Calif., Feb.). IEEE, New York, 1982, 87-90.

41. Hwang, K., and Briggs, F.A. Computer Architecture and Parallel Processing. McGraw Hill, New York, NY., 1984.

42. Hwang, K., Ghosh, J., and Chowkwanyun, R. Computer architectures for artificial intelligence processing. IEEE Computer (January 1987), 19-27.

43. Hwu, W.W., and Patt, Y.N. Checkpoint repair for out-of-order execution machines. The 14-th Annual International Symposium on Computer Architecture (Pitsburgh, Pennsylvania, June 2-5, 1987). IEEE Computer Science Press, Washington, D.C., 1987, 18-26.

44. Intel iPSC Data Sheet, Intel Scientific Computers.

45. Irwin, M.J. Reconfigurable pipeline systems. ACM Computing Surveys (Jan. 1978), 86-92.

46. Jordan, H.F. Experience with pipelined multiple instruction streams. Proceedings of the IEEE vol. 72, no. 4, (Jan. 1984), 113-123.

47. Jordan, H.F. Performance measurement of HEP - a pipelined MIMD computer. Proceedings 10th Ann. Symp. Computer Architecture (June 1983), 207-212.

48. Kartashev, S.P., and Kartashev, I.S., eds. Designing and Programming Modern Computers and Systems, Vol.I, LSI Modular Computer Systems. Prentice-Hall, Inc., Englewood, N.J., 1982.

49. Keller, R.M., and Lin, F.C.H. Simulated performance of a reduction-based multiprocessor. IEEE Computer (July 1984), 70-82.

50. Keller, R.M., Lin, F.C.H., and Tanaka, J. Rediflow processing. Proceedings of the 26-th Computer Conference (February 1984), 410-417.

51. Kluge, W.E. Cooperating reduction machines. IEEE Transactions on Computers (November 1983), 1002-1012.

52. Kogge, P.M. The Architecture of Pipelined Computers. McGraw Hill, New York, 1981.

53. Krajewski, R. Multiprocessing: an overview. Byte (May, 1985), 171-181.

54. Kuck, D.J. Illiac IV software and application programming. IEEE Trans. on Computers (Aug 1968), 746-757.

55. Kuck, D.J. The Structure of Computers and Computations, Vol. 1. John Wiley and Sons, New York, N.Y., 1978.

56. Kumar, M. Performance enhancement in buffered delta networks using crossbar switches and multiple links. Journal of Parallel and Distributed Computing 1, (1984), 81-103.

57. Lee, R.L., Yew, P.C., and Lawrie, D.H. Multiprocessor cache design considerations. The 14-th Annual International Symposium on Computer Architecture (Pitsburgh, Pennsylvania, June 2-5, 1987). IEEE Computer Science Press, Washington, D.C., 1987, 27-34.

58. McGraw, J.R. Data flow computing - software development. IEEE Transactions on Computers (Dec. 1980), 1095-1103.

59. Mokhoff, N. Parallelism breeds a new class of supercomputers. Computer Design (March 15, 1987), 53-64.

60. Myers, W. Getting the cycles out of a supercomputer. IEEE Computer (March 1986), 89-92.

61. Oldehoeft A.E., Allan S., Thoreson, S.A., Retnadhas C. and Zingg, R.J. Translation of high level programs to data flow and their simulated execution on a feedback interpreter. Iowa State University Technical Report #78-2 (1978).

62. Paseman, W.G. Applying data flow in the real world. Byte (May 1986), 201-214.

63. Peterson, J.L. and Silberschatz, A. Operating System Concepts. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

64. Poplett, J., and Kurver, R. The DSI transputer development system. Byte (Feb. 1988), 249-254.

65. Pountain, D. Parallel processing: a look at the ALICE hardware and Hope language. Byte (May 1985), 385-394.

66. Ramamoorthy, C.V., and Li, H.F. Pipeline architecture. ACM Computing Surveys (March 1977), 61-102.

67. Ravishankar, C.V., and Goodman, J.R. VLSI considerations that influence data flow architecture. Proceedings of the 24-th IEEE Computer Conference (San Francisco, Calif., Feb.). IEEE, New York, 1982, 87-90.

68. Ryder, B.G., and Paull, M.C. Elimination algorithms for data flow analysis. ACM Computing Surveys (September 1986), 277-316.

69. Seban, R.R., and Siegal, H.J. Shuffling with the Illiac and PM2I SIMD Networks. IEEE Trans on Comp. (July 1984), 619-625.

70. Seitz, C.L. The Cosmic Cube. Communications of the ACM (January 1985), 22-33.

71. Smith, B.J. A pipelined shared resource MIMD computer. Proc. 1978 Int'l, Conf. on Parallel Proc., 1978, 6-8.

72. Smith, B.J. Architecture and applications of the HEP multiprocessor computer system. Real Time Signal Processing IV (Aug. 1981), 241-248.

73. Sohi, G.S., and Vajapeyam, S. Instruction issue logic for high-performance, interruptable pipelined processors. The 14-th Annual International Symposium on Computer Architecture (Pitsburgh, Pennsylvania, June 2-5, 1987). IEEE Computer Science Press, Washington, D.C., 1987, 27-34.

74. Srini, V.P. An architectural comparison of data flow systems. IEEE Computer (March 1986), 68-88.

75. Stalling, W. Computer Organization and Architecture. MacMillan Publishing Co.,Inc., New York City, New York, 1987.

76. Stone, H. High Performance Computer Architectures. Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

77. Syre, J.C. The data flow approach for MIMD multiprocessor systems. Parallel Processing Systems. Cambridge University Press, 1982, 239-274.

78. Tanenbaum, A.S. Structured Computer Organization. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

79. Tanenbaum, A.S. Computer Networks. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

80. Thoreson, S.A. and Long, A.N. Applying program restructuring to dataflow environments. The Proceedings of Workshop on Applied Computing (Stillwater, Ok., Oct. 1986), 58-62.

81. Thoreson, S.A. and Long, A.N. Modeling a virtual memory in a dataflow environment. Proceedings of the Sixteenth Annual Pittsburgh Conference on Modeling and Simulation 16 (1985), 951-957.

82. Thoreson, S.A. and Long, A.N. A feasibility study of a memory hierarchy in a dataflow environment. Proceedings of the 1985 International Conference on Parallel Processing (1985), 356-360.

83. Thoreson, S.A. and Oldehoeft, A.E. Instruction reference patterns in data flow programs. ACM '80 Proceedings (1980), 211-216.

84. Treleaven, P.C. Parallel models of computation. Parallel Processing Systems, Cambridge University Press, 1982. 274-282.

85. Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P. Data-driven and demand-driven computer architectures. ACM Computing Surveys (Mar.1982), 93-143.

86. Treleaven, P.C., and Lima, I.G. Japan's fifth-generation computer systems. IEEE Computer (Aug. 1982), 79-88.

87. Treleaven, P.C., and Lima, I.G. Future computers: logic, data flow, ..., control flow? IEEE Computer (March 1984), 47-58.

88. Treleaven, P.C., and Mole, G.F. A multiprocessor reduction machine for user-defined reduction languages. Proceedings of the 7-th Annual Symposium on Computer Architecture (May 6-8). ACM, New York, 1980, 121-130.

89. Treleaven, P.C., Hopkins, R.P., and Rautenbach, P.W. Combining data flow and control flow computing. The Computer Journal (May 1982).

90. Turner, D. Combinator reduction machines. Proceedings of the 10-th Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 13-17). ACM, New York, 1983, 5.26-5.38.

91. Veen, A.H. Dataflow machine architecture. ACM Computing Surveys (December 1986), 365-396.

92. Vegdahl, S.R. A survey of proposed architectures for the execution of functional languages. IEEE Transactions on Computers (December 1984), 1050-1071.

93. Walker, P. The transputer. Byte (May 1985), 219-235.

94. Watson, I., and Gurd, J. A practical data flow computer. IEEE Computer (Feb. 1982), 51-57.

95. Welty, L.L., and Patton, P.C. Hypercube architectures. National Computer Conference (1985), 259-265.

APPENDIXES

APPENDIX A

GLOSSARY

accumulator: A holding register for the results of arithmetical and logical operations. Usually, the accumulator is loaded with the value of an operand while any other required operands of an instruction remain in memory; the result of the operation is placed in the accumulator by the arithmetic/logic unit.

address: An identifier of a memory location, register, or device.

address bus: A unidirectional bus over which is transmitted digital information that identifies either a device or a memory location.

aliasing: In procedural languages, two or more names are used to denote the same memory address.

alignment network: A network that allows the simultaneous connection of any two or more distinct module pairs. For example, in an array or a multiprocessor system, any memory may be connected to any processor.

arbitration network: A network allowing data from any input to be routed to one of several possible outputs as specified by information included with the data.

architecture: See computer architecture.

arithmetic/logic unit (ALU): A unit of the central processing unit (CPU) that performs arithmetic and logical operations.

ARPANET: One of the first large scale packet switched networks produced by the ARPA project and funded by the U.S Defense Advanced Research Project Agency.

array processor: A computer with one control unit, multiple arithmetic/logic units, and multiple memory units. The control unit fetches instructions from the memories, decodes them and broadcasts the instructions to the arithmetic/logic units. Each arithmetic/logic unit can fetch its own data for processing. An array processor performs duplicate operations on multiple data items simultaneously.

associative memory: See content addressable memory.

associative processor: A computer system much like an array processor with the distinction that it operates on associative memories.

asynchronous: The starting and stopping of processing based on the sending and receiving of acknowledgement signals between dependant modules.

barrel shifter: An interconnection network with the interconnect function defined as follows:

$$B(j) = (j \pm 2^i) \pmod N$$

where N is the number of modules connected, $0 \leq j \leq N-1$, $0 \leq i \leq n-1$, and $n = \log_2 N$.

bus: A common connector. In data communications, a network topology in which workstations are connected by T junctions to one main cable. In computing, an electrical connection between the components of a computer system along which data is transmitted.

cache: A very high speed buffer memory into which instructions and data anticipated for use in the near future are loaded from main storage. The processor has direct access to the cache.

call-by-reference: Method of passing parameters wherein the function receives the address of the real parameter value. Changes to the formal parameter in the function results in changes to the real parameter in the calling routine.

call-by-value: Method of passing parameters wherein the function receives a copy of the real parameter. Changes to the formal parameter in the function results in changes to the parameter's local copy and not to the real parameter in the calling routine.

central processing unit (CPU): The unit of a computer containing the control unit, the arithmetic/logic unit, and a number of registers.

computer architecture: The arrangement of the parts of a computer system, their interconnections, dynamic interactions, implementations, and management.

content addressable memory: A memory that is content addressable; that is, where every memory register that contains a specified string of symbols (key) is accessed rather than the single register whose location is specified.

control unit (CU): The unit of the central processing unit responsible for fetching and decoding of instructions, operand address calculation, and driving the arithmetic/logic unit and other system elements.

crossbar switch: A telephone switching network. An alignment network that allows simultaneous conflict free transmissions between two sets of modules. For example, if there are M memories and P processors, data may be transmitted on an MxP crossbar switch from any memory to any processor, assuming there is a one to one mapping. An MxP crossbar switch has M inputs and P outputs.

data dependency: The state of being dependent or conditional on the value of the data read or written in a single instruction or in a block of code. Data dependencies exist between operations when the action of one operation on the data affects the outcome of the other operation and vice versa.

data-driven computer: See data flow computer.

data flow computer: Computer in which instructions are executed based on data dependencies. Programs are represented by data flow graphs. Availability of operands triggers the execution of operations.

data flow graph: A directed graph used to represent a data flow program, where nodes are instructions or processes whose outputs pass along links to subsequent processes. A node executes, or fires, if all its input links are carrying values. The graph represents the data dependencies inherent in the computer program.

data parallelism: The capability of a computer to process multiple data items at the same time.

delta network: An alignment network establishing a path of constant length from any one of its a^n inputs to any one of its b^n outputs. This is an $a^n X b^n$ switching network with n stages consisting of aXb crossbar switches. It is cheaper to construct than an $a^n X b^n$ crossbar switch but provides less speed as the number of terminals increases.

demand-driven computer: See reduction machine.

distributed data processing: The processing of jobs at a number of geographically separate locations.

distribution network: A network that allows data from an input to be dispensed to one or more outputs.

emitter-coupled logic: In microelectronics, a transistor logic circuit characterized by fast action and high power dissipation. The fastest of the widely used technologies for LSI and VLSI chips.

fault: In systems, a condition that causes a device, component, or element to fail to perform in a required manner. The fault may be either physical or algorithmic.

fire: The execution of a node in a data flow graph.

flip-flop: A simple circuit that can maintain one of two possible stable states.

front-end: In computing, a front-end processor is used to handle communication interfacing.

Goodyear-Aerospace: The division of Goodyear Tire, Akron, Ohio, that designs and builds parallel computers (as well as other unrelated things). Notables it has built are STARAN (1974) and the Massively Parallel Processor (MPP) (1982).

graceful degradation: Components already in the system assume some or all of the responsibilities of failed components. The system can continue to operate although there may be some reduction of performance.

graph reduction: a form of reduction in which each instruction that accesses a particular definition will manipulate references to the definition. That is, graph manipulation is based on the sharing of arguments using pointers. When a functional value is demanded the reference is traversed in order to reduce the definition and return with the actual value.

host: A computer used to prepare programs to be run on other systems. Within a network, it may provide services such as computation, database access, or allow use of special programming languages. Within a distributed system, it may be the primary controlling computer within the multiple computer installation.

image processing: The processing of digitized image data by a computer to obtain information about the image or to change the representation of the image.

immediate operand: Constant stored in the machine instruction.

instruction parallelism: The capability of a computer to execute multiple instructions at the same time.

interconnection network: In general, an interconnection network allows communication between modules. In parallel systems such as array processors, there is a specific one to one and onto function defined, say f. If there are N modules, the interconnection network allows simultaneous communication between module i and module f(i) where i= 1,2,...N. The specific function is a constant for the network and designed for the application of the system.

interleaved memory: If n memory modules are numbered 0, 1, 2, ..., n-1, and if words at address i are located in memory module number i (mod n), then the memory is n-way interleaved. The n memory modules may be operated independently and timeshare the memory bus.

large scale integration (LSI): The fabrication of 100 to 1000 gates on a single chip.

LOCK and UNLOCK operations: Process synchronization primitives. Used so that each process accessing shared data excludes all others from doing so simultaneously. Processes attempting to initiate access to shared data while another process has access is forced into busy waiting. See P and V operations.

machine instruction: An instruction in binary code that can be executed directly by a computer.

main memory: The memory in a computer that stores instructions and data that are in active use by the processor.

multiprocessor: A computer system with more than one central processing unit. Used to decrease the time to completion for a single job.

network: Either a series of interconnected points or a system of interconnected communication facilities.

outermost function: In reduction, in an expression, the operation of lowest precedence.

P and V operations: Process synchronization primitives. Used so that each process accessing shared data excludes all others from doing so simultaneously. Processes attempting to initiate access to shared data while another process has access is removed by the operating system from the list of ready processes (put to sleep).

packet: A self contained component of information. In communications, the information is a message comprising address, control, and data that can be transferred as an entity within a network.

packet switching: A method of message transmission in which each complete message is assembled into one or more packets that can be sent through a network, collected and reassembled into the original message at the destination. The individual packets need not be sent by the same route. The channels are seized only during the duration of packet transmission and are then released.

parallel computer: A computer that can perform multiple operations at the same time.

pipelining: The process of partitioning a job into distinct steps and streaming inputs through the steps. The mechanism is like that of materials moving through an assembly line.

process: A program or some more or less self-contained transformation that is actually being executed by a processor.

processor: A device or system capable of performing operations upon data.

program counter: The register in the control unit of a von Neumann computer that holds the address of the next machine instruction to be executed.

random-access memory (RAM): A memory system which accepts as input the location of a memory word and returns as output the contents of that word. The time to access one word is the same as that required to access any other word.

reduction: A computation system in which programs are built from nested expressions. The nearest analogy to an instruction is a function application where the function returns its result in place (a CALL-RETURN pattern of control). A function or its arguments may be recursively defined as a primitive operation, such as add or multiply, as a constant, as an expression, or as another function. In reduction, a program is equivalent to its result in the same way that 2+2 is equivalent to 4. The main points of reduction are that 1) program structures, instructions, and arguments are all expressions, or functions; 2) there is no concept of updatable storage; 3) there are no sequencing constraints other than those implied by demands for operands; 4) demands may return both simple or complex arguments, such as a function.

reduction machine: Computer in which the requirement for a result triggers the operation that will generate it.

referential transparency: A principle which states that the replacement of an expression, or function, by its value is entirely independent of the context in which the function application appears.

shuffle-exchange: An interconnection network with the interconnect function defined as follows: first apply the shuffle function and follow it by the exchange function. The shuffle function may be defined as:

$$S(a_{n-1}\cdots a_1 a_0) = a_{n-2}\cdots a_1 a_0 a_{n-1}$$

where the number of processors is N; $A = a_{n-1}\cdots a_1 a_0$ is a processor address in binary, each a_i is a bit, and $0 \leq A \leq N-1$; and $n = \log_2 N$. The exchange function may be defined similarly as $E(a_{n-1}\cdots a_1 a_0) = a_{n-1}\cdots a_1 \overline{a_0}$ (the right most bit is complemented).

supercomputer: A loose term for an extremely powerful mainframe computer that provides high speed computing.

token: The operand value emitted by a node in a data flow graph.

transputer: A von Neumann computer implemented on a VLSI chip. A processor, 4K bytes local memory, four link interfaces for interfacing to other transputers, interfaces for accessing other devices, and system services such as reset and the clock are all packed onto a single chip. The transputers are programmed in a language called Occam. Each transputer in a system executes its own Occam program using its own local memory.

V operation: See P and V operations.

very large scale integration (VLSI): The fabrication of 100,000 or more gates on a single chip.

von Neumann computer: A computer based on the work of mathematician and computer designer John von Neumann. The computers are characterized by 1) a single computing element incorporating processor, communications, and memory, 2) linear organization of fixed size random-access memory cells, 3) a sequential, centralized control of computation. A machine instruction program is loaded sequentially in main memory and executed under the sequencing of a program counter.

APPENDIX B

LIST OF ACRONYMS

ADVAST	In the ILLIAC control unit, ADVAnced instruction STation.
ALICE	Applicative Language Idealized Computing Engine.
ARPA	Advanced Research Project Agency.
CE	In Alliant, Computational Element.
CIR	Current Instruction Register.
CPU	Central Processing Unit.
CU	Control Unit.
ECL	Emitter-Coupled Logic.
EDVAC	Electronic Discrete Variable Automatic Computer.
ENIAC	Electronic Numerical Integrator and Calculator.
FFT	Fast Fourier Transform.
FLOPS	FLoating Point Operations Per Second.
GFLOPS	Giga (one billion) Floating Point Operations Per Second.
HEP	Heterogeneous Element Processor
IAS	Institute for Advanced Studies computer.
IBM	International Business Machines.
ILLIAC	ILLInois Array Computer.
IP	In Alliant, Interactive Processor.
iPSC	Intel Personal SuperComputer.

IPU	In the HEP, Instruction Processing Unit.
LAN	Local Area Network.
LINPACK	LINear equations software PACKages.
LSI	Large Scale Integration.
MAR	Memory Address Register.
MBR	Memory Buffer Register.
MFLOPS	Mega (million) FLoating Point Operations Per Second.
MIMD	Multiple-Instruction-stream Multiple-Data-stream.
MIPS	Million Instructions Per Second.
MISD	Multiple-Instruction-stream Single-Data-stream.
MR	Mask Register.
MSIMD	Multiple Single-Instruction-stream Multiple-Data-stream.
PC	Program Counter.
PE	Processing Element.
PEM	In array processor, Processing Element Memory. In the HEP, Process Execution Module.
PSW	In the HEP, Process Status Word.
PT	In the HEP, Process Tag.
RAM	Random-Access Memory.
SDI	Strategic Defense Initiative.
SFU	In the HEP, Scheduler Function Unit.
SIMD	Single-Instruction-stream Multiple-Data-stream.
SISD	Single-Instruction-stream Single-Data-stream.
SSW	In the HEP, Scheduler Status Word.

TSW	In the HEP, Task Status Word.
VLSI	Very Large Scale Integration.
XR	indeX Register.

v

VITA

Phyllis Johnson Thornton

Candidate for the Degree of

Doctor of Education

Thesis: AN INTRODUCTION TO PARALLEL COMPUTER ARCHITECTURES

Major Field: Higher Education

Area of Specialization: Computing and Information Sciences

Biographical:

Personal Data: Born in Sherman, Texas, December 16, 1947, the daughter of Phillip and Fern L. Johnson. Married to Michael C. Thornton on June 5, 1971. The mother of Justin Glenn and Jennifer Dawn Thornton.

Education: Graduated from Yuma Union High School, Yuma, Arizona, in May 1966; received a Bachelor of Arts Degree in Mathematics with a minor in Physics from San Diego State University, San Diego, California, in June 1970; attended graduate school at San Diego State University from 1970 to 1974; completed the requirements for the Doctor of Education at Oklahoma State University in July, 1988.

Professional Experience: Teaching Assistant, Department of Mathematics and Computer Science, San Diego State University, September, 1970, to June, 1973; Part Time Instructor, Department of Mathematics, Arizona Western College, Yuma, Arizona, 1979 and 1980; Instructor and Assistant Professor, Department of Computer Science, Central State University, Edmond, Oklahoma, January, 1981 to present; current member of the Association for Computing Machinery and The Institute of Electrical and Electronics Engineers' Computer Society.