HIGH-LEVEL LANGUAGE CONCEPTS IN DATA FLOW ARCHITECTURE

By

TARANEH BARADARAN-SEYED

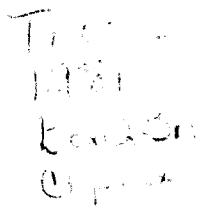
Bachelor of Science

Aryamehr University of Technology

Tehran, Iran

1976

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 1981



HIGH-LEVEL LANGUAGE CONCEPTS

IN DATA FLOW ARCHITECTURE

Thesis Approved:

Donald D. Fisher
Thesis Adviser

D. E. Helmich

J. R. Phillips

Norman N. Durham
Dean of Graduate College

PREFACE

This study is concerned with the aspects of data flow architecture. Asurvey of the data flow architecture proposed by Dennis and Misunas is presented. A survey is made of the semantic gap in the classical von Neumann architecture. Methods to represent high-level languages concepts in data flow base language are presented. Existing semantic gap in the data flow architecture is studied and methods to overcome this gap are discussed.

I wish to thank my advisor, Dr. Donald Fisher, for his invaluable guidance, assistance, and encouragement throughout this study. Also I express my appreciation to the other committee members, Dr. G. E. Hedrick and Dr. J. R. Philips, for their assistance and encouragement. Finally, I express my gratitude to my parents Mr. and Mrs. Baradaran-Seyed, and my husband, Bijan Karimi, for their support, patience, and encouragement.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

A. Introduction

The short history of computing as a science is unique in its unparalleled rate of technological growth. In response to this, the demand for greater levels of computing power has risen as rapidly. Anticipating the continuation of this trend, research in the area of parallel computation seeks to achieve high performance by manipulating programs to exploit the parallelism inherent in many problems.

It is well known that LSI technology is capable of economically producing large numbers of similar, small and complex devices. It is equally clear that use of LSI technology has not yet provided a breakthrough in the computing power available in a single system. Rather, the best that has been accomplished is simple reduction in the physical size of all familiar systems.

Many computing systems have departed from conventional computer organizations to improve capability for concurrent execution. A class of such processors belong to the category of SIMD (Single Instruction Multiple Data) machines. For instance, there are array processors represented by ILLIAC IV, associative processors like the

STARAN, and vector processors such as CDC STAR 100. These processors perform well only when the computation can be expressed in program and data structures which are easily mapped onto the particular machine structures. Array processors require that data structures be mapped onto a fixed structure imposed by the physical arrangement of the processors, such as a two-dimensional array. Associative processors require that data structures be linear lists of words so that associative operations on parts of these words can be efficient. For vector processors, data structures must be in the form of one-dimensional arrays to allow pipelining of operations on successive array elements. Furthermore, programs must exhibit a high degree of locality of reference such that a significant amount of data structure movement is not necessary during the execution.

There are concurrent processors that belong to the category of MIMD (Multiple Instruction Multiple Data) machine. A typical realization of this form of machines is based on multiple processor and shared multiple memory organization. The predominant problem of these processors is that the system performance is based on the assumption of locality of reference achieved by a programmers explicit partitioning of a computation. Furthermore, because the semantics of the languages supported by these systems are based on the notion of sequential execution and operations which have side-effects, concurrency is achieved through careful analysis of programs to prevent possible deadlocks

and bottlenecks in memory references.

A number of inadequacies may be noted with currently proposed and operational multi-processor computers, including:

1. the poor utilization of program parallelism by the architecture,
2. an incompatibility in the way that these architectures and their programming languages represent parallelism,
3. the difficulty of programming the computers using conventional languages.

When a closer examination is made of multi-microprocessor systems, it is possible to identify three problem areas in their design:

1. the possible contention of concurrent processes for the physical resources (processors, memories, input-output) of the computer,
2. the difficulty of partitioning the programs to be executed so as to maximize the utilization of the resources provided,
3. the need to supply mechanisms so that concurrent processes may interact to communicate data and synchronize their operation.

The conventional approach to multi-microprocessor systems is to base their design on extensions to the inherently sequential "control flow" or von Neumann concept of a stored program computer. This organization, however,

may be inapplicable for multiprocessor computers. This design has some architectural deficiencies which were studied by Myers :28: in 1978. These problems contribute in a phenomenon known as the semantic gap. The semantic gap shows the difference between the concepts in computer architecture and high-level languages and causes software unreliability, performance problems, excessive program size, and compiler complexity.

Two particularly troublesome attributes of the von Neumann model are sequential control and memory cells. Sequential control is troublesome since it prohibits the asychronous behavior and distributed control that is essential to a multiprocessor. It also burdens the programmer with the need to explicitly specify exactly where concurrency may occur. The concept of a memory cell, along with the idea of assigning a value, presents a difficulty since its existence forces the programmer to consider not only what value is being computed, but also where that value is to be kept.

An alternative organization, namely, dataflow, exists. In this organization:

1. an instruction executes when and only when all operands needed for that instruction become available;
2. instructions, at whatever level they might exist, are purely functional and produce no side effects.

Data flow computation is therefore "data driven" as opposed

to "control driven" as exemplified by the conventional von Neumann machines. A data flow program may be represented as a directed graph with certain restrictions on interconnections between nodes. The nodes of the graph represent instructions and the directed arcs represent paths for operands. Data flow language is asynchronous except when synchronization is explicitly specified, and in which values are the subject of computation rather than the locations where those values are kept (i.e., no memory addresses). An asynchronous language assumes computations are unrelated, and thus concurrent, unless otherwise specified. The absence of memory cells ensures that only simple control mechanisms are needed to consider access to data, since races to "store" data never occur. Such a semantic basis should work well with a machine composed of many asynchronous cooperating processors.

This report dicusses the basic concepts of data flow architecture proposed by Dennis [11, 12, 13, 14]. It also includes a study of problems that occur in von Neumann architectures known as semantic gap [28]. Then the representation of high-level language concepts in data flow base language are studied and coded. Also the semantic problems in this architecture are dicussed. Finally, two application processes, Discrete Fourier Transform, and SIN function are studied and coded in data flow base language.

B. Literature Review

The theoretical basis for the data flow architecture was established during the 1960s. In 1975, a preliminary architecture for a basic data flow architecture was proposed by Dennis [12]; this machine executes programs coded in data flow base language proposed also by Dennis [13]. Information flow in the Dennis architecture is done through packet communication features presented in 1975 [14]. Misunas extended this model to make it suitable for handling data structures [24, 26] and published a performance analysis of the machine [25]. In 1977, Arvind and Gostelow proposed a data flow architecture [6], and both a high-level data flow programming language and a base machine language [5, 7]. Miranker [23] presented a method to implement procedures on a class of data flow processors; and Rumbaugh [30] presented a detailed data flow multi-processor.

In 1978, a structure processing facility for data flow computers was proposed by Ackerman [1]; and an asynchronous programming language and computing machine was presented by Arvind, Gostelow, and Plouffe [5]. Davis proposed a recursively structured data-driven machine called DDM1 (Data Driven Machine #1) [10]. Design of an arithmetic processor compatible with a data flow computer was proposed by Feridum [17] in 1978. An architecture for a loosely-coupled parallel processor was presented by Keller, Lindstrom, and Patil [21]. Software for a data flow computer proposed by Arvind was developed by Thomas [34]. Additional research

was conducted Manchester University by Treleaven [36].

In 1979, a high-level language [2], an intermediate form [22], and a machine language set were devised for the M.I.T data flow architecture; the Manchester data flow architecture was improved [16, 18]; and the Texas Instruments research group proposed and built the first computer using the data flow concepts [19, 20, 32].

In 1980, a data flow architecture with tagged tokens was proposed by Arvind, Kathail, and Pingali [4]. Safety and optimization transformations of data flow programs was studied by Montz [27]. Semantics of data-driven loops was analyzed by Ruth [31]. Thomas [34] presented a performance analysis of two classes of data flow computing systems.

CHAPTER II

BASIC BACKGROUND FOR THE EXISTENCE OF SEMANTIC GAP

In 1978, Myers [28] proposed a new approach to the study and design of computer architectures in his book. The main premise of Myers book is that the architectures of most computing systems have not been designed according to the computational and structural needs of high-level languages. Rather than taking a global look at system functions and its hardware/ software tradeoff, most architects have based their designs on tradition and the bottom-up view of "minimize the cost of hardware and let the programmers solve all the difficult problems". Most of the shortcomings caused are attributable to a phenomenon known as the semantic gap.

A. Semantic Gap

The semantic gap is a measure of the difference between the concepts in the high-level languages and the concepts in the computer architecture. Most current systems have an undesireably large semantic gap in that the objects and operations reflected in their architectures are rarely closely related to the objects and operations provided in

the programming languages and used with them. This semantic gap contributes to software unreliability, performance problems, excessive program size, and compiler complexity, all of which contribute negatively to the economics of data processing.

To understand the presence of the semantic gap, the major and heavily used concepts in high-level languages (PL/I, COBOL, FORTRAN) and a computer architecture can be picked up and the relationship between the two can be studied. As an example, we analyze PL/I and the IBM S/370. The example is not PL/I oriented, however, since most or all the PL/I concepts discussed also exist in such languages as COBOL, FORTRAN, and ALGOL. Neither is the example S/370 oriented; the S/370 was selected because it is representative of most conventional architectures.

The following is a list of a few major and heavily used concepts in PL/I (or any other language for that matter). The question for each is determining to what S/370 (or most other architectures for that matter) concepts it is related.

A.1. Arrays

The array is the most frequently used language data structure. PL/I provides such concepts as multidimensional arrays, performing operations on entire arrays, referencing cross-sections (sub-arrays within arrays), and, the option of, ensuring that subscripts do not fall beyond the bounds of the array dimensions. The question is, what S/370

concepts directly relate to these concepts? the answer is, very few. The only architectural concept that seems indirectly related in a primitive way is the concept of index registers. This means that it is left to the compiler to create the widely used concept of an array out of the rather distant S/370 instruction set.

A.2. Structures

A second frequently used data concept is the structure, a collection of heterogeneous data elements (also known as a record in some programming languages). One finds absolutely nothing in the S/370 that is related to structures and operations performed on structures.

A.3. Procedures

The basic program structure in PL/I is the procedure (subroutine). A procedure call entails saving the state of the calling procedure, dynamically allocating and initializing local storage for the called procedure, transmitting arguments, and beginning execution of the called procedure. One finds next to nothing in the S/370 that corresponds to these concepts. One exception is the branch-and-link instruction, but this contributes so little to the procedure-call operation (one of many instructions that must be executed) that its absence would never be missed (the compiler could just as easily generate two instructions, load-address and branch-register, in its

place).

A.4. Data Representation

PL/I has decimal and binary fixed-point data representations (integer,fraction). The S/370 has none, but it does have decimal and binary integer representations out of which the compiler must create the fixed-point concept. PL/I decimal numbers can contain anywhere from 1 to 15 digits, but the S/370 can only represent decimal numbers with an odd number of digits. PL/I binary numbers can contain anywhere from 1 to 31 binary digits, but the S/370 provides for only binary numbers of 15 or 31 digits. PL/I floating-point numbers can be declared as having 1 to 53 digits of significance, but these must be mapped into one of three fixed-size S/370 representations.

This discussion could be continued indefinitely by looking at other PL/I concepts such as string processing, block structures, controlled storage (a push-down stack concept), generic procedure calls, program-tracing functions, and automatic data conversion, but by now there is an understanding of the semantic gap between high-level -language concepts and current computer architectures. The cause of large semantic gaps is more difficult to discover, but the usual causes are bottom-up system design and the computer architects lack of knowledge and appreciation of programming languages, what programs do, what programmers do, the difficulty of program debugging , and the causes and

consequences of software errors.

Given the existence of this large semantic gap, the next step is to discuss some of its consequences.

B. Consequences Of Semantic Gap

B.1. Software Unreliability

The semantic gap is a significant contributer to software unreliability in the sense that a large set of programming errors that theoretically could be prevented or detected by the computing system are not prevented or detected in current systems. A few examples suffic .

One common programming error that arises under a large variety of circumstances is a reference to a variable that has an undefined or unset value. This error is not detected by most current systems; since execution continues using some unpredictable value, the error is difficult to debug. Although some instances of the error could be detected at compilation time by doing a flow analysis of the program, in general it cannot be detected until execution time. Since conventional machines have no way of distinguishing a defined variable from an undefined one, the architects have, in effect, deferred the problem to the compiler writer. The compiler writer finds no easy and efficient solution to the problem; thus he or she defers the problem to the application programmer.

Some compilers have attempted to solve the problem, but the solution has turned out to be complicated, inefficient,

and not foolproof. For instance, IBM's PL/I Checkout compiler initializes all character strings with hexadecimal FE characters and all fixed-point binary numbers with the smallest negative number and then checks for these values whenever these variables are referenced. However, not only does this add overhead (execution time and storage), but it can cause "errors" to be detected in correct programs and does not cover all data.

A second common error is referencing an array element where one of the subscripts falls beyond the bounds of the corresponding dimension. Again, since the conventional machine does not recognize the structure array, the problem is deferred to the compiler writer. The compiler writers see no easy solution, thus the problem is ignored or the decision is left to the application programmer by making the check optional.

As an example of the overhead of this software check, IBM s PL/I optimizing compiler normally generates 17 machine instructions (occupying 62 bytes of storage) for the statement

```
C(i,j) = A(i,j) + B(i,j);
```

when A,B, and C are arrays of fixed-binary elements of identical size. If the optional SUBSCRIPTRANGE check is enabled, the compiler generates 75 machine instructions (274 bytes), and 57 of these instructions would be executed if the subscripts were within the array bounds.

B.2. Performance Problems

The large semantic gap also leads to significant performance problems because of the large number of instructions that must be generated by the compiler to implement the language concepts out of the rather primitive machine-instruction repertoire. This has a negative effect on performance because it increases the amount of information that must be transmitted between storage and the processor, and this has been found to be a good first-order measure in comparing the performance of different machines.

Since this effect is not widely understood, it is worthwhile to look at a simple example. Assume that we wish to add two 100 by 100 element fixed-binary PL/I arrays together. Hopefully we would write this as A=A+B; (writing nested DO loops to accomplish this is much more inefficient). IBM s S/370 optimizing compiler generates efficient object code for this statement: six instructions followed by a loop of four instructions executed 10,000 times. The number of 32-bit words that must move between memory and the processor is 40,004 (the instruction; the first six instructions fit into four words, and the loop body occupies four words) plus 30,003 (two data fetches and one store for the element plus a few additional fetches), for a total of 70,007.

Although this example applies only to array operations, one can find analogous examples in the excessive number of instructions generated to implement almost every

programming-language concept on a conventional architecture.

B.3. Excessive Program Size

The large semantic gap affects program size in the same way. For instance, it was seen earlier that it takes 62 bytes of storage to represent the statement

$$C(i,j) = A(i,j) + B(i,j)$$

if no subscript checking is done and 274 bytes if subscript checking is desired. In addition to being a problem itself, excessive program size is another contributing factor to system performance problems (e.g., in a virtual storage system, by increasing the programs, working-set sizes and thus increasing the number of page faults incurred).

B.4. Compiler Complexity

From the previous two points, the effect of the large semantic gap on compilers should be obvious; the code-generation portion of compilers must be extremely complex to generate code that bridges the semantic gap as efficiently as possible.

C. A Critique of the Conventional von Neumann Architecture

The basic reason for the existence of the large semantic gap in current systems is that most architectures are simply modifications of the von Neumann architecture

derived in the 1940s. This is not to imply that the von Neumann architecture was not a stroke of genius when it was developed. However, the world has changed tremendously since the 1940s. The feasibility of even constructing electronic computers was still in doubt at that time, and hardware costs and reliability were of utmost concern; thus the motivation was to design as primitive a processor as possible. Also, factors that are taken for granted today, such as high-level programming languages and the sophistication and critical nature of most computing applications, were not even envisioned at that time.

It is common today to talk of a class of machines as von Neumann machines and to say that most current machines belong to this class. A von Neumann machine is said to have these properties:

1. A single sequential memory. A program and its data are stored in a single memory and the memory is referenced with sequential (0,1,2,...) addresses.

2. A linear memory. The memory is one-dimensional, that is, it has the appearance of a vector of words (or bytes).

3. No explicit distinction between instructions and data. One can, for instance, treat an instruction as data (e.g., modify it), add an instruction to a data word, or branch to a data word and execute it as if its bits represent an instruction.

4. Meaning is not an inherent part of data. There is nothing, for instance, that explicitly distinguishes a set

of bits representing a floating-point number from a set of bits representing a character string. Rather, the meaning of data is assigned by program logic. If a machine fetches a floating-point add instruction, it assumes that the operands represent floating-point numbers and performs a floating-point addition with the operands. Hence one can perform a floating-point addition on two operands that actually represent a character string or an address.

Although the von Neumann architecture was a reasonable design for the first stored-program computer, it is alien to the execution of programs written in high-level languages. Internal structures of data in high-level languages are distinguished from von Neumann machines by the following:

1. Storage, as represented in high-level languages, consists of a set of discrete named variables. With the exclusion of certain questionable language constructs (e.g., the FORTRAN COMMON area) there is no concept of one variable being "next" to another variable. There is no reason to believe that the variables in one subroutine are located in the same storage device as the variables in another subroutine. In short, the concept of a single sequential storage bears little resemblence to the concept of storage in programming languages.

2. programming languages deal with multidimensional, not just linear, data types (e.g., arrays, structures, and lists).

3. In programming languages there is a sharp

distinction between data and instructions. In a high-level language, there are no concepts of executing data or referencing instructions as if they were data.

4. In a high-level language, meaning is an inherent part of data. One does not write a PL/I program as

```
DECLARE A WORD;
DECLARE B WORD;
A = A "floating-point add with" B;
```

Instead one writes

```
DECLARE A DECIMAL FLOAT (6);
DECLARE B DECIMAL FLOAT (6);
A = A + B;
```

That is, in high-level languages the meaning of the data is associated with the data itself, and the operators are generic (i.e., the meaning of "+" is determined by examining the attributes of its operands).

Thus the attributes of a von Neumann architecture are not related, and are even contradictory, to the concepts in languages. Intutively, one can observe that a von Neumann machine is a poor vehicle for the execution of high-level-language programs because

1. Excessive mapping is required in software (i.e., by the compiler in the form of compiler-generated code) to match the language concepts to the von Neumann view of storage. This has been referred to as "absorbing the structure (of the data) into the logic of the program". This should be apparent to anyone who has examined the

output of a compiler; the amount of code generated by the compiler to map the language concepts of storage and data to the underlying architecture usually greatly outweighs the amount of problem-solving code generated.

2. A von Neumann machine is excessively overgeneral (e.g., one can use a word that has no currently defined value, address anything in storage, add a character string to an instruction); since this generality fortunately is absent from programming languages, the compiler (and its generated code) is left with the task of removing the generality and ensuring that it does not interfere with the definition of the language.

3. Because the concept of storage in a von Neumann machine is rather primitive, the operations (instruction set) performed by the machine are constrained to be equally primitive.

D. Other Undesirable Features Of Classical Architectures

Although the von Neumann model is the major cause of the large semantic gap, there are additional undesirable architectural properties of current systems that contribute to the gap.

D.1. Binary (Base Two) Arithmetic

In current machines, binary arithmetic is treated as almost sacred, but it almost goes without saying that humans

find base-two arithmetic quite distasteful. Since proposals for decimal arithmetic often evoke emotional arguments, it is worth exploring the traditional arguments for and against decimal arithmetic.

Two arguments may be presented in favor of decimal arithmetic. First, since todays computing environment is highly input/output oriented and since few, if any, people would consider forcing human beings to communicate with computers in base-two terms, current systems waste an enormous amount of time performing conversions between decimal and binary representations. Second, the fact that a machine represents numbers in base-two form cannot be hidden completely from the human, since, for instance, most rational decimal fractions are represented as infinite-digit base-two fractions. This means that finite-length base-two numbers are often approximations of decimal numbers, a source of programming difficulty, programming errors, and confusing language definitions.

The traditional arguments against decimal arithmetic are that it is slower than binary arithmetic and that binary numbers can be stored more compactly than decimal numbers.

The two arguments against decimal arithmetic are subject to question. First, one must weigh the speed of arithmetic algorithms against the overhead of converting decimal numbers to binary and back again. Second, decimal arithmetic circuits have been devised that are competitive with binary circuits in terms of speed and only slightly

less competitive in terms of cost. The second argument (space) has some merit, but it is not insurmountable.

D.2. Fixed Size Storage Words

In an architecture with fixed-size storage words, deciding on the word size is probably the most difficult tradeoff facing the architect. If the word size is too small, the maximum value of numbers that can be represented is too small, fractional (e.g.,floating-point) numbers are excessively imprecise, and larger addresses are needed. On the other hand, larger words tend to waste storage, because studies of the distributions of data values in programs indicate that values are not uniformly distributed; they are heavily skewed in favor of small values (e.g.,the values zero and one are common, the values in the ranges 10-20 are more common than values in the range 59470-59480). Hence large words waste storage because their high-order bits or digits are likely to be zero.

The second problem with fixed-size words is that many languages (e.g., PL/I and COBOL) allow the programmer to declare the size of each variable, and the possible sizes usually vary over a large range (e.g., a PL/I decimal floating-point variable can be declared as having anywhere from 1 to 53 mantissa digits). If the compiler is able to accurately map this concept into a fixed-size-word machine. Performance problems (excessive generated code) are a likely consequence. If the compiler designer decides that the

concept of variable-size data cannot be efficiently and accurately mapped into fixed-size words, the underlying machine architecture shows through and distorts the language. Of course, one might argue that languages should not contain this concept, but the argument has little validity. The concept assists one in defining machine-independent languages, allowing programs to be transferred from one machine type to another.

D.3. Registers

Another concept that is alien to the concepts in programming languages is the presence of program-addressable register (e.g., the concept of general-purpose registers in the S/370). If the machine requires the use of registers for all arithmetic operations and if the number of registers is small (both are the case in most machines), the compiler is left with the task of generating code to manage the registers and optimize their use. This code is extraneous in that it contributes nothing toward the expression of the source program's logic.

Since the 1950s, except for a few machines (e.g., some made by Burroughs Corp.), there have been no advances in the computer architectures of current systems. However, there have been some advances in the implementation of particular architectures exploiting the inherent parallelism in operations.

During 1970s, a new approach in computer architecture

was proposed by Dennis and others [11, 12, 13, 14]; it is known as data flow architecture. This approach is a radical change from the traditional von Neumann architecture and is a well designed system to perform parallel processing. The data flow approach changed the process of selecting the instruction for execution, and consequently, other related concepts have been changed as follows:

1. Execution of instructions is based on their readiness for execution instead of their location in the program. In this approach any instruction may be executed as soon as all its operands become available.

2. There is a distinction between instruction and data. Instructions are located in a special memory called an instruction memory, constants reside in an instruction cell, and variables are either a portion of the instruction cell or float in the architecture as results. Data may not be treated as instructions and vice versa.

3. Instruction memory contains both instructions and simple variables , data structures are held in a separate memory called structure memory.

4. Data structures (arrays, matrices, ...) are stored in structure memory as binary or n-ary trees, that consequently, makes most of the existing methods to implement and handle data structures invalid.

5. Meaning is an inherent part of data. Data items contains a type tag which specifies its meaning.

CHAPTER III

BASIC BACKGROUND FOR DATA FLOW

A. Architecture of Parallel Systems

Highly parallel computer systems have evolved in a manner which often necessitates the placing of unusual constraints on program and data. Parallel machines such as ILLIAC IV and the CDC STAR can realize their full potential only for data represented in array or vector formats.

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

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

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

The pipelining of an arithmetic operation distributes the performance of the operation over time rather than

space. That is, rather than utilizing several functional units of a specific type to increase the processing rate, one larger functional unit is employed, and the operation is broken into a number of smaller operations which are performed simultaneously upon a stream of values. Although the performance of a single operation can actually take longer in a pipelined functional unit, the fact that a large number of operations are being performed concurrently can produce a very high processing rate.

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

Dependencies between successive instructions of a process complicate attempts to utilize pipelining for the instruction stream of a processor. For example, the execution of an instruction which references a memory cell modified by a previous instruction must await the completion of the previous instruction. An instruction pipeline must detect dependencies dynamically. When it finds a dependency, it must either stop accepting new instructions or rearrange the order of execution; in either case, the degree of concurrency is reduced or the pipeline becomes

complicated.

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

The use of instruction lookahead in a processor allows the exploitation of multiple arithmetic units by decomposing the instruction stream into independent elements. For example, consider the arithmetic expression A+B + (C*D). The two computations A+B and C*D can be performed simultaneously in separate functional units. The IBM 360 model 91 and the CDC 6600 have developed techniques for exploiting this property for short instruction sequences; however, once again, any branching in the program disrupts the flow of instructions to the functional units and decreases the processing capability of the architecture.

In illustration of the problems encountered in

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

The methods of structuring multiple processor systems and improving the performance of a processor all have serious drawbacks to the full exploitation of the capabilities of the processors. In this regard , data flow approach offers attractive solutions to many of these problems.

B. The Dataflow Approach

Studies of concurrent operations within a computer system and of the representation of parallelism in a programming language have yielded a new form of program representation, known as data flow. Execution of a data flow program is data-driven; that is, each instruction is enabled for execution just when each required operand has been supplied by the execution of predecessor instructions.

In order to take advantage of the parallelism inherent

in an elementary data flow representation, the architecture of the elementary data flow processor was developed by Dennis and Misunas [11, 12, 13, 14, 24, 25, 26].

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

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

The processor does not utilize an instruction register or instruction decoder in the von Neumann sense; an instruction proceeds on its own when its operands are ready and delivers its results to other instructions which are waiting for them. No software operating systemis necessary within the architecture [24]. Processor allocation, the formation of instructions into streams for the functional units, and the transfer of information between levels of

memory is efficiently accomplished by the hardware of machine.

The exploitation of data dependencies in programs has been investigated previously, indeed, such is the goal of the lookahead techniques utilized in architectures such as the IBM 360/91 and the CDC 6600. The approach taken in the data flow processor differs from these approaches in that it utilizes a radically different concept of computer organizations which offers attractive solutions to many of the problems encountered in adapting von Neumann machines for parallel computation, an architecture in which parallelism and concurrency are inherent in the structure of the processor.

C. The Data Flow Language

The data flow language presented in this section serves as the base language for the architecture to be described in the next chapter. The semantics of the language is developed by Misunas.

In order to represent the exact serial/parallel nature and existing inherent instruction level parallelism of the program, the directed graph representation has been selected as an alternative to the traditional serial list of instructions. The longest path through the graph is the critical path which is the ultimate limit on the speed of execution no matter how many parallel processors are available. The width of the graph represents the program

parallelism at that point.

A directed graph consists of nodes that represent the operations to be done and links that show how results move from operation to operation. A directed graph node denotes an operation to be executed and is not involved with the sequencing mechanism. Therefore, the internal contents of a node (opcodes, operands, subroutine calls, etc.) are directed by the hardware implementation of the processor independent of the mechanisms that sequences that node. A directed graph link denotes movement of data between nodes and is crucial to any sequencing mechanism based upon the flow of data. Therefore, links are logically pointers associated with each node.

Execution of a directed graph follows the flow of data through the graph (hence, data flow). No instruction can start execution before all of its inputs arrive; no instruction must wait after its inputs and a processor are available. Data flow sequencing guarantees only the minimum constraints necessary to assure logically correct execution. As soon as an instruction can correctly execute, it is flagged ready for execution. All ready instructions can be executed in parallel, if a sufficient number of processors is available.

As soon as a result is calculated and available, it is immediately forwarded to each of the succeeding instructions that need it. An instruction never has to fetch its operand. All input operands are collected into the body of

instruction before it begins execution. Therefore, there is no extra operand fetch time needed after instruction fetch. The memory accesses needed to update results are done by dedicated hardware in parallel with useful work. The pending instruction list allows the next instruction fetch to be overlapped with execution.

C.1. Elements

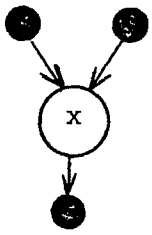
The data flow language is composed of two kinds of elements, called actors and links. An actor of language can be one of the following:

- operator
- decider
- gate

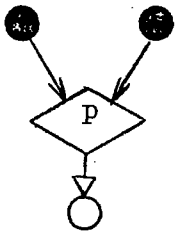
which are represented in Figure 1.

Each actor has a number of input arcs which supply values necessary for its execution and one output arc upon which results are placed. A small dot or circle represents a link which has one input arc upon which it receives results from an actor and a number of output arcs over which it distributes copies of the result to other actors (Figure 2).

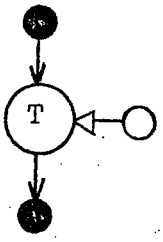
Values are conveyed over the arcs of the program by tokens which are represented by large solid dots. An actor with a token on each of its input arc, and no token on its output arc, is enabled and somtimes later will fire, removing the tokens from its input arcs, computing a result



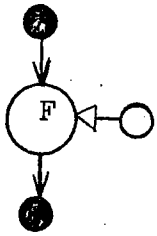
(a) operator



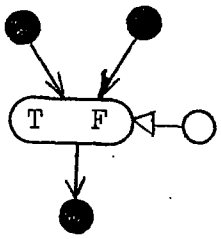
(b) decider



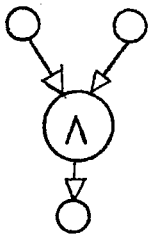
(c) T-gate



(d) F-gate



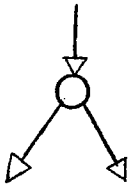
(e) MERGE



(f) boolean operator

Figure 1. Actors of the Data Flow Language

(a) data link



(b) control link

Figure 2. Links of the Data Flow Language

using the values carried by the input tokens, and associating the result with a token placed on its output arc. In a similar manner, a link is enabled when a token is present on its input arc, and no token is present on any of its output arcs. It fires by removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs. The data flow language utilizes two types of tokens:

- data tokens
- control tokens

A data token carries a data value which is produced by an operator (Figure 1a) as a result of some arithmetic operation. A control token is generated at a decider (Figure 1b) which, when the decider receives a data value on each input arc, applies its associated predicate and produces either a true-or-false-valued control token on its output arc.

Control tokens direct the flow of data tokens by means of either a T-gate, F-gata or MERGE actor (Figure 1c,d,e). A T-gate passes a value on its output arc if it receives the value true at its control input arc; the received data value is discarded if false is received. The merge actor allows a control value to determine which of two sources supplies a data value to its output arc. If the control value false arrives at the control arc, the merge passes on the value present or next to arrive at the false-input arc. A value

present at the true-input arc is left undistributed. The complementary action occurs for the control value true.

C.2. Structures

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

$$E = T, I, R, Q$$

where :

T = truth values

I = integers

R = reals

Q = strings

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

$$S = (s1,v1),(s2,v2),.....(sn,vn)$$

where:

$si \in I \cup Q$

$vi \in E \cup S \cup nil$

and si is the selector of node vi. An elementary node has no emanating arc; rather, an elementary value is associated with the node. A node with no emanating arcs and no associated elementary value has value nil. A structure value is represented by a data token carrying a unique pointer to the node of the structure. In Figure 3 the structure contains three elementary values a,b, and c, designated by the simple selector L and the compound selectors R.L and R.R respectively. Structure node C of structure A is shared with structure B and is designated by a different selector in B than in A.

A simple selector associated with a node can be either an integer or a string consisting of letters L and R (indicating left and right respectively). A compound selector is formed by the concatenation of a number of simple selectors and specifies a path through the structure which can be followed by applying the simple selectors in the stated order.

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

In order to generate and perform operations upon structure values, a number of new actors must be defined. Structures are created through use of the CONSTRUCT actor

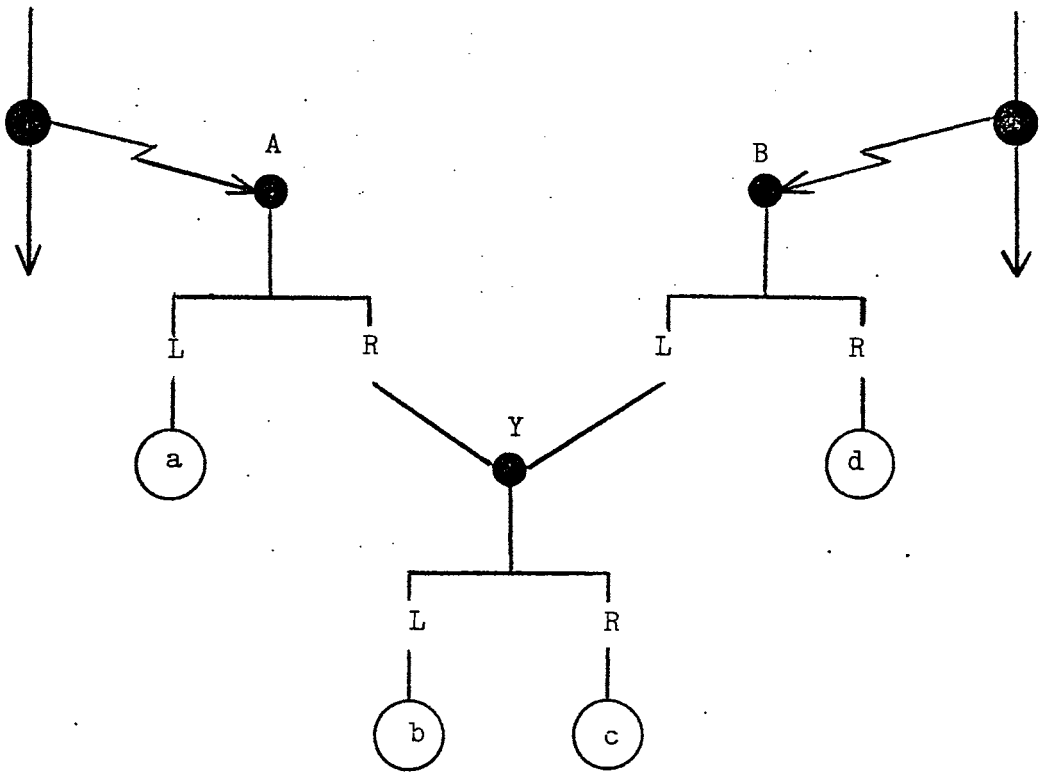


Figure 3. An Example of Two Structures Sharing a Common Substructure

(Figure 4). The actor accepts an elementary or structure value from each input and places on its output a structure containing the input values as components. Each input is labeled with the selector in the new structure to be associated with the value arriving on that input.

A value is retrieved from a structure by a SELECT actor (Figure 5). The value in the input structure designated by the selector argument is placed on the output of the actor. The result can be either an elementary value or a structure value. If the argument of the actor is a multiple selector, the actor produces on its output the value at the end of the path designated by the multiple selector. The action of the actor is undefined if the input structure does not contain the specified selector(s).

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

The structure produced by the firing of an APPEND actor is a version of the input structure which contains a new or modified component (Figure 6). If the specified node of the input structure has a selector corresponding to the selector argument of the actor, the value designated by that selector in the new structure is the input value. Otherwise the specified selector-value pair is added to the node of the new structure. Identical elements of the input and output

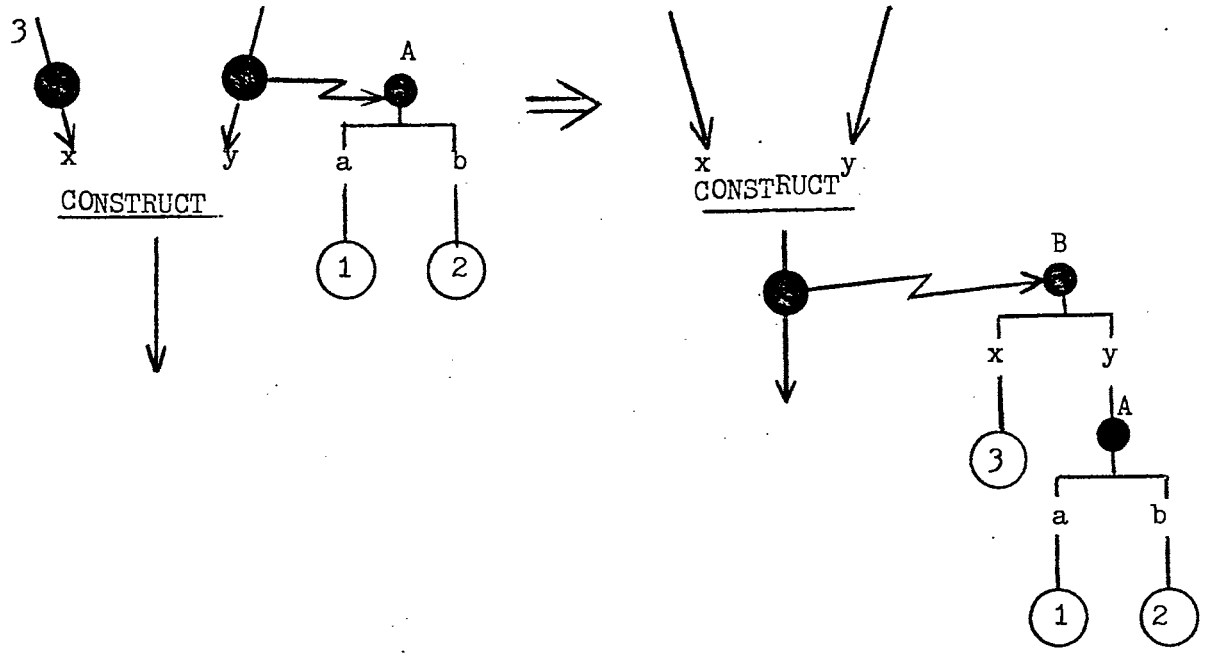


Figure 4. Operation of the CONSTRUCT Actor

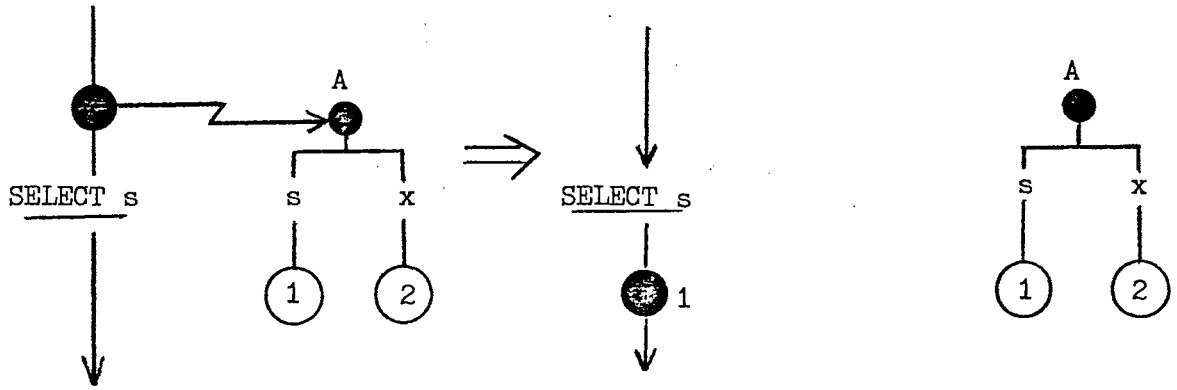


Figure 5. Operation of the SELECT Actor

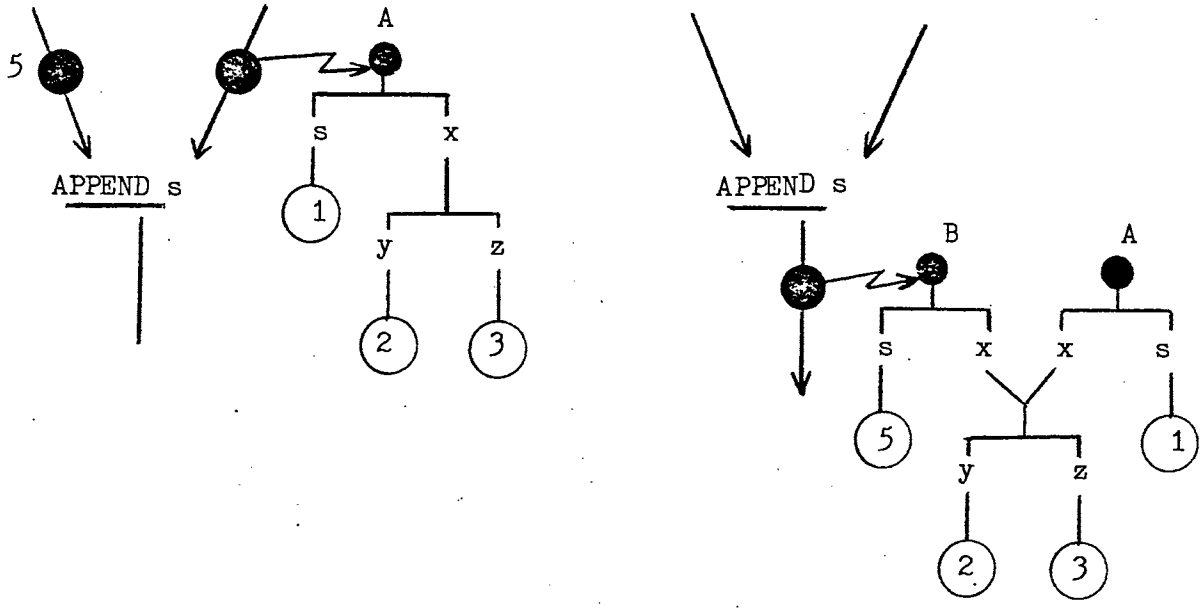


Figure 6. Operation of the APPEND Actor

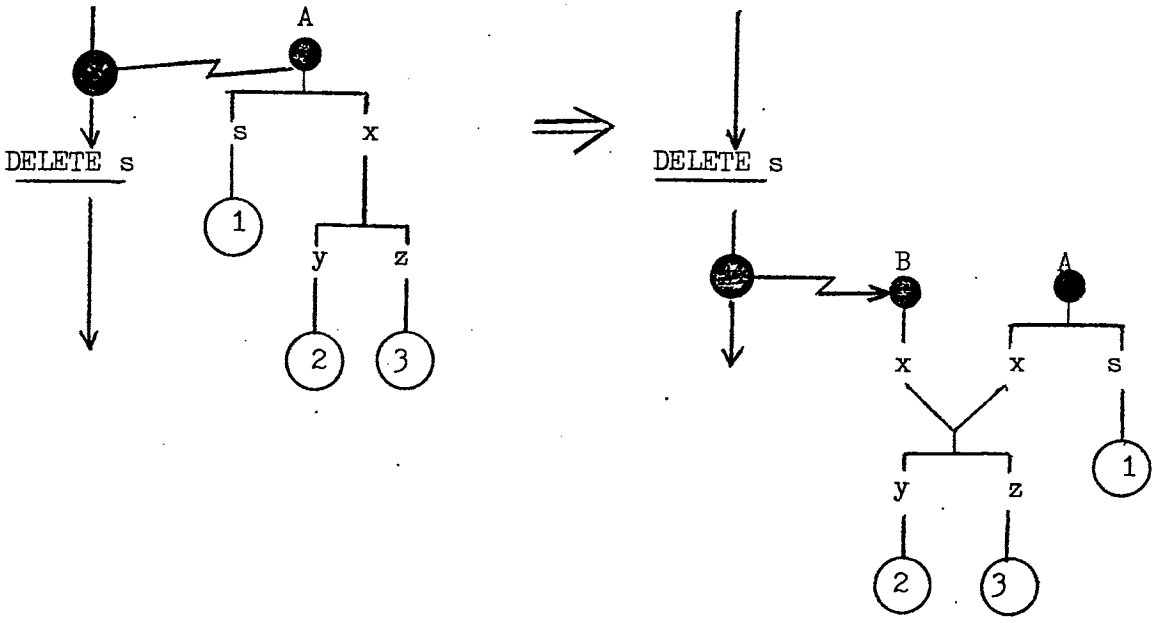


Figure 7. Operation of the DELETE Actor

structures are shared between the two structures.

In a similar manner, the structure appearing on the output arc of a DELETE actor is a version of the input structure in which the specified node contains one fewer component (Figure 7). The specified node in the new structure is missing the selector-value pair designated by the selector argument. As with the APPEND actor, identical elements are shared between the input and output structures.

C.3. Data Flow Procedure Representation

Procedures of the language are represented as acyclic directed graphs in a manner which is very attractive from both a semantic viewpoint and an implementation viewpoint. A data flow procedure is a data flow program with a single input arc over which the argument arrives and a single output arc upon which result is placed. The body of a procedure is represented as a data structure, and the procedure is referenced by a token carrying a pointer to the structured representation. Every procedure in the language is determinate that is, the same result is produced by every activation of the procedure which receives the same input values.

To provide for procedure activation and termination, the APPLY and RETURN actors are introduced into the data flow language. The operation of these actors is shown in Figure 8. The APPLY actor receives two inputs, a procedure and an argument, which may be either an elementary value or

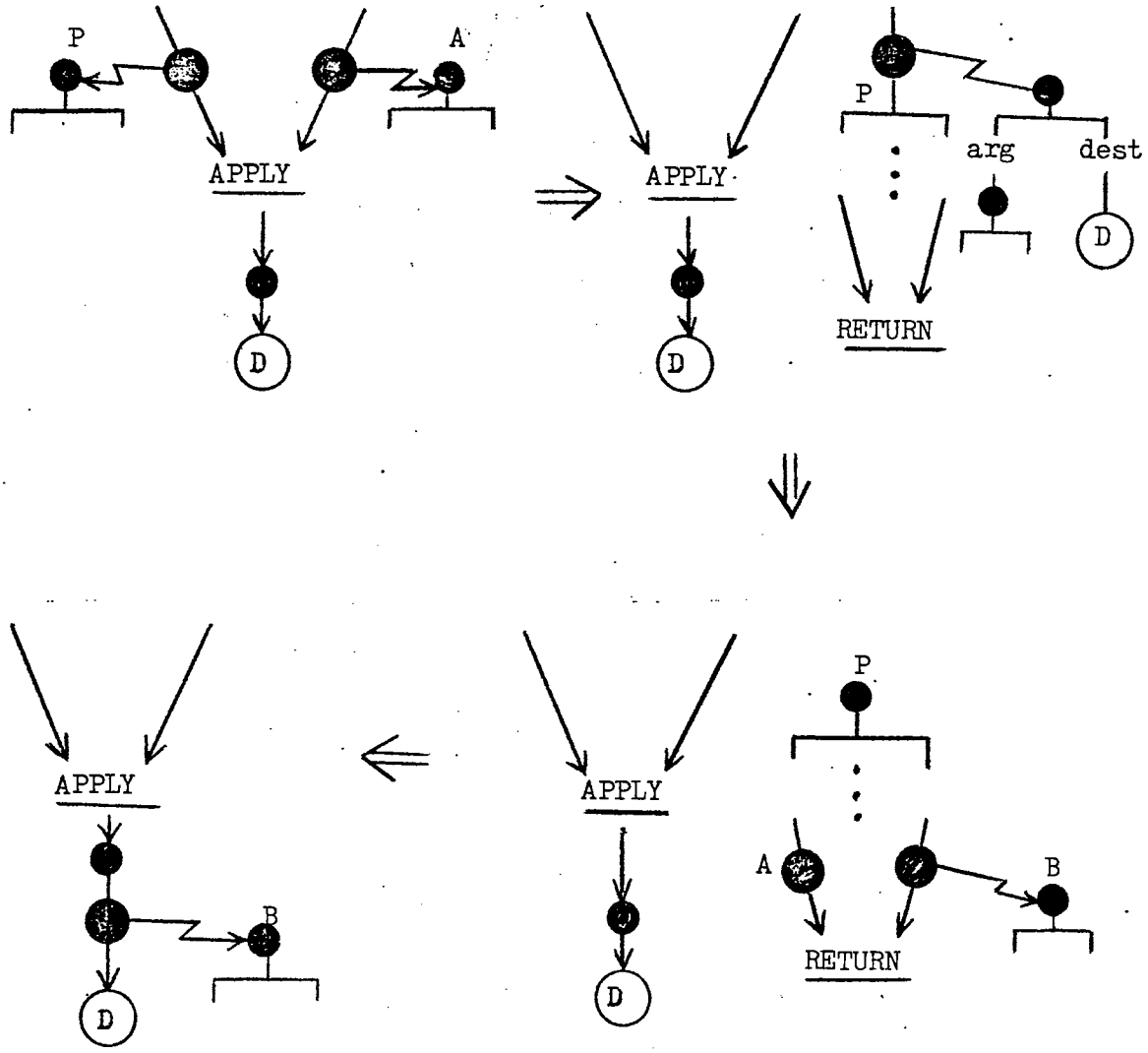


Figure 8. Operation of the APPLY and RETURN Actors

a structure value. Upon firing, the actor creates an argument structure of the argument and the destination for the result of the application, and this argument structure is given to the procedure as input. If no instruction follows the APPLY actor in the program, the value designated by the destination selector in the argument structure passed to the procedure is nil. Upon completion of the execution of the procedure, the result is sent to the specified destination by a RETURN actor within the procedure body.

The data flow representation of the following simple procedure is shown in Figure 9:

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

When the procedure of Figure 9 is applied, it receives on its input arc a structure containing two elements. The first element, designated by the selector arg, is the argument x of the procedure. The second element, dest, is the destination address for the result. The procedure shown in Figure 9 has been called with the argument 5 and the destination D.

The first operations performed by the procedure are select operations which send the argument to the procedure body and the destination address to the return instructions.

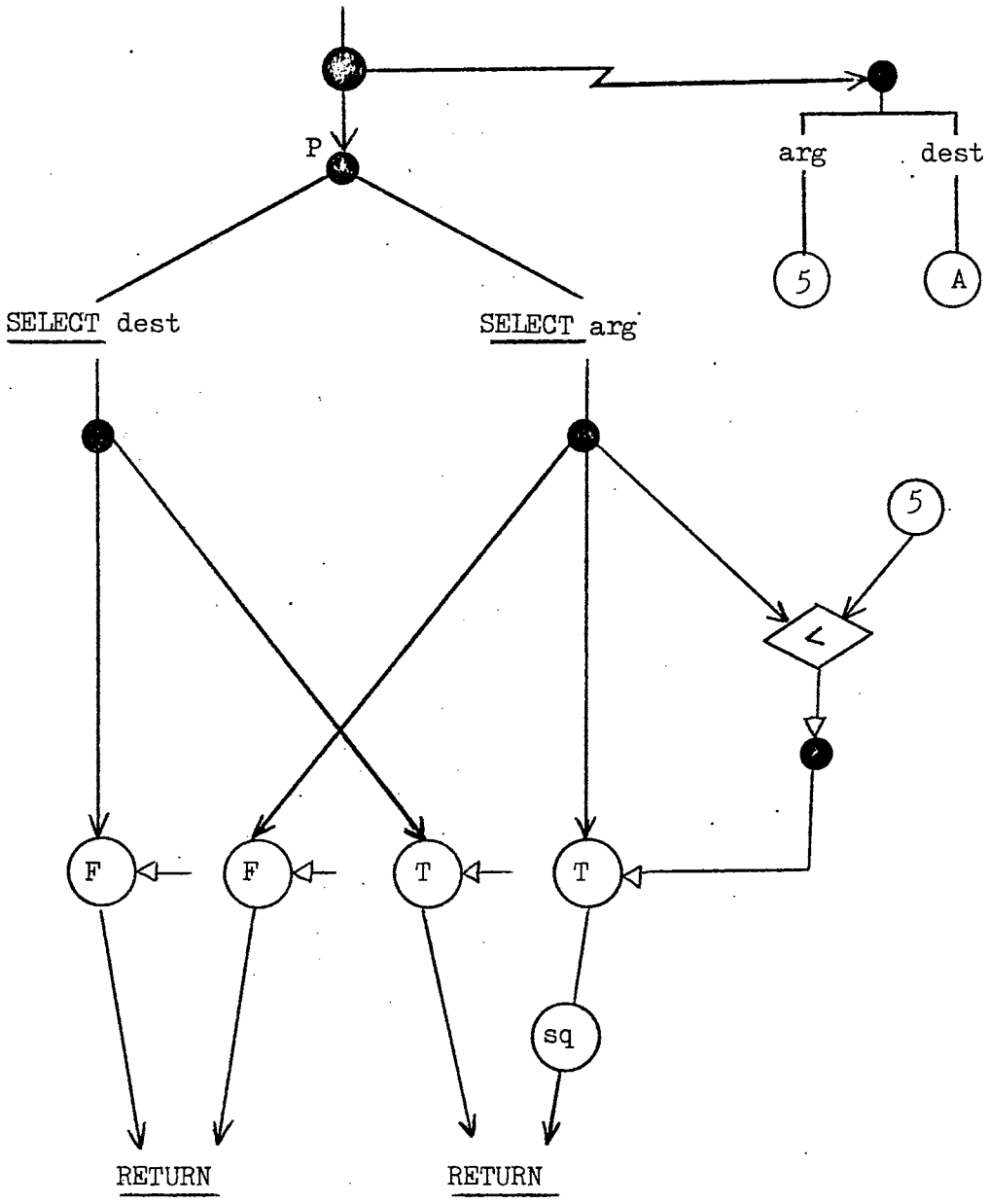


Figure 9. Data Flow Representation of a Simple Procedure

The procedure body tests the argument to see if it is less than five. If so, it is squared, and the resulting value is returned. If the argument is greater than or equal to five, the original value is returned.

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

Basic definitions of elements of the data flow language were described in this chapter. The complete data flow architecture, internal instruction representation and structure operations are discussed in Chapter IV.

CHAPTER IV

ARCHITECTURE OF THE DATA FLOW PROCESSOR

A. Introduction

The data flow processor described in this chapter is designed to directly execute programs expressed in the data flow language presented in Chapter III. The structure of the processor is presented in two stages. The first section of the chapter discusses the representation of instructions within the processor and the execution of individual instructions representing operators and deciders of a program. The next section extends the description to include the processing of structures.

B. Instruction Processing

The instructions of a data flow program are stored and executed in the instruction processing section of the processor (Figure 10). Instructions awaiting execution are contained in the instruction memory. Upon becoming ready for execution, an instruction enters the arbitration network and is conveyed by the arbitration network to the correct operation or decision unit. The results of an operation are distributed to the desired destination instructions by a distribution network. Similarly, the results of a decision

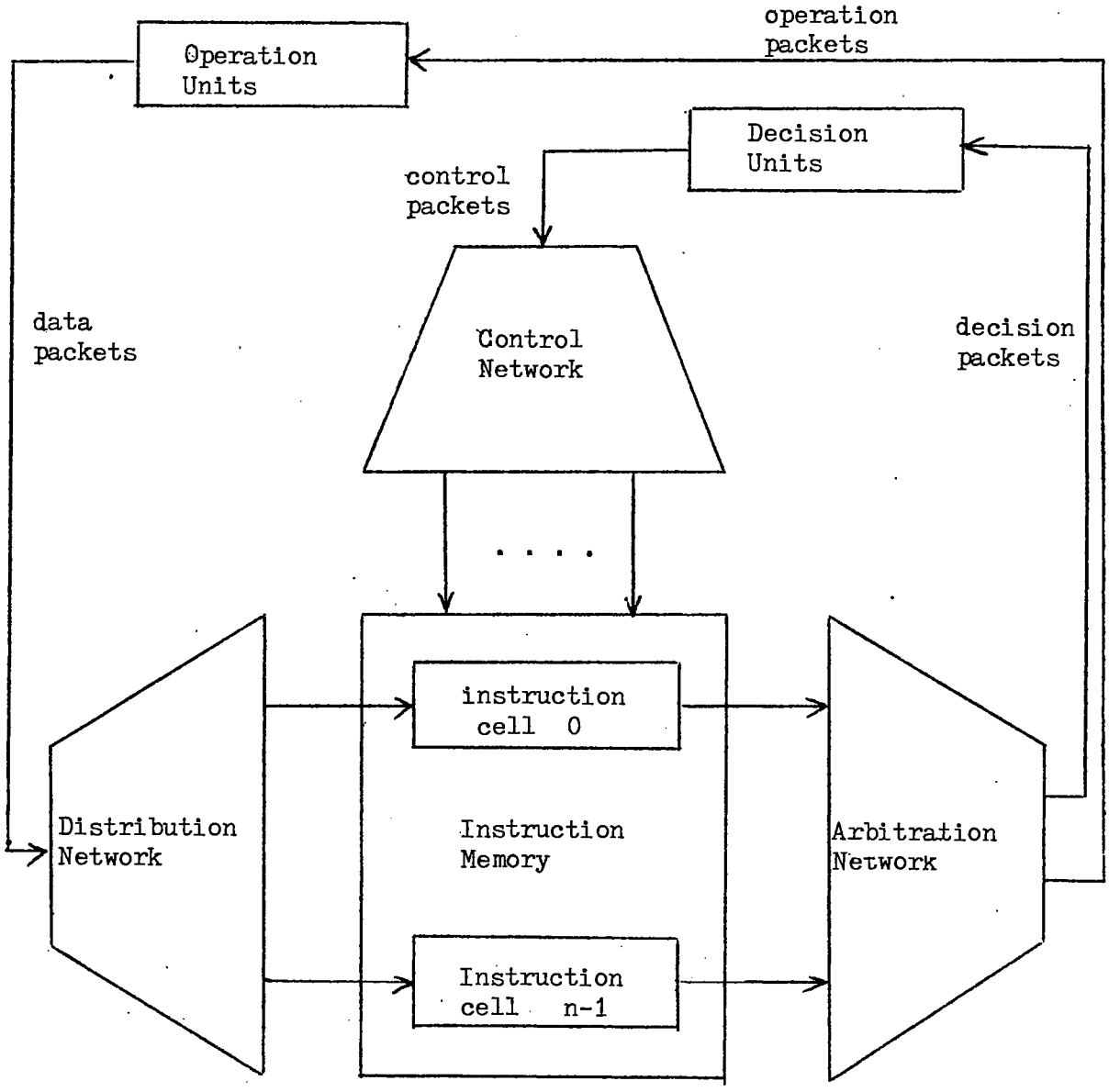


Figure 10. Organization of the Instruction Processing Section of the Data Flow Processor

are distributed by a control unit.

B.1. Instruction Representation

The instructions of a program being executed are stored in the instruction memory of the processor. The instruction memory contains a number of instruction cells, each holding one instruction of the data flow program. Each instruction cell consists of a number of registers, say five (Figure 11) and holds the instruction in the specified format together with spaces for receiving its operands. An instruction cell is designated by an identifier which specifies a path to that cell through the distribution and control networks.

Each instruction corresponds to an operator, a decider, or a boolean operator of a data flow program. The first register of an instruction cell holds an instruction which encodes in its operation code the function to be performed; that is, the type of actor represented by cell. The register specifies in its destination field the cell identifier of an instruction which is to receive one copy of the result.

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

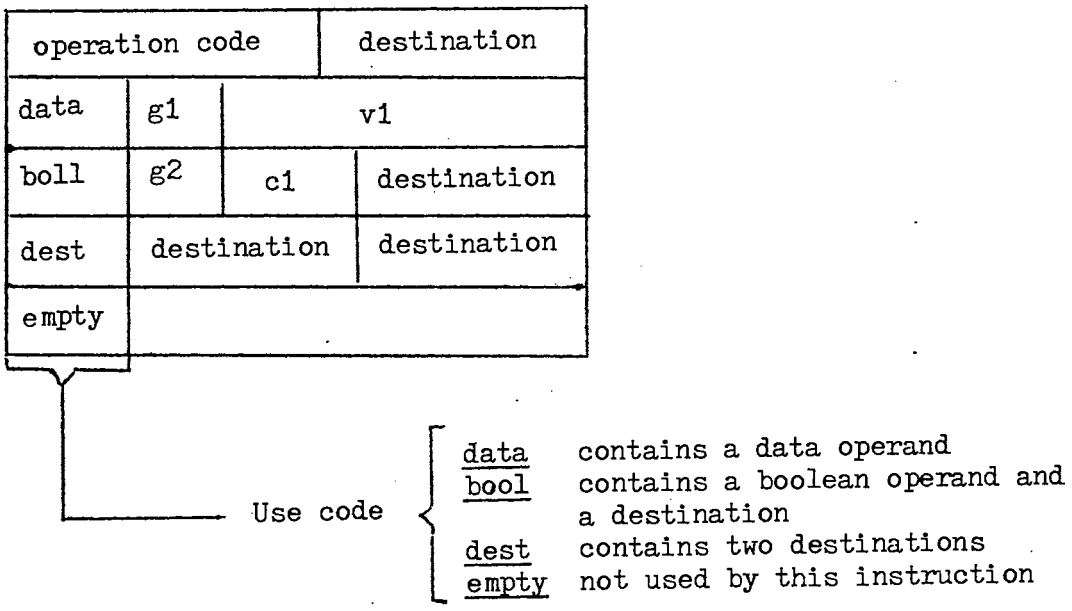


Figure 11. Format of Fields in an Instruction Cell

specified, and that destination must be a distribution instruction (Figure 12) if more than one destination is desired for the result.

A register containing the components designated by an operand selector in an instruction consists of two parts, a gating code g1,g2 and either a data receiver v1 or a control receiver c1. The gating codes permit representation of gate actors that control the reception of operand values by the operator or decider represented by the instruction cell. The meaning of the code values are as follows:

code value	meaning
no	the associated operand is not gated
true	an operand value is accepted by arrival of a true control value; discarded by arrival of a false control value
false	an operand value is accepted by arrival of a false control value; discarded by arrival of a true value
const	the operand is a constant value

The structure of a data or control receiver (Figure 13) provides space to receive a data or boolean value, and two flag fields in which the arrival of data and control values is recorded. The gate flag is changed from off to true or false by a true or false control value. The value flag is changed from off to on by a data or boolean value according

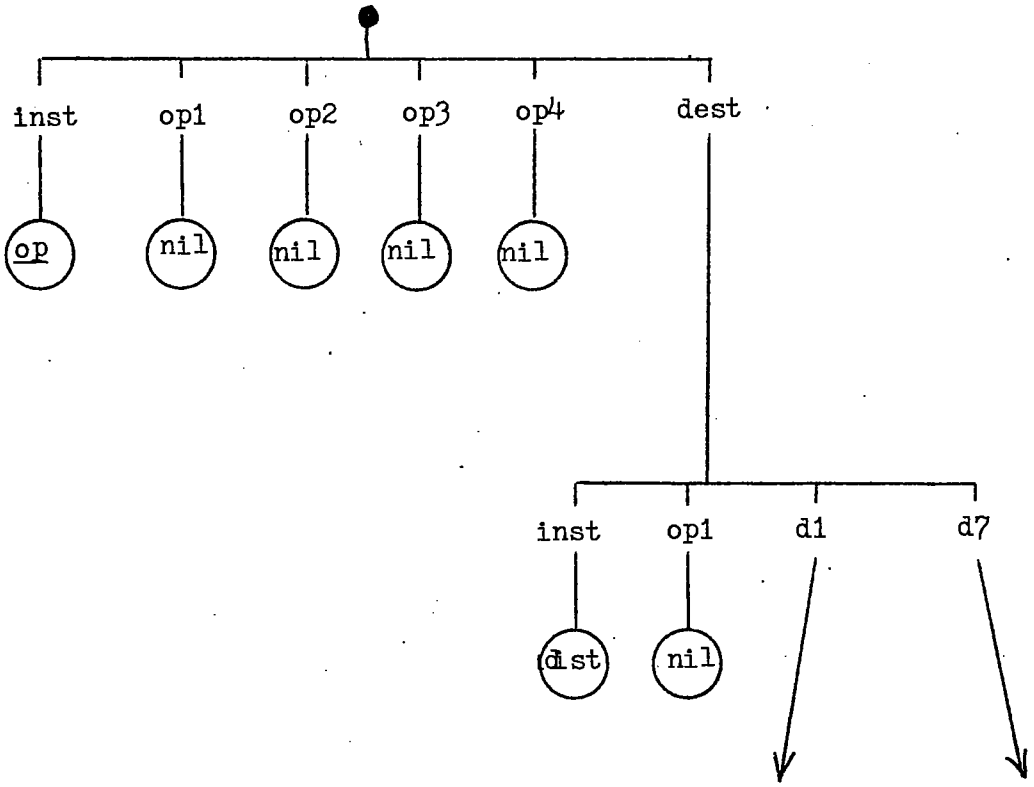


Figure 12. Use of the Distribution Instruction

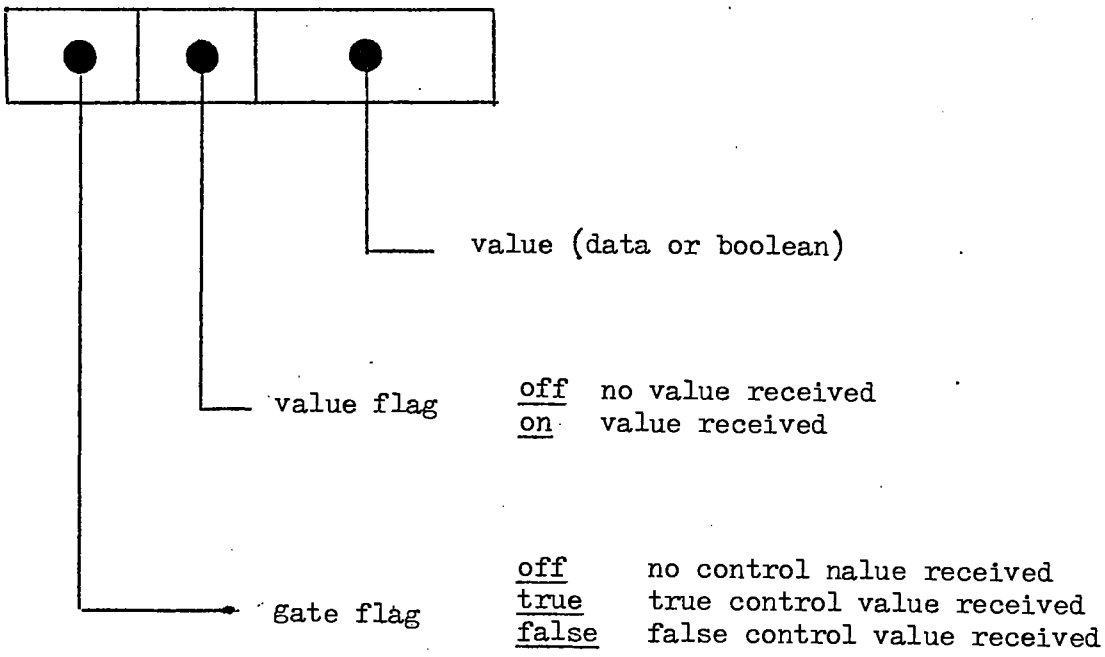


Figure 13. Structure of a Receiver

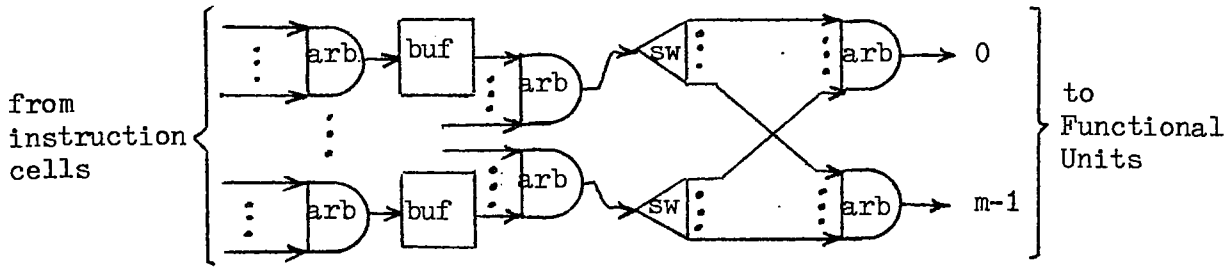
to the type of receiver.

B.2. Network Structures

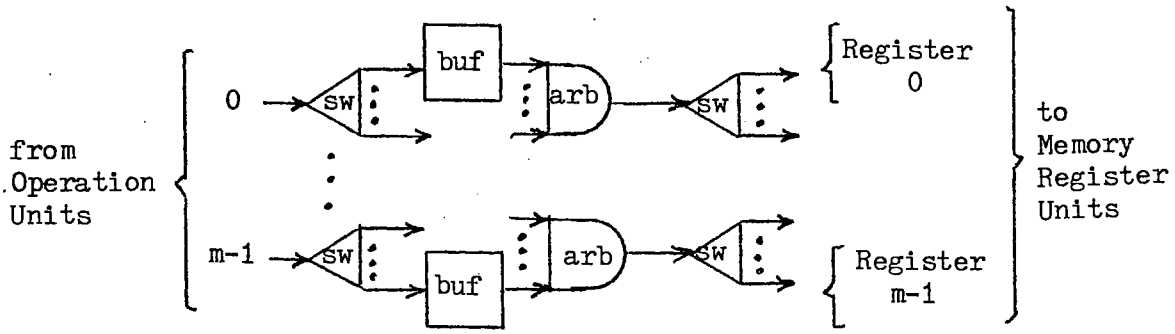
To connect the instruction cells of the memory to the operation and decision units, a network, called the arbitration network, provides a path from each instruction cell to each operation or decision unit. Operation and decision packets are transmitted from instruction cells into the arbitration network. The network is capable of accepting many packets simultanously and delivers each packet to the correct Functional Unit.

Upon receiving an operation packet, an operation unit performs the function specified by the operation code on the operands of the packet and produces a data packet for each destination specified in the instruction. A distribution network concurrently accepts data packets from the operation units and, using the destination address of each packet, delivers it to the specified instruction cell. Similarly, the control packets produced by a decision unit are sent to the control network for delivery to the designated instruction cells.

A simplified structure of the arbitration and distribution networks is presented in Figure 14. The networks are composed of three types of units. An arbitration unit passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any conflicts. A switch unit passes a



(a) Arbitration Network



(b) Distribution Network

Figure 14. Structure of the Arbitration and Distribution Networks

packet at its input to one of its outputs, controlled by some property of the packet. In the arbitration network this property is the operation code, whereas in the distribution network , the switch units are controlled by the destination address. A buffer unit stores a packet until the succeeding switch or arbitration unit is ready to accept it.

C. Structure Handling

The physical representation of a structure within a computer system may be viewed in several different ways. One extreme involves implementing the structure as it is represented in the data flow model, that is, as an acyclic directed graph in which each node is either a structure node or an elementary node. In such an implementation, each node of the graph occupies a number of storage locations within the processor. The location(s) containing a structure node hold the identifiers of the locations containing nodes which are successors of that node. The location representing an elementary node holds an elementary value. The nodes of a structure represented in this fashion may be scattered throughout the memory of the processor. Alternatively, all elementary values of a structure may be stored together in a group of locations. The first few locations of the group then contain a mapping function which allows one to find the location of a specific element within the group. This method is often used for the representation of arrays within a conventional computer system.

The first approach has the problem that the storage of a structure in such a manner can occupy a great deal of space within the memory. Not only must the data be stored, but a large number of structure nodes and associated pointers must also be located within the memory. Accessing an elementary value in a graph can take a long time as a path is followed over the arcs of the graph to the desired node. On the other hand, a single structure represented by the second approach occupies much less room, but the representation of several structures in such a manner can be very expensive in terms of space since components of a structure cannot be shared as they can in the graph approach. It would seem that perhaps a combination of these two methods could be efficiently utilized; that is, a structure representation in which each node of the structure is a small block of data.

C.1. Simple Structures

The storage of structures and the execution of the structure actors occurs in a separate structure processing section within the data flow processor. The structure processing section consists of a structure operation unit and a structure memory and attendant arbitration and distribution networks. This section of the processor is viewed as an operation unit by the instruction memory; that is, packets specifying structure operations are sent to the section, and data packets are returned. The organization of

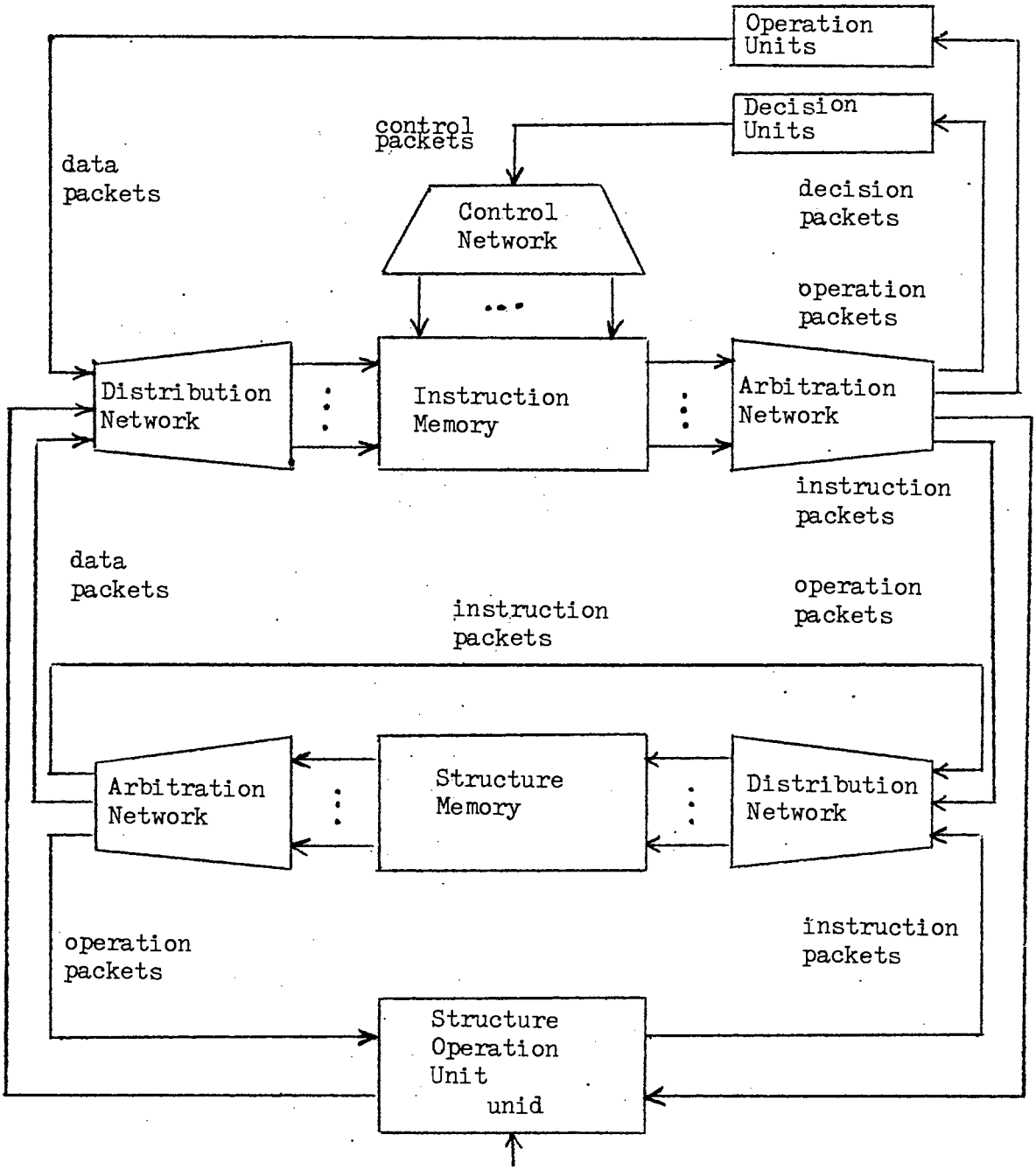


Figure 15. Organization of the Data Flow Processor With Structure Processing Capability

the data flow processor with the addition of the structure processing capability is shown in Figure 15.

Packets specifying structure operations are received by the structure memory and the structure operation unit. Instructions which require the creation of new structure nodes are processed by the structure operation unit. The unit controls the performance of the instruction specified in each operation packet through instruction packets sent to the structure memory and sends as data packets the identifiers of the resulting structures to the instruction processing section. All structure operations other than the allocation of a new node are performed within the structure memory.

To illustrate the operation of the structure processing section of the processor, in this section we shall limit our consideration to structures represented as binary trees. A selector of such a structure can have one of two values, L (left) and R (right).

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

Cell A

elem	a
struc	Y

Cell B

elem	d
struc	Y

Cell Y

elem	b
elem	c

Figure 16. Memory Representation of the Structure of Figure 3

structure of Figure 5 is presented in Figure 16.

The structure memory is composed of a number of structure cells in a manner similar to the way the instruction memory is formed of a number of instruction cells. Each structure cell is capable of holding one node of a structure, and the identifier of the cell specifies a path through the distribution network to the cell. The structure memory receives instruction packets from the instruction memory and the structure operation unit commanding a specific structure cell to execute some structure operation upon the node located in the cell.

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

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

$$\left\{\begin{array}{l} \text{SELECT dest} \\ \quad\quad\ \ s \end{array}\right\}$$

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

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

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

which is presented to the arbitration network for transmission to the instruction processing section of the processor.

b. If s is a compound selector s1s2....sn, the content B of the register designated by s1 is the identifier of some other structure cell and is used to form the instruction packet

$$\left\{\begin{array}{c} B \\ \text{SELECT dest} \\ s2.....sn \end{array}\right\}$$

which is presented to the arbitration network for transmission to the input distribution network of the structure memory. The process is then repeated with the selector s2 at structure cell B.

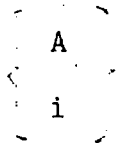
2. ALTER. The receipt of an ALTER instruction

$$\left\{\begin{array}{c} Y \\ \text{ALTER} \\ \left\{\begin{array}{c} s \\ x \end{array}\right\} \\ B \end{array}\right\}$$

indicates that the structure cell is to contain a copy of the node B with the component of B designated by the selector s set to x. First, a copy of node B is retrieved from the memory. Once the copy of B is present in the Cell, the value contained in the register designated by the selector

s is changed to x, and the use code of the register is set to the appropriate value (elem, struc, or empty), designated by the tag of x, and the result is linked to Y.

The format of an instruction packet received at the input distribution network of the structure memory differs from the format of an operation packet transmitted to a functional unit or the structure operation unit due to the fact that the operation code of an instruction packet does not control the switching within the distribution network; rather, the cell identifier is used to direct an instruction packet toward the correct structure cell. Hence, an instruction packet in the distribution network has the following format



where A is the identifier of some structure cell in the structure memory and i specifies one of the two operations which can be performed by a structure cell and contain the necessary operands.

Packets containing instructions that designate structure operations are transmitted to the structure processing section of the processor from the instruction memory. A packet specifying a select instruction is transmitted directly to the structure memory as an instruction packet. Structure operation packets representing the other structure instructions are

transmitted to the structure operation unit. The necessity of processing each operation packet within the structure operation unit is due to the required allocation of one or more free structure cells for the execution of each instruction with the exception of the select instruction. The structure operation unit performs the allocation of a free Cell simply by accepting the identifier of a cell over the unid port in structure operation unit.

Now that we have considered the operation of a structure cell within the structure memory, we can describe the execution of each of the remaining structure actors merely by listing the procedure followed by the structure operation unit in processing the instruction. For the purposes of this discussion, it is assumed that all selectors are simple selectors.

A CONSTRUCT instruction

```
{ CONSTRUCT  dest }
{      s1:   A    }
{      s2:   Y    }
```

specifies that a new node is to be created with components A and Y, designated by the selectors s1 and s2. The instruction is implemented by the structure operation unit as a number of ALTER operations in the following manner:

1. Accept an identifier B from the unid port.
2. Transmit to the structure memory the instruction packets

$$\left\{\begin{array}{c} B \\ ALTER \\ s1 \\ A \\ 0 \end{array}\right\} \quad \text{and} \quad \left\{\begin{array}{c} B \\ ALTER \\ s2 \\ Y \\ 0 \end{array}\right\}$$

transferring the values A and Y to the correct registers of B.

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

$$\left\{\begin{array}{c} dest \\ B \end{array}\right\}$$

An operation packet containing an APPEND instruction is of the following format:

$$\left\{\begin{array}{cc} APPEND & dest \\ & s \\ & x \\ & A \end{array}\right\}$$

where s is the selector of the element in structure cell A which is to be replaced by x in the new structure. The procedure followed by the structure operation unit to execute the instruction is as follows:

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

$$\left\{\begin{array}{c} B \\ ALTER \\ s \end{array}\right\}$$

```
| x |
| A |
```

to the structure memory to copy node A into cell B and change the component of B designated by the selector s to x.

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

```
dest
 B
```

An operation packet specifying a DELETE instruction

```
DELETE  dest
   s
   A
```

is processed in a similar manner:

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

```
   B
ALTER
   s
   0
   A
```

to the structure memory, indicating that the use code of the register designated by s in cell B is to be set to EMPTY.

3. Transmit the data packet

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

to the instruction processing section.

C.2. Extension to More Complex Structures

The extension of the described techniques for the implementation of data structures to larger and more complex structures is straightforward. In order to implement structures with a fixed maximum number of arcs emanating from each node, the size of a structure cell is increased to accomodate the new node size. The use of arbitrary (to a fixed maximum size) integers or character strings as selectors can be accomodated through the addition of a selector field to each register. A structure cell must then have the capability to choose from the node contained in the cell an item whose selector matches a specified selector. These extensions allow the representation of fairly powerful structures. A further extension to allow a node to have arbitrary number of emanating arcs introduces a great deal of complexity since it might be necessary to use several cells to hold the identifiers of all cells which contain successors of the node. To avoid this complexity, a node of a structure in the data flow processor is of fixed size, and each arc emanating from the node has a fixed size selector associated with it.

CHAPTER V

IMPLEMENTATION OF HIGH-LEVEL LANGUAGE CONCEPTS IN DATA FLOW ARCHITECTURE AND EXISTING SEMANTIC GAP

A. Data Representation

Data representation and arithmetic processing of a highly parallel, asynchronous data flow computer should be designed in a manner compatible with the architecture of the computer. The data flow within the processor occurs in terms of packet flow. Packet format consists of a group of bytes (8 bit each) travelling sequentially along byte-width channels. Hence, a convenient way to manipulate or examine these packets is to provide byte-serial operation units [17].

The arithmetic processing unit uses signed digit arithmetic which uses algorithms with the following properties:

1. The operation can begin before the operands are available in complete form,
2. The first result digits are produced (most significant first) after a certain number of result digits are available.

For example, in the addition operation, the most significant

result digit is available after the first operand digits arrive. This is made possible by the property of Signed Digit arithmetic that limits carry propagation to adjacent digits. As a result, the processor accepts bytes of input, and produces output bytes, consistent with the structure of data packets in the data flow computer. Pipelining allows a high byte processing rate.

A.1. Signed Digit Number Representation

Various options for number representations are available for fast arithmetic. Conventional number representation such as 2s complement are such that for an arbitrary base r, each digit of a number can have r values, chosen from the digit set (0,1,...,r-1). These representations have the property that carries generated by the summation of digits can propagate from right to left along the whole number, e.g., 999+1 = 1000. This property limits digit-by-digit computations to representations where the least significant digit is available first; otherwise the result can only be obtained as a whole. For example, lets consider the operation (9863 + 0199)

1. two digits at a time, right to left

63 + 99 = 1 62

98 + 01 = 99

100 62

result available

2. two digits at a time left to right

98 + 01 = 99

63 + 99 = 1 62

10062

result available

in parts as a whole

The arithmetic processor designed for data flow computer is a byte-level pipelined processor with on-line properties i.e., a processor that would receive operands as bytes and output the results also as bytes, in both cases most significant byte first. Such algorithms exists for Signed Digit number representation.

A signed digit number system is a redundant system, i.e., each number can have more than one representation. For a chosen base r, this can be achieved by allowing each digit to assume more than r values. For example, a symmetric digit set of 2r-1 elements -a,...,-1,0,1,...a where a=r-1. This representation is called maximally redundant, and it is the largest possible digit set for the chosen base. For example, for base 8 arithmetic, the maximally redundant signed digit set is S = -7,...,-1,0,1,...,7, while the conventional digit set is A = 0,1,...,7. Hence A is a subset of S. Using the digit set S, redundancy can be shown:

$$0.6432_8 = 0.7\bar{4}32_8 = 0.7\bar{4}4\bar{6}_8$$

Characteristics of signed digit numbers are as follows:

1. A signed digit number X is represented by n+m+1 digits x_i (i=-n,...,0,...,m) and $X = \sum_{-n}^{m} x_i r^{-i}$ where r=integer base,
2. X = 0 if and only if all x_i =0,
3. Sign(X) = Sign of the most significant digit, and

4. Inverse of X, i.e. -X is obtained by changing the sign of each x_i in X.

Since fixed format floating-point operations are used, representation of the number X can be redefined as consisting of m digits x_i $(i=1,2,\ldots,m)$ so that $X=\sum_{1}^{m} x_i r^{-i}$. This way there are no digits to the left of the radix point Now definitions for parallel addition and subtraction are given as follows:

1. Addition of digits z_i, y_i is parallel if
   a. Sum digit s_i is a function of only z_i, y_i and the transfer digit t_i from the (i+1)th position on the right (Figure 17), i.e., $s_i = f(z_i, y_i, t_i)$.
   b. The transfer digit t_i is a function of z_{i+1} and y_{i+1} only.
2. Subtraction is done by negating the subtrahend according to property (4) above and then adding, so that $z_i - y_i = z_i + \bar{y}_i$.

The transfer digit t_i is the carry generated when the digits are added. Since negative sums can be used, there can be negative carry as well. Therefore, t_i can assume $(\bar{1},0,1)$ as values.

Interim sum digit, w_i, is defined to be a subsum such that:

$$z_i + y_i = r t_{i-1} + w_i \qquad (1)$$

and sum digit

$$s_i = w_i + t_i \qquad (2)$$

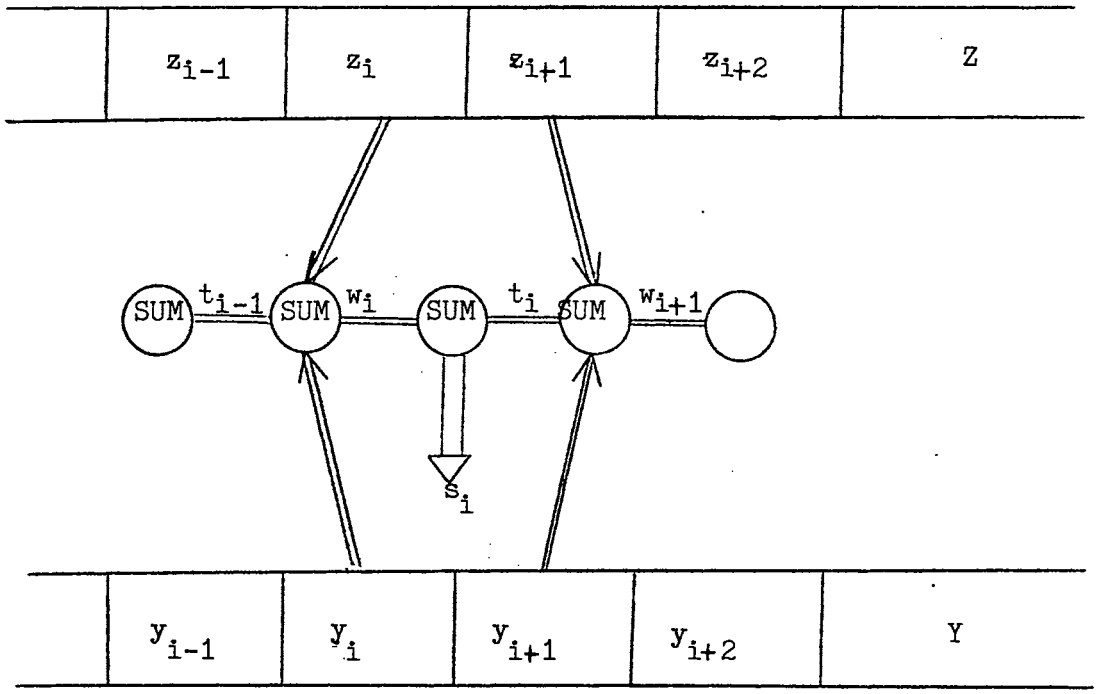


Figure 17. Signed Digit Addition

Since $t_i = (\bar{1},0,1)$ and since s_i must also be in the same digit set as z_i and y_i (namely $s_i < r$), the $w_i < r-1$ because otherwise $(w_i + t_i)$ will not be in the digit set S. For example, using r=8, $|w_i| < r-1 = 8-1 = 7$

for $t_i = 1$ and for unallowed value $w_i = 7$,

$s_i = t_i + w_i = 7+1 = 10$ which is clearly not in S

So far nothing has been said about the base limit, however because of the restriction on $|w_i|$, it can be seen that r=2 is not allowed. For base 2,

$$|w_i| < r-1=1$$

If $|w_i|=0$, then there is no t_i to satisfy $z_i + y_i = 1 = 2t_i$ (from Eq.1). Therefore, signed digit representation and algorithms are valid for r>2.

Advantages of using Sined Digit number representation are as follows:

1. Carry propagation chains in a conventional number representation are eliminated because s_i is a function of adjacent digits. Since there is no operand width carry, addition and subtraction time is independent of operand precision.
2. Most significant digits can be available before least significant ones and they can be processed further before an operation ends. Hence computations can begin before all of the digits are available, and therefore digit level pipelining is possible for arithmetic operations using sined digit number

representation.

Disadvantages of using signed digit number representation are as follows:

1. The adders are more complicated and therefore require more hardware than for example 2's complement adders.
2. Machine representations of numbers are larger than in conventional machines because of the digit set chosen, which requires an extra sign bit for each digit.

For the arithmetic processor designed for data flow computer, base-8 , fixed format, floating-point, sined digit representation is used. The digit set chosen is maximally redundant and consists of 15 integers (-7,...,-1,0,1,...,7). Machine representation is chosen as 16s complement base-8 binary form where each digit occupies 4 bits (Figure 18) Therefore two digits from an 8-bit byte and the purpose of the design is to acheive a byte-level pipelined, "two-digit-at-a-time" arithmetic processor.

As in all floating-point numbers, an exponent and mantissa are required. A sign bit for the whole number is not necessary: the sign of the number is the sign of the most significant digit of the mantissa. The exponent is represented by a binary byte (8-bit): one bit is the exponent sign and seven bits form the exponent, giving an exponent range of $8^{\pm 127}$ (approximately $5 \times 10^{\pm 114}$). Larger

Given base-8 Signed Digit set

$$S \quad (\bar{7},\ldots,\bar{1},0,1,\ldots,7)$$

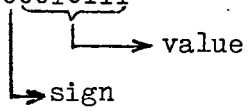
possible machine representation (16s complement)

0	0000		
1	0001	$\bar{1}$	1111
2	0010	$\bar{2}$	1110
3	0011	$\bar{3}$	1101
4	0100	$\bar{4}$	1100
5	0101	$\bar{5}$	1011
6	0110	$\bar{6}$	1010
7	0111	$\bar{7}$	1001

Floating-point number representation

number $.73\bar{4}\bar{6}$ E+23

exponent 00010111



mantissa 0111 0011 1100 1010

Figure 18. Machine Representation of Numbers

exponents can be obtained by the addition of more bytes as required. Conventional binary representation is used for the exponent because it makes exponent manipulations such as overflow and underflow detection easier. The format for the mantissa is selected as 4 digits or 2 bytes. The small number of digits is for clarity; increasing the precision does not change the structure of the processor.

Various operations result in either an error or in other special conditions e.g. exponent overflow, divide by zero, etc. When these are detected, they can be either handled through an error routine, or be unreported and indicated as a special result value (operand). Since the aim is to design a fast processor, error routines are not appropriate due to the fact that in a pipelined asynchronous system, it is hard to find means to report the error. therefore, various special operands are defined :

$\pm\infty$ (infinity for overflow cases)

$\pm\epsilon$ (0^+ and 0^- for underflow cases)

E (error, for indefinite cases)

These operands can be represented by special exponents and since these exponents are processed first, unnecessary operations can be discovered early. For example, for base-8 number format, one can limit the exponent range to $8^{\pm 120}$. In this case 8^{+121} would be overflow, while 8^{-121} be underflow. To the remaining 12 possibilities, the following values may be assigned

exp := ± 123 := $\pm\infty$

exp := +125 := $\pm\epsilon$

exp := 127 := E

When a special operand is detected, the normal operation is not completed, rather a special operand is selected and sent out as a result. For example let N be a normal operand, then:

$+\epsilon - N = -N$ $\qquad$ $0 - (-\infty) = +$

$E * (-\infty) = E$ $\qquad$ $0 / (+\infty) = 0$

This method is used in the CDC 6600.

Special operands can also be used or created in case of overflow occuring after an operation. In such cases the sign of the special operand is chosen to be the sign of the over or underflow result.

A.2. Arithmetic operations

In this section normalization, addition-subtraction and multiplication algorithms used in an arithmetic processor for a data flow computer are described.

A.2.a. Normalization. In floating-point arithmetic, normalization is basically the adjustment of a result to a specified format. A normalized number is such that the most significant digit of its mantissa is non-zero, i.e., for mantissa m and base r,

$$r^{-1} < |m| < 1$$

An exception to this rule is the zero mantissa (the number 0).

Usually in machine arithmetic involving conventional number representation, the result is ready as a whole and the normalization is done as follows:

1. If there is mantissa overflow then right shift the mantissa 1 digit; increment the exponent, check for overflow. If there is no overflow, pack the exponent and mantissa according to the format.
2. If the most significant digit of the mantissa is non-zero, then pack the exponent and mantissa according to the format.
3. If the most significant digit of the mantissa is zero then left shift the mantissa, decrement the exponent, check for underflow. If there is no underflow, check the new most significant digit; repeat until either the most significant digit is zero or the exponent underflows. Then pack the exponent and mantissa. The zero case is detected before normalization.

In the arithmetic processor designed for a data flow computer, the result is not available as a whole. Rather, digits are available one-by-one (in the adder-subtracter) and two digits at-a-time (in the multiplier). Since the most significant digits arrive first, this does not change the above algorithm, except that no shifting is done. For example given result 1.8734 E+72 in an on-line addition-subtraction operation:

.1 E+73 · mantissa overflow, increment exponent

.18 E+73

.187 E+73

.1873 E+73 done; exponent and mantissa packed

As seen above, normalizing involves also the construction of the mantissa according to the format. In some cases, exponent overflow or underflow may occur during such operation. In the overflow case, $\pm\infty$ is sent out according to the sign of the mantissa overflow digit. If there is underflow, then all result digits have to be examined for the sign until a non-zero digit is found; then $\pm\epsilon$ is sent out according to the sign of this digit. For example let E+100 be overflow and E-100 be underflow, then:

$\bar{1}$.7344 E+99 .$\bar{1}$7344 E100 negative overflow

therefore result $\longrightarrow$ $-\infty$

.00034$\bar{5}$ E-98 $\longrightarrow$ .0034$\bar{5}$ E-99

$\longrightarrow$ .034$\bar{5}$ E-100 underflow

therefore result $\longrightarrow$ $+\epsilon$

Unfortunately all zero results cannot be detected easily in a digit-by-digit environment and therefore can cause unnecessary normalizing operations. The proposed method of handling these is to:

1. Provide mechanisms to check operands pre-operation to discover zero-result cases, e.g. 0+0, 10*0, and
2. Continue normalizing post-operation until the last result digit is produced. In this case a zero exponent and zero mantissa can be packed and sent.

For case 1, 789*0 = 0 can be detected before the operation is performed. For addition and subtraction, there can be pre-operation detection of all zero operands only, i.e. 0+0.

A.2.b. Addition and Subtraction. Signed digit addition and subtraction has been described previously. What follows is an algorithmic description.

Given operands Z and Y, signed digit addition is done at two levels. First

$$w_i + rt_{i-1} = z_i + y_i$$

where z_i and y_i are ith digits of Z and Y respectively (i digits right of the radix point, t_{i-1} is the transfer digit and w_i is the interim sum digit).

The second level produces the ith sum digit:

$$s_i = w_i + t_i$$

Since $|w_i| < r-1$, a value for |wmax|, the largest magnitude, has to be selected. In this design, wmax is chosen to be r-2. Now a stepwise description of addition can be made:

1. Add z_i to y_i to obtain x_i, i.e. $x_i = z_i + y_i$.
2. Generate the transfer digit t_i using s_i and wmax where wmax < r-1.
   a. If x_i > wmax, there is positive carry; i.e. $t_{i-1} = 1$.
   b. If -wmax < x_i < wmax, then there is no carry; i.e. $t_{i-1} = 0$.
   c. If x_i <-wmax, then there is negative carry; $t_{i-1} = -1$.

3. Obtain ith interim sum digit w_i:

$$w_i = x_i - rt_{i-1}$$

4. Finally, compute ith sum digit:

$$s_i = w_i + t_i$$

Figure 19 summaraizes the above. It should be noted that using this algorithm, given ith operand digits z_i and y_i, ith sum digit s_i is produced when t_i is available, which is to say when (i+1)st digits are available. Once s_i is produced, it can be used up in another process before s_i is available. Initially w_0 is zero so that carry produced by the first most significant digits z_1 and y_1 indicates overflow; i.e. if $s_0 \neq 0$, then there is no overflow. Subtraction is done by negating the subtrahend.

In the adder-subtracter, bytes will be produced. Since a byte is two digits, a two digit parallel adder can be used as shown in Figure 20. Only variation is the extension of the transfer digit of A2 to B1 to enable sequential byte-level addition. Computation sequence is indicated next to each port in Figure 20. For example let |wmax| < 6, also let the digit set be maximally redundant, given Z = $.651\bar{3}$ and Y = $0.4\bar{7}\bar{1}\bar{4}$ the sum is:

s_0 s_1

1. 6 + 4 = 12 $w_0 = 0$

$t_0 = 1$ 1

$w_1 = 2$

2. $5 + \bar{7} = \bar{2}$

$t_1 = 0$ 2

$w_2 = \bar{2}$

3. $1 + \bar{1} = 0$

$t_2 = 0$ $\bar{2}$

$w_3 = 0$

4. $\bar{3} + \bar{4} = \bar{7}$

$t_3 = \bar{1}$ $\bar{1}$

$w_4 = 1$

$t_4 = 0$ 1

RESULT : $0.651\bar{3} + 0.4\overline{714} = 1.2\overline{211}$

A.2.c. Multiplication. An efficient algorithm for signed digit multiplication is used in design. Following is a description of the algorithm :

Operands are defined as

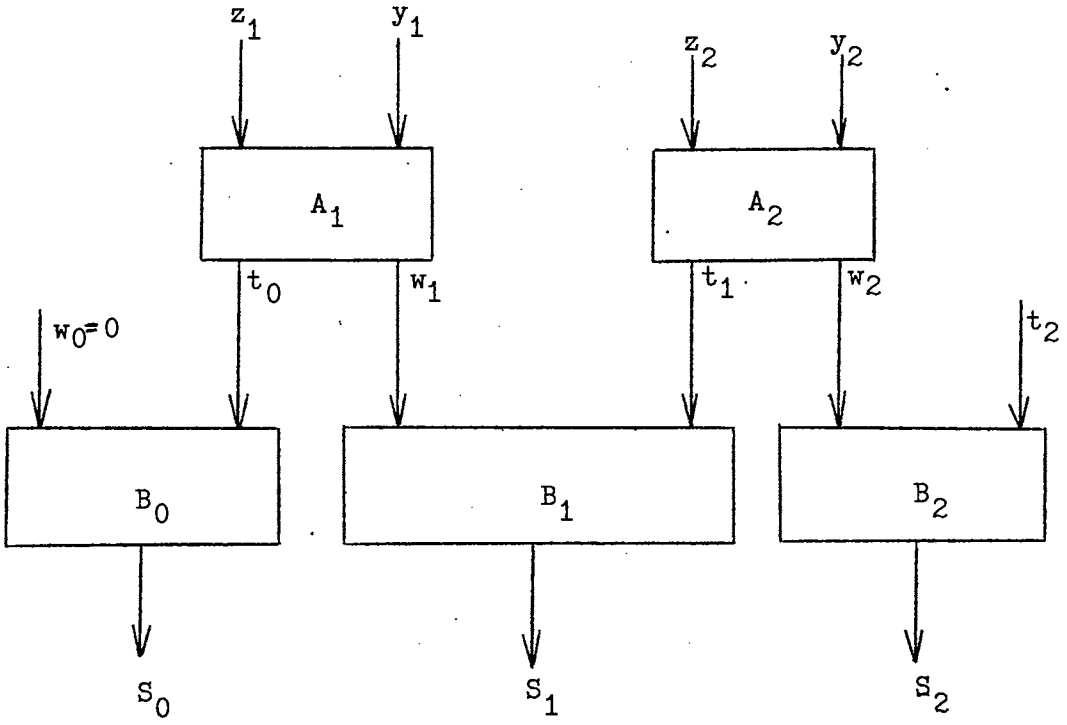
$$X = \sum_{1}^{m} x_i r^{-i} \quad \text{and} \quad Y = \sum_{1}^{m} y_i r^{-i}$$

As explained previously, this representation has no digits to the left of the radix point.

Let X_j and Y_j be the j-digit representation of X and Y respectively. In other words, let

$$X_j = \sum_{1}^{j} x_i r^{-i} = X_{j-1} + x_j r^{-j} \quad \text{and} \quad Y_j = \sum_{1}^{j} y_i r^{-i} = Y_{j-1} + y_j r^{-j}$$

In an on-line environment, X_j and Y_j are considered as the



Let K digit position to the right of the radix point

A_k : $z_k + y_k = x_k$

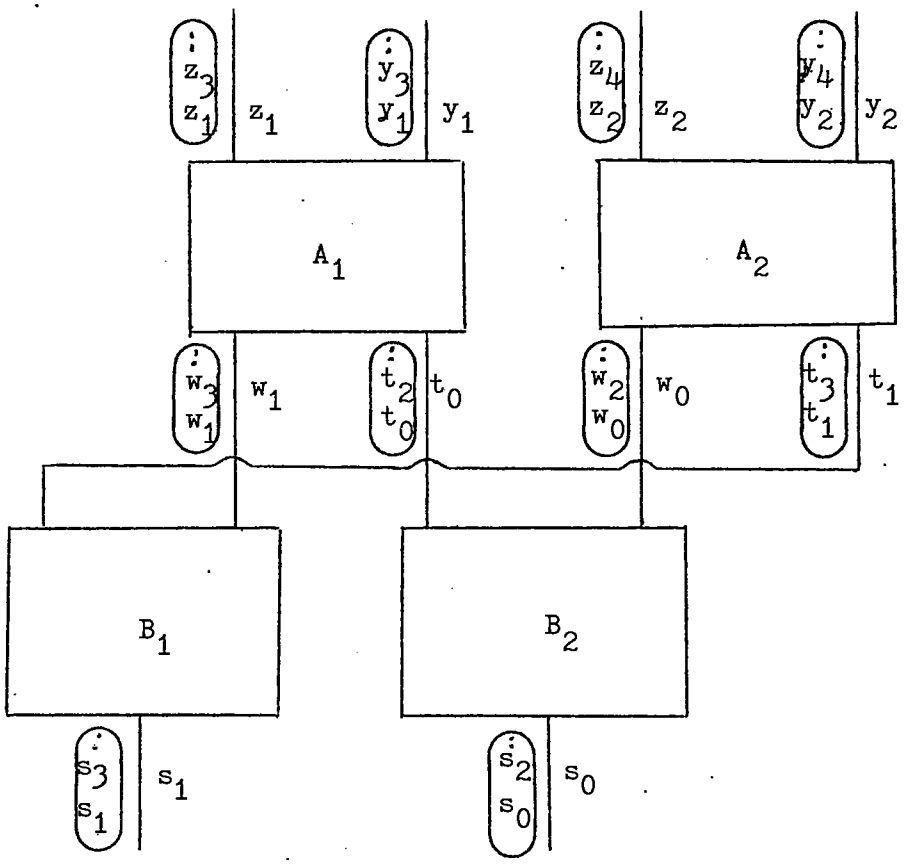
if $x_k > wmax \Rightarrow t_{k-1} = 1$

if $-wmax < x_k < wmax \Rightarrow t_{k-1} = 0$

if $x_k < -wmax \Rightarrow t_{k-1} = -1$

B : $s_k = w_k + t_k$

Figure 19. Parallel Signed Digit Adder



Note :  shows input or output at a given port

Figure 20. Double Digit Parallel Adder Modified for Byte-level Computation

available parts of X and Y respectively on the jth step. Now the partial product

$$X_j Y_j = (X_{j-1} + x_j r^{-j})(Y_{j-1} + y_j r^{-j})$$

$$= X_{j-1} Y_{j-1} + X_{j-1} y_j r^{-j} + x_j y_j r^{-2j} + x_j Y_{j-1} r^{-j}$$

$$= X_{j-1} Y_{j-1} + r^{-j}(X_j y_j + Y_{j-1} x_j) \quad (3)$$

Defining P_j to be the scaled partial product, i.e. $P_j = X_j Y_j r^j$, then

$$P_j = P_{j-1} + X_j Y_j + Y_{j-1} x_j \quad (4)$$

from Eq.3 above. Using this and the fact that $P_0 = 0$, the desired result can be obtained by

$$P_n = X.Y.r^n \quad (5)$$

This algorithm can be used for non-redundant numbers where the result digits are available least significant first in order to cope with carry propagation requirements. Since the interest is on-line computation, a new algorithm can be derived for Signed Digit multiplication with the on-line property, where input and outputs are obtained most significant digit first.

Using the symmetric and maximally redundant digit set S, the following new algorithm can be written using Equation (4):

$$W_j = r(W_{j-1} - d_{j-1}) + X_j y_j + Y_{j-1} x_j \qquad (6)$$

where digits d_j are in S, and

$$d_j = \text{Sign}(W_j) * \lfloor |W_j| + 1/2 \rfloor$$

The result of multiplication can be expressed as

$$XY = r^{-n}(W_n - d_n) + \sum_{1}^{n} d_i r^{-i}$$

In order to meet the restriction that d be in S, the operand bounds are limited so that for maximal redundency,

$$|X||Y| < 1/4$$

Figure 21 illustrates the algorithm.

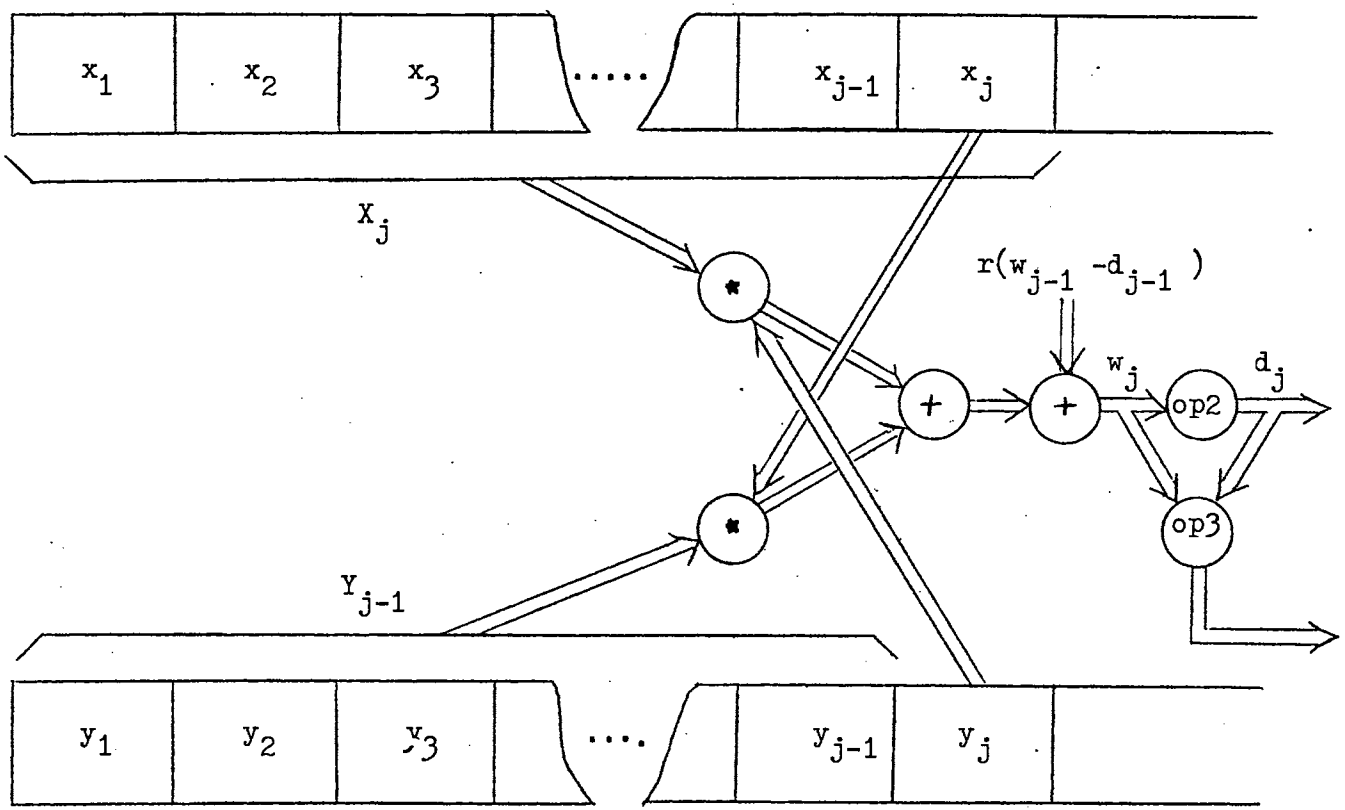
What has been described so far is a digit-at-a-time multiplication algorithm. For the design proposed for data flow computer, a two-digit-at-a-time algorithm is required and this can be made possible by slightly modifying Eq.(6). Since digits arrive as pairs, partial operands are redefined as follows:

$$X_j = \sum_{1}^{j} x_i r^{-2i} = X_{j-1} + r^{-2j} x_j \quad \text{and}$$

$$Y_j = \sum_{1}^{j} y_j r^{-2i} = Y_{j-1} + r^{-2j} y_j$$

Using the same derivation method as before, the new algorithm is defined as follows:

$$W_j = r^2(W_{j-1} - d_{j-1}) + X_j y_j + Y_{j-1} x_j \quad \text{and}$$



op2 : Sign w_j . $\lfloor |w_j| + \frac{1}{2} \rfloor$

op3 : $r(w_j - d_j)$

Figure 21. Signed Digit Multiplication

$$XY = r^{-n}(W_n - d_n) + \sum_{1}^{n} d_i r^{-2i}$$

The new algorithm produces d s that are digit pairs where each digit is in S. Operand bounds still apply, i.e. $|X||Y| < 1/4$. Signed digit multiplication procedures using single and double digits is shown in Figure 22.

A.2.d. Data Type Specification. The data flow computer supports are boolean, integer, and real data types. It is obvious why these types were chosen as the basic data types for data flow computer. Boolean values are required for control, and both integer and real data types are needed for performing practical computations.

Multiple precision and complex data types are not allowed because of storage limitations in the instruction cell, Their infrequent use, and their requirements for a more complicated processing unit. Character operands are not permitted because they typically occur in character strings, which should be handled by Structure Processor and kept in structure memory.

Boolean values will be represented in one byte, integers and reals in four bytes. The first byte of each representation contains an error bit. If the error bit is on, the error value is specified in the first byte. If the error bit is off, the operand is a standard boolean, integer, or real value.

Since there is no control flow to interrupt in data flow programs, programming errors are handled by generating

let $X = 0.025_{10}$ and $Y = 0.129_{10}$

j	x_j	y_j	X_j	Y_{j-1}	X_jy_j	$Y_{j-1}x_j$	SUM	w_j	d_j	$10(w_j-d_j)$
0	0	0	.0	.0	.0	.0	.0	0.	0	0.
1	0	1	.0	.0	.0	.0	.0	0.	0	0.
2	2	2	.02	.1	.04	.2	.24	0.24	0	2.4
3	5	9	.025	.12	.225	.6	.825	3.225	3	-

$w_3-d_3=0.225$

result can be obtained as digit pairs, i.e. by d_1, d_2, d_3 and (w_3-d_3)

therefore, $X.Y = 0.003225$

(a) Signed Digit Multiplication Using Single Digits

let $X = 0.0234_{10}$ and $Y = 0.2463_{10}$

j	x_j	y_j	X_j	Y_{j-1}	X_jy_j	$Y_{j-1}x_j$	SUM	w_j	d_j	$100(w_j-d_j)$
0	00	00	.0	.0	0.0	0.0	0.0	00.	00	00
1	02	24	.02	.00	0.48	0.00	0.48	00.48	00	48.
2	34	63	.0234	.24	1.4742	8.16	9.6342	57.6342	58	-

$w_2-d_2 = 0.\overline{3658}$

result , $X.Y=0.00583\overline{658}$

(b) Signed Digit Multiplication Using Double Digits

Figure 22. Two Procedures for Signed Digit Multiplication

special error values. The error values are:

boolean	_	undefined
integer	_	undefined
		positive/negative overflow
		unknown
		zero-divide
real	_	undefined
		positive/negative overflow
		positive/negative underflow
		unknown
		zero-divide

The element "undefined" results when operand values are not in the domain of an operator. The elements "positive/negative overflow" denote values, positive or negative, too large to be represented in the representation of the type used. The element "unknown" indicates the result of a computation that has exceeded the capacity of the implementation, but whose true value is not known to be out of range. The elements "positive/negative underflow " denotes non-zero values, positive or negative, too small to be represented in the representation of data type. A table of error values is represented in Figure 23.

B. Iterations

B.1. Introduction

Before discussing iteration (loop) structures it is

VALUE	NAME
1000 0010	unknown
1000 0011	undefined
1001 1100	positive-overflow
1000 1100	negative-overflow
1001 0100	positive-underflow
1000 0100	negative-underflow
1000 0001	zero-divide

Figure 23. Error values

useful to establish some terminology. By the term loop in high-level languages we mean a control construct which somehow enumerates a set of values for a loop-index or a loop-condition and which performs a fixed sequence of statements (its body), once for each value of loop-index or until the loop-condition is not satisfied.

A loop may contain one or more loops within its body. The inner loops are said to be nested within the outer (enclosing) loop and the structure as a whole is called a nested loop structure. Each enclosure defines a different level of the nested loop structure. The degenerate case of a nested loop structure, where there is no loop in the body of the outer loop, is called a single-level loop, since there is only one loop level.

B.2. Loop-construct

A loop-construct consists of some initialization code, a body which may be executed several times, and some exit code. There are different loop-constructs in high-level languages (PL/I, FORTRAN, COBOL). Execution of a program loop in high-level languages is controlled by the DO statement. Different DO statements existing in PL/I are the major concern of this study.

One of the PL/I DO statements has the following format:

$$\text{DO index-var = exp-1} \left\{ \begin{array}{l} \text{TO exp-2 BY exp-3} \\ \text{BY exp-3 TO exp-2} \end{array} \right\}$$

```
statement
    .
    .
END
```

in which a loop-index designated by "index-var" is used to control the number of iterations. Loop-index initially contains the computed value for "exp-1". After each iteration the value of loop-index is adjusted by the computed value of "exp-3" and compared with the computed value of "exp-2". The decision to continue or terminate the iteration is based on the result of this comparison.

An example of this DO statement is as follows:

```
DO I=1 TO 30 BY 2;
VOL=3.1416 * I**2;
PRINT VOL;
END
```

This code segment may be expressed in a lower-language notation as:

```
        I=1
LOOP:   VOL=I*3.1416
        VOL=VOL*I
        PRINT VOL
        I=I+2
        IF (I < 31) GO TO LOOP
```

Using this notation the different segments of the loop-

construct can be easily distinguished. Generally, this DO format uses a loop-index with a specified initial value (i.e., 1) which is incremented by an incremental value (a signed integer) after each iteration and compared with the final value (i.e., 50). If the final value is reached the loop is terminated and control value is passed to the next instruction in the program logic, otherwise, the new value of the loop-index is conveyed to the body of the loop for further computations.

To transfer both the initial and adjusted values of the loop-index to the body of the loop, a MERGE gate may be used, in which the false input receives the initial value of the loop-index (since all control values are initially false), and the true input receives the adjusted value of the index (Figure 24a).

After each iteration the value of the loop-index is adjusted by the incremental value. This segment of loop-construct may be represented in data flow base language using an actor (Figure 24b).

Finally, the new value of the loop-index should be compared with the final value. This segment may be represented by a decider gate, which current and final values of the loop-index are its inputs (Figure 24c). The comparison operator may be one of the following:

<	~<
>	~>

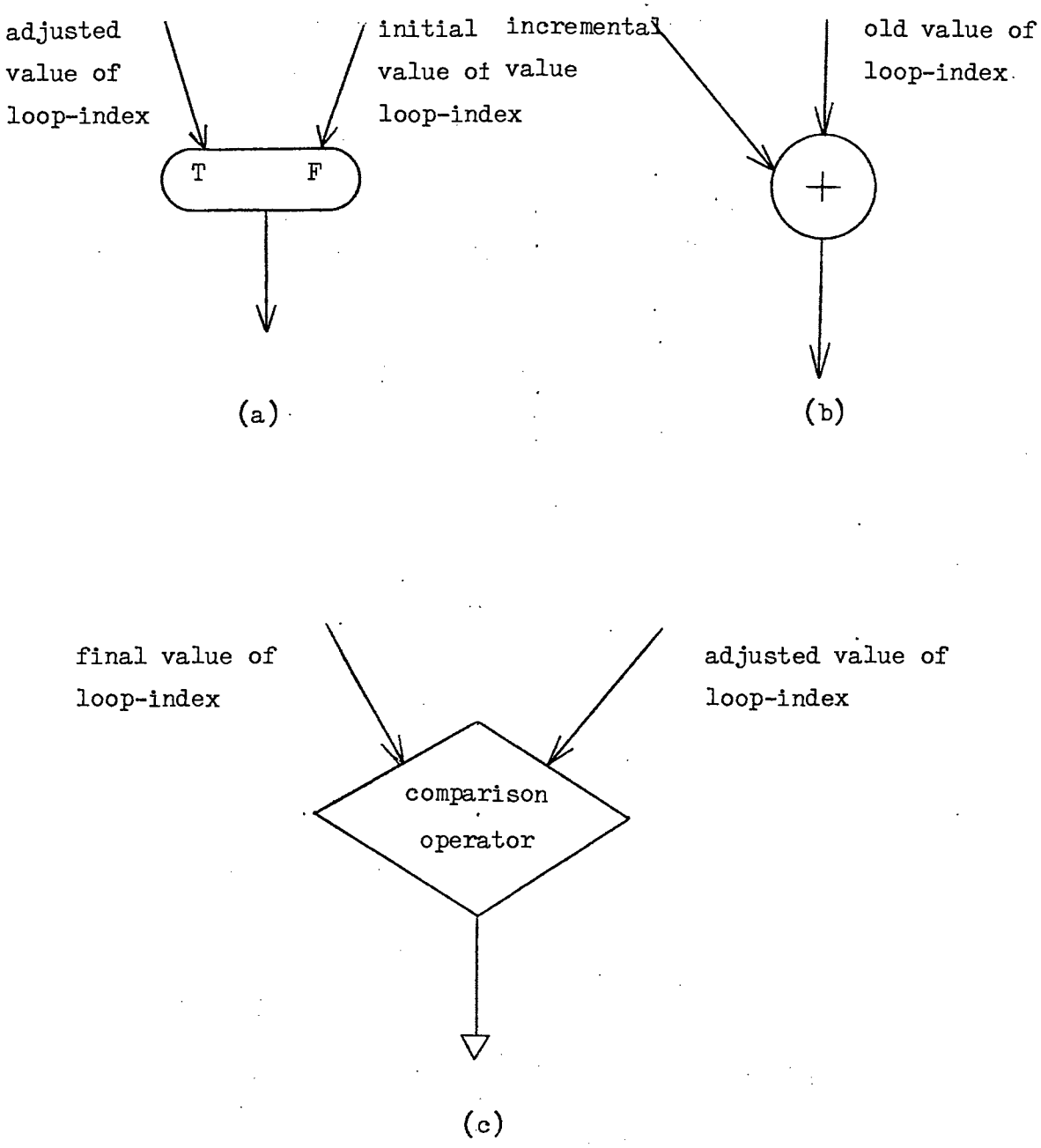


Figure 24. Data Flow Actors Used to Represent Loops

= ~=

<= >=

The result of the comparison is a control value (true or false) which specifies the status of the loop (terminated or not). The control token is then conveyed to the MERGE gate. Note that the comparison operator should be selected such that the resulting true value of the control token could cause the continuation of the loop. A copy of the control token is sent to the instruction immediately following the loop-construct in the program logic. A complete data flow code corresponding to the program segment discussed before is shown in Figure 25.

A more elaborate example of an indexed nested loop construct is presented in the following program segment:

```
DO I=1 TO 11 BY 2;
M=I**2;
   DO J=30 TO 1 BY -1;
   K=M*J**2+1;
   PRINT K;
   END;
END;
```

This PL/I nested loop-construct may be expressed in a lower-language notation as follows:

```
I=1
J=30
```

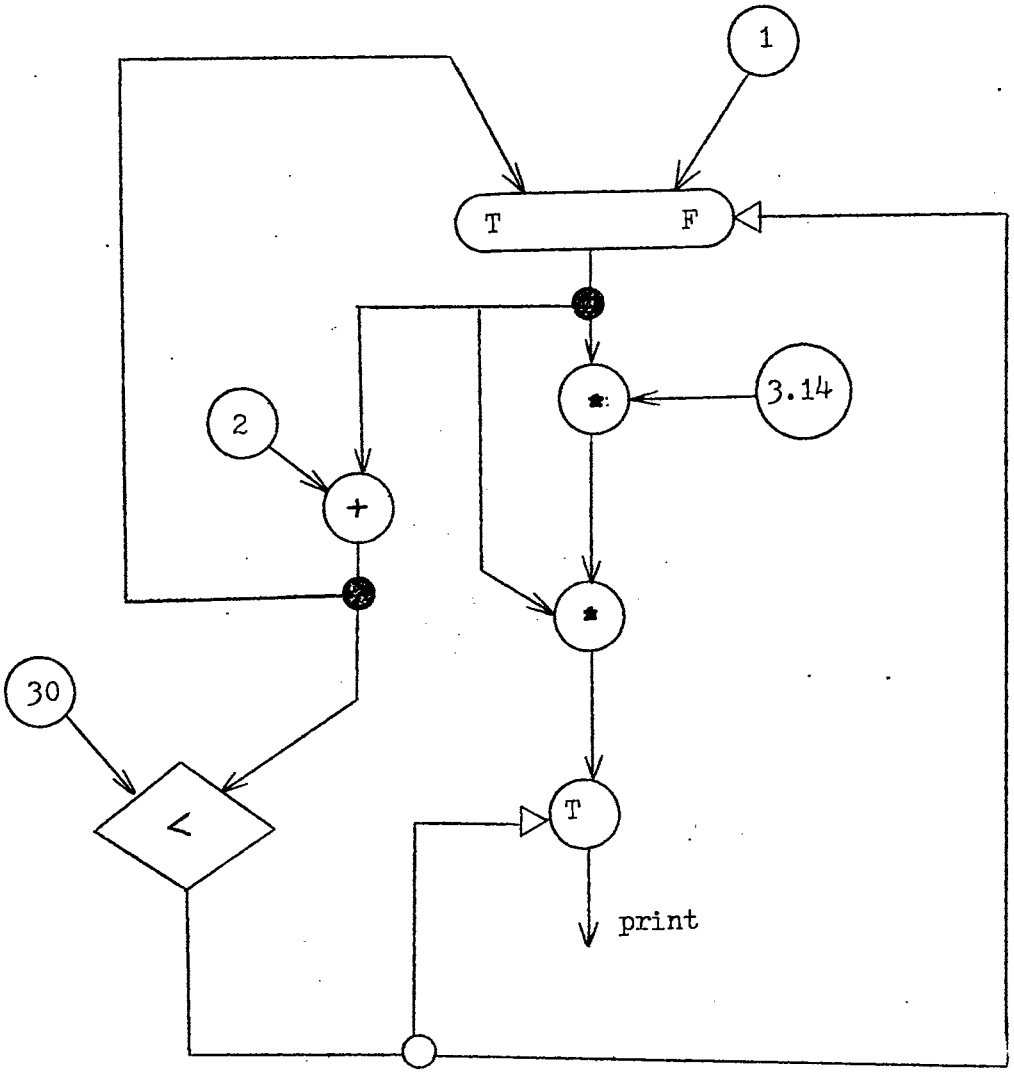


Figure 25. Representation of a Single DO Loop in Data Flow Base Language

```
LOOPO    M=I*I
LOOP1    K=M*J
         K=K*J
         K=K+1
         PRINT K
         J=J-1
         IF (J >= 1) GO TO LOOP1
         I=I+2
         IF (I <= 11) GO TO LOOPO
```

The corresponding data flow code is represented in Figure 26.

There is another form of DO statement in PL/I which instead of using a loop-index to specify the number of iterations uses an expression whose value can be converted to a truth value and as long as its value is true the iteration is continued. This form of the loop-construct has the following format:

```
DO WHILE (expression);
statement;
   .
   .
END;
```

The following code segment is an example of the DO WHILE form of the loop-construct in PL/I:

```
/* This program computes and prints SIN(x) for a
```

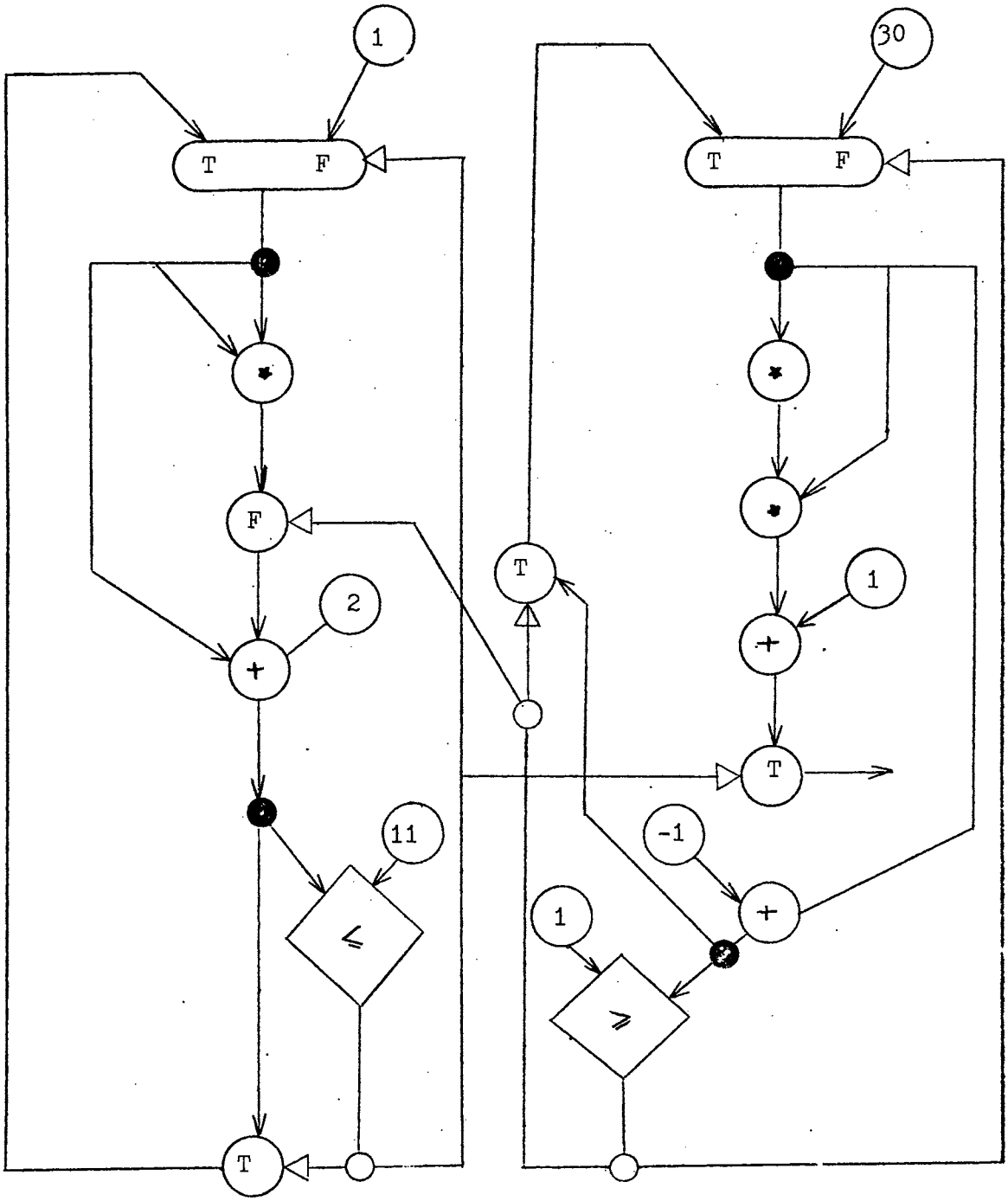


Figure 26. Representation of a Nested DO Loop in Data Flow Base Language

```
      given x with 8 digits of accuracy.          */
          SIN=0; I=1; FACT=1; TERM=X;
          DO WHILE ( TERM > 1.0 E-9 );
          SIN=SIN+TERM;
          FACT=FACT*(I+1)*(I+2);
          TERM=(TERM*X**2)/FACT;
          I=I+2;
          END;
          PRINT SIN;
```

The loop-construct may be expressed in a lower-level language notation as:

```
          SIN=1
          I=1
          FACT=1
          TERM=X
LOOP:     IF (TERM > 1.0E-6) GO TO OUT
          SIN=SIN+TERM
          I=I+1
          FACT=FACT*I
          I=I+1
          FACT=FACT*I
          TERM=TERM*X
          TERM=TERM*X
          TERM=TERM/FACT
          GO TO LOOP
OUT:      PRINT SIN
```

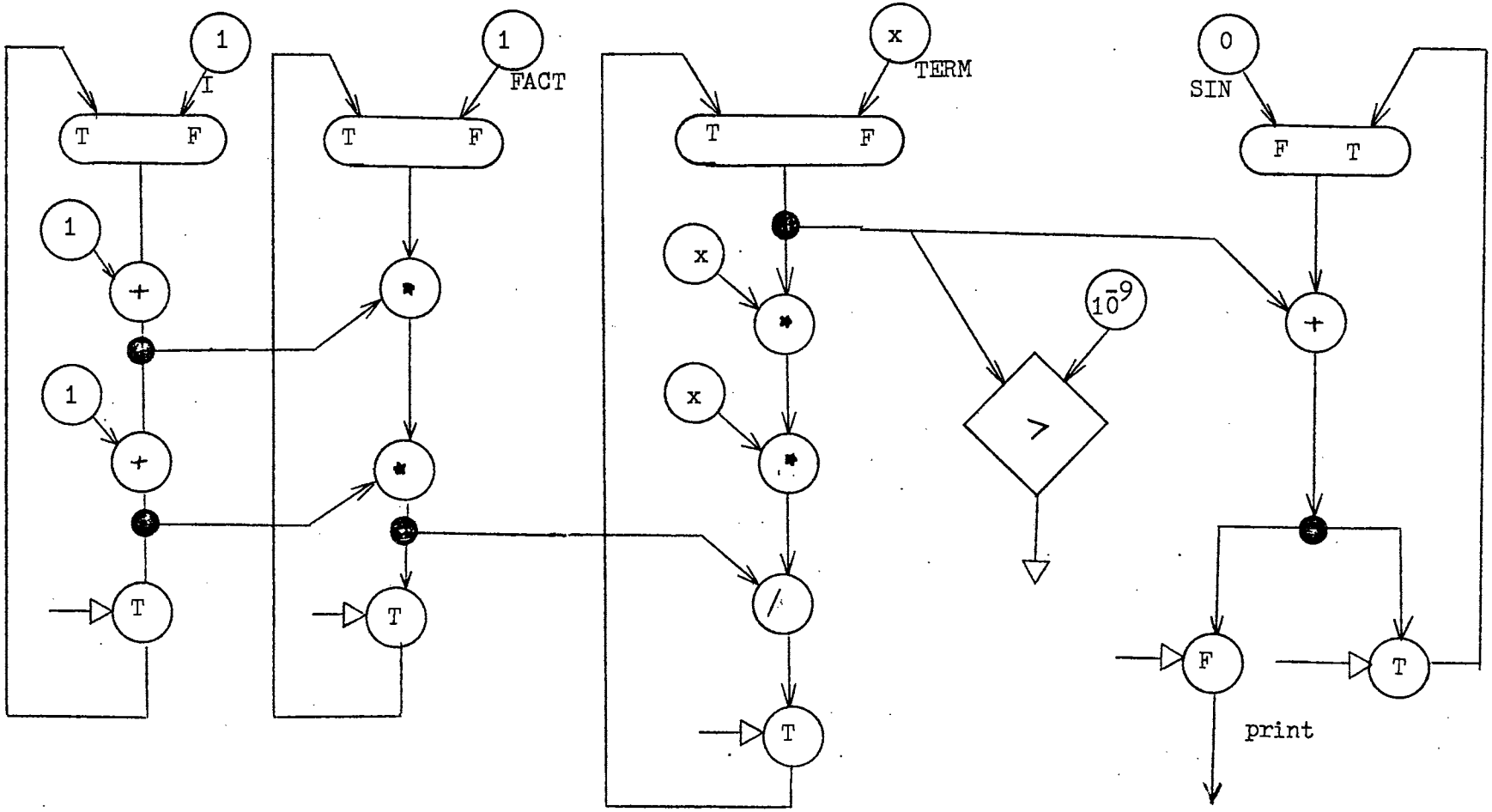


Figure 27. Data Flow Code to Perform SIN

The corresponding data flow code is represented in Figure 27.

C. Data Structures

In this section some basic data structures are studied on the implementation level in two types of computer architectures, conventional von Neumann and data flow. On the logical level, a data structure is a set of primitive data elements and other data structures, together with a set of structural relations among its components.

Difference in implementation of data structures in two different architectures arise from the difference in logical structure of the memories. In conventional von Neumann architectures memory is sequential, one dimensional block with the appearance of a vector. The only data structures that may be implemented directly in these architectures are linear lists. Structural relations in other data structures are implemented by compilers using basic properties of logical memory, and as it was discussed before, this mapping is one of the reasons of existing of the semantic gap.

The data flow architecture proposed by Dennis uses binary tree representation as the basic logical structure of the structure memory. Since the basic logical view of memory in this architecture is different from the von Neumann architecture, all mapping procedures of data structures should be changed or modified to cope with the new logical view of memory.

In the following sections the major and widely-used data structures in high-level languages are examined carefully. The mapping procedures used in compilers written for von Neumann architectures are represented, and new procedures to map data structures onto data flow structure memory are proposed. The data structures which are major concern of this study:

- arrays
- stacks
- queues

C.1. Arrays

An array is a collection of elements of some fixed type, laid out in a k-dimensional rectangular structure. A measure of the distance along the structure is called an index, or subscript, and the elements are found at integer points from some lower limit to some upper limit. An element of an array is named by giving the name of the array and the value of its index.

C.1.a. Allocation And Mapping. In conventional von Neumann architectures , if the size of the array is known at compile time, then it is expedient to implement the array as a block of consecutive words in memory. If it takes k memory units to store each data element, then A(i), the ith element of the array A begins in location

$$BASE + k*(i-LOW)$$

where LOW is the lower bound on the subscript and BASE IS the lowest numbered memory unit allocated to the array, that is, BASE is the location of A(LOW). A compiler recieves the following information from array descriptor in high-level program:

- the data type (i.e., one-dimensional array)
- the element type (i.e.,integer, real, or character)
- the number of memory units per element
- the lower limit on subscript range, and
- the upper limit on subscript range

In the case where everything is of fixed size, all of this information is available in the symbol table at compile time. Thus the compiler can generate a reference to any element of an array by determining its offset from the base of the array.

A two dimensional array is normally stored in one of the two forms, either row-major (row-by-row) or column major (column-by-column). FORTRAN uses column-major form; PL/I uses row-major form. Figure 28 shows the implementation of a 2x3 array called A in (a) row-major form and (b) column-major form.

In the case of a two-dimensional array stored in row-major form, with lower limit of 1 in each dimension, the location for A(i,j) can be calculated by the formula:

LOCATION	ARRAY ELEMENT
BASE	A(1,1)
BASE + 1	A(1,2)
BASE + 2	A(1,3)
BASE + 3	A(2,1)
BASE + 4	A(2,2)
BASE + 5	A(2,3)

(a) Row-major Form

LOCATION	ARRAY ELEMENT
BASE	A(1,1)
BASE + 1	A(2,1)
BASE + 2	A(1,2)
BASE + 3	A(2,2)
BASE + 4	A(1,3)
BASE + 5	A(2,3)

(b) Column-major Form

Figure 28. Two Forms to Represent a Two Dimentional Array

$$BASE + k*((i-1)*r + j-1)$$

where k is the number of memory units per element and r is the number of elements per row. In column-major form the formula is:

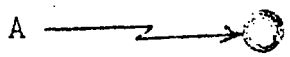
$$BASE + k*((j-1)*c + i-1)$$

where c is the number of elements per column.

Row-or column-major forms may be generalized to many dimensions and to arrays with a lower bound of subscript other than 1. The generalization of row-major form is to store the elements in such a way that, as we scan down the block of storage, the rightmost subscripts appear to vary fastest. Column major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest.

In the data flow architecture a binary tree is the basic logical representation of structures and other data structures must be mapped by compiler to a binary tree. To implement a data structure in data flow base language, the compiler should map its descriptor to a (pointer,selector) pair. An array is declared in a high-level language by a (name,dimension) pair. An array name may be directly used to create a pointer to the root of the associated binary tree. The dimensions of the array may be used to realize the length of the selector which identifies individual elements of the array. In the case of one-dimensional arrays, a binary representation of the index may be used as a selector, interpreting Os as left and 1s as right with

slight modification in index. For example consider the array declared as A(16). First A may be used as a unique pointer to the root of binary tree representation



Then the number of bits required to represent 16 different indices (4) specifies the number of elements in the selector. The following algorithm generalizes the mapping algorithm for one-dimensional arrays:

- create a pointer to an allocated cell using the name of the array
- find v such that

$$2^{v-1} < \text{dimension of array} <= 2^{v}$$

Then v is the number of elements in the selectors used to reference the array. To reference each individual element of the array, its index is first decremented by one and then its binary representation is used as a selector. The mapping algorithm may be generalized as follows:

- decrement index by 1
- convert index to a v-bit binary number
- use binary representation of the index as a selector (interpreting 0s as left and 1s as right)

For example references to the elements of array A may be shown as:

array element	selector
A(1)	LLLL
A(2)	LLLR
A(3)	LLRL
.	.
.	.
A(15)	RRRL
A(16)	RRRR

The complete structure of the array A is shown in figure 29.

Multidimensional arrays may be mapped using the above procedure with some modifications. The name of the array may still be used as a pointer to the root node of the binary tree. In this case, the concatenation of indices may be used as a selector. The allocation algorithm may be represented as follows:

_ create a pointer to an allocated cell
  using the name of the array
_ find v and w such that

$$2^{v-1} < \text{first dimension of the array} < 2^{v}$$

$$2^{w-1} < \text{second dimension of the array} < 2^{w}$$

Then v+w is the number of elements used in the selectors to reference the array. For example, array B(3,3) is pointed by a pointer B, and two bits is assigned to represent each index. The mapping algorithm may be represented as follows:

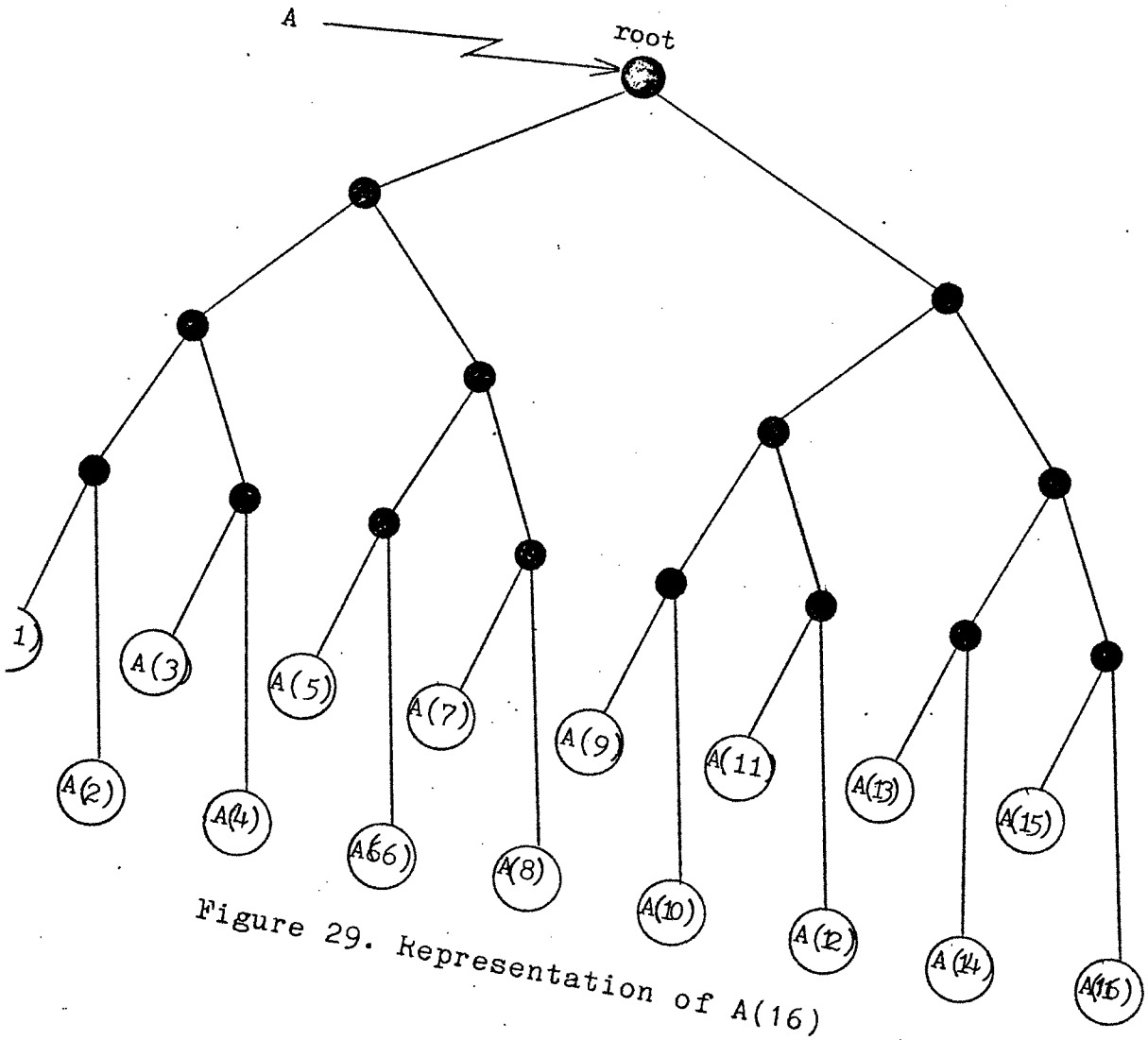


Figure 29. Representation of A(16)

_ decrement first index by 1

_ decrement second index by 1

_ convert indices to binary

_ concatenate binary representation of the indices to form the selector (interpreting Os as left and 1s as right)

_ use the selector and pointer B to address the element

Note that the concatenation procedure determines the allocation type. If row index represented first, the allocation is row-major; otherwise it is column-major. References to array B in row-major form is as follows:

array element	selector
B(1,1)	LL LL
B(1,2)	LL LR
.	.
.	.
B(3,2)	RL LR
B(3,3)	RL RL

The complete structure of array B is shown in figure 30.

Some high-level programming languages like PL/I allow zero or negative indices. If the index range starts with zero the first step of the mapping algorithm (decrementing index by 1) is eliminated. If index range starts with a negative integer the index should be decremented by the starting value of the index. Let array A be declared as:

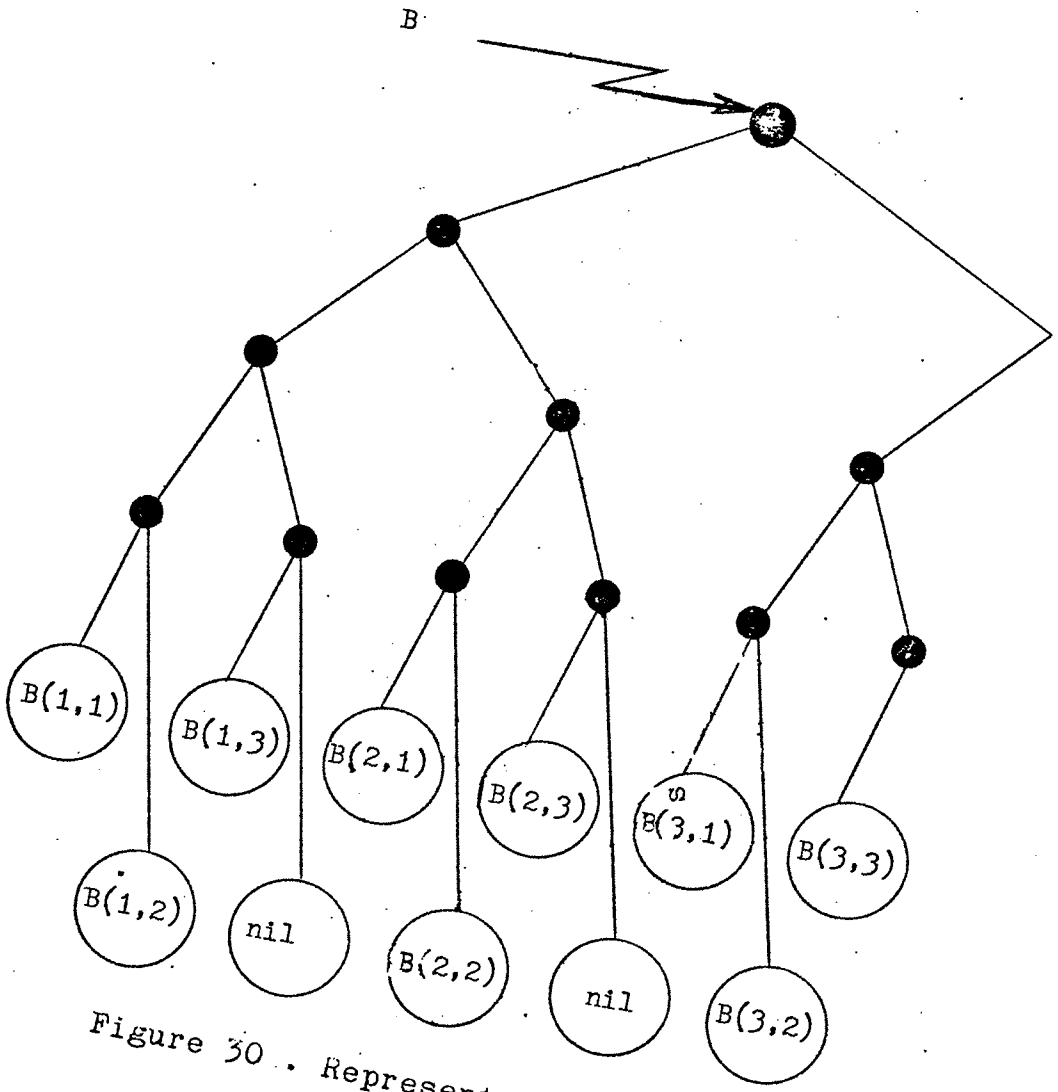


Figure 30 . Representation of B(3,3)

dcl A(m:n)

where m and n are signed integers, then the allocation algorithm may be generalized as follows:

- use the name of the array as a poiter to the root node
- compute n-m+1 and find v such that

$$2^{v-1} < n-m+1 <= 2^{v}$$

- represent any reference to the array A by v bits

Similarly the mapping algorithm may be generalized as follows:

- decrement index by m
- convert index to a v-bit binary number
- use converted binary number as a selector to reference the elements of the array

The allocation and mapping of the multidimensional arrays may be generalized by few modifications. Let the two-dimensional array B be declared as follows:

dcl B(m:n,p:q)

Where m,n,p, and q are signed integers, then the allocation algorithm may be generalized as follows:

- use the name of the array as a pointer to the root node
- compute n-m and q-p and find v and w such that

$$2^{v-1} < n-m+1 <= 2^{v} \qquad 2^{w-1} < q-p+1 <= 2^{w}$$

_ represent any reference to the array B in v+w bits

Similarly the mapping algorithm may be generalized as follows:

_ decrement first index by m

_ decrement second index by p

_ convert first index to a v-bit binary number

_ convert second index to a w-bit binary number

_ concatenate two numbers to form the selector

Although the proposed mapping function for multidimensional arrays is the easiest method, it is not the best. When the index ranges are not actual powers of 2, the depth of the binary tree grows more than it is required to represent all elements of the array. Consider the array A(3,17), using the concatenation method 7 bits (2 for rows and 5 for columns) are required to represent the selector and the tree will grow up to 7 levels. However, A containns only 51 elements that may be represented in 6 levels. To reduce the depth of the tree, a mathematical function may be used to map the indices to the range of product of the subscript ranges. Assume array A declared as A(m,n), then element A(i,j) may be selected by the selector

binary equivalent of ((i-1)*n+j-1)

This method needs 4 arithmetic operations to map an index to the corresponding selector. The concatenation method uses

only two simple mathematical operations (subtractions). Since speed is the major goal in the design of the data flow architecture, the first approach seems more attractive.

C.1.b. Operations. The array operations normally consist of accessing and/or modifying an individual element or a specific group of elements of an array and may be categorized as follows:

- accessing/modifying an individual element
- accessing/modifying a specific row or column of a matrix
- accessing/modifying the whole array

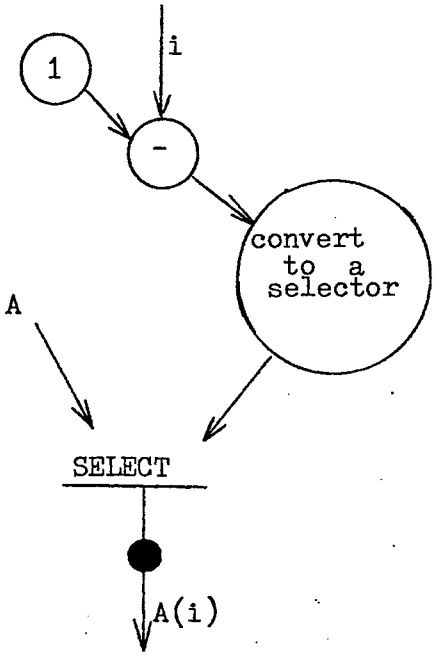
Methods of mapping the individual elements of an array have been discussed previously. Accessing an individual element follows the previously described methods; converting indices to a proper selector, and using that to reference the element. The SELECT actor is used as a basic operator to activate architectures structure handling mechanism to fetch and transfer referenced element. For example, the data flow code segment shown in Figure 31a is used to reference A(i). Individual elements may be modified using the ALTER basic operator which modifies an individual element designated by a specific selector to the given value, the result is another structure. For example, to modify the value of A(i) to 5, the data flow code segment shown in Figure 31b is used. Since modifying element(s) of an array includes accessing too, only the modify algorithms and the associated

only two simple mathematical operations (subtractions). Since speed is the major goal in the design of the data flow architecture, the first approach seems more attractive.

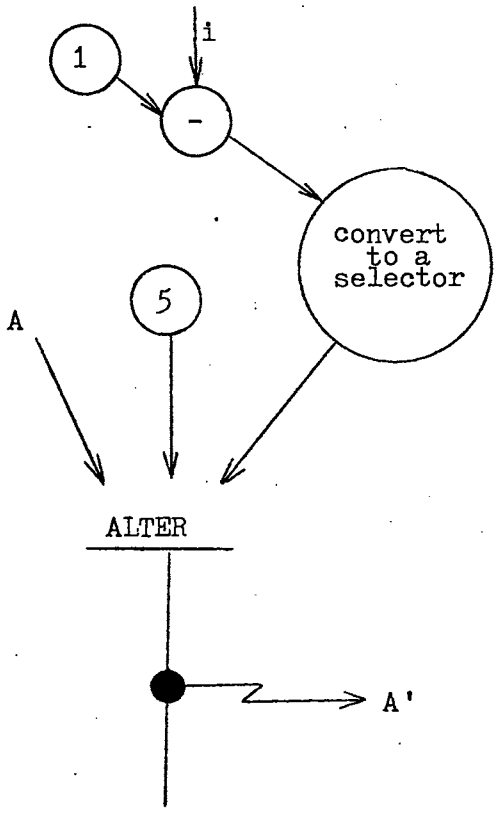
C.1.b. Operations. The array operations normally consist of accessing and/or modifying an individual element or a specific group of elements of an array and may be categorized as follows:

- accessing/modifying an individual element
- accessing/modifying a specific row or column of a matrix
- accessing/modifying the whole array

Methods of mapping the individual elements of an array have been discussed previously. Accessing an individual element follows the previously described methods; converting indices to a proper selector, and using that to reference the element. The SELECT actor is used as a basic operator to activate architectures structure handling mechanism to fetch and transfer referenced element. For example, the data flow code segment shown in Figure 31a is used to reference A(i). Individual elements may be modified using the ALTER basic operator which modifies an individual element designated by a specific selector to the given value, the result is another structure. For example, to modify the value of A(i) to 5, the data flow code segment shown in Figure 31b is used. Since modifying element(s) of an array includes accessing too, only the modify algorithms and the associated



(a) Access A(i)



(b) Modify A(i)

Figure 31. Codes to Access/Modify an Individual Element of Array A

code segments are represented in next sections.

Some programming languages allow reference to a specific group of items (a row or column). A reference to a special row or column of an array may be done by keeping one of the indices fixed and changing the other index from the lower limit of the corresponding dimension up to the upper limit of that. For example, consider the array A declared as A(4,6), then a reference as A(2,*) is interpreted as a reference to all elements of the second row and A(*,3)is interpreted as a reference to all elements of the third column. The following code segment represents an example of this type of array reference:

```
dcl  A(3,6),B(4,6)
  .
A(2,*)=2*B(*,3)
```

This process may be represented in detail as

```
       i=1
loop:  A(2,i)=2*B(3,i)
       i=i+1
       if (i<7) go to loop
```

The corresponding data flow code is shown in Figure 32.

Reference to the whole array is possible in some high-level programming languages by using the name of the array without any index. Consider the following code segment:

```
dcl  A(3,4),B(3,4),c(3,4)
```

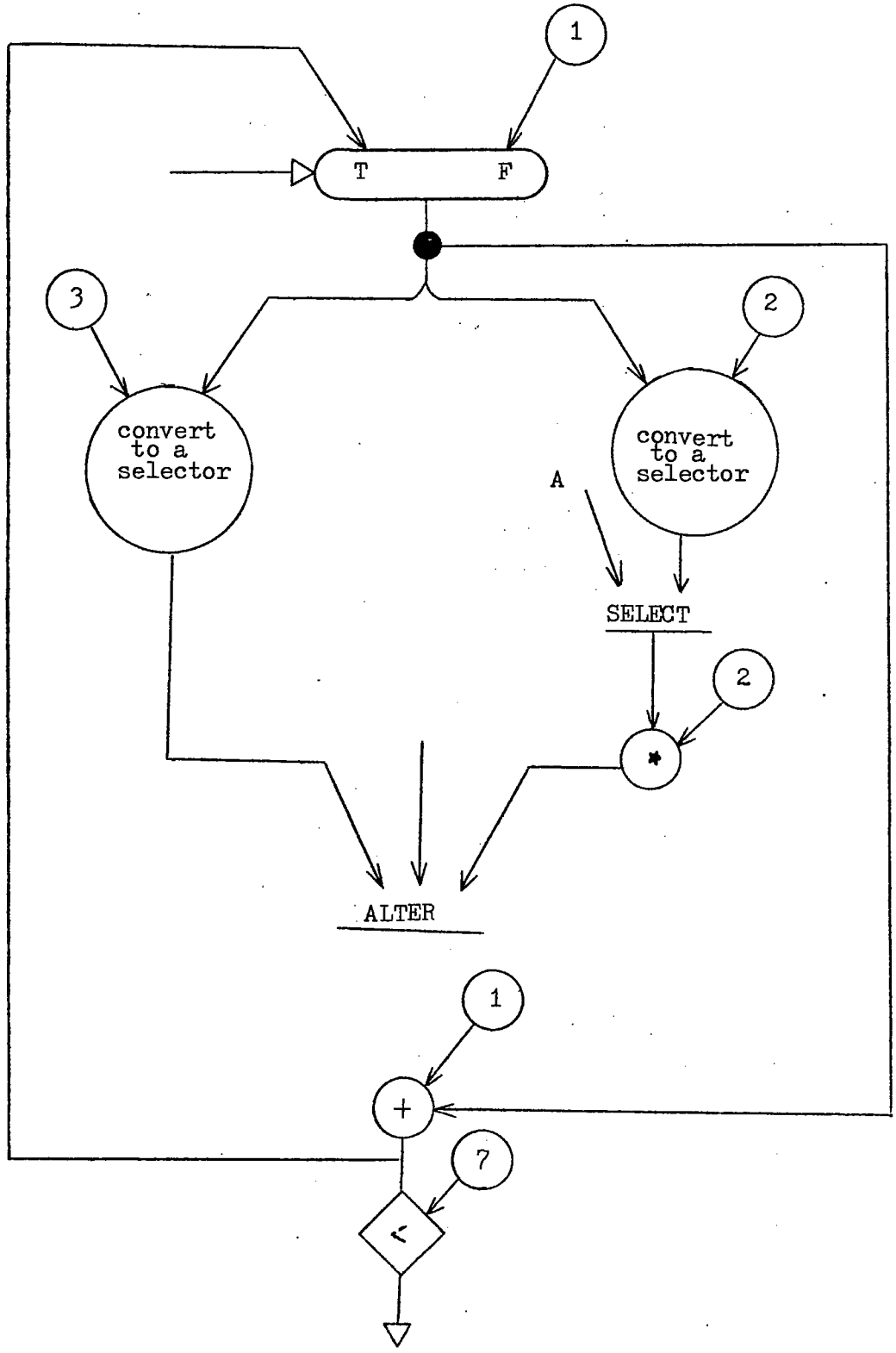


Figure 32. Data Flow Code to Perform A(2,*)=2*B(*,3)

```
C=A+B
```

In the above program, the statement C=A+B is equivalent to the following code segment

```
        i=1, j=1
loop:   C(i,j)=A(i,j)+B(i,j)
        j=j+1
        if (j<5) go to loop
        j=1
        i=i+1
        if (i<4) go to loop
```

The corresponding data flow code is shown in Figure 33.

C.2. Stacks

Stack is a sequence of items, which is permitted to grow only by special disciplines for adding and removing items at its endpoints. As the name stack suggests, it is conventional to think of the items in a stack as being piled on top of one another, with the most recently inserted item at the top and the least recently inserted item at the bottom. Deleting the topmost item is often called popping the stack, and inserting a new item on the top is often called pushing the item onto the stack. There are two different methods to implement a stack; linear implementation, in which stack is treated as a sequential list of items together with a pointer (stack pointer) which

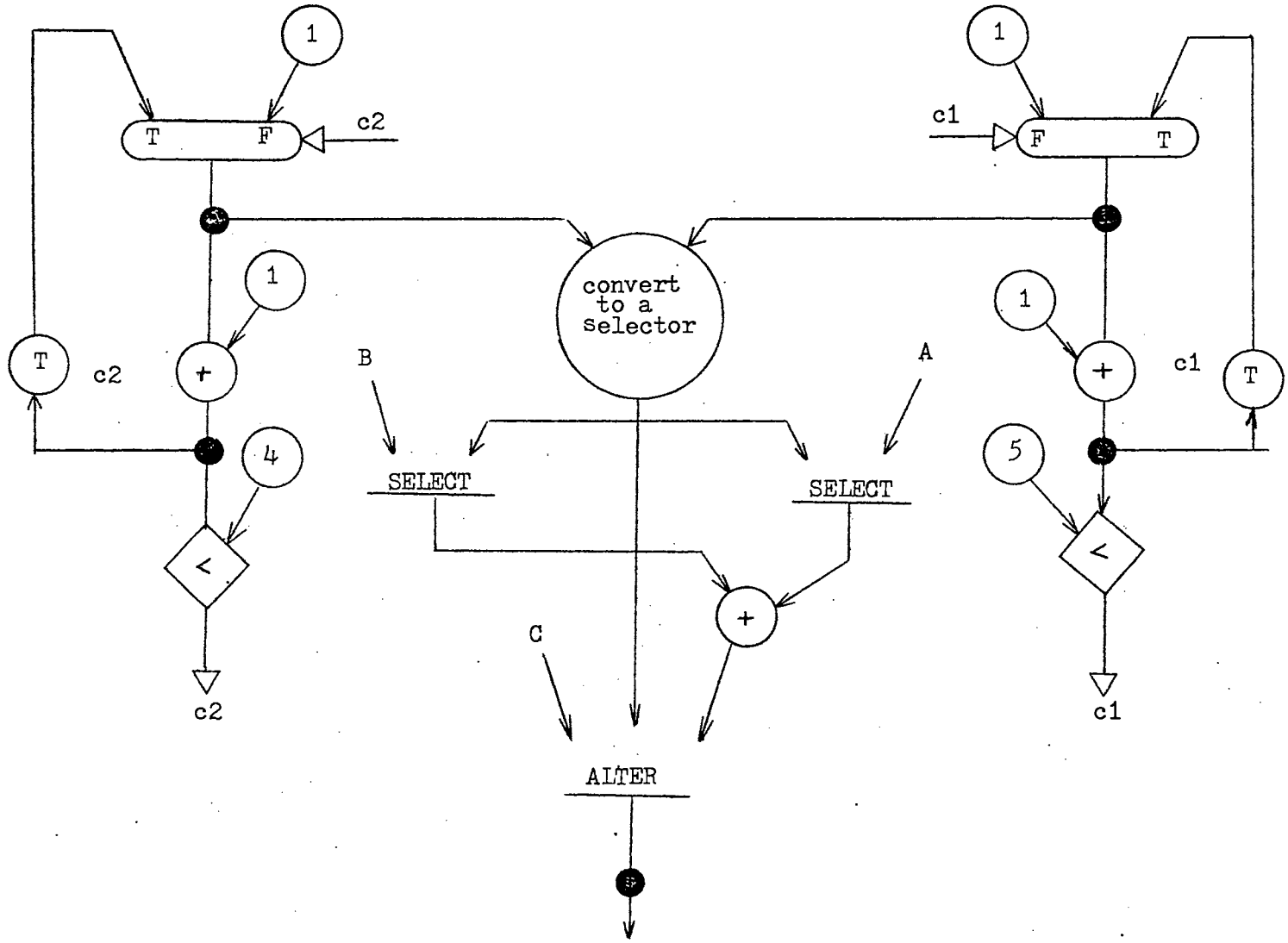
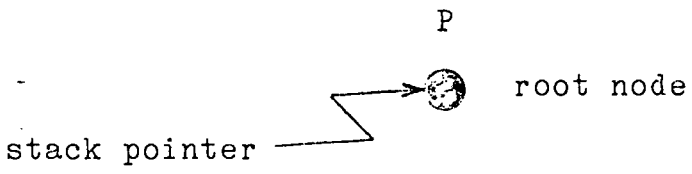


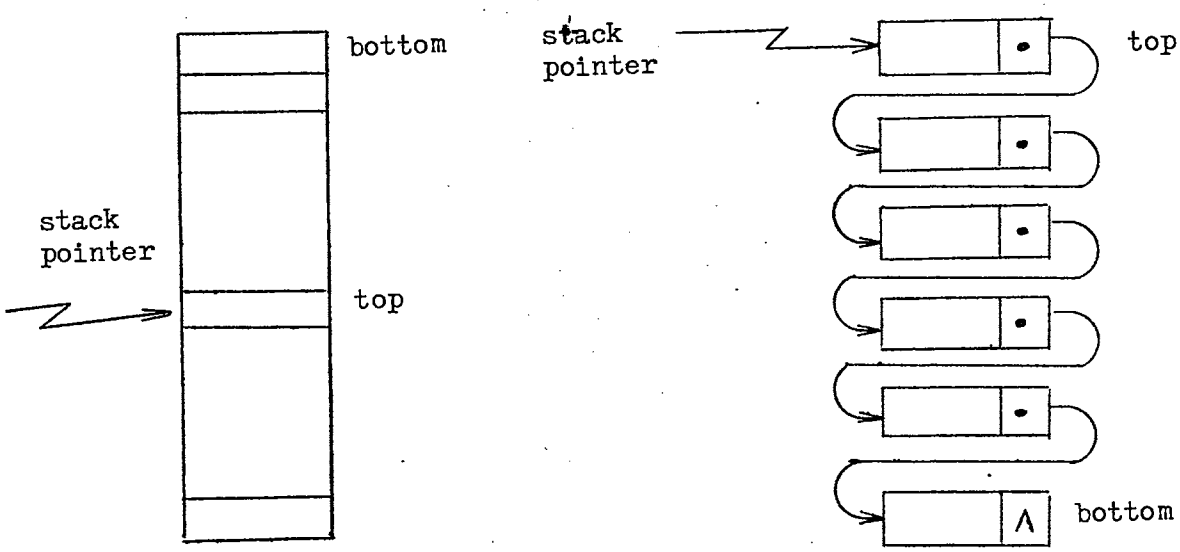
Figure 33. Data Flow Code to Perform C=A+B

points to the topmost element of stack and linked representation, in which elements of stack are linked to each other. Figure 54 represents these methods.

Since in data flow architecture, basic structure representation is the binary tree, stacks should be mapped onto a binary tree. To manipulate a stack, two pieces of information are required. First a pointer to the top of the stack, second a method to update the pointer so that it always points to the most recently inserted item. The structure representing the stack is always pointed by a pointer say S. Stack manipulation may be performed using sequential stack manipulation rules, that is, initializing stack pointer to zero, incrementing it by 1 after any insertion (PUSH) , and decrementing by 1 before any deletion (POP). Using this method the numeric value of the stack pointer may be used as a selector to select the topmost element. The value of the stack pointer should be saved either together with pointer S (pointer to the root of the structure) or in root node of the structure (by adding one more field to the root). Assume that the stack pointer is kept together with the pointer S, then for stack P, structure pointer looks like



As previously discussed, the initial value of the stack



Linear Representation

Linked Representation

Figure 34. Stack Allocation Methods

pointer may be set to zero. The dimension of the stack (maximum number of elements in stack) specifies the length of the selector. For example, when the dimension is specified as 16, the maximum length of stack pointer would be 4 elements varying from 0000 to 1111 (interpreting 0s as left and 1s as right). Using these assumptions algorithm for pushing an item into stack is as follows:

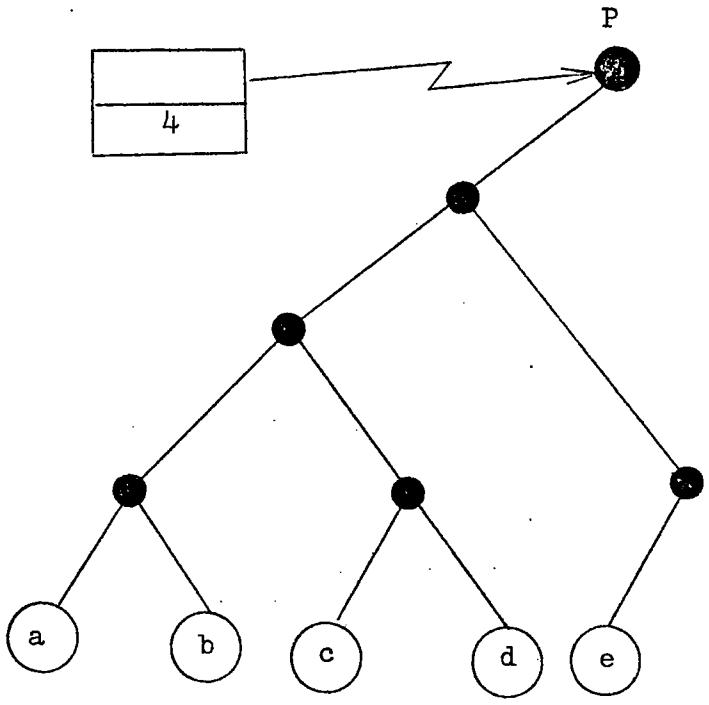
_ convert stack pointer to a selector
_ APPEND the topmost item using the selector
_ increment stack pointer by 1

For example, lets stack P with maximum length of 16 be empty, then inserting items a,b,c,d,e into stack produces the structure represented in Figure 35a.

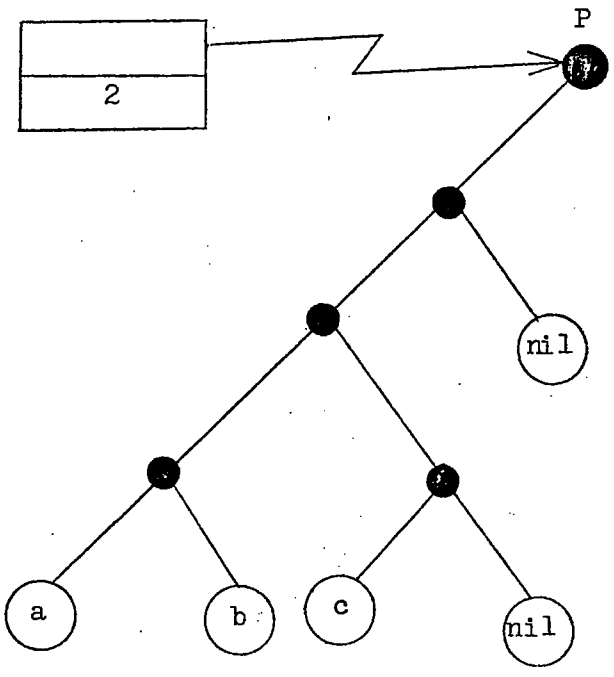
Algorithm to pop an element from a stack is as follows:

_ decrement 1 from stack pointer
_ convert stack pointer to a selector
_ SELECT the element using the selector
_ DELETE the element

For example, poping the two topmost elements from stack P in Figure 35a produces a structure shown in Figure 35b. Special conditions like overflow or underflow of the stack may be handled by checking the value of stack pointer with the lower and upper limits of the stack boundary, i.e., zero and 15 in case of stack P. The data flow code segment to push and pop an element is illustrated in Figure 36.

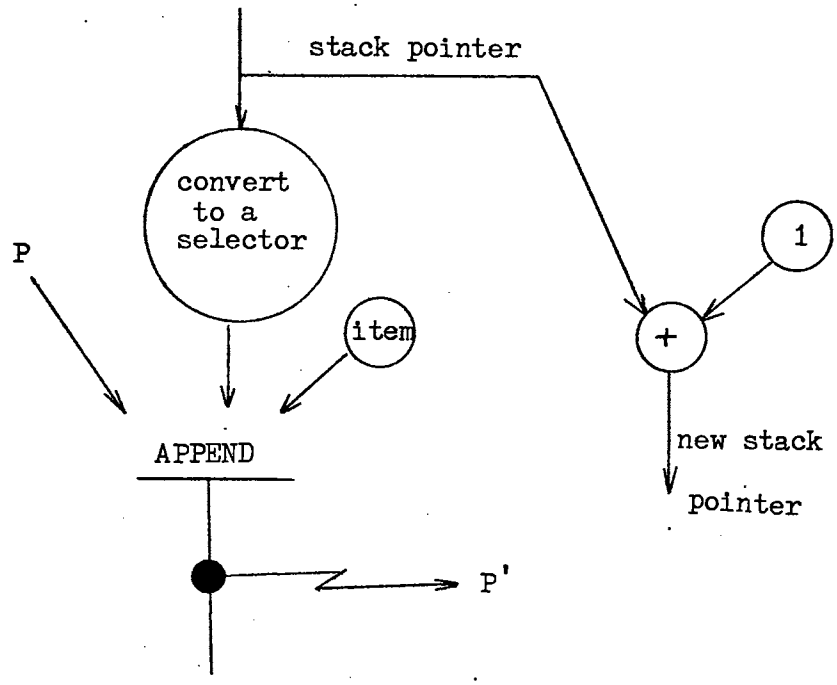


(a) PUSH Items a,b,c,d, and e Into Stack P

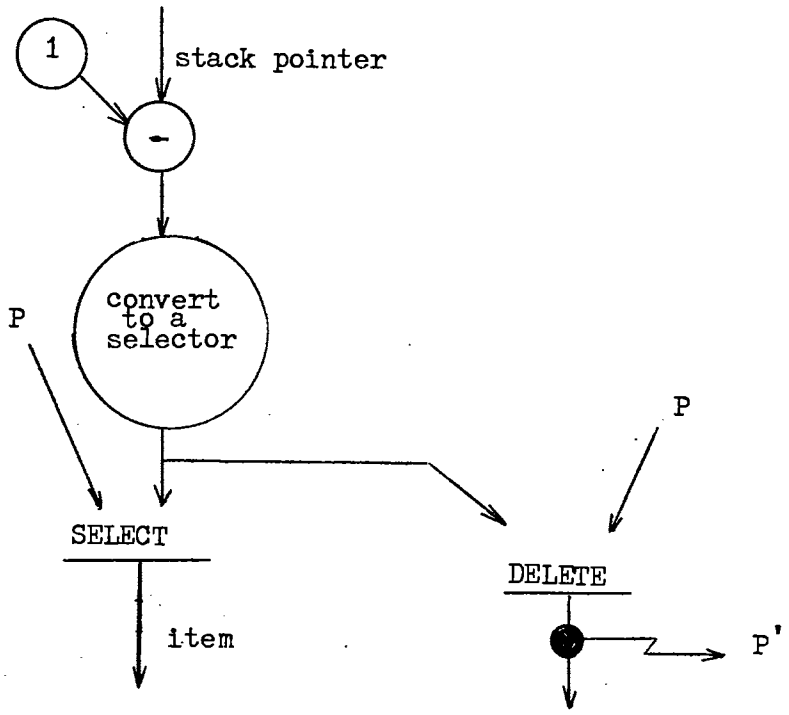


(b) POP items e and d from Stack P

Figure 35. PUSH/POP into/from Stack



(a) Data Flow Code to Perform PUSH



(b) Data Flow Code to perfomorm POP

Figure 36. Data Flow Codes to Perform PUSH and POP Operations

Selector may be constructed using another method:

_ initialize stack pointer to L(R)

_ concatenate a L(R) to stack pointer after any PUSH

_ delete a L(R) from selector before POP

Using this method, the structure grows on one side not like a complete binary tree, consequently, the depth of the tree is higher than the previous case and search time increases accordingly. The length of the selector is much longer in this case but the selector processing routine is much simpler. The number of memory spaces used to hold data items and structure pointers decreases. For example, to push items a,b,c,d,and e, selectors L,LL,LLL,LLLL, and LLLLL are used. Structure produced using this method is represented in Figure 37.

C.3. Queues

A queue is a sequece of items which grows under special disciplines. Items are added to the rear of queues and deleted from the front, this is analogous to a waiting line. Methods presented to implement a stack may be used in queue implementation with slight modifications.

To implement a queue two pointers are required to point to the front and rear of the queue. These pointers may be saved together with the pointer to the root of the structure representing the queue. For a linear implementation of a queue, both of these values may be initialized to zero. The

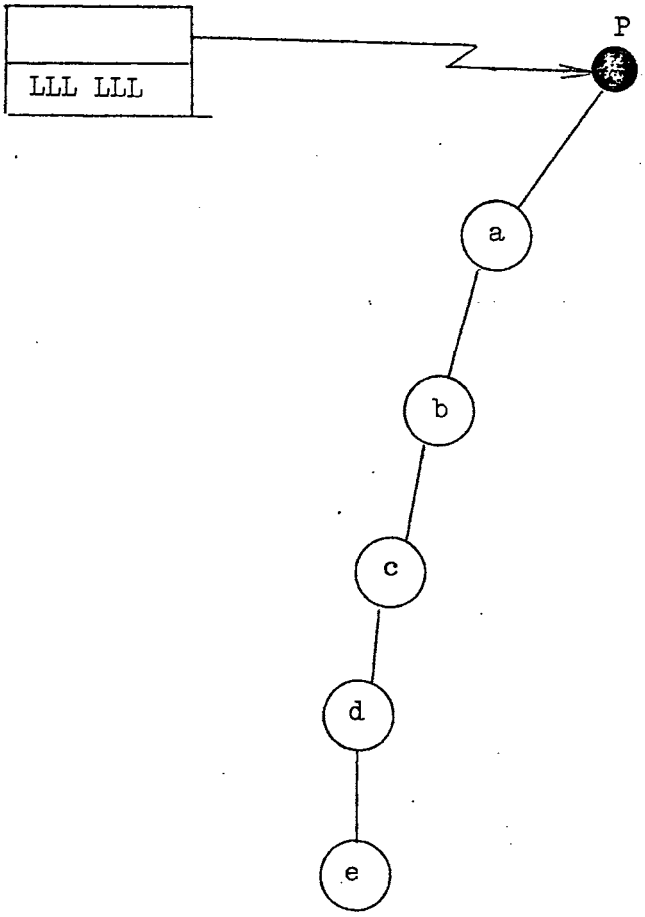


Figure 37. Stachk Constructed Using Non-linear Concepts

value of these pointers is incremented/decremented by 1 after any insertion/deletion of an item to/from the queue, and the value is used as a selector to access the item. The dimension of the queue is used to realize the length of the selector. The insertion algorithm is as follows:

_ convert Qrear to a selector
_ APPEND item to tree using the selector
_ increment Qrear by 1

The deletion algorithm is as follows:

_ convert Qfront to a selector
_ select item from queue using the selector
_ DELETE the item pointed to by Qfront
_ increment Qfront by 1

Special conditions like overflow and underflow may be handled comparing the values of Qrear or Qfront with the boundaries of queue. For example lets assume queue Q has at most 8 items, then associated selector consists of 3 identifiers. The values of Qfront and Qrear are initially zero, then the sequence of operations:

_ insert a
_ insert b
_ insert c
_ delete
_ insert d
_ insert e

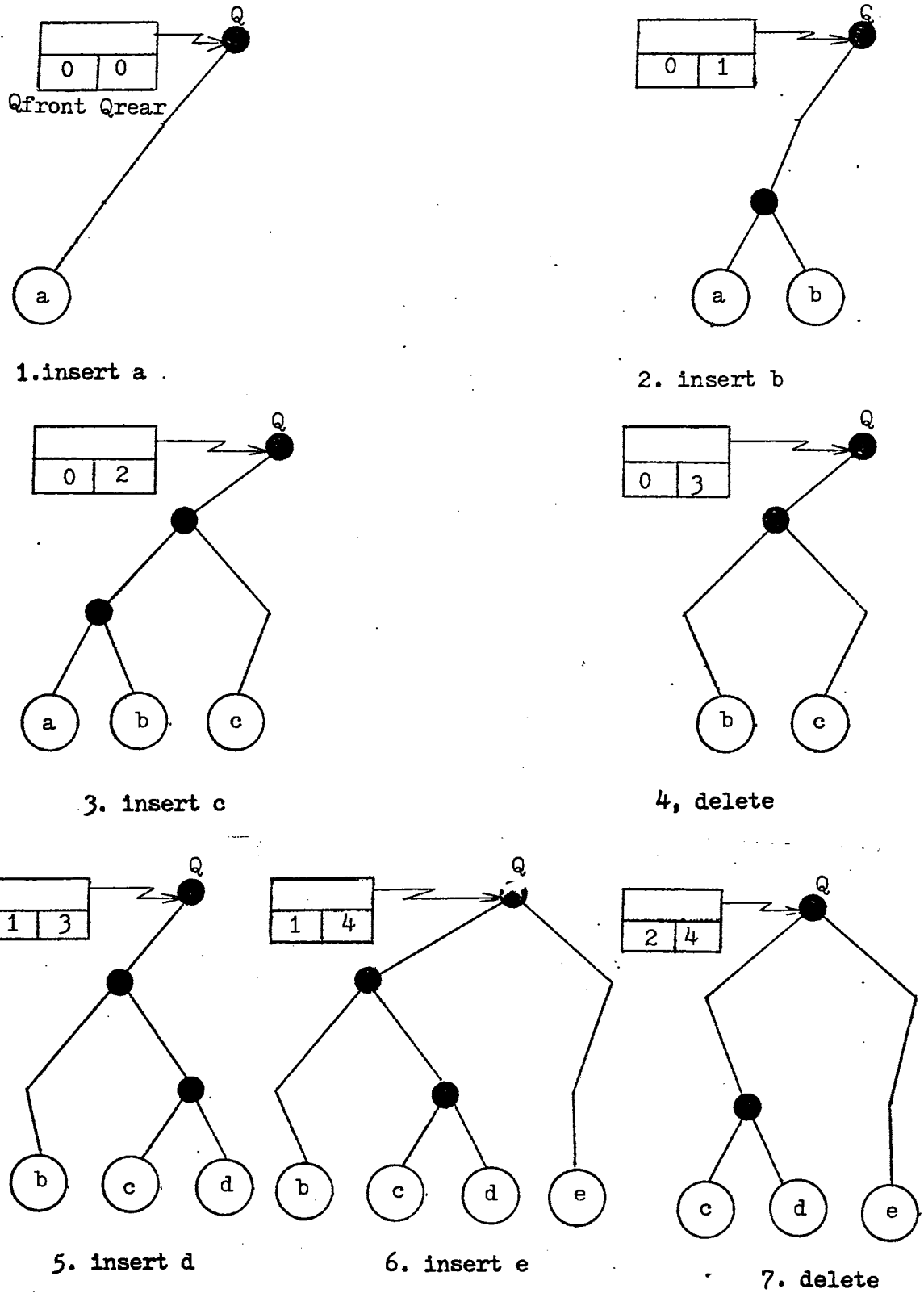


Figure 38. Structures Produced by a Sequence of Insertions and Deletions into and from Queue

_ delete

produces structures shown in Figure 38.

The concatenation method may be used to construct selectors for queues, but a slight modification is required in deletion algorithm. The deleted element may not be actually deleted unless the queue is reconstructed making root node point to the Qfront and modifying Qfront and Qrear accordingly. Since these operations take a considerable amount of time and the data flow computer is intended to be as fast as possible architecture, this method is not an appropriate one.

D. Procedures

In sequential programming languages, the abstraction obtained by using procedures is a useful one. The ability to define and call procedures is a great assest in a programming language. procedures:

- Permit modular design of programs, by allowing large tasks to be broken into smaller units.
- Permit economy in size of programs and in the total programming effort, since similar computations need be specified only once.
- Add extensibility to a language, since operators can be defined in terms of procedures, which can then be used as functions within expressions.

One problem arising from the introduction of procedures is that a method of transmitting information to and from

procedures must be defined and established.

In data flow base language APPLY actor is used to call a procedure. It has m inputs, n outputs and is labeled with a procedure name P. The APPLY actor when enabled to fire, substitutes for itself a copy of the procedure whose name matches that of the actor. This action takes place only if a procedure exists with name P and the procedure has the same number of inputs and outputs as the actor.

To completely understand how the APPLY actor works, the enabling condition, the mechanism for transmitting input values to the copied procedure, and the return mechanism for results must be defined. There are two alternatives:

1. The APPLY actor is enabled, as soon as its first argument token is arrived. It then copies the procedure (a procedure copy is called an instantiation) and passes argument tokens as they arrive. An argument is passed by absorbing a token from an input arc to the APPLY, and placing a copy of it onto the procedure instantiation s corresponding input link s output arc. The RETURN actor copies output values from the procedure copy as soon as they become available on the output links and the corresponding link to the calling program is empty. When values from each output link have been returned the copy is destroyed.

2. The APPLY actor is enabled only when all its argument values have arrived and its output links are empty. When these two conditions are met, the procedure is copied and the argument tokens are passed. When all argument

tokens are available they are copied by the RETURN actor to the output arcs of the APPLY actor. The copy of the procedure is then destroyed.

In both cases it is assumed that the n input links are numbered left to right, 0,1,...,n-1, for both the APPLY actor and the procedure it invokes. The jth link of the APPLY is associated with the jth link of the procedure it invokes. The m output links are treated in a similar fashion.

The semantics of the two approaches to procedure activation are quite different. In the first approach an APPLY actor can be thought of as replaced inline by the graph of the procedure it invokes. In the second approach an APPLY actor behaves exactly like a primitive function, except that it may have multiple outputs and computes a function that is not necessarily in the repertoire of primitive functions. The first approach is called immediate copy rule (ICR), and the second is called deffered copy rule (DCR). The DCR most closely corresponds with one s idea that a procedure is some sort of a functional abstraction, whereas the ICR is more like a macro expansion. The DCR has the advantage of simplicity of implementation. It also lends an additional homogeneity to the set of actors, since its enabling rule is that of a primitive function. However, the ICR clearly allows greater parallelism than DCR.

The ICR has one potential problem. Suppose an argument token arrives on the jth link and the execution of some

procedure is initiated. Consider what happens if another argument arrives on the jth link before the previously invoked copy of the procedure terminates. In order to be consistent another copy of the procedure must be created and this newly arrived token must be passed to its input. Thus the APPLY actor must "keep track of" an arbitrary number of concurrently executing instantiations of the procedure, and this poses some serious implementation questions. If we can demonstrate for every APPLY actor A that

$$\forall\,(i,j) \quad |P(i)-P(j)| <= 1$$

$$0 <= i,j <= \text{number of inputs of A}$$

where

P(i) = number of tokens that have arrived on the ith input of A

for any configuration of a data flow program, then we can show for any APPLY actor A of a data flow program that at most one instantiation can exist at any time, and consequently the state information is bounded. In general, data flow programs do not exhibit this behavior. However, certain large syntactic subclasses of data flow programs satisfy the above arc condition. One such class is known as well formed data flow programs. Besides having the above property, a well formed data flow program, when it terminates, will be in its initial configuration. In particular, the only tokens left on the arcs of a terminated program, will be the initial "F" tokens on the gating inputs

of MERGE gates of iterative loops.

A procedure implementation scheme was proposed by Miranker [23] based on ICR approach. This procedure is rather simple, and overhead in terms of storage, or extra packets in the system, is almost zero The deficiency of this scheme for procedure implementation is that it supports a rather primitive form of the APPLY actor _ only one input and one output. Multiple input values and multiple output values could be encoded as structures. However, such a form of procedure invocation would be undesireable because it would limit the degree of parallelism achievable.

E. Semantic Gap in Data Flow Architecture

Data flow computer architecture proposed by Dennis is designed to perform about 200 Megaflops (million floating-point operations per second). Since speed was the major goal in this design, architecture deficiencies leading to semantic gap have not been resolved. The semantic gap in a data flow computer and existing solutions to reduce this problem is studied in this section.

Logical memory structure is one of the properties of the conventional computers which contributes in causing the semantic gap. Incompatibility of linear memory structure with data structures presented in high-level languages cause performance problems and excessive program size. Memory of the data flow computer is separated into two different parts, instruction memory and structure memory, with

different logical structures.

Instruction memory is composed of fixed size instruction cells. During execution of a data flow program most of the nodes fire once. A large number of nodes of the program will not fire at all if any decider is present. Thus it would be wasteful to assign an instruction cell to each instruction of a procedure when the procedure is activated. To solve this problem the instruction processing section of the data flow computer incorporates a multi-level memory system such that only the active instructions of a program occupy the instruction cells of the processor.

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

A packet destined for the instruction memory or structure memory can no longer identify its destination by use of a cell identifier. The identifier is divided into two parts, a major address and a minor address, each containing a portion of the identifier.

All instruction cells having the same major address belong to the corresponding cell block. Thus, the distribution and control networks use the major address to direct data packets and control packets to the appropriate

instruction cell blocks. The packet delivered to a cell block includes the minor address, which serves as an identifier for that packet within the cell block.

Although multi-level memory reduces the size of the active memory, it causes some implementation problems. Tables which are used to indicate the status of each node (free, engaged, and occupied), minor address of the node, and the candidates for displacement by more active nodes occupy considerable space and delays memory access considerably. Node access and placement algorithms becomes very complicated and slow.

An instruction cell in instruction memory is composed of five registers capable of holding at most four operands at the same time. Increasing the number of registers helps to decrease the packet travel time through arbitration and distribution networks and to save memory spaces used to represent complete operation in more small cells. The proposed instruction cell can hold at most 8 destinations. When an instruction requires more destination fields, one or more extra distribution instructions must be used to convey results to all destinations. Since distribution instructions fire only after the completion of the instruction and distribution of the result, it takes as many distribution instructions required extra cycles to distribute result. Consequently, all instructions waiting for results must wait more extra time than required. A large memory cell provides enough room to hold more pointers and

prevent the delay time. Although a large instruction cell solves above problems, it causes space waste for short instructions. A variable size instruction cells may be used as a compromise.

Structure memory is composed of structure cells. Each structure cell is capable of holding one node of a structure contained in a two register cell. The two registers of the cell contain the left and right components of the structure, respectively. This organization uses a binary tree as the basic logical structure of the structure memory.

Data structures used in high-level languages may not be represented directly in the memory, then special mapping functions must be used. The allocation and mapping function was discussed previously. Including this packages in software (i.e., compiler) increases program size and packet travel time in a network tremendously. An alternative is to add these capabilities to structure processing section of the computer.

By increasing the number of structure processing units and adding a special processor to determine the type of the process and distribution of the instruction among units, structure processing time decreases considerably (Figure 39). Special purpose processing units (array, stack, and queue) perform allocation and mapping algorithms discussed previously. Ring type networks of structure memories and structure operation units increases cuncurrency specially in operations.

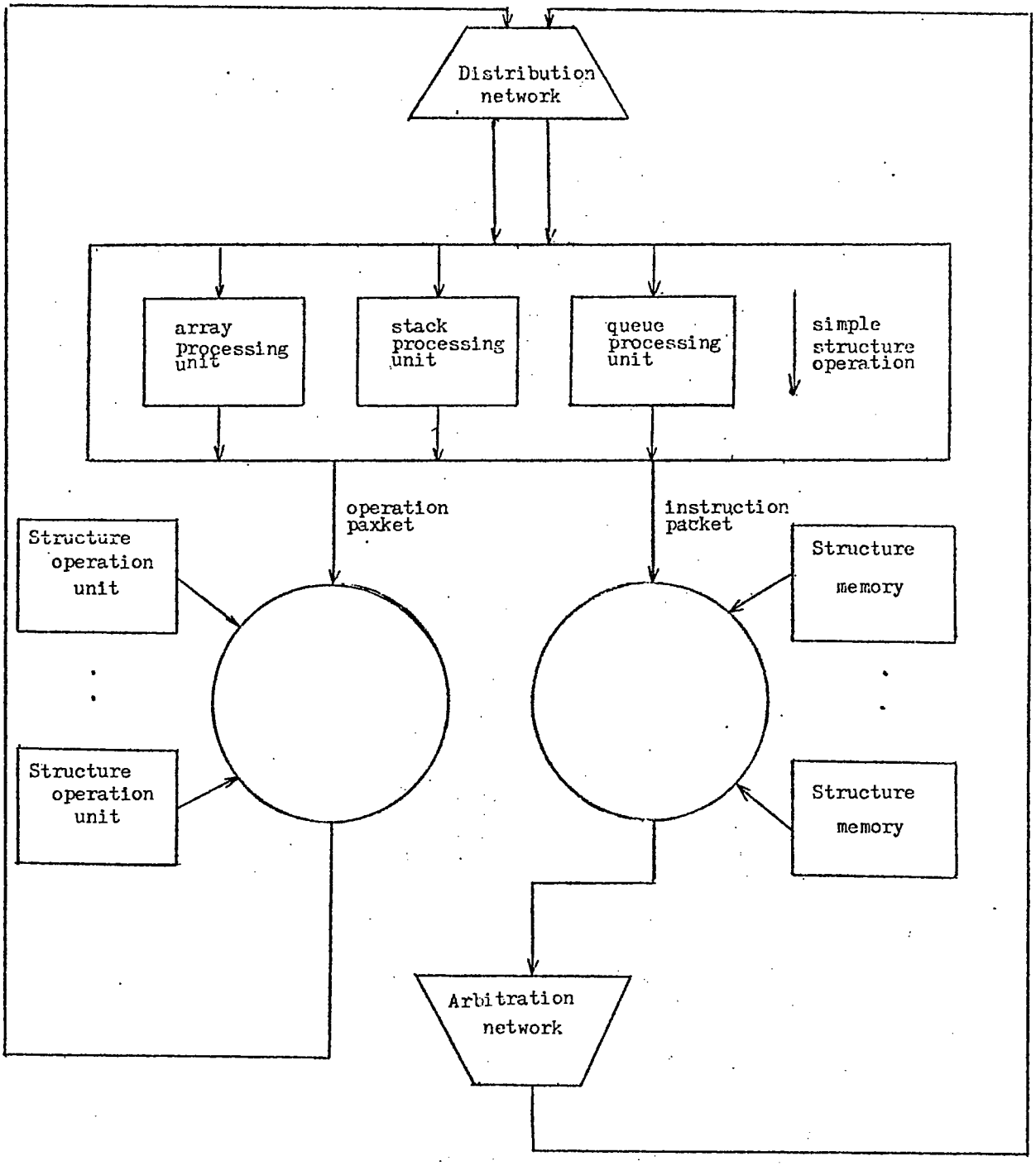


Figure 39. Expanded Structurre Processing Unit

The arithmetic processing unit of a data flow computer uses signed digit number representation to perform arithmetic operations. Although, this representation enables system to take advantage of serial properties of the representation, the computation time is not very impressive. Complex arithmetic is not available and must be handled by a compiler using multiple real arithmetic operations. Multiple-precision arithmetic is left out and no division algorithm is proposed.

Although data flow architecture is a radical and attractive approach to computer architecture, it has some shortcomings. The principles of Dennis data flow architecture was discussed in this chapter. Major high-level language concepts were coded in data flow base language, and finally, existing shortcomings were studied.

Although speed is the major goal in this design, the shortcomings contribute in many ways in reducing the speed and also creating a form of semantic gap. In Chapter VI two application programs coded in data flow base language are represented and a performance analysis of them are studied.

CHAPTER VI

TWO APPLICATIONS

A. Fast Fourier Transform

A.1. Introduction

The Discrete Fast Fourier transform plays an important role in the analysis, the design, and the implementation of digital signal processing algorithms. One of the reasons that Fourier analysis is of such wide-ranging importance in digital signal processing is because of the existence of efficient algorithms for computing the Discrete Fourier Transform.

The Discrete Fourier Transform (DFT) is

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{kn} \qquad k=0,1,\ldots,N-1 \qquad (1)$$

where $W_N = e^{-J(2\pi/N)}$. The Inverse Discrete Fourier Transform (IDFT) is

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)\, W_N^{-kn} \qquad n=0,1,\ldots,N-1 \qquad (2)$$

In equations (1) and (2) both x(n) and X(k) may be complex. The expressions of Eqs. (1) and (2) differ only in the sign of the exponent of W_N and in a scale factor 1/N. Thus a discussion of computation procedures for Eq.(1) applies with

straightforward modifications to Eq.(2).

To indicate the importance of efficient computation schemes, it is instructive to consider the direct evaluation of the DFT equations. Since x(n) may be complex we can write

$$X(k) = \sum_{n=0}^{N-1} \left(\mathrm{Re}(x(n))\,\mathrm{Re}(W_N^{kn}) - \mathrm{Im}(x(n))\,\mathrm{Im}(W_N^{kn})\right) + J\left(\mathrm{Re}(x(n))\mathrm{Im}(W_N^{kn}) + \mathrm{Im}(x(n))\mathrm{Re}(W_N^{kn})\right) \qquad k=0,1,\ldots,N-1 \qquad (3)$$

From Eq.(3) it is clear that for each value of k, the direct computation of X(k) requires 4N real multiplications (N complex multiplications) and 4N-2 real additions (N-1 complex additions). Since X(k) must be computed for N different values of k, the direct computation of the Discrete Fourier Transform of a sequence x(n) requires $4N^2$ real multiplications, or alternatively N^2 complex multiplications and N(4N-2) real additions or, alternatively, N(N-1) complex additions. In addition to the multiplications and additions called for by Eq.(3) the implementation of the computation of the DFT on a general-purpose digital computer or with special purpose hardware of course requires provision for storing and accessing the input sequence values x(n) and values of the coefficients W_N^{kn}. Since the amount of accessing and storing of data in numerical computation algorithms is generally proportional to the number of arithmetic operations, it is generally

accepted that a meaningful measure of complexity, or, of the time required to implement a computational algorithm, is the number of multiplications and additions required. Thus, for the direct computation of the Discrete Fourier Transform, a convenient measure of the efficiency of the computation is the fact that $4N^2$ real multiplications and $N(4N-2)$ real additions are required. Since the number of computations, and thus the computation time, is approximately proportional to N^2, it is evident that the number of arithmetic operations required to compute the DFT by the direct methods becomes very large for large values of N. For this reason, computational procedures that reduce the number of multiplications and additions are of considerable interest.

Most approaches to improve the efficiency of the computation of the DFT exploit one or both of the following special properties of the quantities (W_N):

1. $W_N^{k(N-n)} = (W_N^{kn})^*$

2. $W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n}$

Computational algorithms that exploit both the symmetry and priodicity of the sequence (W_N) were known long before the era of high-speed digital computation. At that time, any scheme that reduced hand computation by even a factor of 2 was welcomed.

The possibility of greatly reduced computation was generally overlooked until about 1965, when Cooley and Tukey

published an algorithm for the computation of the Discrete Fourier Transform that is applicable when N is a composite number; i.e., N is the product of two or more integers. The publication of this paper resulted in the discovery of a number of computational algorithms which have come to be known as Fast Fourier Transform, or simply FFT, algorithms.

The fundamental principle that all these algorithms are based upon is that of decomposing the computation of the Discrete Fourier Transform of a sequence of length N into successively smaller Discrete Fourier Transforms. The manner in which this principle is implemented leads to a variety of different algorithms, all with comparable improvements in computational speed.

A.2. Decimation-In-Time Algorithm

To achieve the dramatic increase in efficiency to which we have alluded, it is necessary to decompose the DFT computation into successively smaller DFT computations. In this process we exploit both symmetry and the priodicity of the complex exponential $(W_N^{kn}) = e^{-j(2\pi/N)kn}$. Algorithms in which the decomposition is based on decomposing the sequence x(n), into successively smaller subsequences, are called Decimation-In-Time algorithms. The principle of Decimation-In-Time is most conveniently illustrated by considering the special case of N an integer power of 2; i.e.,

$$N = 2^{v}$$

Since N is an even integer, we can consider computing X(k) by separating x(n) into two N/2-point sequences consisting of the even-numbered points in x(n) and the odd-numbered points in x(n). With X(k) given by

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{kn} \qquad k=0,1,\ldots,N-1 \qquad (4)$$

and separating x(n) into its even-and-odd-numbered points we obtain

$$X(k) = \sum_{n \text{ even}} x(n)\, W_N^{kn} + \sum_{n \text{ odd}} x(n)\, W_N^{kn}$$

or with the substitution of variables n=2r for n even and n=2r+1 for n odd,

$$X(k) = \sum_{r=0}^{(N/2)-1} x(2r)\, W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x(2r+1)\, W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{(N/2)-1} x(2r)(W_N)^{2rk} + W_N^{k} \sum_{r=0}^{(N/2)-1} x(2r+1)\,(W_N)^{2rk} \qquad (5)$$

but $(W_N)^2 = W_{N/2}$ since

$$W_N^2 = e^{-2J(2\pi/n)} = e^{-J2\pi/(N/2)} = W_{N/2}$$

consequently Eq.(5) can be written as

$$X(k) = \sum_{r=0}^{(N/2)-1} x(2r)\, W_{N/2}^{rk} + W_N^{k} \sum_{r=0}^{(N/2)-1} x(2r+1)\, W_{N/2}^{rk} \qquad (6)$$

$$= G(k) + W_N^{k} H(k)$$

Each of the sums in Eq.(6) is recognized as an N/2-point DFT, the first sum being the N/2-point DFT of the even-numbered points of the original sequence and the second being the N/2-point DFT of the odd-numbered points of the original sequence. Although the index k ranges over N values, $k=0,1,\ldots,N-1$, each of the sums need only be computed for k between 0 and N/2-1, since G(k) and H(k) are each periodic in k with period N/2.

After the two DFTs corresponding to the two sums in Eq.(6) are computed, they are then combined to yield the N-point DFT, X(k). Figure 40 indicates the computation involved in computing X(k) according to Eq.(6) for an eight-point sequence, i.e. for N=8. In this figure, branches entering a node are summed to produce the node variable. When no coefficient is indicated, the branch transmittance is assumed to be one. For other branches, the transmittance of a branch is an integer power of W_N. Since G(k) and H(k) are both periodic in k with period 4, then

$$H(4) = H(0) \qquad G(4) = G(0)$$
$$H(5) = H(1) \qquad G(5) = G(1)$$
$$H(6) = H(2) \qquad G(6) = G(2)$$
$$H(7) = H(3) \qquad G(7) = G(3)$$

With the computation restructured according to Eq.(6), we can compare the number of multiplications and additions required with those required for a direct computation of the DFT. Previously we saw that for direct computation without

exploiting symmetry, N^2 complex multiplications and additions were required. By comparison, Eq.(6) requires the computation of two N/2-point DFTs, which in turn requires $2(N/2)^2$ complex multiplications and approximately 2(N/2) complex additions. Then the two N/2-point DFTs must be combined, requiring N complex multiplications corresponding to multiplying the second sum by W_N and then N complex additions, corresponding to adding that product to the first sum. Consequently, the computation of Eq.(6) for all values of k requiers $N+2(N/2)^2$ or $N+(N^2/2)$ complex multiplications and complex additions. It is easy to verify that for N > 2, $N+N^2/2$ will be less than N^2.

Equation (6) corresponds to breaking the original N-point computation into two N/2-point computations. If N/2 is even, as it always is when N is equal to a power of 2, then we can consider computing each of the N/2-point DFTs in Eq.(6) by breaking each of the sums in Eq.(6) into two N/2-point DFTs, which would then be combined to yield the N/2-point DFTs. Thus G(k) and H(k) in Eq.(6) would be computed as indicated below:

$$G(k)=\sum_{r=0}^{(N/2)-1} g(r)W_{N/2}^{rk}=\sum_{l=0}^{(N/4)-1} g(2l)W_{N/2}^{2lk}+\sum_{l=0}^{(N/4)-1} g(2l+1)W_{N/2}^{(2l+1)k}$$

or

$$G(k) = \sum_{l=0}^{(N/4)-1} g(2l)\, W_{N/4}^{lk} + W_{N/2}^{k} \sum_{l=0}^{(N/4)-1} g(2l+1)\, W_{N/4}^{lk} \qquad (7)$$

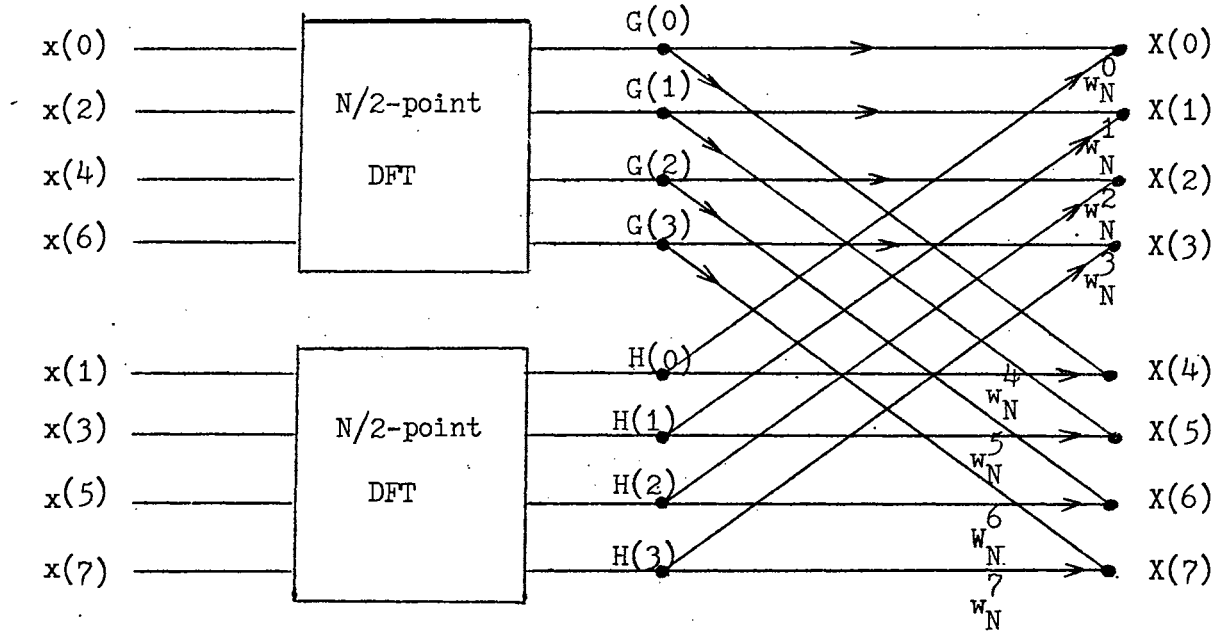


Figure 40. Flow Graph of the Decimated-In-Time Decomposition of an 8-point DFT Computation

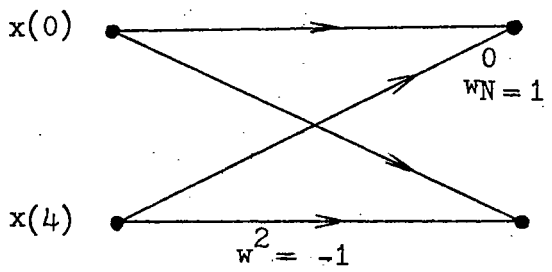


Figure 41. Flow Graph of a 2-point DFT

similarly

$$H(k) = \sum_{l=0}^{(N/4)-1} h(2l)\, W_{N/4}^{lk} + W_{N/2}^{k} \sum_{l=0}^{(N/4)-1} h(2l+1)\, W_{N/4}^{lk} \qquad (8)$$

Note that we have used the fact that $W_{N/2} = (W_N^2)$.

For the eight-point DFT that we have been using as an illustration, the computation has been reduced to a computation of two-point DFTs. The two-point DFT of, for example x(0) and x(4) is depicted in Figure 41. A complete flow graph for computation of the eight-point DFT is shown in Figure 42.

For the more general case with N a power of 2 greater than 3, we would proceed by decomposing the N/4-point transforms in Eq.(7) and (8) into N/8-point transforms, and continue until left with only two-point transforms. This requires v stages of computation, where v = log (N). Previously we found that in the original decomposition of an N-point transform into two N/2-point transforms, the number of complex multiplications and additions required was $N+2(N/2)^2$. When the N/2-point transforms are decomposed into N/4-point transforms, then the factor of $(N/2)^2$ is replaced by $N/2 + 2(N/4)^2$, so the overall computation then requires $N + N + 4(N/4)^2$ complex multiplications and additions. If $N = 2^v$, this can be done at most v = log (N) times, so that after carrying out this decomposition as many times as possible the number of complex multiplications and additions is equal to N log (N).

It is useful to note that each stage of the computation takes a set of N complex numbers and transforms them into another set of N complex numbers. When implementing the computation we can imagine the use of two arrays of (complex) storage registers, one for array being computed and one for the data being used in the computation. We shall denote the sequence of complex numbers resulting from the mth stage of computation as Xm (l), where l=0,1,...,N-1 and m=1,2,...,v. Furthermore, for convenience, let us define the set of input samples as XO (l). We can think of Xm (l) as the input array and Xm+1 (l) as the output array for the (m+1)th stage of computations; thus for the case of N=8,

$$XO\ (0) = x(0)$$
$$XO\ (1) = x(4)$$
$$XO\ (2) = x(2)$$
$$XO\ (3) = x(6)$$
$$XO\ (4) = x(1)$$
$$XO\ (5) = x(5)$$
$$XO\ (6) = x(3)$$
$$XO\ (7) = x(7)$$

Using this notation and ordering, it can be seen that the basic computation is shown as Figure 43. The equations represented by this flow graph are of the form

$$Xm+1\ (p) = Xm\ (p) + W_N^r\ Xm\ (q)$$

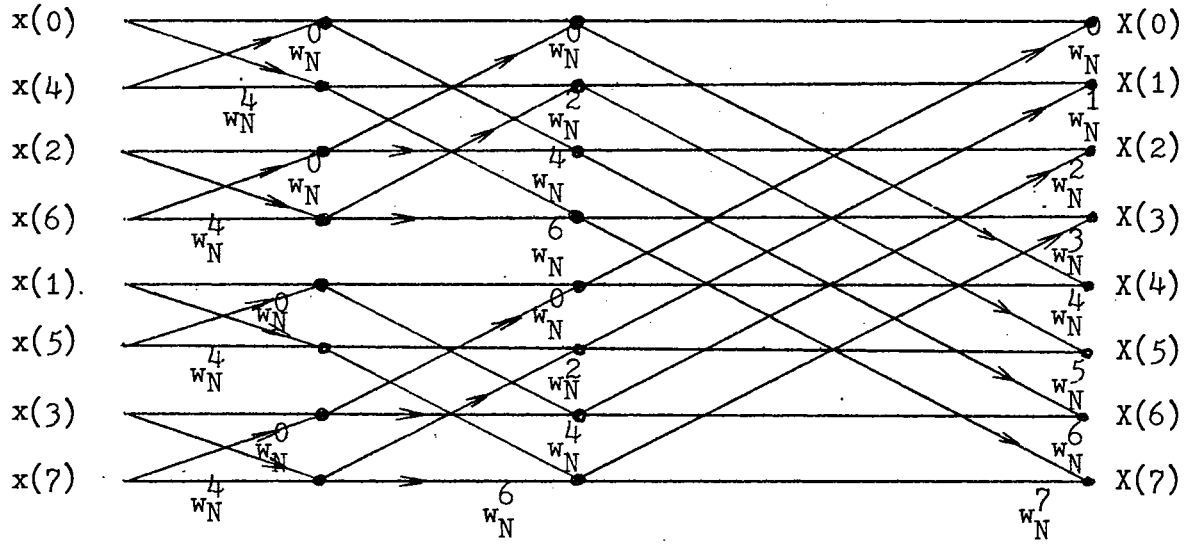


Figure 42. Flow Graph of Compltete Decimated-In-Time Decomposition of an 8-point DFT

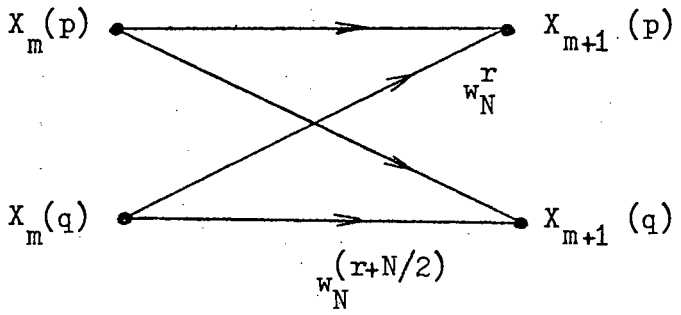


Figure 43. Flow Graph of Basic Butterfly Computation

$$Xm+1\ (q) = Xm\ (p) + W_N^{(r+N/2)}\ Xm\ (q) \qquad (9)$$

Because of the appearance of the flow graph, this computation is reffered to as butterfly computation.

Equation (9) suggests a means of reducing the number of multiplications by a factor of 2. To see this we note that

$$W_N^{N/2} = e^{-J(2\pi/N).N/2} = e^{-J\pi} = -1$$

so that the equations (9) becomes

$$Xm+1\ (p) = Xm\ (p) + W_N^{r}\ Xm\ (q)$$

$$Xm+1\ (q) = Xm\ (p) - W_N^{r}\ Xm\ (q) \qquad (10)$$

Since there are N/2 "butterflies" per stage and log (N) stages, the total number of multiplications required is (N/2) log (N). Using the new approach the flow graph of 8-point DFT is illustrated in Figure 44.

In order that computation may be done in place using a single array we note that input data must be stored in non-sequential order. In fact the order in which the input data are stored is in bit-reversed order. To see what is meant by this terminology, we note that for the eight-point flow graph, three binary digits are required to index through the data. If we write the indices in binary form, we obtain the set of equations

$$XO\ (000) = x(000)$$

$$XO\ (001) = x(100)$$

$$XO\ (010) = x(010)$$

$$X0\ (011) = x(110)$$
$$X0\ (100) = x(001)$$
$$X0\ (101) = x(101)$$
$$X0\ (110) = x(011)$$
$$X0\ (111) = x(111)$$

If (n2,n1,n0) is the binary representation of the index of sequence x(n), then the sequence value x(n2 n1 n0) is stored in the array position X0(n0 n1 n2). That is, in determining the position of x(n2 n1 n0) in the input array, we must reverse the order of the bits of the index n.

In realizing the computations, it is clearly necessary to access elements of intermediate arrays in non-sequential order. Thus, for greater computational speed, the complex numbers must be stored in random access memory. For example, to compute the first array from the input array, the inputs to each butterfly computation are adjacent node variables which are thought of as being stored in adjacent storage locations. In computing the second intermediate array from the first, the inputs to a butterfly are separated by two storage locations, and in computing the third array from the second, the inputs to a butterfly computation are separated by four storage locations. If N is larger than 8, the separation between butterfly inputs is 8 for the fourth stage, 16 for the fifth stage, etc. The separation in the last (vth) stage is N/2.

A rearrangement of the flow graph, that is particularly useful when random access memory is not available is shown

in Figure 45. This flow graph represent the Decimation-in-Time algorithm. Note first that in this flow graph the input is again in bit-reversed order and the output in normal order. The important feature of this flow graph is that the geometry is identical for each stage; only the branch transmitances change from stage to stage. This makes it possible to access data sequentially.

A.3. Data Flow Representation of the DFT Algorithm

The general form of DFT algorithm may be described as follows: let U(m,k) be the kth component of the vector of values computed by the mth stage of the computation. Then B(m,q) the qth butterfly of stage m computes

$$U(m,q) = U(m-1,2q) + U(m-1,2q+1)\, W^{e(m,q)} \tag{11}$$

$$U(m,q+2^{(n-1)}) = U(m-1,2q) - U(m-1,2q+1)\, W^{e(m,q)} \tag{12}$$

where the exponent e(m,q) of each phase factor is given by

$$e(m,q) = 2^{n-m}\ \text{quo}\ (q, 2^{n-m}) \tag{13}$$

and

$$0 < q < 2^{n-1}$$

$$0 < m < n$$

$$n = \log (N)$$

The symbol "quo" denotes the function quo(i,j) which yields the integer quotient of i divided by j. The input values for stage one are related to the data samples by

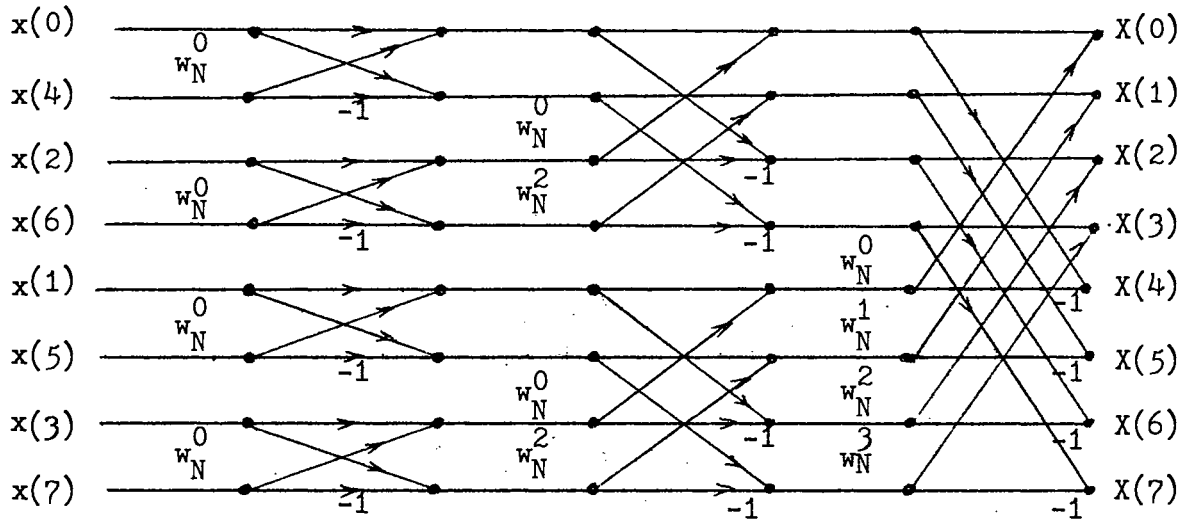


Figure 44. Flow Graph of 8-point DFT Using the Butterfly Computation of Figure 43.

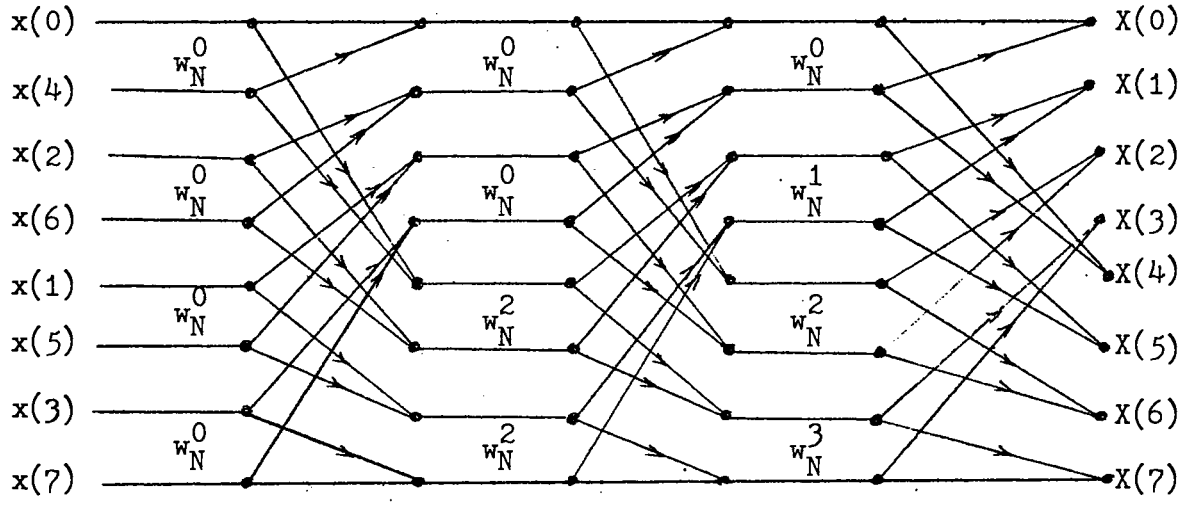


Figure 45. Rearrangement of Figure 44 Having the Same Geometry for Each Stage

$$U(0,k) = x(i) \quad \text{where } i = rev(k)$$

in which "rev" is the operation on integers such that the n-bit binary representation of i is the reverse of the n-bit representation of k. The output values are

$$f(k) = U(n,k) \qquad 0 < k < 2$$

Using new terminology the eight-point, constant geometry decimated-in-time is shown in Figure 46.

The goal is to take maximum advantage of parallelism in representing the FFT as a data flow program, but since each actor will take space in the machine representation, we dont want to use a larger program than necessary to exploit concurrency. Since each stage of the computation uses values computed by the preceeding stage, it is appropriate to write the program as an n-cycle iteration in which the body consists of the $2^{(n-1)}$ butterflies comprising one stage of computation written out explicitly. The form of the corresponding data flow program is shown in Figure 47 for the eight-point case. This is fairly easy because the constant geometry of the computation over all stages makes it possible to use a fixed routing of values from the outputs of the butterflies to their inputs where they become operands for the next cycle. Generating the phase factors for each butterfly, however presents a problem. The usual technique is to use a table lookup in a table of powers of W, but our present data flow language includes no suitable

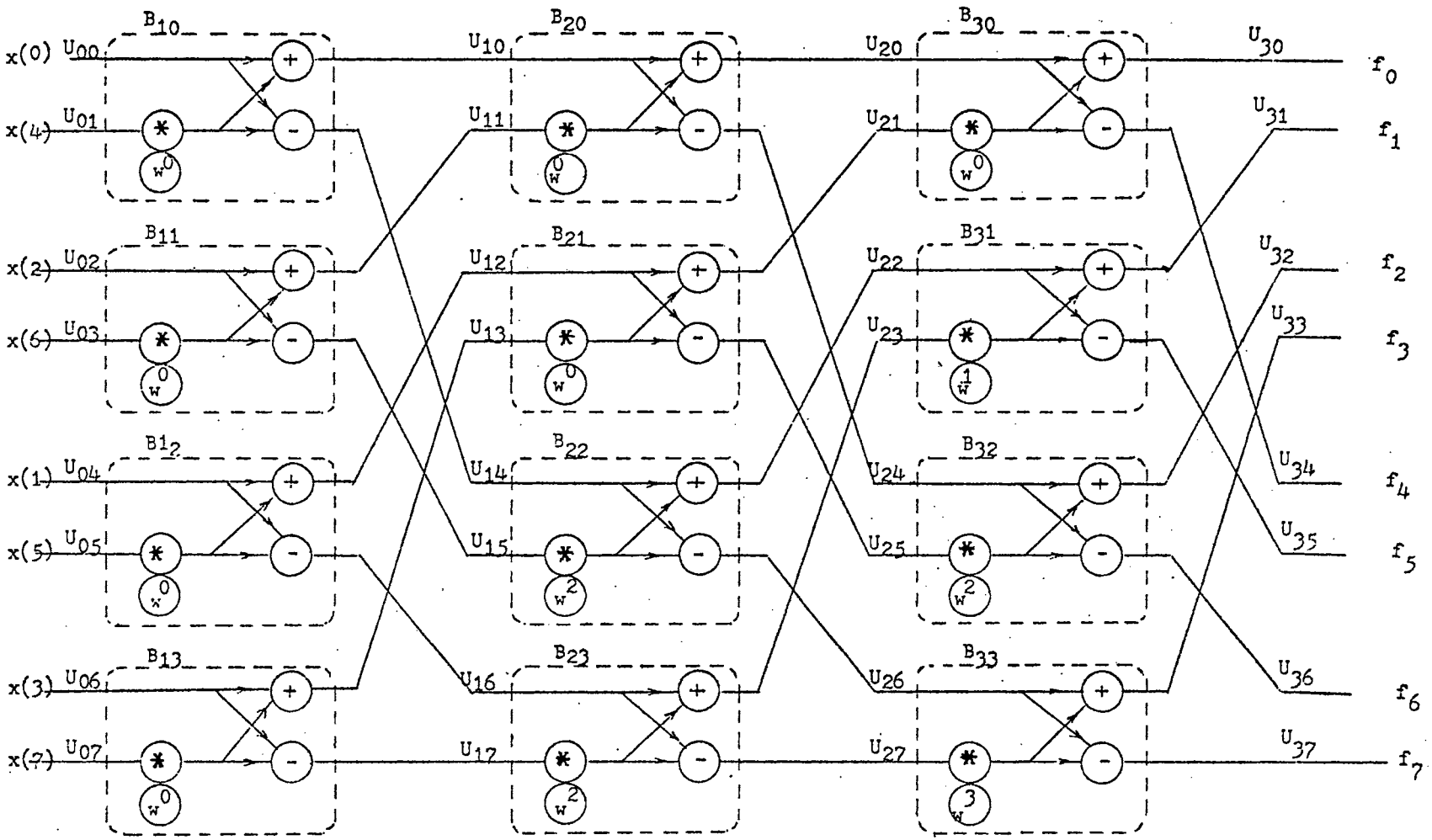


Figure 46. The Eight-point, Constant Geometry, Decimated-In-Time DFT

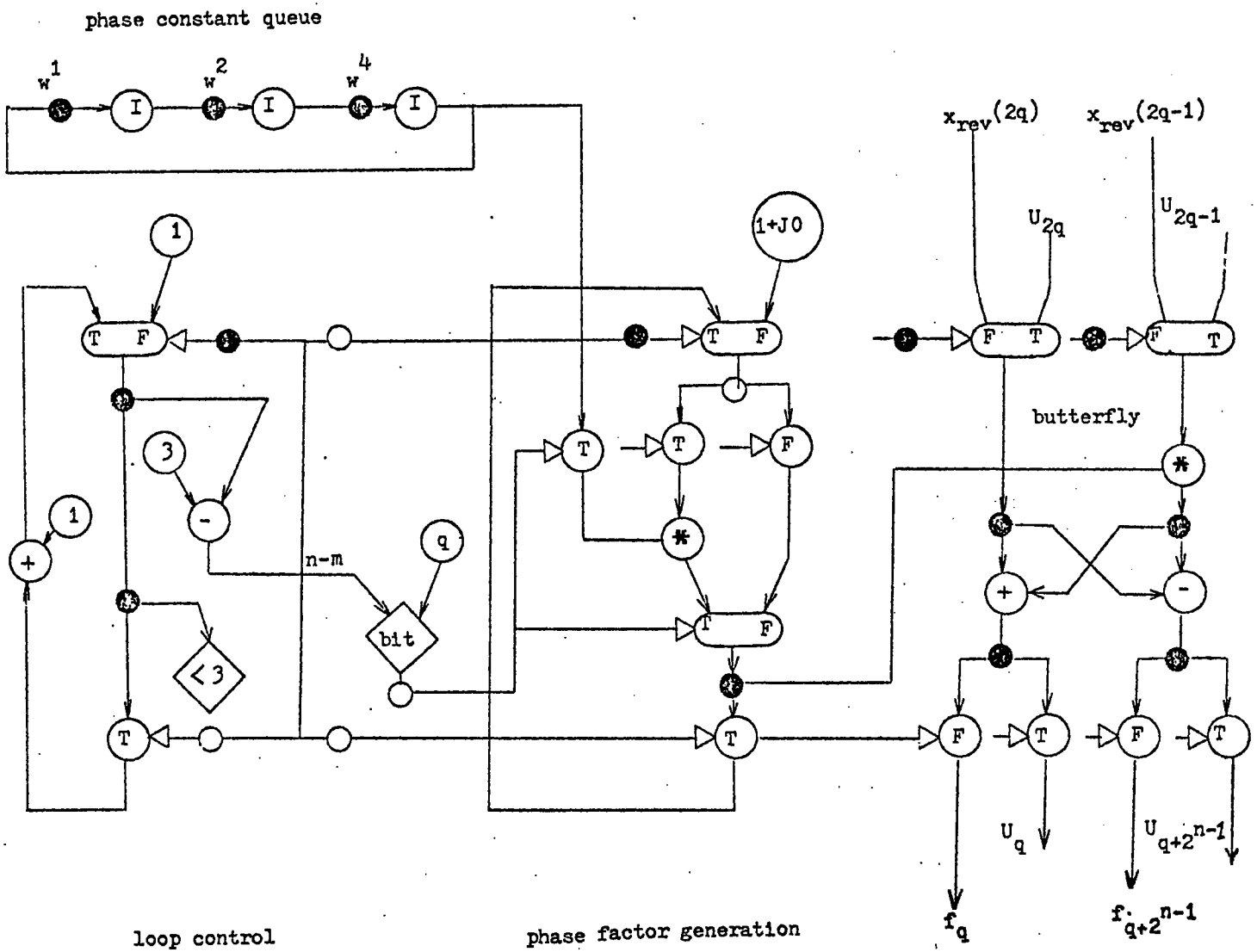


Figure 47. Iterative Data Flow Program for Eight-point DFT

mechanism. Instead, the factor W(m,q) used for butterfly q in stage m may be computed from the factor W(m-1,q) used for the previous stage by a simple rule derived as follows: the exponents of W for W(m,q) and W(m-1,q) are

$$e(m,q) = 2^{n-m} \text{ quo } (q, 2^{n-m})$$

$$e(m-1,q) = 2^{n-m+1} \text{ quo } (q, 2^{n-m+1})$$

then

$$e(m,q) = e(m-1,q) + e(m,q) - e(m-1,q)$$

$$= e(m-1,q) + \underbrace{2^{n-m} (\text{quo } (q,2^{n-m}) - 2 \text{ quo}(q,2^{n-m+1}))}_{T(m,q)}$$

Careful study of the factor T(m,q) reveals that

$$T(m,q) = \begin{cases} 0 & \text{if } \text{rem } (q, 2^{n-m}) \text{ is even} \\ 1 & \text{if } \text{rem } (q, 2^{n-m}) \text{ is odd} \end{cases}$$

Thus T(m,q) is the (n-m)th bit in binary representation of q. Let bit(r,q) be a primitive function that yieldes the rth bit of q. Then we have

$$W(m,q) = W(m-1,q) \times \begin{cases} \text{if bit}(n-m,q) = 1 \\ \text{then } W^{2^{(n-m)}} \quad \text{else } 1 \end{cases}$$

The initial value of the phase factor for the qth butterfly is

$$W(1,q) = W^{e(1,q)} \quad \text{where} \quad e(1,q) = 2^{(n-1)} \text{ quo}(q, 2^{(n-1)})$$

$=(1 + JO)$

The computation of the phase factors W(m,q) is performed by the sections of data flow program labelled "phase factor generation" and "phase constant queue".

A.4. Description of the Program

The data flow program consists of four copies of the code shown previously. Each copy performs one of the butterflies (0,1,2,3) in stage m, and consists of four sections:

A.4.1. Loop Control. This section controls the number of iterations (3 in this case) and computes the (n-m) which wil be used to recognize the (n-m)th bit of q in the computation of W(m,q). Two control values CL1 and CL2 will be produced and distributed in this section:

$$CL1 = \begin{cases} \text{True} & \text{if } m<3 \quad \text{(more iterations)} \\ \text{False} & \text{if } m>3 \quad \text{(no more iterations)} \end{cases}$$

$$CL2 = \begin{cases} \text{True} & \text{if ((n-m)th bit of q) = 1} \\ \text{False} & \text{if ((n-m)th bit of q) = 0} \end{cases}$$

A.4.2. Butterfly. This section computes U(q) and U(q+2^{n-1}) (f(q) and f(q+2^{n-1}) at the end of program) using U(2q) and U(2q+1) (x rev(2q) and x rev(2q+1) initially) and W(m,q) produced by phase factor generation section according

to the following equations:

$$U(m,q) = U(m-1,q) + U(m-1,2q+1)\ W^{e(m,q)}$$

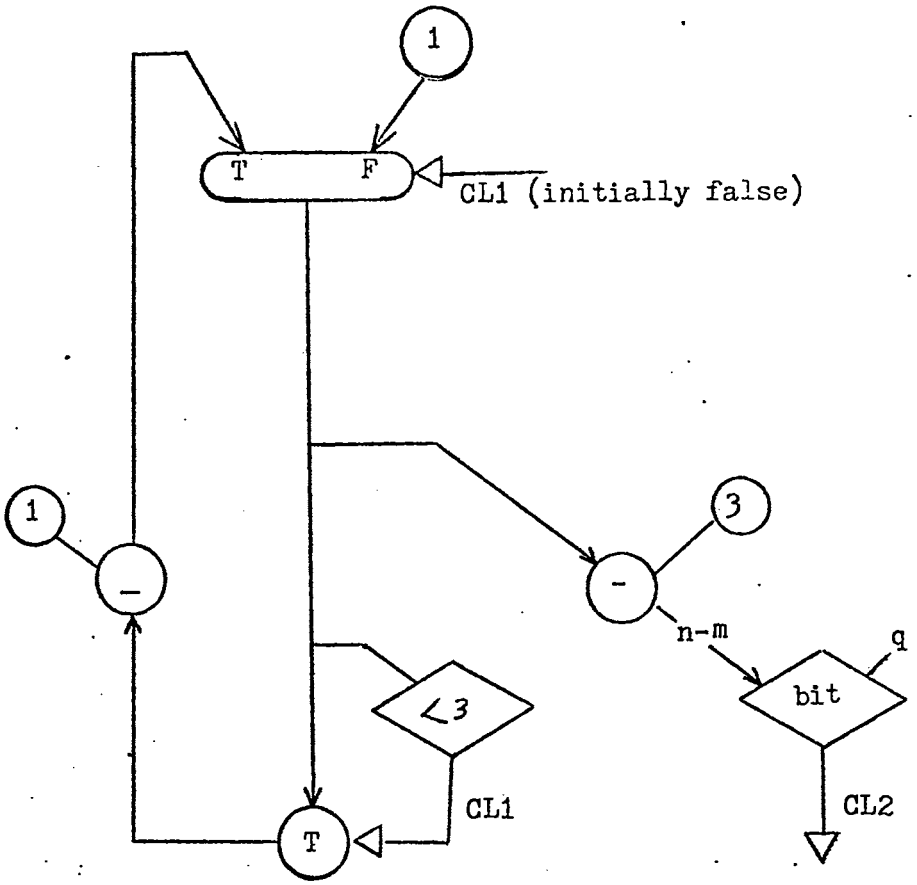
$$U(m,q+2^{n-1}) = U(m-1,2q) - U(m-1,2q+1)\ W^{e(m,q)}$$

A.4.3. Phase Constant Queue. This section operates a queue like structure. The phase constant queue consists of three distribution cells which are linked to simulate a circular queue. The front node of structure always contains W**(2**(n-m)) which will be used in the computation of W(m,q) in phase factor generation section.

A.4.4. Phase Factor Generation. Phase factor W(m,q) will be computed in this section usin the following equations:

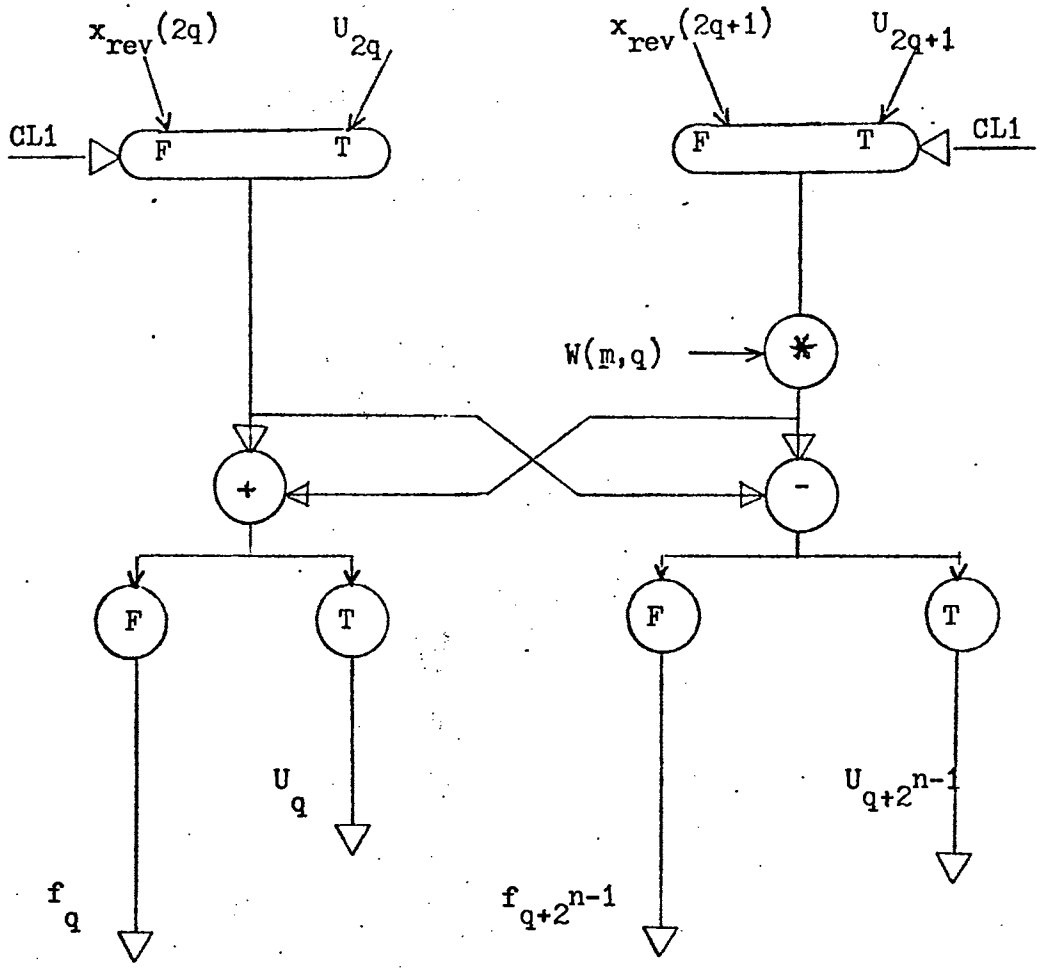
$$W(m,q) = W(m-1,q)\ x \begin{cases} \text{if bit}(n-m,q)=1 \text{ then } W^{2^{n-m}} \\ \text{else } 1 \end{cases}$$

Loop Control Section.



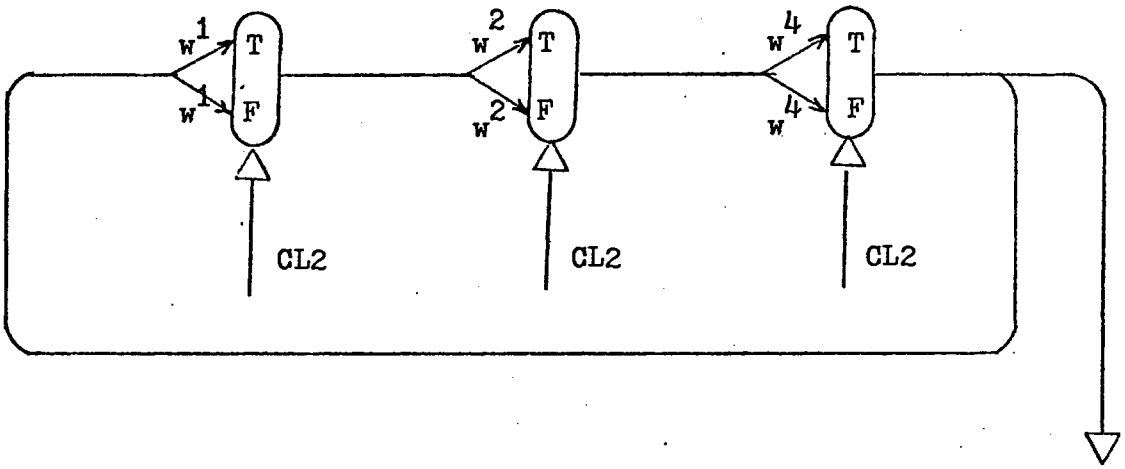

```
        M=1
LOOP:   N-M=3-M
        IF M>3 THEN GO TO OUT
             IF bit 'N-M,Q) = 1 THEN "CL2 = 'True'"
                                ELSE "CL2 = 'False'"
              ELSE "CL1 = 'True'"
        M=M+1
        GO TO LOOP
OUT:    "CL1 = 'False'"
```

Butterfly Section.



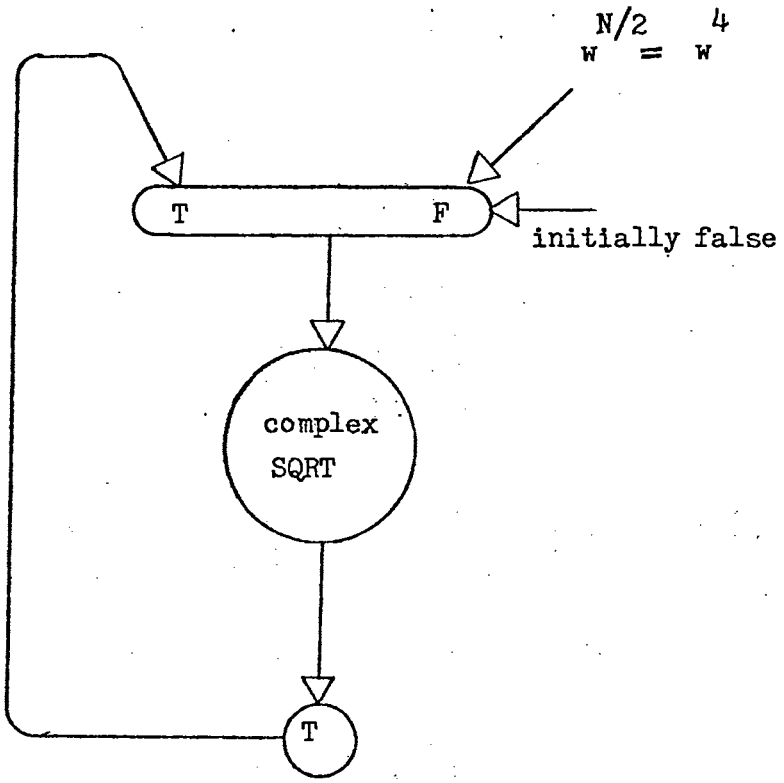

```
c1= "false", a= xrev (2q), b= xrev (2q+1)
DO WHILE (c1 = "true)
     U(q) =          a + b*W(m,q)
     U(q+2**(n-1) = a - b*W(m,q)
     a= U(2q)
     b= U(2q+1)
end
```

Phase Constant Queue




```
CL2 = 'false'
  a = W**1
  b = W**2
  c = W**4
  IF (CL2 is activated) THEN
                              temp = a
                                 a = c
                                 c = b
                                 b = temp
```

Using queue structure to produce W (phase constant) is the best approach for small values of N, but when N is large, which normally is very large in FFT problems, it tends to be very space-consuming and uneconomical. The alternative approach takes advantage of the fact that phase constant of stage m is the square root of the phase constant in stage m-1. This approach which spends more execution time but much less space is shown in Figure 48.



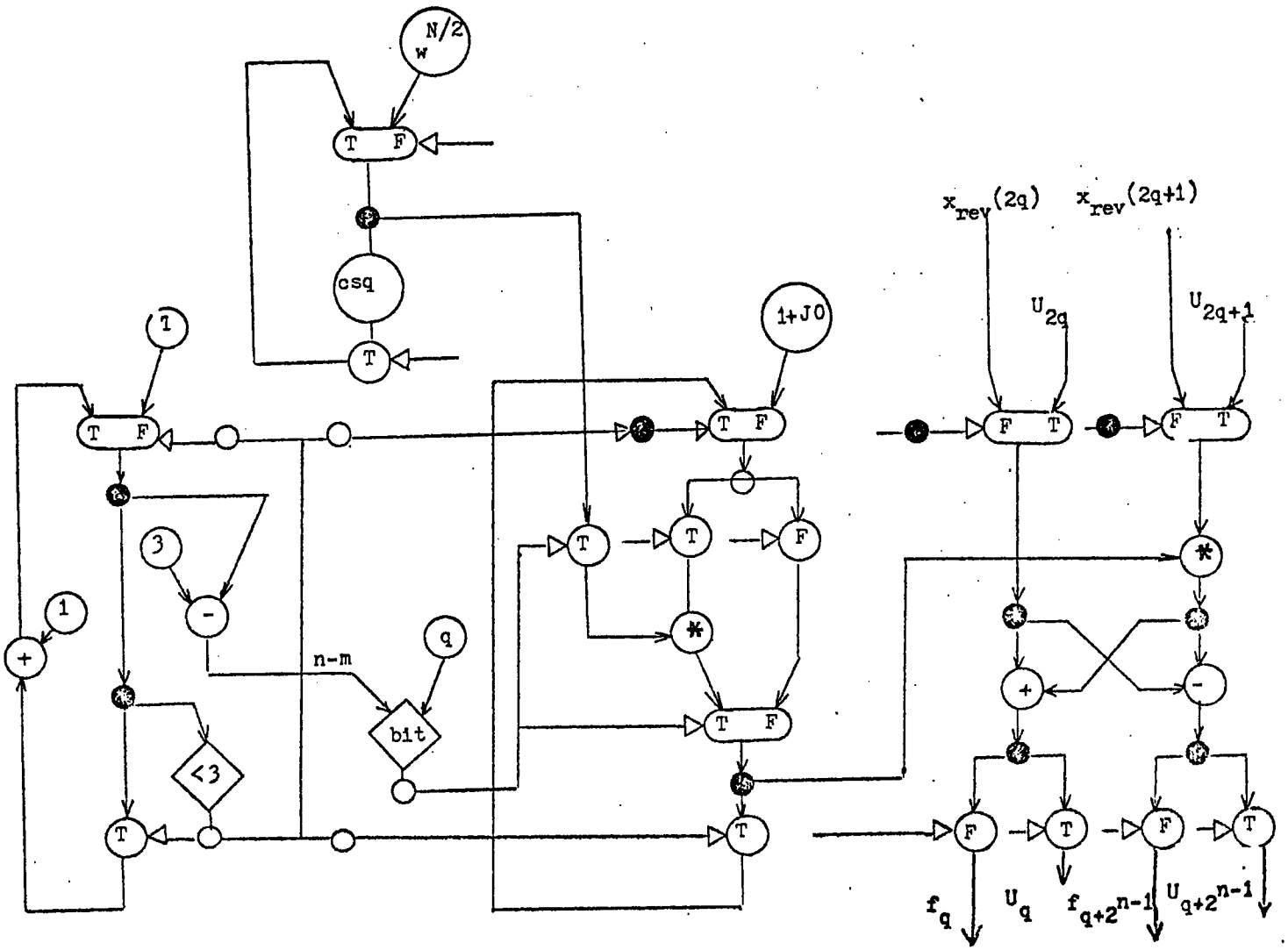
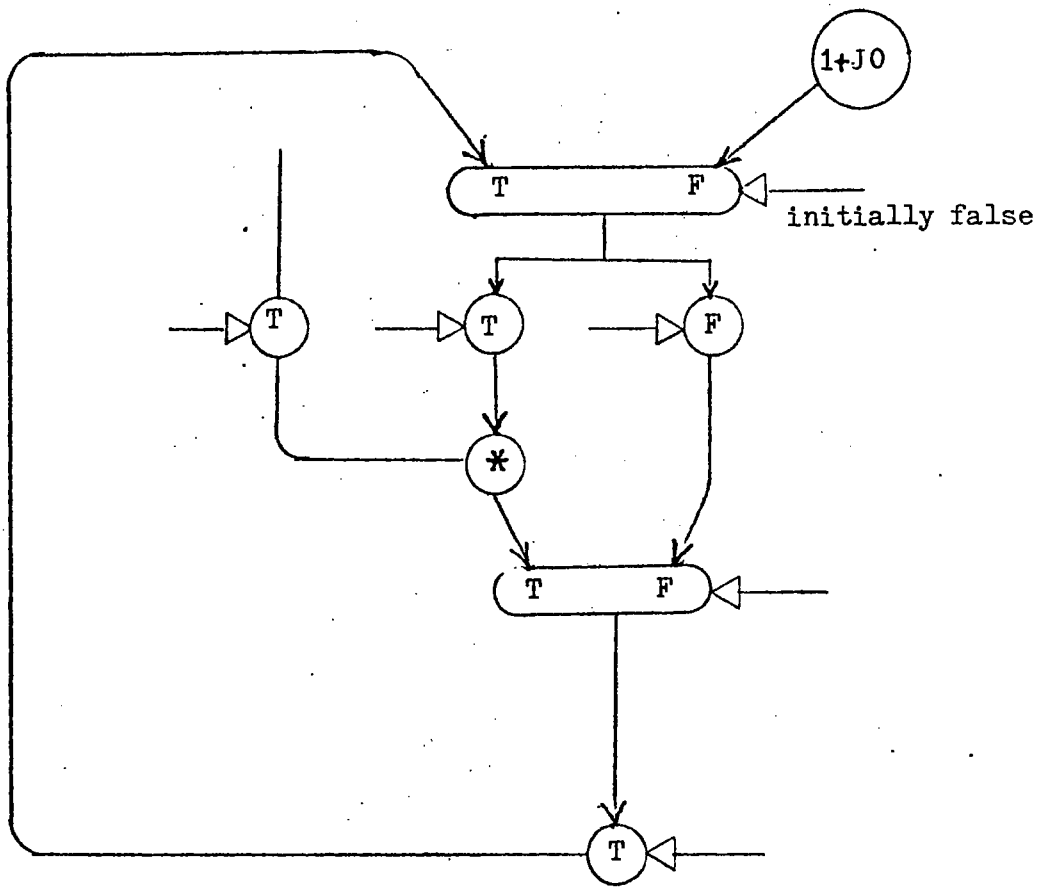


Figure 48. Alternative Data Flow Program for the 8-point DFT

Phase Factor Generation Section




```
      W(m-1,q) = W**0 = 1+J0
LOOP: IF (CL2 = 'false') THEN W(m,q) = W(m-1,q)
                         ELSE W(m,q) = W(m-1,q) *
                                       W**(2**(n-m))
      GO TO LOOP
```

A.5. Program Performance Analysis

Direct computation of the Discrete ast Fourier Transform on a sequential computer may be performed using the following program:

time spend			
			K=0
N(6N+4)		1	DO WHILE (K < N)
	6N+4	3	N=0, F=(2* /N)*K
		1	DO WHILE (n < N)
		4	X(k)=X(k)+x(n)*(e**(0,-F*n))
		1	n=n+1
			END
		1	k=k+1
			END

total time spend = N*(6N+4) = 6N**2 + 4*n

for N=8, total time spend = 416

Note : assignment and initialization statements are considered no time statements

Computation of the Fast Fourier Transform using Decimation-in-Time algorithm consists of two segments, first is a segment to rearrange the input array, second is the segment to compute the values of X.

time spend

			i=0
N+N(10logN+1)		1	DO WHILE (i<N)
			k=1, j=0, l=i
	10logN	1	DO WHILE (k<p+1)
		7	l=k/2**(k-1)-2*(i/2**k)
		1	j=j+1
		1	k=k+1
			END
			X(j)=x(i)
		1	i=i+1
			END

The second segment of program is as follows:

```
     time spend
                      (         2  pi=2*3.14/N, i=1
                      |         1  DO WHILE (i<p+1)
                      |         1   k=0, mp=p-i
                      |      (  1   DO WHILE (k<(N/2+1)
                      |      |  4     t=2**mp*(k/2**mp)
logN+logN(5N+2)+2 <   5N  <  3     wpq=(cos(pi*t),
                      |      |              -sin(pi*t)*X(2*k+1))
                      |      |  1     X(k)=X(2*k)+wpq
                      |      (  1     X(k+2**(N-1))=X(2*k)-wpq
                      |              END
                      |         1   i=i+1
                      (            END
```

total time spend = 15NlogN + 2N + 3logN + 2

for N=8, total time spend = 387

Using data flow program represented, the flow of information may be shown by the following table:

step#	parallel processes
1	p0,c0,w0,u0,u1
2	c1,c2,p1
3	c3,c5,p2
4	w1,w2,w3,c4,p0
5	w4,c0,p1
6	w5,c1
7	u2,c5
8	u3,u4
9	u5,u6,u7,u8

After 9 cycles the first set of results is ready then:

time spend for computations in one stage = 9

total time spend =3x9=27

B. SIN Function

Trigonometric functions are the most widely used arithmetic functions. Some numerical methods to compute these functions are inherently parallel and may be easily converted into parallel procedures. In this section Taylor series representation of SIN function is studied and

programmed in data flow base language.

Taylor series for SIN function is as

$$SIN(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots\ldots \quad (14)$$

Each term in the seri is independent from others and may be computed separately, but independent computation of each term turns to be very inefficient.

A careful study of the terms of the series reveals a special relationship between the two consecutive terms. If nth term is represented by T(n), then

$$T(n+1) = - T(n) * ((n+1)*(n+2)) \quad (15)$$

A data flow code segment using this property is shown in Figure 27. Using this direct approach, computation of each term requires 6 operations; then for N terms the number of arithmetic operations is 6N.

In a multiprocessor environment, more than one term can be computed at the same time. The new approach that is presented in this section involves the computation of 4 terms simultaneously, using the relationship between the terms of the Taylor series. First, divide the terms of the series into the groups of 4 terms, then each term may be represented by T(n,m), where n=0,1,...,N/4, and m=0,1,2,3. If the denominator of each term is represented by D(n,m), then the relationship between the first denominator of a group and the last denominator of the preceeding group is as follows:

$$D(n+1,0) = D(n,3) * (8(n+1)) * (8(n+1)+1) \quad (16)$$

and the relationship between the denominator of the first term of a group and the others is as:

$$D(n,m) = D(n,0) * (8(n+1)+2m)*(8(n+1)+2m+1) \qquad m=1,2,3 \quad (17)$$

If denominators of the terms of a group is stored in an array, F say, then F should initially contain (1!, 3!, 5!, 7!). The values of the denominators of each group may be computed using Eq.s (16) and (17). The numerator of each term in nth group is the product of the numerator of the corresponding term in (n-1)st group and x . Using the facts represented above, a data flow code segment is written which is illustrated in Figures 49, 50, 51.

The first step in computation of the SIN function is the generation of the first 4 powers of x and x^8 . This segment is performed only once at the beginning of the process and takes 3 cycles, is represented in Figure 49.

Generating the denominators of each term using the last denominator of the previous group is done by a code segment represented in Figure 50. The process of generating and adding the terms of a group and making the decision whether to continue or terminate the process is represented in Figure 51. Each section of the code segment is labelled by a letter and each step is labelled by a number to clarify the analysis of the process, for example, in program analysis tables, instructions are specified by code segment label at the top of the table and associated step under that column.

Two different methods to process this code segment are analyzed and results are shown in Figures 52 and 53. In the first method the process is not controlled and the values of the powers of x are transmitted to the next section as soon as they are generated. Figure 52 shows that using this method, 4 consecutive terms of the Taylor series are calculated in 10 cycles. In the second method the powers of x are not transmitted to the destinations before all powers are calculated. In this controlled method, for the first 3 cycles processor utilization is not efficient, but the execution of 4 term groups takes only 7 cycles. Using this method 7N/4 cycles are required to compute N terms, which is obviously less than 6N cycles in the direct approach solution. The maximum number of parallel processes in one cycle is 12, which determines the minimum number of processors to achieve the 7N/4 execution time.

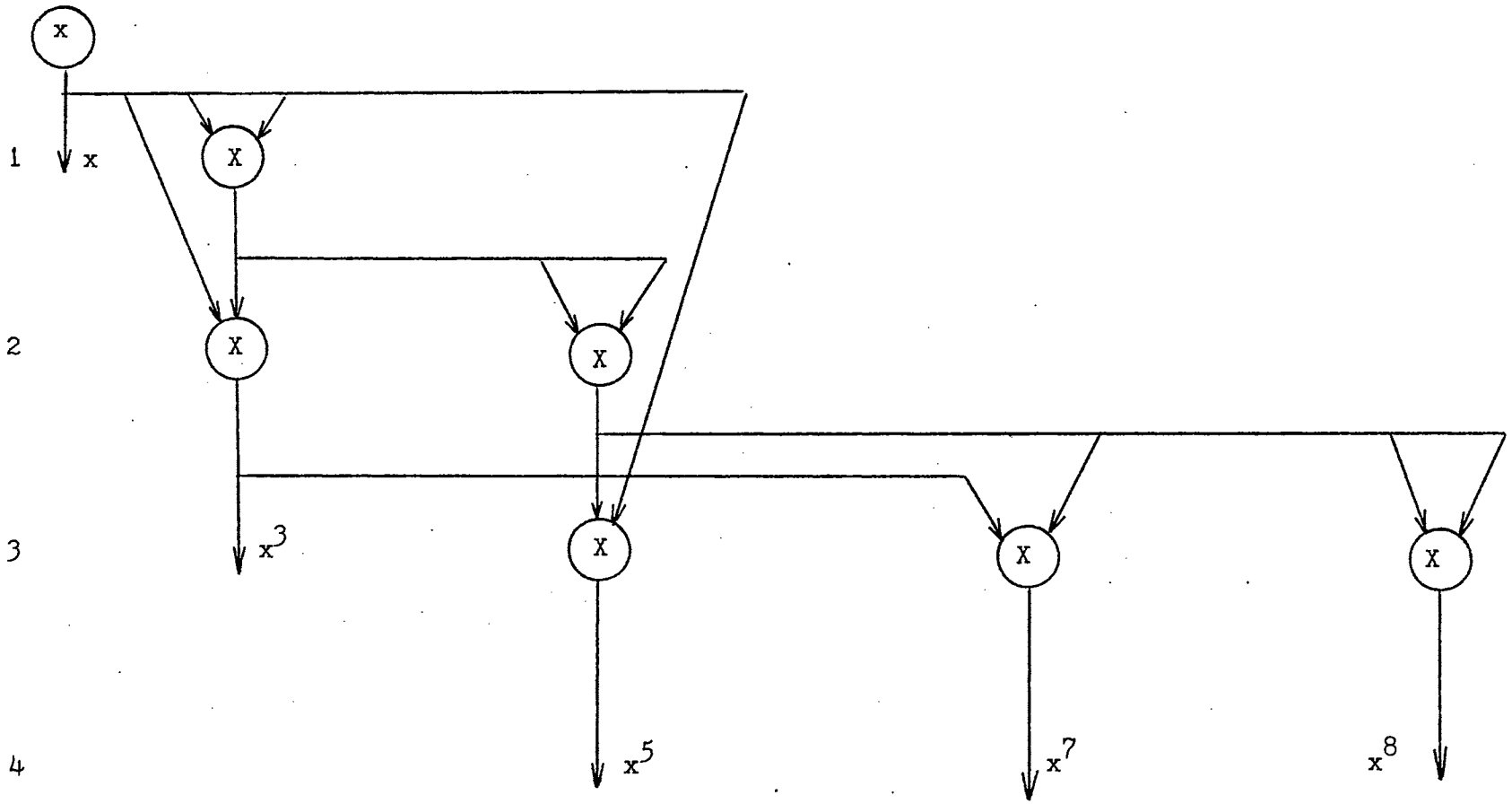


Figure 49. Generation of Powers of x

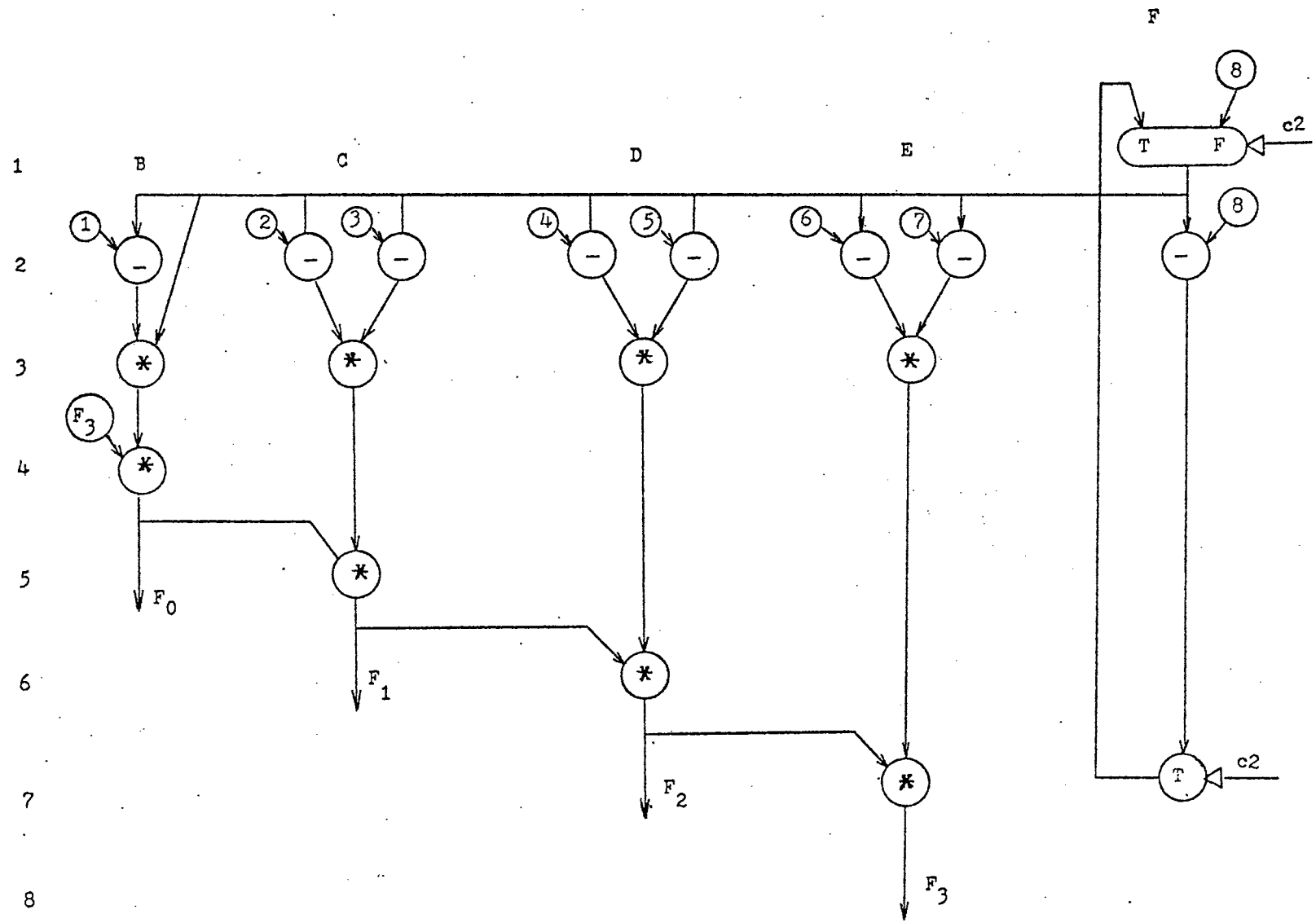


Figure 50. Coefficient Generation

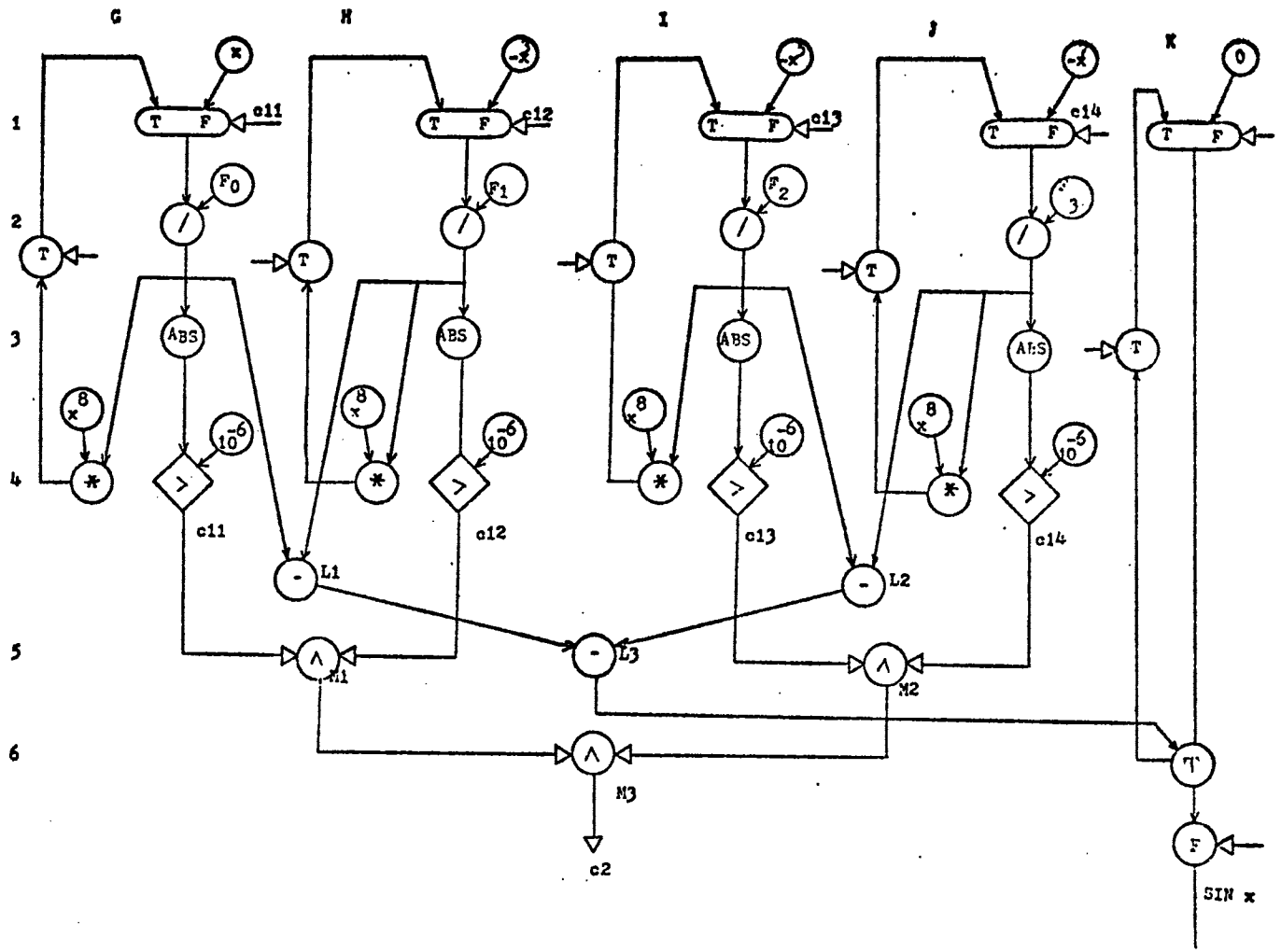


Figure 51. Computation of Four Consecutive Terms of Taylor Series

step #	f	A	B	C	D	E	F	G	H	I	J	K	L	M
1		1					1	1				1		
2		2	2	2	2	2	2	2						
3		3	3	3	3	3		3	1					
4			4					4,5	2	1	1			
5	f0			5				6	3,4	2	2		1	
6	f1				6			1	5	3,4	3,4		2	
7	f2					7		2	6	5	5		3	1
8	f3							3,4	1	6	6	6		2
9								5	2	1	1			3
10							7	6	3,4	2	2	7,8	1	
11							1	1	5	3,4	3,4	1	2	1
12			2	2	2	2	2		6	5	5		3	
13			3	3	3	3			1	6	6	6		2
14			4							1	1			3
15	f0			5			7	2				7,8		
16	f1				6		1	3,4	2			1		
17	f2					7	2	5	3,4	2			1	
18	f3							6	5	3,4	2			
19								1	6	5	3,4		2	1
20									1	6	5		3	
21										1	6	6		2
22											1			3
23							7					7,8		
24							1					1		
25			2	2	2	2	2							
26			3	3	3	3								
27			4											
28	f0			5				2						
29	f1				6			3,4	2				1	
30	f2					7		5	3,4	2				
31	f3							6	5	3,4	2			1

Figure 52. Computation Analysis of SIN Program

step #	f	A	B	C	D	E	F	G	H	I	J	K	L	M
1		1												
2		2												
3		3												
4							1	1	1	1	1	1		
5			2	2	2	2	2	2	2	2	2			
6			3	3	3	3		3,4	3,4	3,4	3,4		1,2	
7			4					5	5	5	5		3	
8	f0			5				6	6	6	6			1,2
9	f1				6			1	1	1	1			3
10	f2					7	7	2	2	2		7,8		
11	f3						1	3,4	3,4	3,4	2	1	1	
12			2	2	2	2	2	5	5	5	3,4		2	
13			3	3	3	3		6	6	6	5		3	1
14			4					1	1	1	6	6		2
15	f0			5				2			1			3
16	f1				6		7	3,4	2			7,8		
17	f2					7	1	5	3,4	2		1	1	
18	f3		2	2	2	2	2	6	5	3,4	2			
19			3	3	3	3		1	6	5	3,4		2	1
20			4						1	6	5		3	
21	f0			5				2		1	6	6		2
22	f1				6			3,4	2		1			3
23	f2					7	7	5	3,4	2		7,8	1	
24	f3						1	6	5	3,4	2	1		
25			2	2	2	2	2	1	6	5	3,4		2	1
26			3	3	3	3			1	6	5		3	
27			4							1	6	6		2
28	f0			5				2			1			3
29	f1				6		7	3,4	2			7,8		
30	f2					7	1	5	3,4	2			1	
31	f3		2	2	2	2	2	6	5	3,4	2			
32			3	3	3	3		1	6	5	3,4		2	1
33.			4						1	6	5		3	2

Figure 53. Computation Analysis of Controlled SIN Program

CHAPTER VII

SUMMARY, CONCLUSIONS AND FUTURE WORK

A survey of a data flow architecture was presented as a solution to many of the problems of highly parallel computer systems. The use of interconnection networks between sections of the processor provides an attractive approach to the communication of information between units. Due to the radical nature of architecture, many questions range from ones about the use of certain methodsof representation or design choices to deep semantic issues.

A survey of a phenomenon known as the semantic gap was presented. The effect of the semantic gap on system performance was discussed. The semantic gap which represents the gap between the concepts presented in the architecture and high-level languages concepts, contributes to perfcrmance problems in conventional computers.

Methods to represent high-level language concepts in data flow base language was presented. The data flow base language, while appearing to be a semantically elegant method of expressing parallelism, is not yet an appropriate one to represent high-level language concepts, and is open to further study and extensions. The language needs to be expanded by the addition of such actors as "forall"

construct to enable it to better express concurrent processing of the elements of a structure. Also, the language does not currently contain the capability to express nondeterminate computations.

Further investigation of the use of the data flow language is necessary. The representation of algorithms such as Fast Fourier Transform and SIN function in data flow appears very attractive (Chapters V and VI). However, the data flow representation for other computations need to be developed and examined.

The data flow language is designed to serve as the base language of the data flow processor. The development of a user language which can be readily translated into a data flow representation is necessary. Much more work needs to be done to identify concurrency in problems and to take advantage of that through use of the data flow representation. New actors and features must be added to the architecture to cope with high-level languages and reduce the semantic gap.

BIBLIOGRAPHY

1. Ackerman, William B. "A structure processing facility for data flow computers." Laboratory For Computer Science, M.I.T, Proceedings of the 1978 international conference on parallel processing, Aug. 1978, pp. 166-172.

2. Ackerman, William B., and Jack B. Dennis. "VAL-- A value-oriented algorithmic language: preliminary reference manual." Laboratory For Computer Science, M.I.T, Technical Report #218, Jan. 1979.

3. Aoki, Donald J. "A machine language instruction set for a data flow processor." Laboratory For Computer Science, M.I.T, Technical Memo #146, Dec. 1979.

4. Arvind, Vinod Kathail, Keshav Pingali. "A data flow architecture with tagged tokens." Laboratory For Computer Science, M.I.T, Technical Memo #174, Sept. 1980.

5. Arvind, Kim P. Gostelow, and Will Plouffe. "An asynchronous programming language and computing machine." University of California, Irvine, Technical Report #114A, Dec. 1978.

6. Arvind and Kim P. Gostelow. "Data flow computer architecture: research and goals." University of California, Irvine, Technical Report #113, Feb. 1978.

7. Arvind and Kim P. Gostelow. "Semantics of loop expressions in ID." UCI data-flow project, data-flow note #11, March 1977.

8. Cornish, Merrill."The TI data-flow architecture: the power of concurrency for avionics." Texas Instruments Corp., Texas, Austin, 1979.

9. Cote, William F. and Richard F. Ricceli. "The design of a data-driven processing element." Wayne State University, Michigan.

10. Davis A. L. "The architecture of DDM1: A recursively structured data-driven machine." University of Utah, UUCS-77-113, April 1978.

11. Dennis, J. B., David Misunas, and Clement K. Leung. "A highly parallel processor using a data-flow machine language." Laboratory For Computer Science, M.I.T, Technical Memo #134, Jan. 1977.

12. Dennis, J. B., and David P. Misunas. "A preliminary architecture for a basic data-flow processor." Project MAC, M.I.T, the second annual symposium on computer architecture, Jan. 1975, pp. 126-132.

13. Dennis, J. B. "First version of a data-flow procedure language." MIT/LCS/TM-61, May 1975.

14. Dennis, J. B. "Packet communication architecture." Project MAC, M.I.T, Technical Memo #130, Aug. 1975.

15. Dennis, J. B. "Programming generality, parallelism and computer architecture." M.I.T, Information Processing 68, pp. 484-492.

16. Farrel, Edward P., Noordin Ghani, and Philip C. Treleaven. "A concurrent computer architecture and a ring based implementation." University of Newcastle, Upon Tyne, England 1979.

17. Feridum, Arif Metin. "Design of an on-line byte-level pipelined arithmetic processor." Laboratory For Computer Science, M.I.T, Technical Memo #162, July 1978.

18. Hopkins, Richard, Paul W. Rautenbach, and Philip C. Treleaven. "A computer supporting data-flow, control-flow and updateable memory." The University of Newcastle, Upon Tyne, England, Technical Report #144, 1979.

19. Jenson, John C. "Basic program representation in the Texas Instrument data-flow test bed compilor." Texas Instruments Corp., Austin, Texas, Jan., 1980.

20. Johnson, Douglas. "Automatic partitioning of programs in multi-processor systems." Texas Instruments Corp., Austin, Texas, 1979.

21. Keller, Robert M., Gary Lindstrom, and Suhas Patil. "An architecture for a loosely-coupled parallel processor." University of Utah, UUCS-78-105, Oct. 1978.

22. Leth, James Williams. "An intermediate form for data-flow programs." Laboratory For Computer Science,

M.I.T, Technical Memo #143, Nov.1979.

23. Miranker,Glen Seth. "Implementation of procedures on a class of data-flow processes." Laboratory For Computer Science, M.I.T, proceedings of 1977 international conference on parallel processing, Aug. 1977, pp. 77-86.

24. Misunas, David P. "A computer architecture for data-flow computation." Laboratory For Computer Science, M.I.T, Technical Memo #100, March 1978.

25. Misunas, David P. "Performance analysis of data-flow processor." Laboratory For Computer Science, M.I.T, proceedings of the 1976 international conference on parallel processing, Aug. 1976, pp. 100-105.

26. Misunas, David P. "Structure processing in a data-flow computer." Project MAC, M.I.T, Technical Memo #129, Aug. 1975.

27. Montz, Lynn Barbara. "Safety and optimization transformations of data-flow programs." Laboratory For Computer Science, M.I.T, Technical Report #240, Jan. 1980.

28. Myres, Glenford J. Advances in computer architecture. 1st Ed. New York: IBM systems research institute, Wiley-interscience publications, 1978.

29. Oppenheim, Alan V. and Roland W. Schafer. Digital signal processing. 1st Ed. New Jersey: Prentice-Hall Inc., 1975.

30. Rumbaugh, James. "A data flow multi-processor." IEEE transactions on computers, Vol. C-26, No.2, Feb. 1977, pp. 138-146.

31. Ruth, Gregory R. "Data driven loops." Laboratory For Computer Science, M.I.T, Technical Report #244, Aug. 1980.

32. Saubar, William. "A data-flow architecture implementation." Texas Instruments Corp., Austin, Texas, 1980.

33. Thomas, Robert Eugene. "An activity assignment scheme for a data-flow computer." UCI data-flow project, Note #29, April 1978.

34. Thomas, Robert Eugene. "Performance analysis of two classes of data-flow computing systems." UCI, Technical Report #120, May 1980.

35. Treleaven, P. C., DAvid R. Brownbridge, and Richard P. Hopkins. "Data-driven and demand driven computer architecture." The University of Newcastle, Upon Tyne, England, 1978.

36. Treleaven, P. C. "Principle components of a data-flow computer." The University of Newcastle, Upon Tyne, England, 1978.

37. Weng, Kung-song. "An abstract implementation for a generalized data-flow language." Laboratory For Computer Science, M.I.T, Technical Report #228, May 1979.

APPENDIX

Actor
: Data flow operator

Arbitration network
: Receives operation packets from the instruction cells, present in the memory, and sends them to the appropriate operation unit.

Cache
: Cache is a scratch pad random access memory usually semiconductor type, holds the information that are most often required by the processor. Other information about the program is kept in a slower memory. The information is passed to the cache, based on certain policies whenever it is required.

Concurrent
: The occurence of two or more events within the same time period, i.e., two computers or programs simultanously.

Control network
: A network which handles control packets. The network consists of arbitration and distribution units.

Data driven
: The class of data flow in which the instructions are executed when all the operands required by the instruction are ready.

Data flow structure
: Structured data residing on conventional memory.

Data packets
: Instruction cells containing data values are known as data packets.

Decision unit
: It is a hardware unit which performs boolean operations and gives the result in the form of control packets.

Distribution network
: Receives results from the operation unit in the form of data packets and places them in the instruction cells, present in the memory.

Fired
: When tokens are present at the input arcs of a data flow graph, the node is enabled and the operands are removed from the input arcs. i.e., the operands are fired.

Instruction cell
: The memory is organized into instruction cells. Each instruction cell consists of three or more registers to hold the data and operator.

Instruction packet
: A packet containing a data flow instruction is called an instruction packet.

Link
: The program in elementary data flow language is a directed graph in which the nodes are operators. The nodes are interconnected by means of links.

Locality
: Working set or the working area in the memory, i.e.,physical locality or program locality.

LSI
: Abbreviation for Large-Scale Integeration. High-density integrated circuits for complex logic functions.

Operation packet
: Operation packet is one of the types of instruction packet that is handled by the operation unit.

Operator
: Operators are the data flow instructions.

Packet
: The information, may be either data or operator, sent from one unit to another unit in data flow machine.

Selector
: Used in the representation of data flow structures- an integer or a string. The structure node is represented as <selector : value>.

Side-effect
: Effect of an instruction on data elements which is to be used by other instructions.

VITA

Taraneh Baradaran-Seyed

Candidate for the Degree of

Master of Science

Thesis: HIGH-LEVEL LANGUAGE CONCEPTS IN DATA FLOW ARCHITECTURE

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Tehran, Iran, October 20, 1952.

Education: Graduated from Shirin High School, Tehran, Iran, in June, 1970; received Bachelor of Science degree in Mathematics and Computing Science from Aryamehr University of Technology in 1976; completed requirements for the Master of Science degree at Oklahoma State University in December, 1981.

Professional Experience: Instructor, ISIRAN Institute, 1976-79; graduate teaching assistant, Oklahoma State University, 1979-81.