5-12-1986

DIVA, a data flow language

Edith Lawson

DIVA, A DATA FLOW LANGUAGE

by
Edith A. Lawson

Submitted to the Faculty of the School of
Computer Science and Technology in partial fulfillment
of the requirements for the degree of Masters of Science
in Computer Science

Rochester Institute of Technology
1986

Rochester Institute of Technology
School of Computer Science and Technology

DIVA, A DATA FLOW LANGUAGE

by
Edith A. Lawson

Submitted to the Faculty of the School of
Computer Science and Technology in partial fulfillment
of the requirements for the degree of Masters of Science
in Computer Science

Approved by:

Lawrence A. Coon 5/12/86
Dr. Lawrence Coon, Chairman

John L. Ellis 5/14/86
Dr. John L. Ellis

Warren R. Carithers 5/14/86
Warren R. Carithers

Title of Thesis: DIVA, A DATA FLOW LANGUAGE

I Edith A. Lawson hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 5-23-86

ABSTRACT

The underlying principles of concurrency and data flow are summarized along with a survey of the current data flow languages.

A high level data flow language, DIVA, is developed that provides the basic data types and language constructs of traditional languages as well as some unique features of data flow.

The organization and data structures of the compiler and assembler are also discussed.

OUTLINE

LIST OF FIGURES

1.0 INTRODUCTION

1.1 BACKGROUND

The development of VLSI technology has increased computing power and reduced the size of present day computers, but there has not been a corresponding advancement in speed that is needed for many computer applications. The failure of modern VLSI technology to produce this breakthrough can be traced to the basic underlying "Von Neumann" concept of computer organization. To fully utilize VLSI technology a different underlying computer architecture concept is necessary.

The need for large, high-speed computations has prompted researchers to investigate alternative architectures and programming languages. Data flow has attracted a great deal of attention in various parts of the world as a viable alternative. It is a means of meeting the needs for faster computation speeds by extending the concept of concurrency on a massive scale. [McGraw '82]

While a number of languages have tried to take advantage of concurrency by including some mechanism for simultaneous processing, it is not natural to their basic design. The languages have, therefore, become large and complicated while providing only a pseudo form of concurrency. [Backus '77] The bottleneck of the Von Neumann organization prevents traditional languages from capitalizing on the parallelism that exists in many algorithms. This bottleneck is the connecting

tube between the CPU and memory that can transmit only a single word or address. [Backus '78]

In addition to the "bottleneck", two major obstacles in the Von Neumann model are the concepts of:

1. sequential control--represented by the program counter
2. memory cells

Sequential control with its program counter does not allow concurrency. In order to overcome sequential execution in traditional languages, the programmer must explicitly indicate the concurrency using a special mechanism such as monitors or semaphores. Assignment and referencing of variables in traditional languages is accomplished by access to and operations on memory cells. These accesses and operations must be communicated through the "Von Neumann bottleneck," and the programmer must keep track of what value is being assigned and where it is stored. The memory cell concept also burdens the programmer with the responsibility of coordinating the access of shared variables by asynchronous processes. [Arvind, Gostelow, Plouffe '78]

Traditional languages evolved to this state because they had to fit themselves to an existing computer architecture. This was the result of a method that designed the hardware first, without regard for the programmer's point of view, and fit the language to it later.

In developing data-flow architectures, a different approach was used. The language concepts are constructed before attempting to construct the hardware. [Gostelow & Thomas '79] A specific base language is developed, which serves as the

target representation for higher level languages, and then the hardware is designed to interpret that base language.

In data flow there is no global updatable memory and no program counter; these principles are discarded in favor of a model based on:

1. Asynchronous processing except where synchronization is explicitly stated--instead of a program counter to determine the order of execution, instructions execute as soon as all their operands (input data) are available. A natural consequence of this is that the order of statements is irrelevant.

2. Single-assignment--assignment statements exist only as a labeling device to identify values which correspond to data paths in the program graph. In this sense an assignment statement can be thought of as a definition. [Ackerman '79] An identifier name can be assigned to only once within its scope.

3. Functionality--all operators and procedures are functional, thus producing freedom from side effects. There are no global or "common" variables--a procedure may not even modify its own arguments. All procedure arguments are call-by-value, and any output produced by the procedure is returned as a value token or tokens if there is more than one value returned.

Single assignment and functionality ensure that the data dependencies are the same as the sequencing constraints, thereby promoting concurrent computation. [Ackerman '79]

Concurrency in data flow is on a more basic and all-encompassing basis than the concurrency achieved in multiprocessing by traditional languages. Programs written using data flow allow concurrency to be detected at the function level or instruction level, because the notion of a variable being equal to a storage location is not present and the functions are data driven. [Srini '84]

All programs in a data-flow computer are directed data-flow graphs. The nodes on the graph represent operations to

be performed, and the arcs (or links) between the nodes represent the pathways of values between operations. When all input tokens arrive at a node, it is activated. The sequence in which nodes fire is based soley on the data dependencies. The lack of an arc between two nodes indicates potential concurrency. [Davis & Keller '82]

New programming languages are needed that identify sources of concurrency in algorithms and map them into data-flow graphs that can then be interpreted by a data-flow computer.

While the underlying semantics are clearly different, high level textual data-flow languages have been developed that share some properties with conventional languages such as: assignment, arithmetic expressions, conditional statements, iterations, recursion, and function declaration. [Treleaven '79]

1.2 SCOPE OF THE PROJECT

It is the purpose of this thesis to investigate the work in the development of textual data-flow languages and to design and implement a high level data-flow language for a simulated data-flow computer. [Torsone '84]

A compiler, written in "C" (using LEX and YACC) will recognize legitimate data-flow programs and generate data-flow program graphs in mnemonic code. An assembler, written in "C", will then generate machine code for the compiler output that can then be directly run on the simulated data-flow computer.

1.3 OVERVIEW OF FOLLOWING CHAPTERS

Chapter 2 surveys several of the current data-flow languages. A description of the organization and data structures of the compiler and assembler are in Chapter 3. In Chapter 4, there is a detailed user's manual for DIVA. And, Chapter 5 summarizes the results of the project with some ideas for improvement and expansion as well as some suggestions for future research.

2.0 SURVEY OF EXISTING DATA FLOW LANGUAGES

2.1 DATA FLOW MODELS

Data-flow languages have the property that they can do function applications, except termination, in any order; because the outcome does not depend on the sequence chosen to perform the function applications. Two types of data-flow models have been developed to support these languages: demand driven and data driven.

The data-driven model represents the "strict function" version of data flow, which requires that all the inputs of a computation must be present before it can be executed. This will result in a cost in processing efficiency when the result desired does not require all the elements of a data structure.

In the demand-driven model ("non-strict function") only those computations necessary to the results are executed. For example, it permits a function to execute some of its code before all the arguments of the function have been evaluated. This requires that the demands be saved and propagated back to get the input values needed. [Pingali and Arvind '85] This approach results in a loss of some of the parallelism possible in data flow. If the producer is only allowed to compute output when the consumer demands it, there is a loss of the pipeline operation of the producer-consumer relationship that could otherwise be exploited.

The simulator which is the target machine for this project's language is based on the data-driven model.

2.2 HIGH LEVEL DATA-FLOW LANGUAGES

The languages of most interest to this project are those that more closely resemble the structure and syntax of conventional languages and fit the data-driven architecture. The goal is to provide a data-flow framework that is easy to adjust to even though the underlying principles and operation are dramatically different. Two of the major data-flow languages, VAL and ID, fall into this category. For these reasons the language (DIVA) developed in this project is based on a combination of the features and rules of these languages. Because of this, they are reviewed in greater detail than the other languages surveyed.

The languages presented in this section are not an exhaustive list of all the existing data-flow languages, but they are representative of the major ones.

2.2.1 VALUE-ORIENTED ALGORITHMIC LANGUAGE (VAL)

VAL was developed at MIT by the Computation Structures Group, with Ackerman and Dennis the principal architects. It is one of the most fully developed data flow languages. The ideas in the language grew out of a gradual self-education about data driven computation beginning around 1967. [Ackerman and Dennis '79]

A program in VAL is a collection of separately translated modules with the following characteristics:

1. Each module contains the definition of one external function and may contain definitions of internal functions.

2. The internal function definitions are nested similar to the PASCAL program format for function definition.

3. There is no recursion or mutual recursion.

4. Internal functions can be called only from the function enclosing it.

5. External functions can be called from all modules of the program except the one defining it.

See Appendix A, page 125, for a more detailed illustration of the VAL scoping for functions.

VAL is an applicative, single-assignment, side-effect free language; consequently, all data types--boolean, integer, real, character, arrays, and records--are treated as mathematical values. Arrays and records may be nested to any depth. There is also a union type available where tags allow you to identify and choose from a specified set of types.

All variables must be declared and strong type checking is done at compile time; there is no mechanism for automatic conversion of types. There are special error values for every data type; however, a discussion of this is outside the scope of this thesis. For more information see [Wetherell '82].

VAL also includes conditional expressions (if-then-else), iteration expressions (for-iter), and a special parallel iteration expression (forall). The "for-iter" expression implements loops that cannot execute in parallel because of data dependencies between iterations, while the "forall" allows the programmer to specify explicitly that the iterative computations are independent and can be executed simultaneously. Adding one to all members of an array would be a

good application for the "forall" expression. With the exception of this expression, all concurrency and synchronization are implicit thus leaving it to the compiler to decide what forms of concurrency to exploit. [McGraw '82]

Another interesting feature in the language is the "let-in" construct that expands the current environment by the introduction of one or more value names, the definition of their values, and the evaluation of an expression(s) within their scope. Any value names used outside the scope of the "let-in" may be reintroduced in the "let-in" and are considered to be new value names. Any value names outside the "let-in" that are not defined by the "let-in" but are used within its scope, are inherited by the "let-in".

In Figure 1, the "let-in" expands the environment by declaring two new labels, "x" and "t", while the value of "p" is inherited from the outside. The values of "x" and "t" can only be referenced within the scope of the "let-in." Upon leaving the expression, the values cease to exist.

VAL LET-IN EXAMPLE

```
LET x : real; t : real;
    t := p + 3.7;
    x := t + 2.4;
in x * t
endlet
```

Figure 1

Recursion was not permitted so that each function invocation would be strictly independent with no state information from one invocation to the next. This was viewed by the VAL designers as a side effect and, therefore, undesirable. I/O operations are also not included as they produce unpredictable results.

VAL was designed specifically for numerical computation with the data-driven machine architecture as the primary target for translation of programs. However, the design was developed with the view of allowing the language to evolve into a general purpose language in the future when a general purpose data-flow computer is available. [Ackerman and Dennis '79]

Work is currently being done to address the issues of I/O operations and file updating. Weng's [Weng '75] work on computation on stream values is being investigated as a means for communicating between modules.

2.2.2 IRVINE DATAFLOW (ID)

ID was designed and developed by the data flow project at the University of California at Irvine in the late 1970s is the most robust of the data flow languages at this time. and Principle members of the group were Arvind, Kim P. Gostelow, and Wil Plouffe.

ID is a block-structured, expression oriented, single-assignment language. A program in ID is composed of a list of expressions. The language supports values, blocks, conditionals, loops, and procedure applications. In addition, it

supports some new concepts such as streams, functionals, and nondeterministic programming. [Arvind, Gostelow, and Plouffe '78]

As in all data flow languages there is no concept of a memory location that is manipulated; all data types--integer, real, boolean, string, structure, procedure definition, manager definition, manager object, programmer-defined data type, and error--are treated as mathematical values.

In ID there are no explicit declarations and no type checking. Since the internal representation of the values is self-identifying, type is associated with the value and not with the variable. Primitives are available to test the type of a value and convert it to a different type if necessary.

Loops in ID are composed of an initialization section, a body, and a return section indicating the value or values to be returned from the loop. Within this framework ID provides several different loop constructs: for-loop, repeat-until-loop, and for-while-loop. (See Figure 2 below) All concurrency is implicit as each value of the variables is tagged with the iteration number. As the U-interpreter [Arvind and Gostelow '82] (the "unfolding interpreter" detects parallelism by uniquely labeling independent activities) unfolds the loops, any iterations not dependent on one another can process simultaneously. In VAL the "forall loop" must be used to explicitly identify independent loop iterations.

ID LOOP EXAMPLE

The semantics of all loops are encompassed within the general while-loop:

```
( initial x <- f(a)
  while p(x,c) do
        y <- g(x,c);
        new x <- h(x,y,c)
  return r(x,c) )
```

Figure 2

Conditionals can be thought of as selectors of values. Whichever branch is taken, the value or values in the body of that branch are returned. The "then" and "else" clause must contain the same number of expressions to insure the single-assignment rule is not violated. For example:

```
y, z <- (if p(x) then f(x),25 else g(x))          illegal
y, z <- (if p(x) then f(x),25 else g(x),h(x))     legal
```

Procedure definitions are separate subprogram units similar to the format in "C"; they are recursive. Multiple arguments and multiple results are permitted; however, if there are more actual arguments in the invocation than there are formal arguments in the definition, the surplus values are ignored. If there are not enough actual values, error values are generated for the remaining formal parameters. (See Figure 3 for a procedure example.)

ID PROCEDURE EXAMPLE

```
procedure quicksort (a,n)
   (middle <- a[1];
    below, j, above, k <-
       (initial below <-   ; j <- 0;
                above <-   ; k <- 0

        for i from 2 to n do
             new below, new j, new above, new k <-
                 (if a[1] < middle
                     then below + [j+1]a[i], j+1, above, k
                     else below, j, above + [k+1]a[i], k+1)
        return (if j>1 then quicksort(below,j) else below), j,
               (if k>1 then quicksort(above,k) else above), k)

    return (initial t <- below + [j+1]middle
            for i from 1 to k do
                 new t <- t + [i+j+1]above[i]
            return t))
```

Figure 3

Block expressions are lists of expressions followed by a return section indicating the outputs of that block. The inputs to a block expression are those variables referenced but not assigned within the block as shown in the following block expression:

```
(x <- sqrt (b ↑ 2 - 4 * a * c);
 y <- 2 * a
 return (-b + x) / y, (-b - x) / y)
```

In ID there are either simple or stream variables. Simple values are represented in their entirety in a single token (integer, structure, etc.); but stream variables are an ordered sequence of tokens (possibly unbounded), where each token carries a simple value. [Arvind, Gostelow, and Plouffe '78]

Operations on streams can be performed before all input tokens are defined and some output can be produced thus creating pipeline concurrency. Some operating system routines (I/O drivers) and on-line updating of data bases are examples of applications for stream processing.

Weng [Weng '75] first proposed streams in data flow in his masters thesis where he gave formal rules to construct "well-formed" data flow schemata with streams in VAL. ID extends this concept by providing a loop structure (for-each) to operate on streams.

2.2.3 DATA DRIVEN PROGRAMMING LANGUAGE (DDPL)

DDPL was developed by ESL Incorporated (Circa 1982) specifically for the Data Driven Signal Processor (DDSP), also developed by ESL Incorporated. The primary goal of DDSP and DDPL is the programming of multiple processors to achieve very high speeds for applications requiring speeds that cannot be achieved by conventional methods. Data flow techniques were chosen as a basis for the DDSP design because of the ability to program algorithms in multiprocessor environment. DDSP can be configured from one to 32 processors with a full speed capacity of 71 million floating point operations per second (MFLOPS); and, it interfaces with a variety of devices allowing for concurrent data and I/O processing. [Hogenaur '83]

DDPL is a high-level language with syntax modeled after Ada. It is a block structured language with a program block containing one or more procedure blocks. The procedures define groups of code (node definitions) with a common purpose

and communicate via data driven communication structures that are linked lists containing both data (for computation) and control information (used to invoke procedures and route output). A node definition may have one to four input ports and an unlimited number of output tokens.

A unique feature of DDPL is the programmer's ability to specify how label fields are subdivided in the declarations at the beginning of each procedure. These fields allow the programmer to specify the next processor output will flow to and the number of calls to a procedure that can execute simultaneously.

2.2.4 DATA FLOW LANGUAGE (DFL)

DFL is an applicative language. It is a block structured, single-assignment language with strong type checking. It borrows heavily from VAL and closely resembles PASCAL in appearance. Concurrency is basically implicit in DFL except for the "forall" construct that allows the programmer to express some special forms of parallelism in array expressions. The array (of one or two dimensions) is the only complex data type supported by DFL. Records, user-defined types, and stream data types are not available.

DFL is intended primarily for mathematical computations; facilities for string manipulation are not available. [L. M. Paitnaik et. al. '84]

Figure 4 below is a complete DFL program that returns the sum of the squares of the first two arguments and zero if the third argument is true, and it returns two zeros if the third

argument is false. A complete BNF specification of the language and several expression examples are included in Appendix A, pages 126 - 128.

DFL PROGRAM

```
program(input a,b: integer, c: boolean) yield integer,integer;
   define RS:= procedure(input a: integer) yield integer;
      define SQUARE:= procedure(input a: integer)yield integer;
           a*a
      end;
      SQUARE (x) + SQUARE (y)
   end;
   if c then RS (a,b), 0 else 0,0
end;
```

Figure 4

2.2.5 LUCID

LUCID was developed by Ashcroft and Wadge in 1976-77. [Ashcroft & Wadge '76 and '77] It is not only a language, but also a formal system for proving properties of LUCID programs. Primarily developed for program verification rather than programming, it is, however, an appropriate language for data flow computers. [Ackerman '79]

Some of the existing implementations include a compiler in the language B by Chris Hoffman at the University of Waterloo running under TSS on a Honeywell 6060 that translates LUCID programs into an imperative language (assembly code); and, two interpreters (based on a demand-driven data-flow

model): one at University of Waterloo by Tom Cargill and one at the University of Warwick by David May.

The main idea in LUCID is that programs should be "denotational" (all expressions in a program must mean something) and "referentially transparent" (2 or more occurrences of the same expression must represent the same something). [Ashcroft and Wadge '76]

LUCID includes assignment, conditionals, and while loops; but, there are no procedures, no type declarations, and no statements. Programs are made up of expressions (terms), and these terms are constructed by using constants, variables, operation symbols, and user-defined function symbols. Function symbols are defined as values of terms (similar to variables) but with formal parameters. A program is a term where every function symbol that is used is defined in some clause. [Jagannathan and Ashcroft '84]

In a more informal approach one can think of a LUCID program as being built up from simple loops. The program in Figure 5 illustrates most of the features of LUCID syntax.

LUCID PROGRAM

The following program computes the running root mean square of its inputs.

```
sqroot (avg (square (a)))
   where
       square (x) = x * x;
       avg (y) = mean
                 where
                     n = 2 fby n + 1;
                     mean = y fby mean + d;
                     d = (next y - mean) / n;
                 end;
       sqroot (z) = approx asa err < 0.0001
                    where
                        Z is current z;
                        approx = Z / x fby
                            (approx + Z / approx) / 2;
                        err = abs (square (approx) - Z;
                    end;
       end
```

The operators "fby" and "asa" stand for "followed by" and "as soon as", respectively.

Figure 5

2.2.6 CAJOLE

CAJOLE was designed to reflect the basic applicative nature of data flow. A CAJOLE compiler was implemented on a conventional Von Neumann architecture with the hope of developing a compiler to produce code for the Manchester data-flow computer.

A program consists of a list of definitions each of which can be thought of as a function definition, for example:

```
a = 6;
xplusy = [x,y] x + y;
c = xplusy (a,b);
b = 13
```

defines "c" to be 19. The bracketed list is a list of formal parameters. As in all data-flow languages, the order of the definitions is irrelevant. There are conditionals but no iterative constructs; however, functions can be recursive. There are no explicit provisions for data structures, but it is possible to define data structures by writing lists of values in angle brackets. The following definition:

```
vec = < 3, 5, 7, 11 >
```

is equivalent to:

```
vec = [n]  { n = 1 : 3,
             n = 2 : 5,
             n = 3 : 7,
             n = 4 : 11 }
```

Figure 6 presents a larger CAJOLE program that produces a vector of the first n numbers of the form $2^i 3^j 5^k$. The WITH...WEND construct allows names to be defined more than once in a program by localizing them to the expression where the WITH...WEND occurs.

CAJOLE PROGRAM

```
stream = [n] vec
   WITH
      vec = { n = 1 : <1>,
              OTHERWISE : <1> && <SP> && dijk (n,2,1,1,1)
            }
         WITH
            dijk = [n, m, x2, x3, x5]
                   { m = n : <minim>,
                     OTHERWISE : <minim> && <SP> &&
                         dijk (n, m + 1, inc (x2, t2, minim),
                               inc (x3, t3, minim),
                               inc (x5, t5, minim) )
                   }
               WITH
                  t2 = 2 * vec (x2);
                  t3 = 3 * vec (x3);
                  t5 = 5 * vec (x5);
                  inc = [x,y,z] { y = z : x + 2, OTHERWISE : x};
                  minim = min (t2, min (t3,t5))
                     WITH
                        min = [x,y] { x > y : y, OTHERWISE : x}
                     WEND
               WEND
         WEND
   WEND
```

The symbol SP represents a space character.

Figure 6

Continuing research is being conducted on including generalized I/O facilities into the language. This work is being done by the Research Group at Westfield College in London and is part of a coordinated program being supported by the Distributed Computing Panel of the UK Science Research Council. [Hankin & Glaser '81]

2.2.7 DAISY

DAISY was developed by A. T. Kohlstaedt in 1981 at Indiana University and is a descendant of pure LISP. It is a

purely applicative, interpreted language that is part of a programming system called DSI. It was originally implemented in Fortran on a DEC-10 computer and in 1981 was transferred to a VAX 11-780 operating under UNIX. Plans are currently underway to translate the implementation to "C".

It is a functional program language in which the basic form of expression is a system of mutually recursive equations. Daisy has a call-by-value semantics and its interpreter computes by graph reduction. It is side-effect free, therefore, call-by-name is implemented transparently using a "suspending CONS".

The "suspending CONS" makes Daisy's evaluation lazy. This means that it is a demand driven language rather than a data driven language. Computations are not executed unless their output is needed as input to another instruction. When Daisy builds a list, it suspends evaluation of the lists elements. In other words, computations of an element's value is deferred until a probe attempts to access it. [Kohlstaedt 1981]

Daisy looks like a subset of LISP with a somewhat different syntax. Each function receives either a list or an atom as its argument, and variables are bound using pattern matching. Figure 7 below shows the LISP equivalent of several DAISY expressions. [Hall & O'Donnell '85]

DAISY = LISP EXPRESSIONS

```
f : <x y z>   =  (f x y z)
<a ! b>       =  (cons a b)
<a b c d>     =  (list a b c d)
(a b)         =  (quote (a b))
if:<p b1 b2>  =  (cond (p b1) (t b2))
first:L       =  (car L)
rest:L        =  (cdr L)
\ (x . y)     =  (lambda (x) y)
```

Figure 7

Figure 8 illustrates a complete DAISY program that deals a deck of cards to two players in a game like war.

DAISY PROGRAM

```
deal:deck = if: <empty?:deck   <[][]>
     empty?:rest:deck   <deck []>
     (cons    cons):
     << 1:deck     2:deck >
        deal:rest:rest:deck >>
```

Figure 8

See Appendix A, page 129 for more program examples.

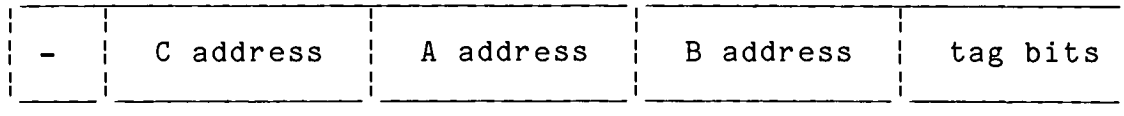
2.2.8 LAU

LAU is a single assignment language (with syntax similar to PASCAL) developed for the LAU system architecture by the Computer Structures Group at the University of Toulouse in Toulouse, France. Both a compiler and a simulator have been implemented. The simulator allows up to 12 Arithmetic Unit processors and 8 Control processors. [Comte, Durrieu, Gelly, Plas and Syre '78 & Plas, Comte, Gelly and Syre '76]

Data entities defined by the programmer are referred to as "objects"; once computed an object value is unique and will not change. Instructions are organized on the concept of the Data Production Set (DPS). A DPS is defined as:

a set of instructions, I
a set of data objects, O

For example: C = A - B is implemented by the following instruction



The DPS for this is:

I : the above instruction
O : C

An instruction is completely defined by one or more DPSs and how it will operate on its DPSs. [Plas, Comte, Gelly and Syre '76] As a result of this, all instructions are considered as assignment instructions though there are different syntactic structures and semantic properties. [Comte, Durrieu, Gelly, Plas and Syre '78]

Language features include mathematical operations, parallel assignment statements, a generalized "CASE" statement,

procedures, and loops. An "EXPAND" statement is included which is comparable to the "FORALL" of VAL, which provides for explicit parallelism in a loop whose iterations are independent of one another and can therefore execute concurrently. [Gajski, Padua, Kuck '82]

There is a "CREATE" command that looks very much like the class definition in SIMULA 67; however, it is not a type definition but an object definition designed to handle such things as shared resources, stacks, etc. (See Figure 9 for sample expressions.)

LAU INSTRUCTION EXAMPLES

Example of LOOP statement (computation of factorial):

```
LOOP L
   OUT : FACT; within the LOOP,
                FACT denotes NEW FACT
   LOCAL : I;
   (START) : FACT = 1; I = 1; L = GO;
   (GO) : I = OLD I + 1;
              CASE (OLD I > N) : STOP
              LOOP L;
                  (ELSE) : FACT = OLD FACT * OLD I;;
              END CASE;
END LOOP L;
```

Example of CASE instruction:

```
CASE X
    (X = 0) : Y = 2;
    (X = 1) : Y = 3;
    (X > 1) : Y = 4; Z = 3;
    (ELSE) : Z = 1;
END CASE;
```

Example of EXPAND statement:

```
EXPAND I = A STEP B TO N:
     TAB (I) = X + I;
END EXPAND;
```

Figure 9

3.0 PROJECT DESCRIPTION

3.1 OPERATIONAL OVERVIEW OF COMPILER

The purpose of the compiler is to convert the source program into an equivalent mnemonic assembly language that represents a directed data-flow graph. The mnemonic code generated by the compiler is then converted by the assembler into machine language that can be executed by the simulator.

The compiler makes a single pass of the source code file, filename, to generate a new file, filename.d, containing the mnemonic directed data-flow graph. In this pass, each high-level expression is converted into one or more tuples that implement that expression at the machine level. Each tuple corresponds to a machine-level instruction that contains information identifying the machine instruction; the input operands (called input tokens in data flow terminology); and the output(s) (also referred to as token(s)), if any.

Two passes are then made of the tuples. On the first pass each instruction that has any input tokens accesses the appropriate symbol table for each and inputs the destination address, which consists of the instruction number and input port number. Type is checked to be sure that the instruction generating the input token is producing what the receiving instruction expects to receive.

On the second pass, each tuple is interpreted into its mnemonic equivalent and written to the newly created file

(filename.d). For each tuple the instruction number, name, number of copies of the output token needed, and the destination addresses for each output copy is written out to the file.

3.2 COMPILER PROGRAM ORGANIZATION

The compiler program is organized into the following modules:

DIVA.Y

```
YACC specification code
explst_proc (in)
explst_other (pt)
voke_folow (pt)
loop_ctrl_var (did,pt,typ)
do_loop1 ()
main (argc,argv)
yyerror (s)
gen (a,b,c,d,e)
```

ASSEMDF.C

```
assemdf
```

INSYMTAB.C

```
insertsymtab (kind,contokens,totalpor,instnum,indexv,
              portno,epyt,name)
stants (newcount,contab,intrno,valu,dportno)
tempin (i,instno,dportno,epyt)
iniotab (index,portno,eypt,instnum,dport)
adder_labe_in (id,top,instnum,i,portno,epyt,contokens,
               totalpor,repeat)
switch (id)
get_io (index,port)
```

MEMORYRTES.C

```
creat_find_labe (pte,top,text,index,toptr)
put_lab_in (top,typ,text,len,did)
put_assign_in (top,value,oper)
io_creat (pte,loc,portno,typ)
new_tem (i)
chk_assg (flag,savflag)
add_in (use,tcval)
fcn_params (ptr,top,type,temp)
install_labes (dup,decling,top,type,yname,size)
```

COND.C

```
find_type (into,top)
arith_tems (temp,operptr,typ)
need_gate (hunt,top)
do_type (p1,top,proptr)
do_proc_tem (p1,in_pt,type)
doub_ops (p1,p2,top,oper)
relop_doub (p1,p2,oper,type)
singl_ops (p1,top,oper)
put_gate (torf,p1,flag)
chk_only_asgn (only)
```

SUBASEMS.C

```
outdests (tp,tem,outport)
outiodests (tp,pt)
outype (tp,pt)
outstants (tp,total,contab)
errorouts (copies,instno,instid)
```

LEX.L

YACGLOBAL.H

TOKENS.H

GLOBALS.H

3.2.1 DIVA.Y

The main subprogram segment reads in the source program file from the command line and creates a new file with the same name with a ".d" appended, which contains intermediate code for the data flow simulator. The intermediate code is a mnemonic representation of a directed data-flow graph.

This is the main "driving" module of the compiler; it specifies the parsing rules and generates the tuples that map the source code into an equivalent data-flow graph. It then calls other modules to insert destination addresses for tokens and to interpret and write out the equivalent intermediate code represented by the tuples.

3.2.2 ASSEMDF.C

"Assemdf" makes two passes of the instruction tuples: the first pass inserts destination information for the output tokens of each instruction in the appropriate tables; the second pass writes out the mnemonic code for each instruction to filename.d.

3.2.3 INSYMTAB.C

On the first pass of the instruction tuples (each instruction tuple represents a node in the data-flow graph), this module inserts the destination information required to identify the arc(s) traveled by each token. For each input operand to a tuple; it identifies the appropriate symbol table, the location of the token in that table, and inserts the instruction number and input port number in the identified table.

The module returns the number of unique start constants.

3.2.4 MEMORYRTES.C

This module contains functions that are called to build the labels symbol table and subsidiary tables containing destination information.

As labels are declared and assigned to, functions are called to allocate symbol table space; store type, scope, and value information; and check for multiple assignment.

3.2.5 SUBASEMS.C

Functions in the "Subasems" module are called to print out the character representation of the type for various instructions, the destination addresses for each token, and all data for the program and start constants at the end of each function and the main program segment.

3.2.6 COND.C

This module is called to build the tuple information for the instruction passed to it, and returns the value of the "operptr" assigned to the output of this tuple. It checks the type of each input operand and whether or not the operand is generated by a function call. The invocation symbol table is updated for invocations with a reference to the instruction's operand for destination input later.

If the instruction is in the branch of a conditional expression, an additional instruction is inserted to check the outcome of the conditional test before the operand value is allowed to arrive at the instruction.

There is a miscellaneous function, chk_only_asgn, that was placed in this module for want of a better place to put it. It is called to output an error message when an expression other than assignment occurs in the assignment-only section of a "let-in" expression.

2.2.7 LEX.L

"Lex.l" is the specification module for the lexical analyzer written in Lex.

3.2.8 YACGLOBAL.H

"Yacglobal.h" contains the data structure definitions and constant definitions used in the ".c" and ".y" files.

3.2.9 TOKENS.H

"Tokens.h" defines the value used by the hash function for function and label names.

3.2.10 GLOBALS.H

"Globals.h" contains constant and character array definitions used in the lexical analyzer and parser as well as in some of the ".c" and ".y" files.

3.3 COMPILER DATA STRUCTURES

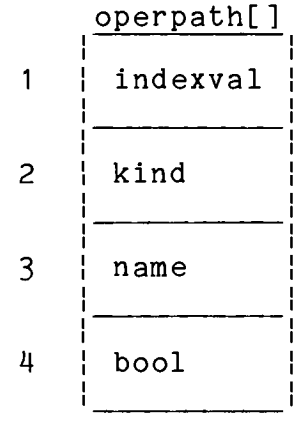
In a data-flow language, compilation must track and store the "data paths" for each value (token) created and referenced in a source program. This section describes the network and data structures used to implement this process.

3.3.1 OPERAND PATH POINTER

"Operpath" is an array of structures that stores the identifying information that is used to access the appropriate symbol table to input and later write out type and destination information. It is the central communication system of the compiler that facilitates the mapping of a source program into a data-flow graph. One of these structures is assigned for each token reference and function invocation. "Operptr" always indexes the next available structure in the "operpath"

array. In one and two operand expressions, the index (represented by "operptr") for each operand is stored in the tuple for that expression. In the case of instructions that have more than 2 operands, a pointer to the "operptr" information is stored. (This is broken down more fully for each instruction in a later section).

The general form of an "operpath" structure is:



1. indexval--For the particular input or output token this structure is assigned to, this value indicates the position in the appropriate symbol table. The following values are stored for the 4 cases (listed in 2 below).

    1. The hash value of the label name.

    2. Integer value of the constant. When a constant is referenced anywhere in a program, the current value of top is the index to the appropriate constant table. A table for program constants and start constants is constructed for each environment. A search is done of the appropriate table to locate the value referenced.

    3. The index value into the "temptab" assigned to the output of an expression evaluation.

    4. The index value into the "invoc" array (see below) assigned to a procedure invocation.

2. kind--This indicates how the token reference was created; there are 4 possibilities:

1. a label reference
2. a constant
3. the result of an expression evaluation i.e.

x + y

4. a procedure invocation

This value identifies the appropriate table to input or access destination addresses.

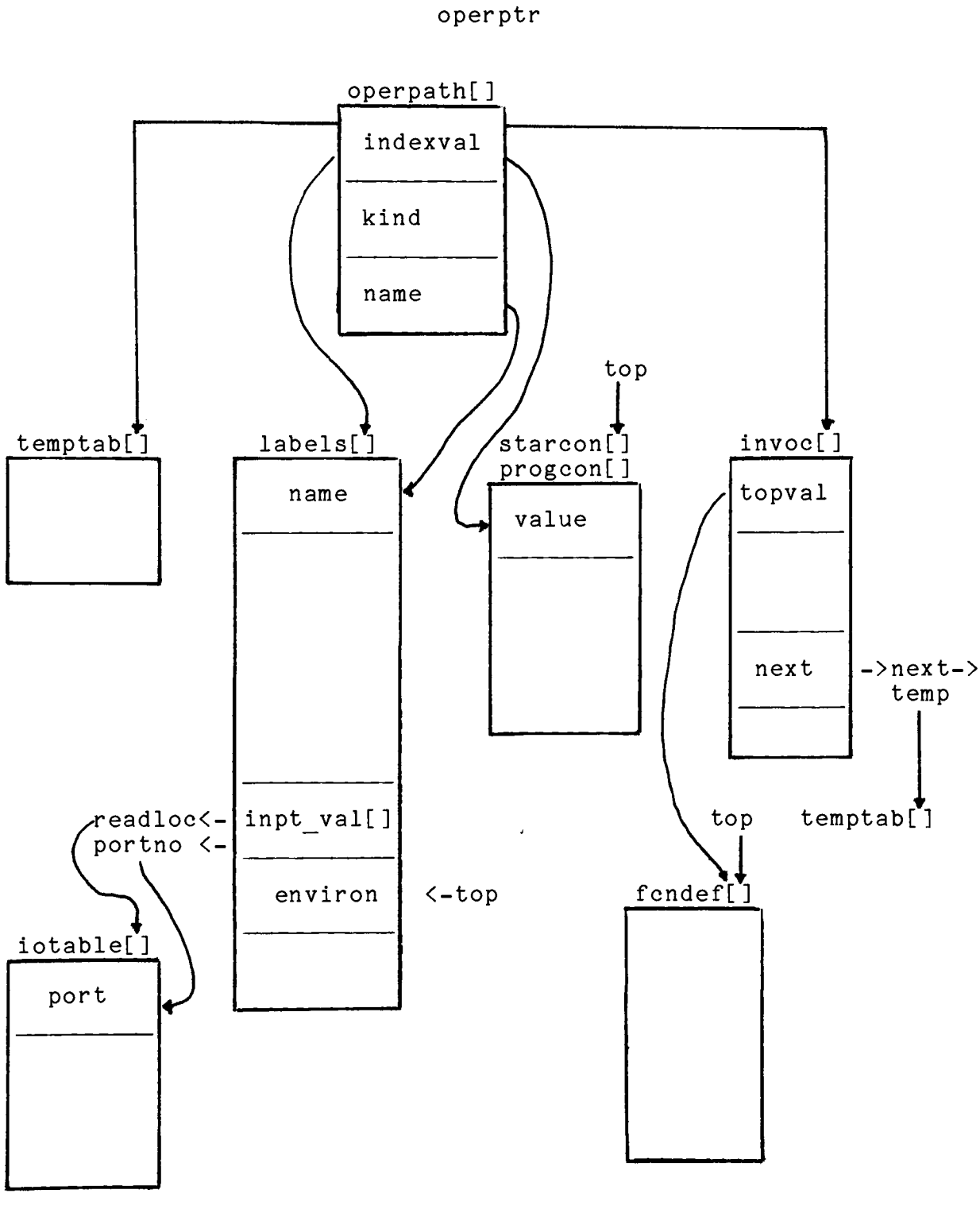
3. name--The character representation of the label or the function name (whichever is relevant) is stored in this field. If the structure represents a constant or temporary value, this is left empty. It is used in conjunction with its hash value to insure the appropriate label is identified in case of collisions.

4. bool--Provided in anticipation of future expansion of the project to include boolean constants.

When an expression creates a value, the "operptr" value stored for the output of that expression points to the path followed to access the destinations when the graph code is output.

An "operptr" for each input operand of an expression points to the path followed in order to input the destination addresses after instructions have been properly ordered and numbered.

The following diagram illustrates the complete "operpath" network.



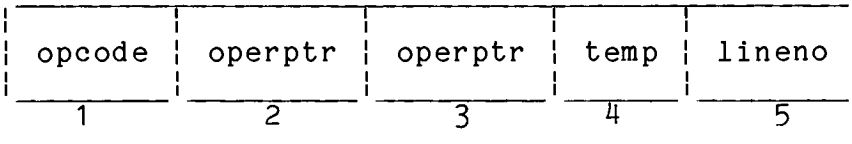
See discussion of "labels" for the role of "top" in that table.

3.3.2 INSTRUCTION TUPLES

For each assembly language instruction that is generated at compile time, a five-field tuple is created that stores the input and output arc information necessary to create that node in the data-flow graph. The following formats and information are stored for each instruction.

3.3.2.1 BINARY OPERATORS

The binary operators: +, -, *, /, %, <, >, <=, >=, !=, ==, and, or, tgate, fgate, switch



(1) The opcode determines the expression to be performed.

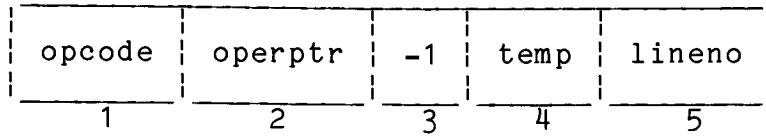
(2 & 3) These two positions represent the inputs to this instruction. These instructions must access the appropriate address table to store the input destination for the instruction(s) that generates the value(s). The destination tables are pointed to via the operptr network.

(4) The destinations of the output of an instruction are stored in a structure "temps". "Temptab" is an array of pointers, each of which points to a "temps" structure. Temp is the index value into "temptab": to locate the structure that acts as the communication point between the instruction outputting the value and any other instructions referencing this value as input.

(5) The line number in the source program that generated the instruction is stored with this tuple in order to facilitate helpful error information to the user.

3.3.2.2 UNARY OPERATORS

The unary operators: not, unary minus, absolute value, do, do1, loop1, loop

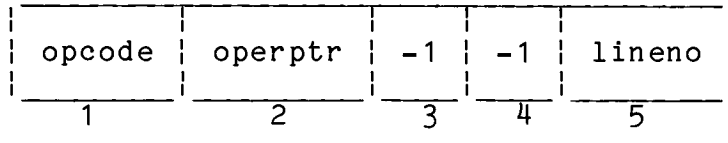


(3) The -1 indicates an empty field.

The do, do1, loop1, and loop operators are used to implement the loop structure (see the section on loops in the DIVA Language Manual).

3.3.2.3 HALT INSTRUCTION

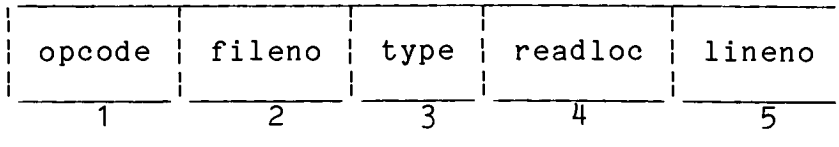
The HALT instruction is used at the conclusion of a loop to consume the output of the loop when it is used as a control flow device and not as an expression returning a value to be used by other instructions. It is a special case of the single operand operators.



3.3.2.4 I/O OPERATIONS

There are two I/O operations:

read



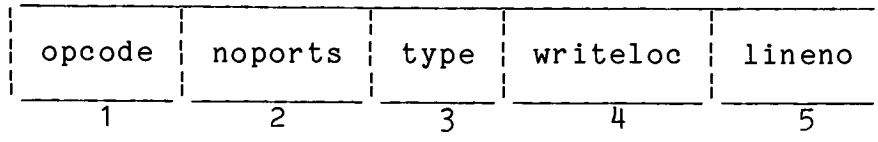
(1) Numeric representation of the expression to be performed.

(2) There may be up to five data files accessed by a DIVA program, this field indicates which file to get the data from.

(3) Indicates whether the data to be read in is integer or character.

(4) There can be from 1 to 20 inputs read in from a file on any one "READ" instruction. The destinations for the values read in are stored in a structure "input", "Iotable" is an array of pointers, each one of which points to an "input" structure. Readloc is the index value into the array to locate the structure that acts as a communication point between the "READ" instruction outputting the values and any other instructions referencing any of the values as input.

write



(1) Numeric representation of the expression to be performed.

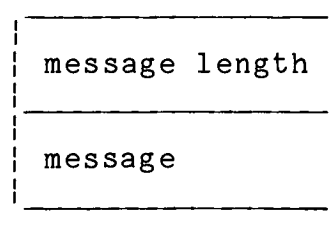
(2) The number of inputs to the write statement is stored in this field.

(3) The input to the "WRITE" statement may be either integer or character.

(4) The "WRITE" statement has two parts: (1) it includes a message that can be up to 20 characters long and (2) from 1 to 20 input values. The message is stored in a structure "missive" pointed to by a position in the array "note". The input pointers are stored in "wtewhere", which is a two-dimensional array. Writeloc indexes both of these structures (1) to get the message and (2) to put the destination address for each input in the appropriate destination address table.

(1) The following illustration is an example of a "missive" structure pointed to by any given location in the "note" array:

note[]



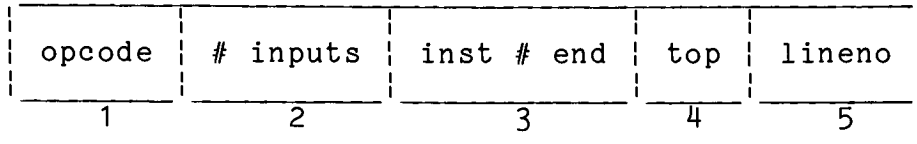
(2) wtewhere [] []

Writeloc indexes the first dimension of the the array "wtewhere". The second dimension has 20 positions, one for each possible input port. The "operptr" for each input value is stored in the corresponding port position in the array.

3.3.2.5 FUNCTION INSTRUCTIONS

There are three instructions related to functions:

begin



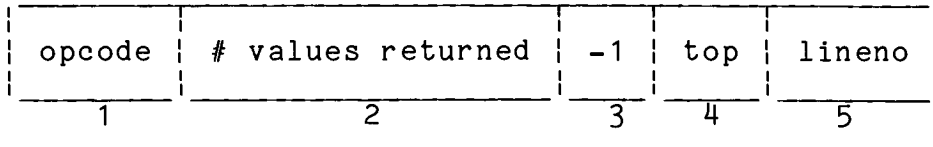
(1) Numeric representation of the expression to be performed.

(2) The number of formal parameters in the function definition.

(3) While there are 20 input ports to the "begin" instruction of a function definition, there can only be up to 19 formal parameters because the first output port is used to transmit the instruction address token to context control token that is sent to the "end" instruction of the function definition. The address of the "end" instruction is needed to transmit this token.

(4) Top points to the description record for this function definition. The description record stores the operptrs (in addition to other information) for the input parameters.

end

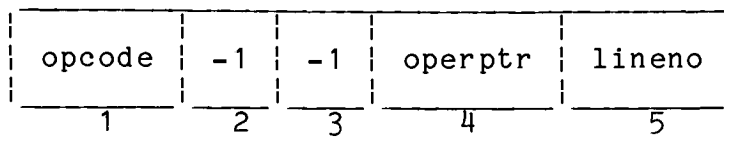


(1) Numeric representation of the expression to be performed.

(2) The number of values returned by the function.

(4) Top points to the description record for this function definition. The "end" statement accesses the description record to get the operptrs for the function's output values that must be sent to the "end" statement. The "end" statement uses the operptrs to put its address in the appropriate destination table for each of the values sent to it.

apply

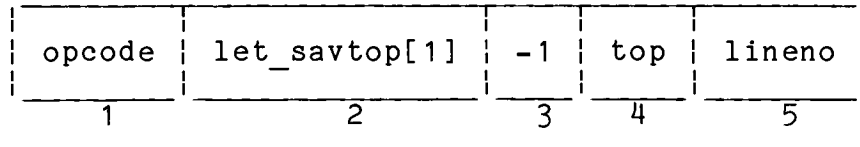


The (4) "operptr" provides access to the invocation record for each function call. This record coordinates type checking and destination information between the function description and the calling environment. See the discussion of function and invocation structures further on for more detail.

3.3.2.6 LET INSTRUCTIONS

The "let" generates two instruction tuples:

let

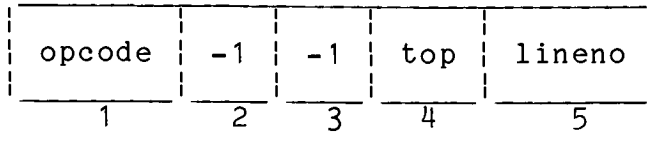


1. Numeric representation of the expression to be performed.

2. let_savtop[1]--Any constants referenced by a "let" expression are listed in the constant list of the main or function block where the "let" appears.

4. top--Contains the number of the activation record of the current "let" expression. Because "let" expressions can be nested, the "let" construct requires a stack to store the "let" activations in order of their invocation, as identifiers from outside the "let" construct (that are not redefined in the "let") can be assigned to or referenced in its body. An identifier search starts at the current "let" activation record and works back through the activation records (indicated by the index values stored in the "let_savtop[]" stack) until the identifier is located.

endlet



1. Numeric representation of the expression to be performed.

4. top--Restores the previous activation record pointer.

3.3.3 IDENTIFIER SYMBOL TABLE

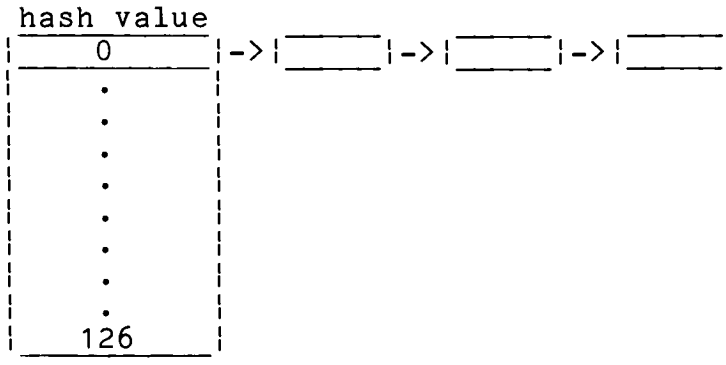
The identifier symbol table coordinates destination information between the assignment statement creating a value and the expressions referencing it. A simple hash algorithm is used to represent the identifier names. The resulting value is used as an index into the table.

The hash value is calculated by:

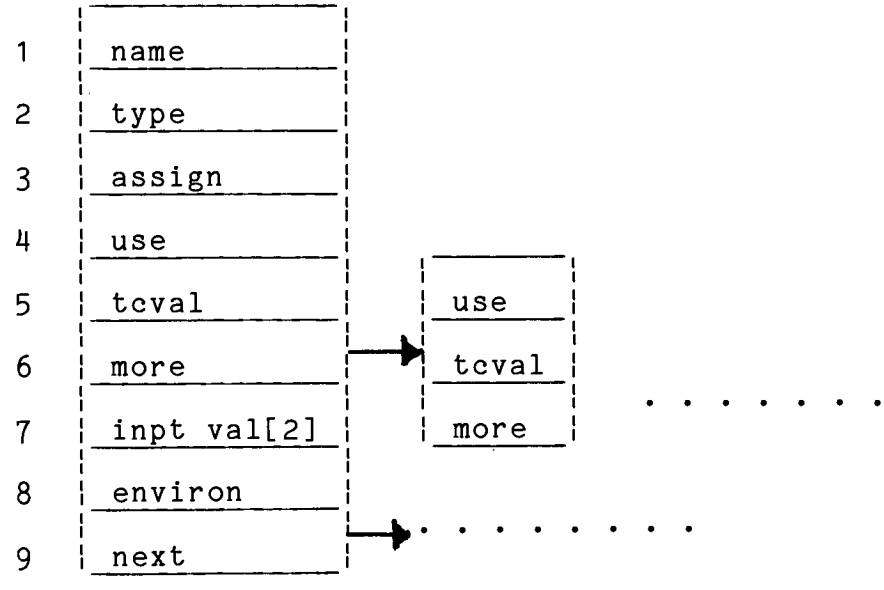
1. Sum the integer representation of each character of the identifier.

2. Divide the sum modulo 127 and use the remainder.

The table is created by an array of pointers "labels", each position contains a pointer to a structure "id_node". If there is a collision, a linked list is constructed at this location and the new identifier is entered at the end of the list.



The "id_node" structure contains the following information:



1. name--The character representation of the label is stored in order to locate the appropriate label, as collision may have created more than one link to the list at the same index location. A search of the linked list is done until a match is found or the list ends. If no match is found, the label (identifier) is undeclared and compilation is terminated.

2. type--The type listed in the declaration for the label (identifier) is stored as an integer.

   boolean = 1
   integer = 2
   character = 3

3. assign--Initially this has a value of 0 indicating that the label has not been assigned a value. This is increased by 1 when the label is assigned to. This field

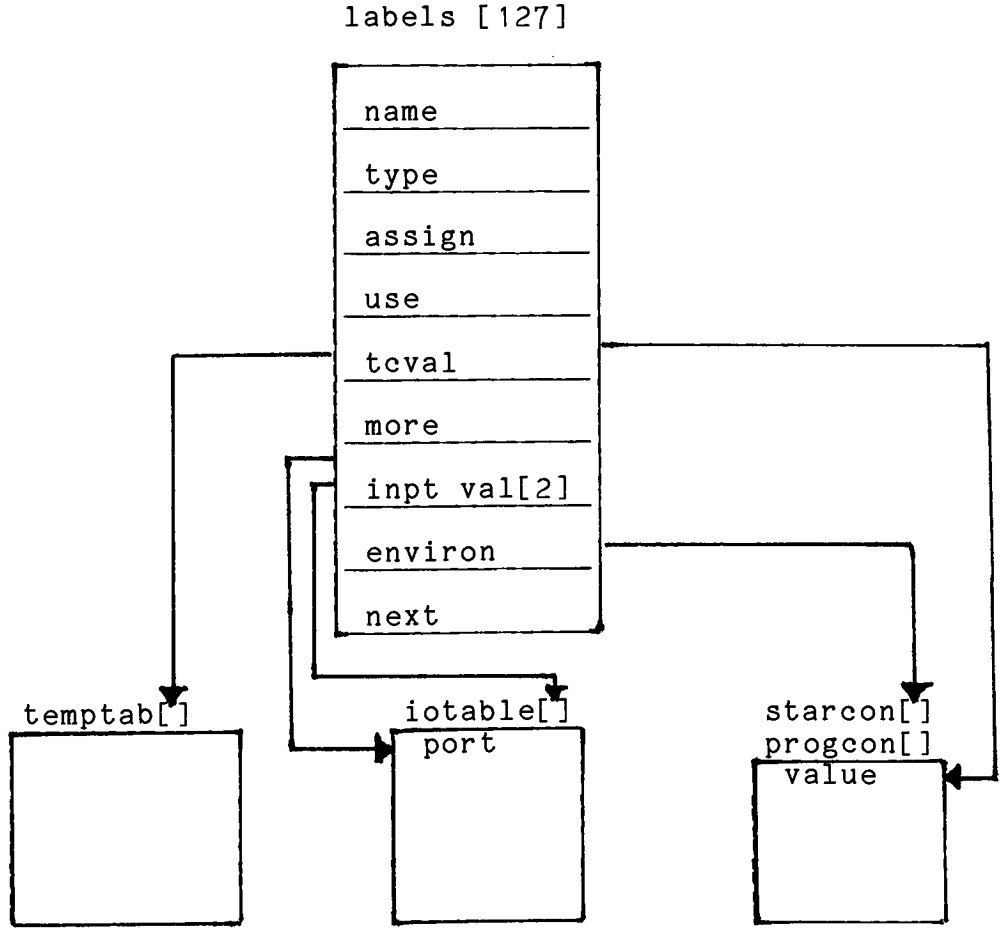
is checked to enforce single assignment. Only in conditional expressions is this value allowed to be greater than one.

4. use--A label can be created in 3 different situations; as an intermediate value, as a constant, or as the result of an input operation (READ).

When a label occurs on the left-hand side of an assignment statement with an expression on the right-hand side, it is considered an intermediate representation (a naming mechanism to the compiler to identify a token for reference elsewhere in the program) and its destination(s) is stored in the "temptab".

When a label occurs on the left-hand side of the assignment statement with a simple constant on the right-hand side, the destination(s) is stored in the appropriate constant table.

If the label occurs in a "READ" statement, its destination(s) is stored in the "iotable".



5. tcval--If the label is used as an intermediate or constant, its value (the constant value or the temp index) is stored in "tcval". The intermediate value directly indexes the appropriate destination record while the constant value is used to search the constant list. The constant list is indexed by "environ" (see 8 below).

6. more--This allows for multiple assignments to labels by the conditional expressions. For each branch of a conditional expression (after the first branch), more points to the next structure storing the use and index value for that assignment.

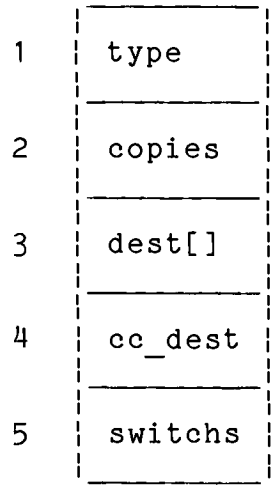
7. inpt_val[]--If the label is assigned to in a "READ" statement, the first number stored here is the index into the "iotable" that stores the destinations of the values read in; the second number is the port location the token comes in and goes out on.

8. environ--This indicates the relative environment (scope) of the label. The main block, each function, and each "let" expression has a unique environment identification number. When a label is referenced, its name is compared (#1 above) and its environment identification is checked for a match. When a constant is referenced, "environ" indexes the appropriate constant tables for that environment. The main block and each "let" expression have both a program constant table and a start constant table. Functions have only a program constant table.

9. next--Hashing may cause more than one identifier to be indexed in the same location in the identifier table. This field points to the next identifier record in the linked list when collision occurs. This field has a value of 0 if it is the last record in the list.

3.3.4 TEMPORARY VALUES DESTINATION TABLE

The "temptab" is the address communication point between the instruction generating a value and the instructions referencing that value as an input. The destination record (pointed to be each "temptab" position) for these values has five fields:



1. type--The type is stored for type checking to insure appropriate use of the value as input to other instructions. It is also necessary to include the type in the destination information provided to the simulator.

2. copies--Each time this value is referenced by another instruction, this field is incremented by 1.

3. dest[]--Dest is an array that stores the instruction number and port number for each copy of the value. The number of slots filled in the array is 2 times copies.

Fields (4) and (5) are only utilized by the loop construct as follows:

4. cc_dest--The "loop" instruction outputs a special context control token that must be sent to the "loop1" instruction at the end of the loop construct, which uses the value to restore the identification tag of the value returned by the loop. The destination address for this token is stored in this field. See the section in the DIVA Language Manual on loops for more complete information on these instructions.

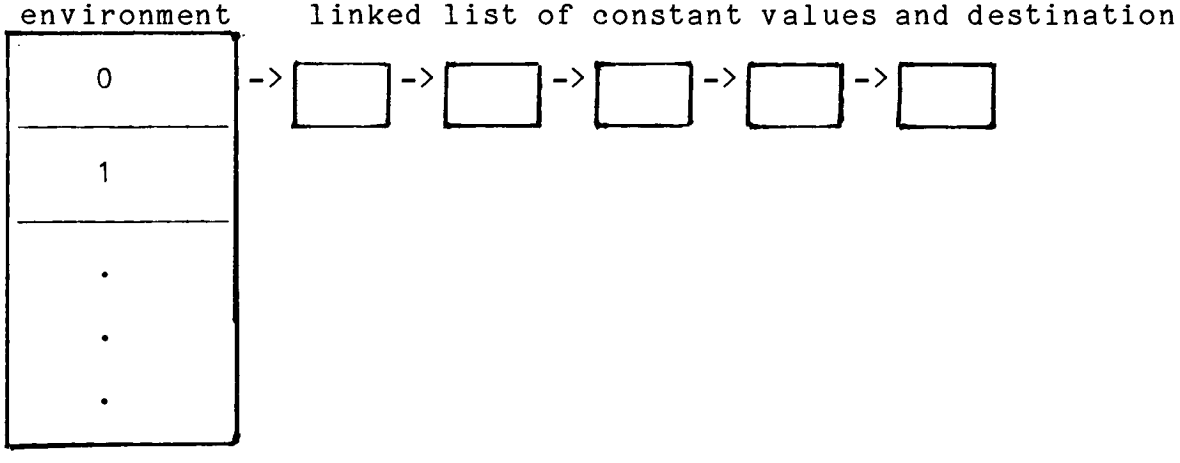
5. switchs--When the loop control test fails, the returned value is routed out of the "switch" instruction to the "do1" instruction, which sets its iteration identification tag field to 1 before sending it on to the "loop1" instruction. This field stores the address of "do1" for the output of the "false" port of the "switch" instruction. See DIVA Language Manual for more information.

As instructions reference a temporary value as input, the "temptab" is accessed to check type compatibility and update the appropriate fields.

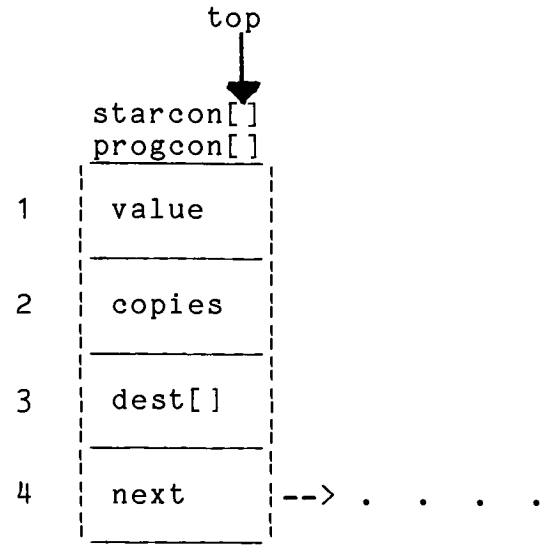
The variable "temp" is initialized to 0 and always points to the next available "temptab" position. As values are created, their type is inserted in the "temptab" and the "temp" value stored with the generating instruction.

3.3.5 CONSTANT VALUES DESTINATION TABLES

For each environment there are two constant lists (except functions, which have only a program constant list), one for program constants and one for start constants. The value of "top" always indicates the current environment; this is used to index the appropriate list for the start or program constants.



"Starcon" is an array of pointers pointing to the start constant list for each environment. "Progcon" is an array of pointers pointing to the program constant list for each environment. The record for each constant contains the following:



1. value--This field stores the integer value of the constant. When inserting values, the list is searched to see if the value is already entered. If it is, the destination is added to this record; if it is not, a new record is created and linked at the end of the list and the value and destination is entered.

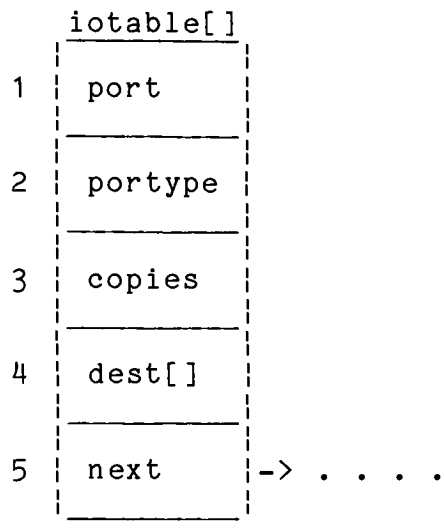
2. copies--Each time a value is entered, this field is incremented by one.

3. dest[]--For each copy of a value, the instruction number and port number are inserted in the next 2 available positions respectively. There can be up to 150 copies of a value.

4. next--This field points to the next value record in the list or to null if it is the last value record in the list.

3.3.6 DESTINATION STORAGE FOR READ INSTRUCTIONS

Each position in the "iotable" contains a pointer to a linked list. Each link represents one of the ports to a "READ" statement. These links are created dynamically at compile time as the number of ports used by each read statement can vary from 1 to 20.

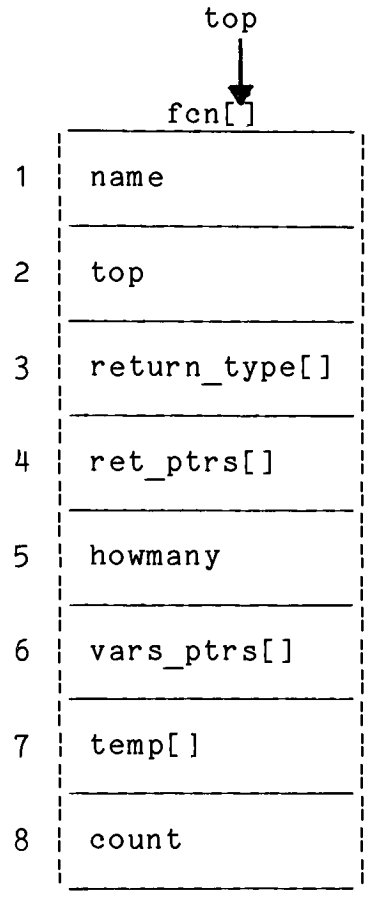


Both the (1) port number and the (2) port type are inserted when a "READ" statement is encountered. "Next" (5) is the pointer to the next link when there is more than one port used. The value of "next" is 0 for the last link.

Each time one of the port values is referenced by an instruction as an input, its (3) "copies" field is incremented by one and the instruction's number and input port number is added in the (4) "dest[]" field.

3.3.7 FUNCTION DEFINITIONS

Information is stored on functions as they are defined in the structure "proctable" pointed to by an array of pointers "fcn[]". Each position in the array "fcn[]" points to a structure "proctable". "Top" is used as an index into this array. Since the value of "top" is initialized to 0 in the main section of the program, this position is never used. In this table is the following information:



1. name--Stores the character representation of the function name. The activation records are searched for a match on this field when a function invocation occurs.

2. top--The function's index value to its description record.

3. return_type[]--There can be up to 19 values returned as the result of a function invocation. For each value returned, the type, as specified in the header, is stored as an integer value.

4. ret_ptrs[]--These are the "operptrs" for the values that will be returned. They are the input tokens for the "end" instruction of the function definition.

5. howmany--This is the number of values the function returns.

6. vars_ptrs[]--These are the "operptrs" for the labels listed in the declaration section of the header for the formal input parameters to the function. The values passed in via these names are the inputs and, therefore, also the outputs of the "begin" instruction of the func-

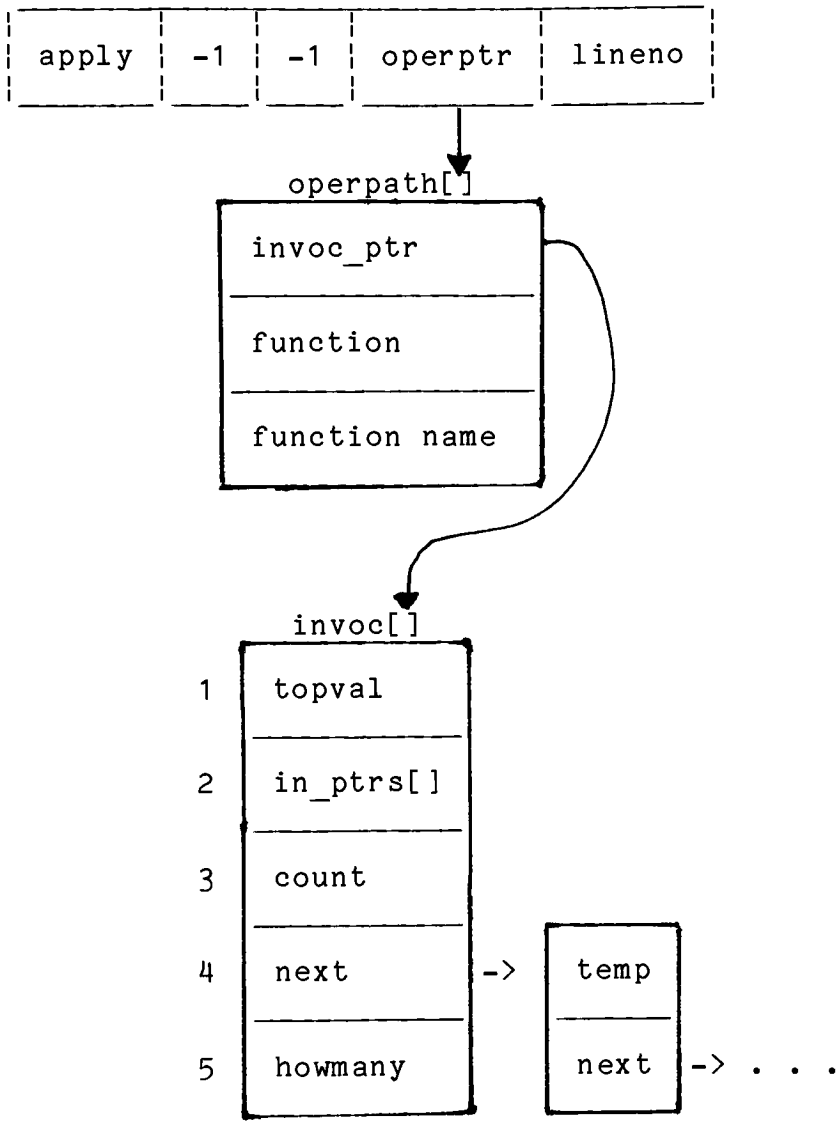
tion definition. The "begin" instruction routes the values to all expressions in the function that reference the formal parameters.

7. temp[]--For each label listed as a formal parameter to the function, the "temp" value assigned to each label for the access of destination information is stored in the corresponding array position.

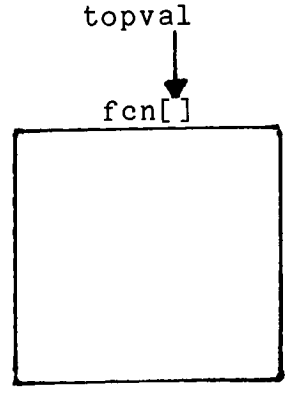
8. count--The number of input parameters.

3.3.8 FUNCTION INVOCATIONS

Information is stored on function invocations in the structure "call" pointed to by an array of pointers "invoc[]". Each position in the array "invoc[]" points to a structure "call". "Invoc_ptr" is used to indicate the next available position in the array "invoc[]". The index value for each function invocation is stored in its "apply" tuple in the "operpath" structure. "Operptr" indexes into "operpath" to access this information and is stored in the 4th position of the tuple. The structure "call" has the following information:



1. topval--This is the "top" value that indexes the description record that defines the function that is being invoked. This is needed to access the input parameter types for type checking and to get the destination addresses for the "apply" expression's arguments.



2. in_ptrs[]--These are the "operptrs" to the actual input parameter values. These are used to input the "apply" expression's address as the destination for these values.

3. count--The number of input parameters in the invocation.

4. next--This is a dynamic structure that stores the "temp" value for each value returned; these are the destinations for the output of the "apply" instruction.

5. howmany--The number of values returned by the invocation.

3.4 OPERATIONAL OVERVIEW OF ASSEMBLER

The assembler requires two passes of the source code to produce the machine language to run on the simulator. The first pass reads in the source code from filename.d and creates an intermediate file, filename.i, containing a stripped down version of the original file consisting of the program statements and their destinations. The second pass reads in the intermediate file, filename.i, and generates the machine language file, filename.m.

On the first pass of the assembler, all program and start constants and the stop, fin, and subp statements have been removed. The program and start constants are stored in their respective symbol tables for output to the machine code file at the end of the next pass. At the same time statements in subprogram definitions are renumbered so all statements are numbered consecutively, except the next statement after apply statements, from 1 to n; and, the name and address of each subprogram is stored in a symbol table. (See Appendix B, page 130, for symbol table descriptions)

The first line expected is the trace specification. This is read in and printed out to the filename.m machine code file.

One instruction and its destinations is read in at a time and stored in a record called "inst". (See pages 131 - 132 in Appendix B) It is identified and the appropriate routines called to process the instruction. If it is the "stop" instruction that is the signal to fall out of the loop reading in the main program section and to read in the constant section following it. If it is a "subp" instruction, the name and the address of its "begin" statement are stored in the symbol table. If it is the "fin" statement, that is the end of program signal that indicates the end of the first pass. All other statements are printed out as read in (with their destinations).

At the end of pass 1, the total number of statements is printed out to filename.m and filename.i is closed and then reopened in read mode for pass 2.

In pass 2 one instruction is read into the record "inst", from filename.i, and the appropriate machine code output to filename.m for the opcode. IDEN is called to identify the opcode and then call the appropriate print routine to write out the associated machine code to filename.m. The type information is added to the destination specifications at this stage. If the opcode is "apply", it is replaced by the activate and terminate operators. The address of the subprogram specified in the "apply" call is looked up in the symbol table to be inserted in the destination of the parameter tokens

passed to the procedure. Function names may be up to 25 characters long.

There can be up to 200 unique program constants and 200 unique start constants with up to 150 copies of any constant. There can be up to 200 subprograms specified.

3.5 ASSEMBLER MNEMONIC PROGRAM FORMAT

The first statement in all mnemonic programs is specification of the trace feature. If a trace is desired, the first line should be "trace"; if not, the first line should be "notrace". The statements of the main section of the program follow the trace specification numbered consecutively from one except when there is a subprogram call (apply). Apply is later replaced by the activate and terminate operators; therefore, the next statement after an apply statement is numbered two greater than the apply statement number. The stop statement indicates the end of the main program section. Following the stop statement are the constants used in the main program. First, the program constants (the number of unique constants followed by their specification); secondly, the start constants (the number of unique constants followed by their specification). Constant specifications are not numbered.

Following the main program section are the subprogram specifications, if any. The statements of each subprogram specification are numbered consecutively starting with 0. The first statement of a subprogram specification is the subp statement followed by the begin statement; the last statement is the end statement. Following each subprogram specification

are the constants used in that subprogram, using the same format as for the main program section.

The very last statement in a mnemonic program is the fin statement.

3.5.1 INSTRUCTION FORMATS

The mnemonic commands fall naturally into 5 pattern groups to identify the required machine code.

3.5.1.1 GROUP I

GROUP I--all these commands generate their output on a single output port; all have either boolean or integer input and output (for each the identification of this is self-evident). For this group it is only necessary to include the op code, number of copies, and their destinations.

The simulator has implemented relational commands for integer data types ONLY.

OPCODE	NUMBER OF COPIES	DESTINATIONS
plus		
minus		
mul		
div		
mod		
abs		
neg		
and		
or		
not		
less		
leql		
gre		
geql		
eql		
neql		

INSTRUCTION FORMAT:

instruction-# opcode number-of-copies destinations

For each destination listed the following information is output:

output port
instruction number
input port

3.5.1.2 GROUP II

GROUP II--the instructions that have more than one input and /or more than one output type, the second type is known upon identification of the op code. (i.e. LOOP1 always receives a context control token on input port 1 and LOOP always puts out a context control token on output port 1). It is necessary to include the data type of the other input data token for this group.

OPCODE	DATA TYPE	NUMBER OF COPIES	DESTINATIONS
tgate			
fgate			
switch			
loop			
loop1			
do			
do1			

INSTRUCTION FORMAT:

instruction-# opcode data-type number-of-copies destinations

3.5.1.3 GROUP III

GROUP III--includes the commands necessary for subprogram implementation. To facilitate the passing of multiple parameters of similar or dissimilar type, it is necessary to include the number of inputs and the type of each one.

OPCODE	NUMBER OF INPUTS	TYPE OF EACH INPUT	NUMBER OF COPIES	DESTINATIONS
apply (name of subprogram)				
begin				
end				

NOTE: activate and terminate are generated by the assembler to replace apply and, therefore, do not appear in this list.

INSTRUCTION FORMAT:

instruction-# opcode #-of-inputs type-of-each #-of-copies destinations

3.5.1.4 GROUP IV

GROUP IV--includes the i/o commands which are each unique in their own right. The simulator requires that all input tokens to be read from the specified file the input command must be of the same type on any given call. The output command is implemented by this assembler so as to be consistent with the input command in this respect. The output command does not generate any output tokens, but it does allow up to a 20-character message to be output with the input tokens.

INSTRUCTION FORMAT:

instruction-# input file-# #-inputs type #-of-copies destinations

instruction-# output string-message #-inputs type

3.5.1.5 GROUP V

GROUP V--includes the subprogram definition identification command and special end commands (end of main section; end of program; end of a line of logic that needs to swallow an output token). All that is needed here is the opcode recognition.

OPCODE

halt
subp (name of subprogram)
stop
fin

INSTRUCTION FORMAT:

instruction-# opcode

See Appendix B, page 133, for a list of all the opcodes and data types.

3.6 ASSEMBLER PROGRAM ORGANIZATION

3.6.1 ASSEM.C

"Assem.c" is the main program module of the assembler. Its function was described in the "Operations Overview of Assembler" section.

3.6.2 GLOBAL.H

"Global.h" contains data structure and constant definitions.

3.6.3 GROUP.C

On the first pass, "group.c" determines which read routine to use to read in the pertinent information for each instruction.

3.6.4 READLN.C

On both passes, "readln.c" reads in the next instruction and the destination information for tokens output by that instruction.

3.6.5 WRITELN.C

On the first pass, "writeln.c" writes out each instruction and the destination information for their output tokens to the intermediate file.

3.6.6 STANTS.C

When a constant is encountered, "stants.c" enters the constant and its destination addresses in the constant symbol table. The module returns the number of unique constants in the constant symbol table.

3.6.7 IDEN.C

On the second pass of the assembler, "iden.c" identifies the opcode of each instruction to determine the value of the parameters to pass to the appropriate print routine to write out the necessary machine code for that particular opcode. Since the halt statement is the only instruction that produces machine code in Group V, "iden.c" writes out the machine code for it.

3.6.8 PRS.C

The module "prs.c" contains four routines:

pr1
pr2
pr3
pr4

The instructions can be organized into five different catagories by the amount and kind of information each include. This necessitates different print routines for each group. Each routine writes out the machine language code for the commands in that group. Group V is not included as that is handled in the "iden.c" module.

3.6.9 MISC.C

"Misc.c", as the name indicates, contains three miscellaneous routines that service the rest of the program.

"Findaddr" searches the symbol table for the address of the indicated subprogram name and returns an integer value equal to the address of the subprogram if it is found in the symbol table. If the name is not found in the symbol table, an error message is printed out to the standard output and execution is terminated.

"Chktyp" returns an integer corresponding to the data type identified in the instruction record "inst", indexed by the integer passed in the function call. If it is not a valid data type, an error message is printed out and execution is terminated.

"Dests" prints out the 4-field destination information for the specified token to the designated file.

3.7.1 COMPLETE PROGRAM EXAMPLES

Pages 134-139 in Appendix B contain two program examples complete with source program, data-flow graph, compiler output, and assembler output.

3.7.2 DATA-FLOW REFERENCE CHART

Page 140 in Appendix B includes a reference chart that summarizes the data-flow instruction formats and codes for programming at the mnemonic code level.

4.0 DIVA LANGUAGE MANUAL

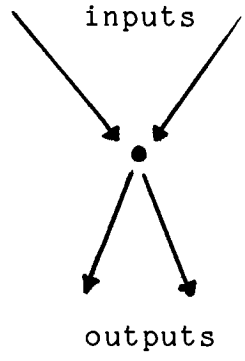
4.1 LANGUAGE OVERVIEW

DIVA (deeva) is a block structured, single-assignment functional language. It is based on the two data flow languages: VAL and ID. There is no updating of memory, no aliasing, and no sequential program control flow by a program counter.

Like all data flow languages it is designed so that concurrency is a natural phenomena rather than requiring explicit indication by semaphores or monitors, etc. This is achieved by providing control that is not sequential but dependent on availability of required data. [McGraw]

While the DIVA language is intended to be a data flow language based on ID and VAL, it does depart from them by including input and output statements and a usage restriction on some of the expressions. Currently, I/O operations are not included in data flow languages as it violates the objective of keeping them free from side effects.

DIVA programs are textual representations of directed graphs. Each node on a directed graph represents an expression or I/O operation. The nodes are connected by arcs over which values travel. Arcs coming into a node are inputs to that instruction, and arcs coming out of a node are its outputs. These arcs are also referred to as the ports of an instruction.



The values that travel over the arcs are called tokens. Tokens can be represented textually by label names (identifiers) via the assignment statement.

The environment of a token is the scope in which it exists. Hence environment and block are interchangeable semantically. A block may contain other blocks that can limit or expand the token values visible inside the internal block. For example, the "LOOP" construct temporarily limits the tokens visible in the block while the "LET" construct is a means to expand the tokens visible. The "LET" provides for redeclaration of outside identifiers or declaration of new ones as well as inheriting any other identifiers from the outside block that are not redeclared. A block can make token values visible outside the block by formally returning its values via the assignment statement.

4.2.1 ELEMENTS AND SYMBOLS

In programs the only control characters are the tab and newline. The program elements are operation and punctuation symbols, integer numbers, reserved words, and names.

The operation and punctuation symbols are:

+	-	*	/	%	=	¦	¦¦
&&	!	==	!=	<=	>=	<	>
(	)	"	:	;	,		

4.2.2 INTEGERS AND CHARACTERS

The simulator restricts all programs to integer constants with a maximum value of 32767 and minimum value of -32768. An integer number is a sequence of digits without a decimal point.

There are no character constants. Individual characters may be read in from an argument file and assigned to a value name. The value name can then be written to the standard output. I/O operations are the only operations available for character data.

4.2.3 RESERVED WORDS

Reserved words are words that have special significance to the compiler and, therefore, may never be used in any other context.

The reserved words are:

begin	endcase	in	read
beginfun	endfun	initial	return
boolean	endif	int	then
by	endinit	let	to
case	endlet	loop	trace
char	endloop	main	when
do	for	new	while
else	function	notrace	write
end	if	of	

Reserved words can either be in all capitals or all lower case letters.

4.2.4 NAMES

Identifiers, function names, and filenames can contain up to 25 alpha-numeric characters; but, the first character must be an alphabetic character. A name may not be the same as a reserved word. It may be used as a value name or a function name.

Upper and lower case letters are distinguished in names. The spelling and capitalization must be consistent in all uses of a name.

4.2.5 SEPARATING CHARACTERS

The acceptable separating characters are the space, tab, and newline. They are necessary to distinguish names, reserved words, constants, and program elements. In the example below, what was intended as an initialization of a loop control variable is identified by the compiler as an assignment statement due to the absence of separating characters. On the left-hand side "forx" is recognized as an identifier; the right-hand side produces a compile-time error because it is recognized as an illegal identifier name.

```
forx=1to10by2do
```

Separating characters are not required between operation or punctuation symbols; however, they are recommended for readability. The statement below is recognized as a correct assignment statement by the compiler.

```
x={a*10+base}
```

4.2.6 COMMENTS

Comments are initiated with a "#" and terminated by a carriage return. They may fill the entire line or only a portion of it as desired. If a comment requires more than one line, each line must begin with a "#".

4.2.7 SIMPLE PROGRAM

The simplest format contains only a main module introduced by a header setting the trace option on (trace) or off (notrace), and indicating any data files that are to be used by the program.

```
MAIN (header information)
     declaration section
BEGIN
     body
END
```

There may be zero to five data files listed in the header. The header for a program with the trace feature turned off and no data files is:

```
MAIN (notrace)
```

The header for a program with the trace feature turned on and two data files to be read from during the execution of the program is:

```
MAIN (trace, file1, file2)
```

The declaration section indicates the identifier names and their associated types for use in the "MAIN" block.

"BEGIN" then signifies the beginning of the body of the main block and "END" signals the end of the block and the program.

The following program computes 2 times the value of "**x**" and prints the result out to the standard output:

```
main (notrace)
     int: x;
     x = 5;
begin
     write ("result is: " : int : x * 2)
end
```

4.3.1 VALUES

The inputs and outputs of all expressions and functions are values. The 3 basic types in DIVA are:

character
integer
boolean

There are no arrays, structures, real numbers, or user-defined types, implemented at this time.

4.3.2.1 BOOLEAN TYPE

The boolean type has only 2 possible values:

true -- represented by a 1
false -- represented by a 0

4.3.2.2 INTEGER TYPE

The simulator implementation limits the integer range from -32768 to 32767. Any expression that produces a number larger or smaller than this will result in underflow or overflow and return a bogus number. The program will continue to execute without any recognition of the error.

4.3.2.3 CHARACTER TYPE

The character set includes 62 alphanumeric characters of the ASCII character set. It is comprised of the 52 uppercase and lowercase alphabetic characters (a-z and A-Z) and the 10 numeric characters (0-9).

4.3.3 TYPE DECLARATIONS & CHECKING

All labels must be declared at the beginning of the block in which they are referenced. Integer labels may be initialized to constant values in the declarations; these values may be declared before the label type is defined. The following examples illustrate the syntax and format of declarations:

```
int : y, x;
char : c, word;
boolean : gate, enter;
x = 5;
```

Example 1

```
int : yak, d = 2, 5;
```

Example 2

```
bold = 7;
int : bold;
```

Example 3

```
int : q, z;
q, z = 10, 3;
```

Example 4

At compile time, type checking is done to insure that the type of each expression matches its context. Mathematical

operators expect integer arguments, boolean operators expect boolean arguments, etc. Arguments to a function must match the argument type specified in the function's definition, and an expression on the right-hand side of the "=" sign in an assignment statement must match the declared type of the label on the left-hand side.

4.3.4 SCOPE

There are no global identifiers; all identifiers are local to the block in which they are defined. When the relevant block of an identifier is exited, the identifier no longer exists. Therefore, a label may be used in more than one environment as it is seen as a completely new identifier.

Function names can also be used as identifier names as their use is different. Reserved words cannot be used as either function names or identifiers.

Parameters to a function are passed using a call-by-value protocol. All values in a function are either passed to the function or locally defined within the function.

4.4 OPERATIONS

In discussing the data types of DIVA and their associated sets of operations, the following notation is used: P and Q for boolean, X and Y for integer. All boolean and integer operations are fully functional and can appear anywhere a value of the same type is expected.

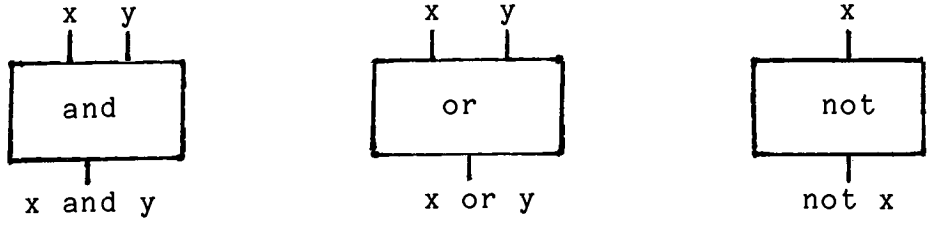
4.4.1 BOOLEAN EXPRESSIONS

Boolean expressions expect either 1 or 2 boolean argument tokens and produce one boolean output token. The following operations and notations are used in DIVA:

Operation	Notation	Inputs/Output
and	P && Q	boolean,boolean->boolean
or	P || Q	boolean,boolean->boolean
not	!P	boolean->boolean

These operations are the only way boolean values can be created in a DIVA program. Setting a label equal to a constant representation of true or false has no meaning in a data flow program as the value of the label cannot be reassigned. If the programmer knows the state of a boolean that determines the flow before execution, then it is unnecessary to test for it.

Below are the data flow graph representations of these operations.



For examples of usage in programs, see Figures 14 and 15 in Appendix C, pages 150-151.

4.4.2 CHARACTER EXPRESSIONS

There are no operations provided for character data other than the ability to read character data from a specified file

and output it to the standard output. See discussion of INPUT/ OUTPUT for more detail.

4.4.3 INTEGER EXPRESSIONS

DIVA provides both arithmetic and relational operations on integer data. With the exception of the negation and absolute value operations that expect only one input token, all other integer expressions expect two integer input tokens. The arithmetic operations produce one integer output value while the relational operations produce one boolean output value. The arithmetic operations and notation are listed in Figure 10 Table a, and the relational operations and notation are listed in Table b.

ARITHMETIC OPERATIONS AND NOTATION

Operation	Notation	Inputs/Output
addition	x + y	integer,integer->integer
subtraction	x - y	integer,integer->integer
multiplication	x * y	integer,integer->integer
division	x / y	integer,integer->integer
modulus	x % y	integer,integer->integer
negation	-x	integer->integer
absolute value	|x|	integer->integer

(Figure 10 Table a)

RELATIONAL OPERATIONS AND NOTATION

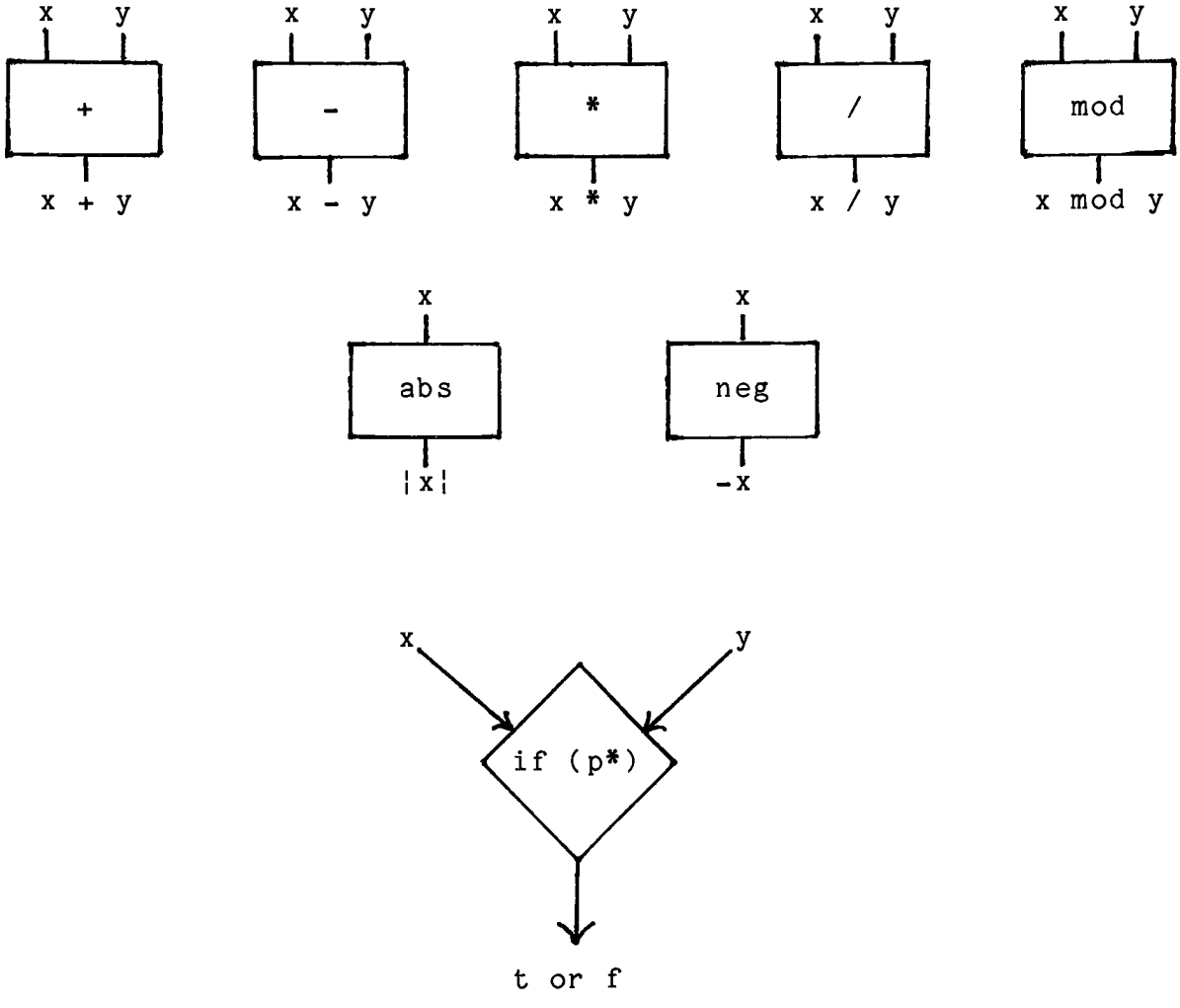
Operation	Notation	Inputs/Output
equal	x == y	integer,integer->boolean
not equal	x != y	integer,integer->boolean
greater	x > y	integer,integer->boolean
less	x < y	integer,integer->boolean
greater or equal	x >= y	integer,integer->boolean
less or equal	x <= y	integer,integer->boolean

(Figure 10 Table b)

Type errors in an expression will be found detected at compile time, an error message output, and the compilation terminated.

Figure 11 shows the data flow graph representation generated for these operations. Numerous examples of programs containing arithmetic and relational operations can be found in Appendix C starting on page 141.

ARITHMETIC & RELATIONAL DATA FLOW GRAPHS



*The possible values of p are <, <=, >, >=, ==, and !=.

Figure 11

4.4.4 OPERATOR PRECEDENCE

Figure 12 summarizes the rules and associativity for all operators. Operators on the same line have the same precedence; rows are in order of decreasing precedence.

OPERATOR ASSOCIATIVITY AND PRECEDENCE

Operator	Associativity
()	left to right
unary minus	right to left
* / %	left to right
+ -	left to right
== != <= >= < >	left to right
!	right to left
&& ||	left to right
=	right to left

Figure 12

4.5 ASSIGNMENT

Assignment has an identifier or list of identifiers separated by commas on the left side of the "=" sign and an expression or list of expressions preceded by a left curly brace "{" and separated by commas on the right side of the "=" sign. The end of the assignment statement is signaled by a "}" curly brace.

```
x , y , z  =  { 10 , 5 , 9 }
var  =  { 25 }
```

If the number of expressions returning values on the right side exceed the number of labels, the superfluous expressions are ignored. The assignment statement only

consumes the number of values it needs. However, since a function returns all its values as a single expression, if a function invocation returns more values than are needed to complete an assignment statement, it will be detected by the compiler, an error message printed out, and compilation terminated.

Because conditionals are used as selectors of values on the right-hand side of the "=" sign in an assignment statement, assignment instructions are not allowed in the branches of conditional expressions. Furthermore, when assignment statements occur in loops, the right-hand side must contain the control label as part of the assigned expression or the instruction will be treated as if it existed outside the loop.

4.5.1 SINGLE ASSIGNMENT RULE

Assignment can be viewed as a labeling mechanism that transmits information to the compiler identifying a value for data transmission between expressions. It is a means of identifying data paths and dependencies.

The single-assignment rule requires that an identifier have only one value assigned to it in the scope of its name. While an identifier can be assigned to only once in the scope of its name, it may be assigned to in both branches of an "if" statement and in all branches of a "case" statement as only one branch of each will fire at execution time, and the single-assignment rule will not be violated. This makes statements such as x = y redundant. Traditionally, this is done where there is a need to save the value of "x" at some point in time.

As the value is bound to the label for the entire scope of the block, this is unnecessary.

4.6.1 READ

"READ" functions like an assignment statement, except that the values are read in from the file name given in the command. All values read in on a single call must be of the same type (either integer or character).

read (x, y : TYPE , FILENAME)

Up to 20 values of the same type may be read in on any one call, and up to five different files can be read from in any one program.

All filenames must be specified in the header of the main section of the program and can be no longer than 25 alphanumeric characters, with the first an alphabetic character.

MAIN (notrace,file1,file2,file3,file4,file5)

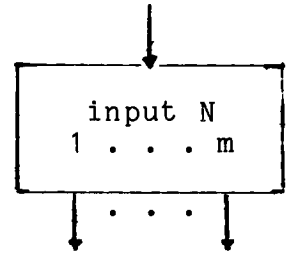
Due to the asynchronous nature of the operation of a data flow computer, there is no guarantee as to the order in which data is read from a file unless the DIVA synchronization mechanism "WHEN" is used to override the concurrency. For more detail see the discussion on synchronization and Figures 22-25, pages 159-161, in Appendix C.

When a "READ" statement occurs in the body of a loop, a copy of the control token is sent to it in order to logically tie its execution to the loop. In all other situations the

compiler generates a program or start constant, which causes the "READ" statement to execute only once.

Figure 13 illustrates the data flow graph generated for the "READ" statement.

DATA FLOW GRAPH FOR READ



where m = the number of values to be read (1 <= m <= 20)
N = the file number (1 <= N <= 5).

Figure 13

4.6.2 WRITE

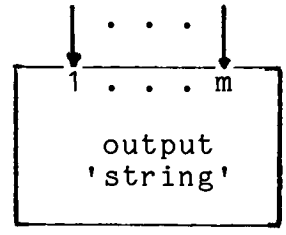
The "WRITE" statement will print out up to 20 values of the same type (integer or character) to the standard output, along with an identifying string of up to 20 characters.

write ("output string" : type : x, y)

At compile time type compatibility of the input tokens is checked and the compile terminated with an error message if they do not match. The "WRITE" statement may appear anywhere in a DIVA program.

See Appendix C, page 141, for examples of usage and Figure 14 below for the data flow graph.

DATA FLOW GRAPH FOR WRITE



where 1 <= m <= 20
string <= 20 characters.

Figure 14

4.7.1 ITERATION EXPRESSIONS

In the data flow simulator and, therefore, the DIVA language, loops are viewed as a special case of a function call where there is only one argument. Only one value can be passed into and circulate through a loop (circulate means the new value is passed to the next iteration of the loop). At this time, loops can only process integer data. Because relational operations only exist for INTEGER data on the simulator, the implementation of the loop control is limited to tests on integer values.

While a function must return at least one value to the calling block, a loop can be used to simply control the number of times a segment of code executes without returning a value. If a loop does return a value, it must appear in an assignment statement.

The general format of a loop is:

```
LOOP
     initialization
DO
     body
[RETURN expression]
ENDLOOP
```

Brackets indicate zero or one occurrences.

The data flow graph generated for this loop is shown in Figure 15. Four operators are required to implement the loop: loop, loop1, do, and do1. "Loop" appears at the beginning of the loop, creates a new context for execution by giving the identification tag (each token carries an identification field that indicates which environment and which instruction it goes to; in a loop it also carries the iteration number) of the input token a unique code block identifier, and sets the iteration number to 1. This token is passed on from output port 2. Output port 1 sends a context control token to "loop1" carrying the old tag value of the input token. "Loop1" restores the old tag value to the data token as it leaves the loop. "Do" increments the iteration count of the token each time it goes around. "Do1" restores the iteration count to one as the token leaves the loop.

DATA FLOW GRAPH FOR A GENERAL LOOP

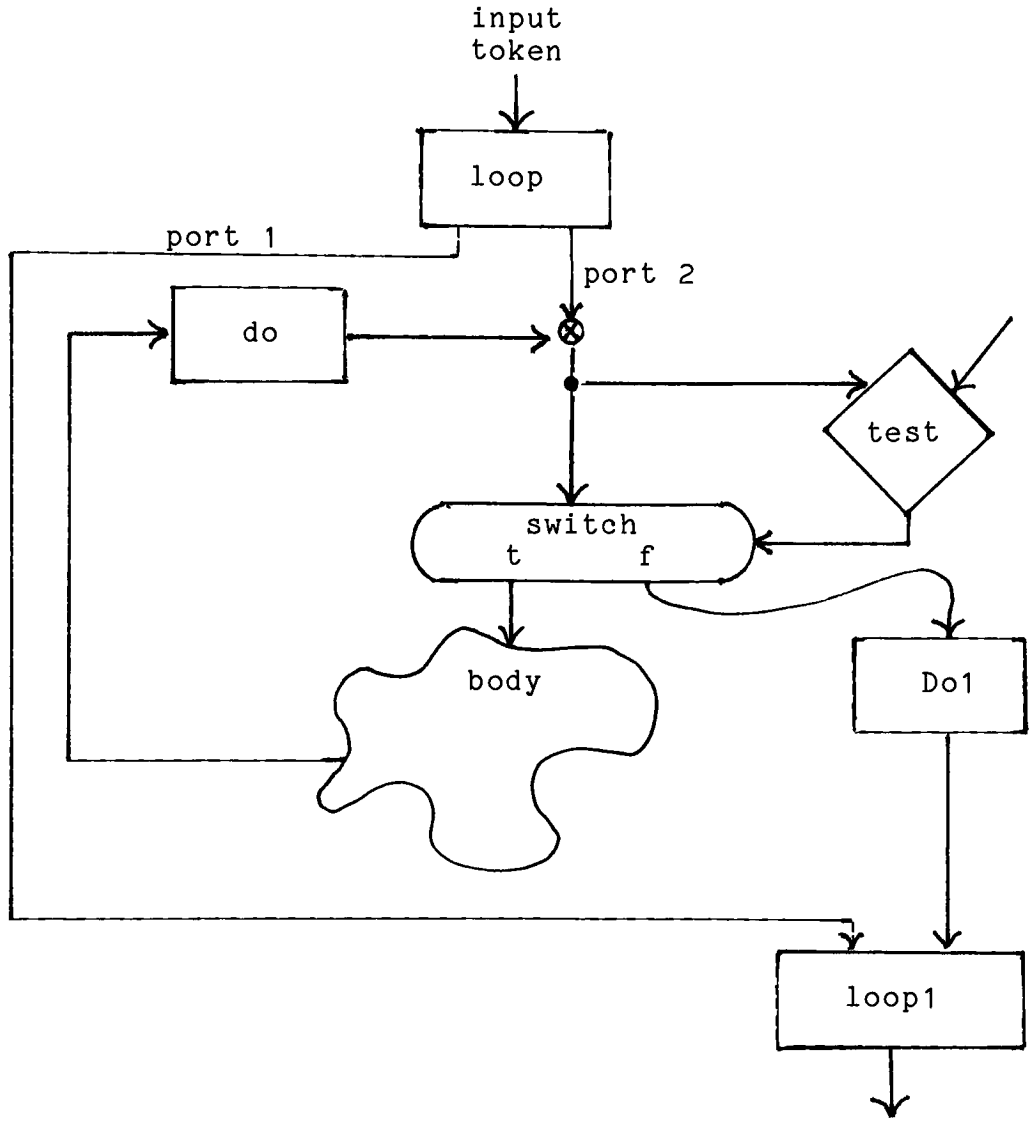


Figure 15

If a loop does not return any values to the outside environment, the exiting control token value is buried in a halt statement. Otherwise, it is routed to those expressions that reference it in the outside environment. The location of the body determines if you have a "WHILE" loop, a "REPEAT-UNTIL" loop or a combination of the two.

4.7.1.1 LOOP FORMAT

There are two variations of the loop construct: the "FOR" loop and the "WHILE" loop. The "FOR" loop is used to control repetition of all code in its body a predetermined number of times as indicated by the control label (which increases each iteration by an amount specified by the programmer). The "WHILE" construct is used to control the number of times certain instructions are repeated based on the value of an integer label as it circulates through the loop iterations. Both may be nested any number of levels.

4.7.1.1.1 FOR LOOP

The "FOR" loop has the format and syntax illustrated in Figure 16.

FOR LOOP FORMAT & SYNTAX

```
LOOP
    FOR x  =  -10 TO 1 BY 2
DO
                haha  =  { x + 12 }
                write ("haha:"  :  int  :  haha)
ENDLOOP
```

or:

```
q = { LOOP
        FOR x  =  -10 TO 1 BY 2
      DO
                haha  =  { x + 12 }
                write ("haha:"  :  int  :  haha)
      RETURN  x
      ENDLOOP }
```

OUTPUT:

```
haha:          2
haha:          4
haha:          6
haha:          8
haha:         10
haha:         12
```

Figure 16

While range values of the control label can be positive or negative integer constants, the incremental change must always be positive.

The first version of the "FOR" in Figure 16 allows the programmer to control the iteration of code without returning any value as the result of the execution of the loop. The second version returns the final value of the loop control label for use in the encompassing environment. This version can be used wherever a value is required for assignment to a label in an assignment expression. Both versions produce the same data-flow graph. (See Figure 17)

DATA FLOW GRAPH FOR FIGURE 16

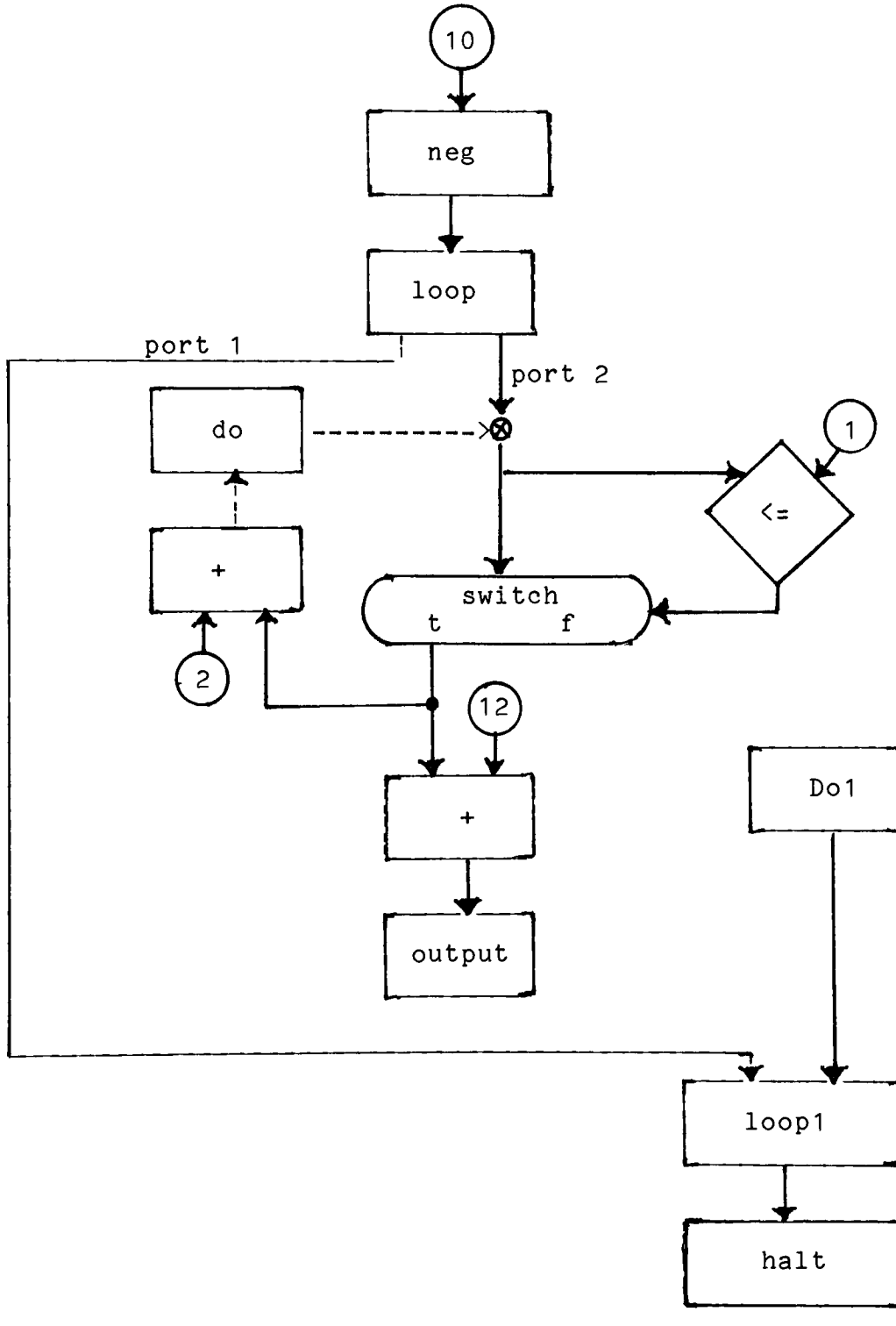


Figure 17

4.7.1.1.2 WHILE LOOP

The "WHILE" construct has the format shown in Figure 18. However, the "WHILE" test, which functions as the exit point for the loop, can be placed anywhere in the body of the loop as long as it precedes the assignment to "NEW" x. The variable placement of the "WHILE" test provides the familiar programming features of the general Ada loop.

WHILE LOOP FORMAT & SYNTAX

```
LOOP
    INITIAL x  =  6;
    DO
          WHILE  x   <=  500
          new x  =  { x * 5 - 20 }
ENDLOOP
```

---

or:

```
q  =  { LOOP
            INITIAL x  =  6;
        DO
             WHILE  x   <=  500
             new x  =  { x * 5 - 20 }
        RETURN x
        ENDLOOP }
```

---

or:

```
q =  { LOOP
            INITIAL x  =  6;
       DO
            write ("x is:"  :  int  :  x)
          WHILE  x   <=  500
          new x  =  { x * 5 - 20 }
       RETURN x
       ENDLOOP }
```

Figure 18

The first two loop examples in Figure 18 generate the same data-flow graph as shown in Figure 19a. The third loop example in Figure 19 generates the data-flow graph as Figure 19b.

DATA FLOW GRAPHS FOR FIGURE 18

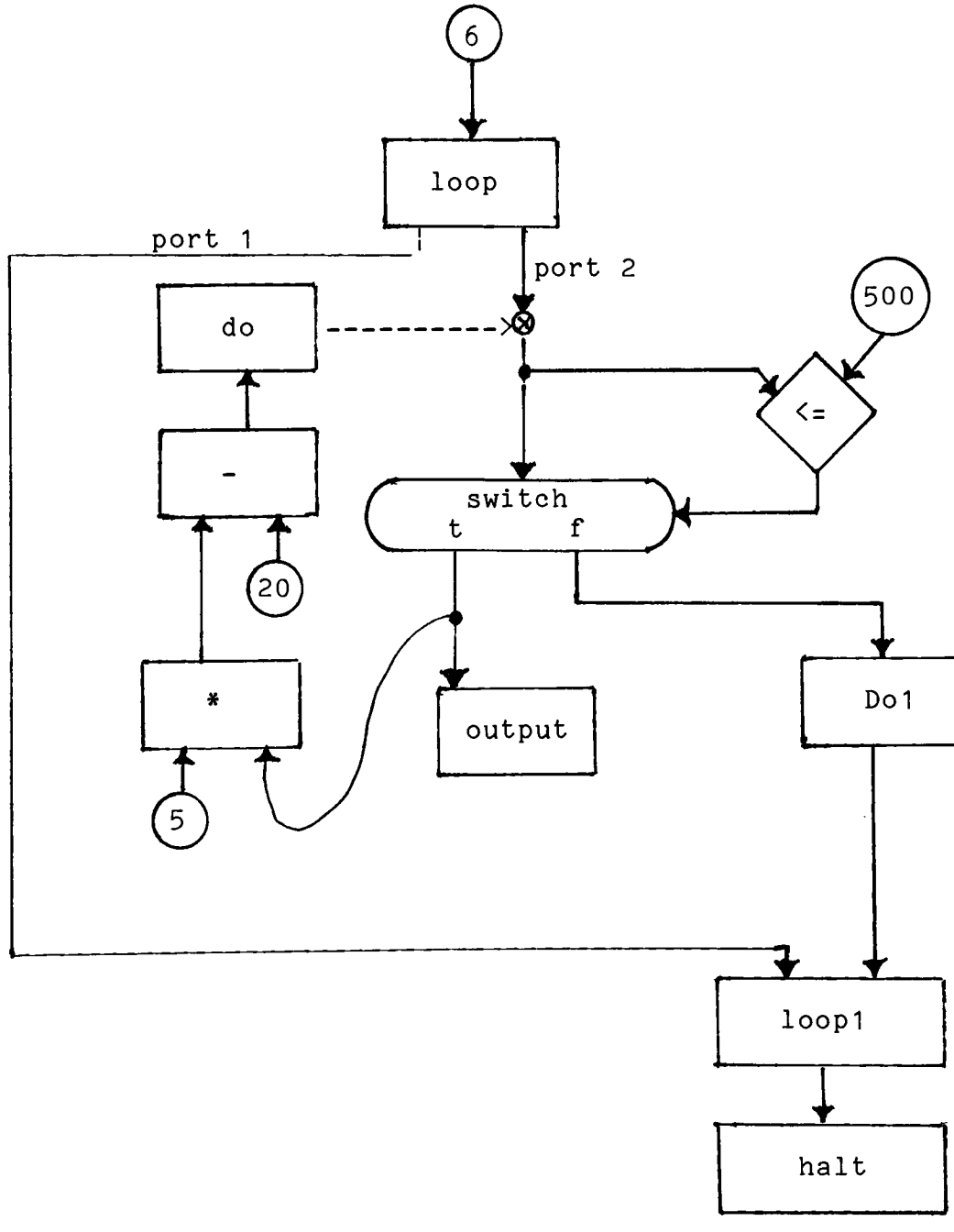


Figure 19a

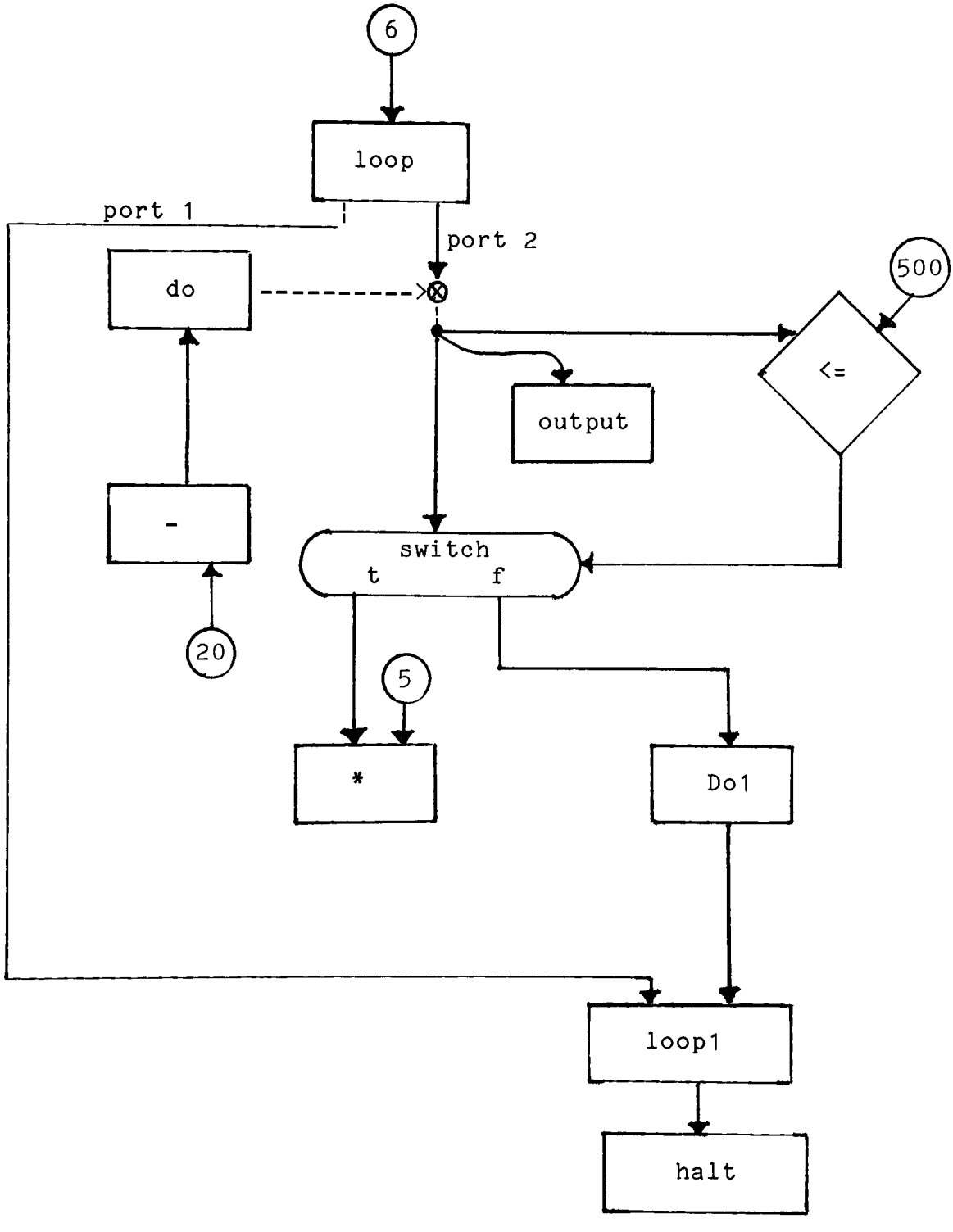


Figure 19b

4.7.1.2 NEW ASSIGNMENT IN LOOPS

The "NEW" assignment instruction can only appear in the "WHILE" loop and is used to calculate the value of the control token for the next iteration. It must be located in the loop body somewhere after the "WHILE" expression; if it does not follow it, an endless loop will be created.

Once the "NEW" assignment has been made, the "NEW" value of the label can be referenced on the right-hand side of the assignment statement in the current iteration by specifying "NEW" label.

```
y = NEW x + 5
```

The "NEW" assignment instruction cannot be used in a "FOR" loop as the new value for the next iteration is already specified in the "FOR" section. If it is used in the "FOR" loop, it produces a run-time error that terminates execution and prints out an error message.

4.7.1.3 RETURN IN LOOPS

"FOR" and "WHILE" loops can return the value of any legitimate integer expression composed of the control label and any number of integer constants.

The following is a legitimate return expression:

```
RETURN x + 5 * 6
```

Any other labels assigned to in the loop cannot be used, because only the control label can be passed back by the loop construct.

4.7.1.4 LABELS IN LOOPS

Loops have been implemented by the simulator as a special case of a function call. They are to be used in situations where there is only one argument and only that argument is manipulated inside the loop. Because of this, only one external value token can be passed in, and only that value can circulate through the iterations of the loop. The value to be passed in must be specified in the "INITIAL" section or "FOR" section. All other tokens in the loop must be program constants, which are always available to their actors, or labels assigned to that include the control token in the expression on the right-hand side. See Figure 16 where "haha" is internalized to the loop by the presence of the control token on the right-hand side.

If an instruction in a loop has all its input operands as integer constants, the compiler recognizes an instruction in this situation as if it exists outside the loop and, therefore, it has no valid identification tag to enable it to fire on repeated iterations of the loop. Any instruction where all the operands are constants fires only once, since one of the operands will be defined as a start constant. Once it has fired there is no recirculating value to trigger this instruction to execute again. See Figure 20.

OUTSIDE CODE INSIDE LOOP

```
loop
   for wong = 1 to 10 by 2
do
          haha = { 2 + 5 }
          write ("haha: " : int : haha)
endloop
```

OUTPUT:

```
    haha:   7
```

Figure 20

The program in Figure 20 generates 2 disjointed data-flow graphs (Figure 21) that can execute concurrently as they do not affect one another.

DATA FLOW GRAPH FOR FIGURE 20

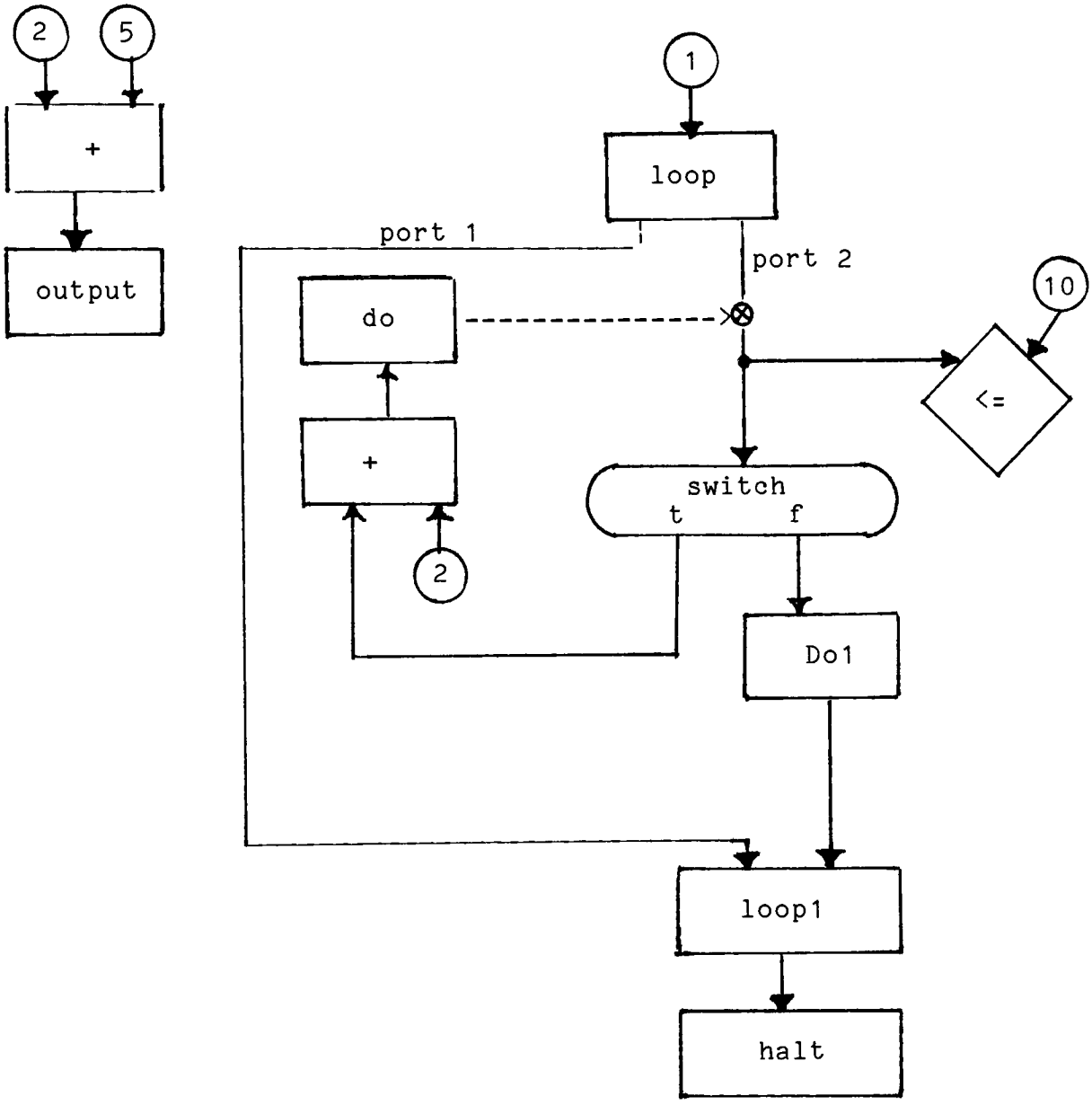


Figure 21

4.7.1.5 SCOPE

The scope of the loop control label is only for the duration of the loop. The final value is returned by the loop to be consumed in a "HALT" instruction (buried with no further use) or assigned to a label as a result of an assignment statement. The loop construct redeclares the identifier that is used as the loop control label in the "INITIAL" section, or "FOR" section, for the life of the loop.

All labels assigned to in the body of the loop must be declared outside the loop. Except for the loop control label, labels assigned to in the body of the loop cannot be used on the left-hand side of the "=" sign again either outside or inside the loop as it would violate the single-assignment rule.

Any reference to a label assigned to in the body of a loop is considered to be logically included in the loop body regardless of the code's physical location. The loop in Figure 22 produces the same output as the loop in Figure 16, and therefore, the same data-flow graph. Because the scope of the loop control label is limited to the loop block, all references to the loop control label must be physically inside the loop.

LOOP CODE OUTSIDE OF LOOP

```
LOOP
    FOR x  =  -10 TO 1 BY 2 DO
                haha  =  { x + 12 }
ENDLOOP

write ("haha:"  :  int  :  haha)
```

Figure 22

To avoid confusion and chaos it is recommended that all instructions performed logically in the loop be organized physically in the loop as well.

If an instruction references a token value generated inside a loop as well as a token value generated outside a loop, the instruction will never fire as all tokens to an instruction must carry the same identification tag. In this case the tokens are identified as going to two different instances of the same instruction in different environments and neither one will receive a second token to enable it to fire. The last "WRITE" instruction in Figure 23 references tokens generated from two different environments, which results in a compile-time error.

REFERENCE OUT OF LABEL'S SCOPE

```
MAIN (notrace)
     INT : x, y, pen, surely, haha;
     pen = 18;
BEGIN
     y = { 7 * pen }

     surely = { LOOP
                     FOR x = -10 TO 1 BY 2
                DO
                     haha = { x + 12 }
                     write ("haha:" : int : haha)
                RETURN x + 12
                ENDLOOP }

     WRITE ("surely is: " : INT : surely)
     WRITE ("y, x: " : INT : y, x)
END
```

Figure 23

4.7.2 CONDITIONAL EXPRESSIONS

The conditional construct can be used in two different ways:

1. as a selector of value(s)

```
if p(q) then value a

if p(q) then value a else value b
```

2. as a guard to decide whether an instruction or group of instructions or expressions should execute.

```
if p(q) then body1

if p(q) then body1 else body2
```

When the conditional is used as a selector of value(s), it must appear in an assignment statement. Once you have

entered a conditional expression in this situation, the assignment instruction is not allowed in the code body of any of the branches of the conditional expressions. If you try to execute an assignment statement in the branch of a conditional expression, you will get an error message at compile time.

When the conditional is used as a guard, assignment statements may be included in the branch bodies. See Figure 25.

There are three conditional constructs:

```
if-then
if-then-else
case
```

4.7.2.1 IF-THEN

Formally, the syntax of the "IF-THEN" is:

```
IF  (test)  THEN
     body
ENDIF
```

The conditional test must be an expression that evaluates to a boolean value of true or false. It may be a simple or compound expression. Some allowable expressions are:

```
! gate
! gate  &&  moon  >  1
a  <  b
x  >  15
x + 6  ==  y * 3
q  >=  5  ||  q  <=  0
```

The data flow graph is shown in Figure 24.

IF-THEN DATA FLOW GRAPH

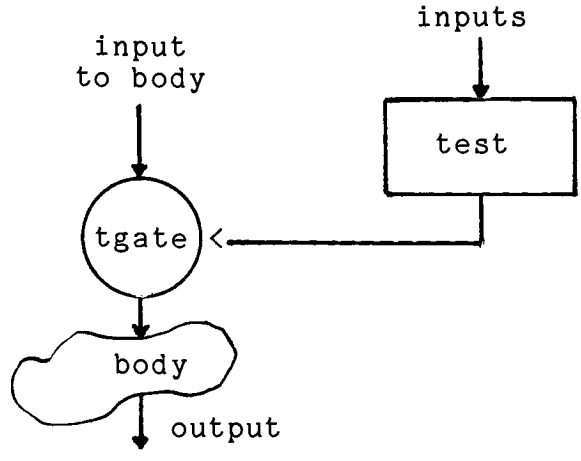


Figure 24

Figure 25 illustrates both uses of the conditional expression.

CONDITIONALS AS VALUE SELECTORS & CONTROL FLOW

As a selector of value:

```
MAIN (notrace)
     INT: zab, zoo;
     INT: xxx, yak = 6, 2;
BEGIN
     zab = { IF  yak  <  xxx  THEN
                  - (xxx/yak)
             ENDIF }

     zoo =  { zab + 3 }
     WRITE ("result is:" : INT : zoo)
END
```

As a guard:

```
MAIN (notrace)
     INT: zab, zoo;
     INT: xxx, yak = 6, 2;
BEGIN
     IF  (yak  <  xxx)  THEN
          zab = { - (xxx/yak) }
     ENDIF

     zoo = { zab + 3 }
     WRITE ("result is:" : INT : zoo)
END
```

Both programs give the following output:

```
    result is:      0
```

Figure 25

Both programs in Figure 25 generate the same data-flow graph (Figure 26).

DATA FLOW GRAPH FOR FIGURE 25

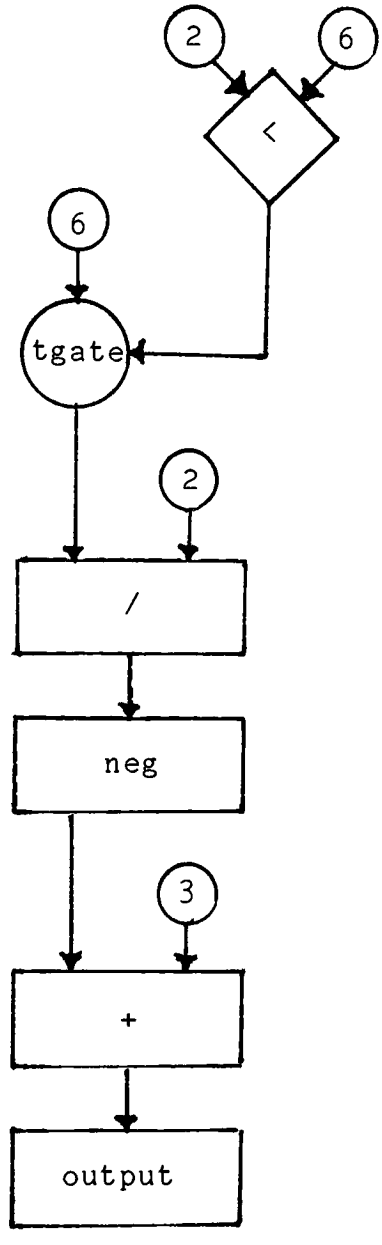


Figure 26

4.7.2.2 IF-THEN-ELSE

Formally the syntax of the "IF-THEN-ELSE" is:

```
IF  (TEST)  THEN
      body1
ELSE
      body2
ENDIF
```

The test can be any legitimate boolean or relational expression. This construct outputs a boolean value that is either true or false. Both branches of the expression must return the same number of values when it is used as a selector of value(s).

For the data flow graph that is generated for this, see Figure 27.

IF-THEN-ELSE DATA FLOW GRAPH

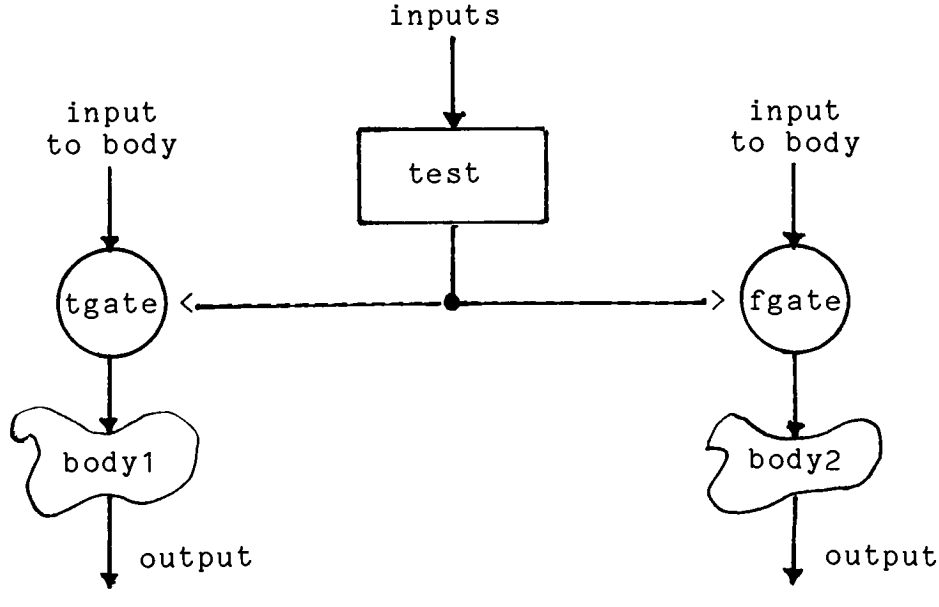


Figure 27

Sample programs are illustrated in Figure 28.

IF-THEN-ELSE SAMPLE PROGRAMS

```
MAIN (notrace)
     INT: zab, zoo;
     INT: xxx, yak = 6, 2;
BEGIN
     zab = { IF  xxx  <  yak  THEN
                - (xxx/yak)
             ELSE
                7
             ENDIF }

     zoo = { zab + 3 }
     WRITE ("result is:" : INT : zoo)
END
```

```
MAIN (notrace)
     INT: zab, zoo;
     INT: xxx, yak = 6, 2;
BEGIN
     IF  (xxx  <  yak)  THEN
          zab = { - (xxx/yak) }
     ELSE
          zab = { 7 }
     ENDIF

     zoo = { zab + 3 }
     WRITE ("result is:" : INT : zoo)
END
```

Both programs give the following output:

```
    result is:      10
```

Figure 28

Both programs in Figure 28 generate the same data-flow graph (Figure 29).

DATA FLOW GRAPH FOR FIGURE 28

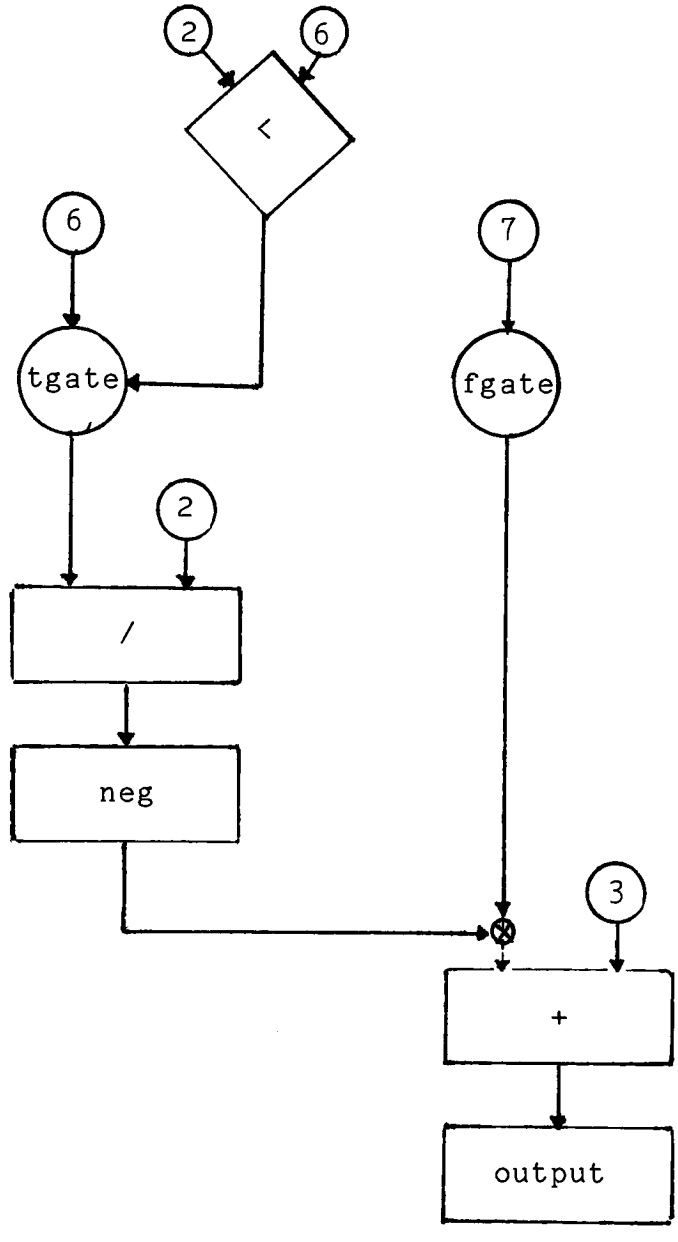


Figure 29

4.7.2.3 CASE

The "CASE" expression allow for multi-way decision making by testing for a match to any one of a number of values. The following example illustrates the general syntax:

```
CASE  expression  OF
      guard1:   body1
                ENDBRANCH
      guard2:   body2
                ENDBRANCH
            .
            .
            .
      guardn:   bodyn
                ENDBRANCH
ENDCASE
```

The language allows any legitimate integer expression in the branch guards; therefore, the programmer must be careful not to create two guards that evaluate to the same value. If two guards evaluate to the same value and that value tests true, a run-time error is generated as the destination(s) receives more than one token.

All guards can evaluate concurrently as they fire as soon as all their input tokens are present (see the data flow graph in Figure 30), but only one body can fire. Again, each branch must return the same number of tokens.

Figure 31 contains two source program examples illustrating the different contexts in which the conditional can be used.

DATA FLOW GRAPH FOR CASE EXPRESSION

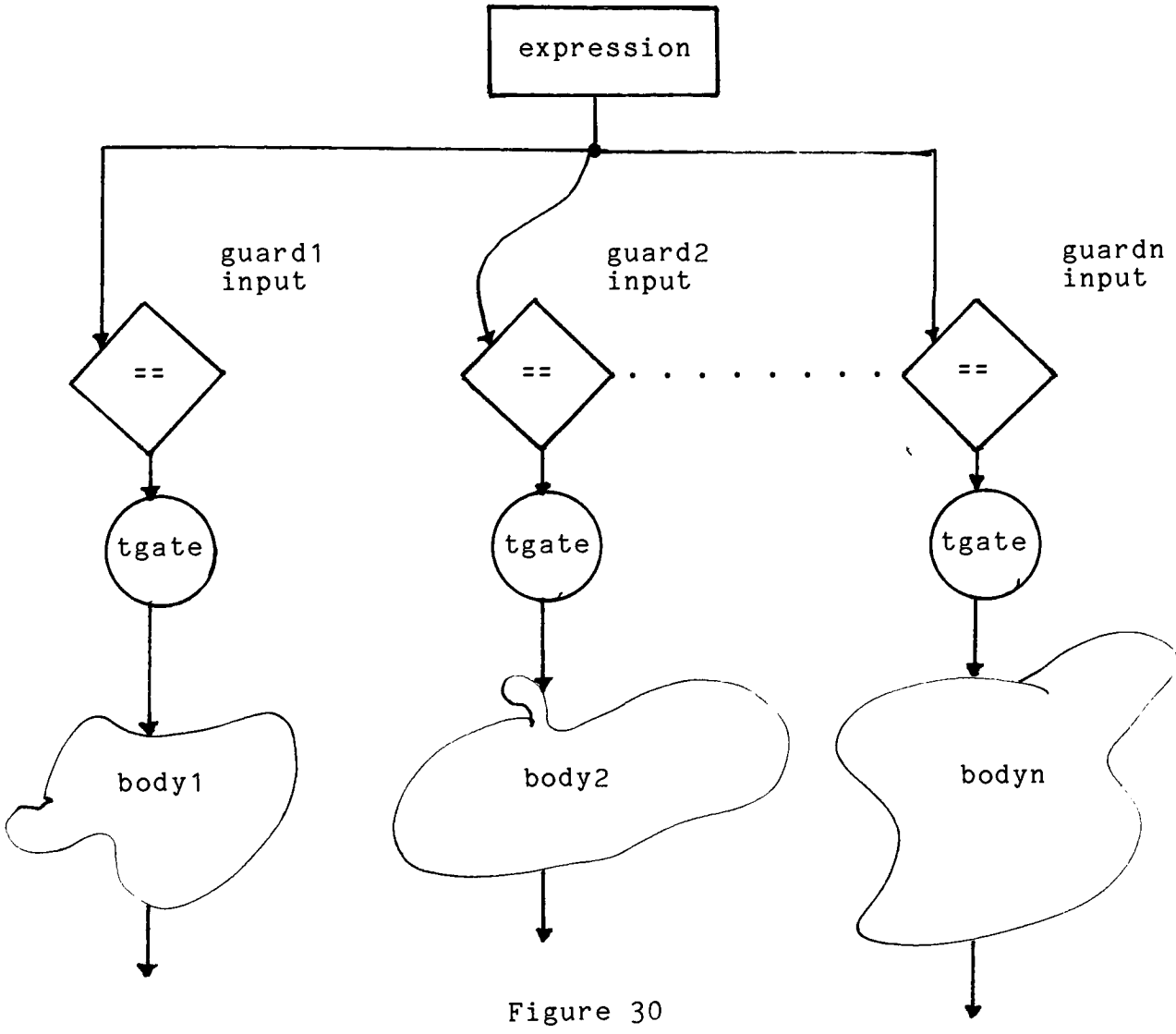


Figure 30

CASE AS VALUE SELECTOR & CONTROL FLOW

(As a selector of value)

```
MAIN (notrace)
     INT: x, y, z;
     INT: a = 5;
BEGIN
     z = { 55 }
     y = { 5 }
     x = { CASE  y  OF
               10:    170
                      ENDBRANCH
                3:    y + a
                      ENDBRANCH
                a:    IF (z < 10) THEN
                           17
                      ELSE
                           y - 3
                      ENDIF
                      ENDBRANCH
          ENDCASE }
     WRITE ("case result: " : INT : x)
END
```

(As a guard)

```
MAIN (notrace)
     INT: x, y, z;
     INT: a = 5;
BEGIN
     z = { 55 }
     y = { 5 }
     CASE  y  OF
          10:    x = { 170 }
                 ENDBRANCH
           3:    x = { y + a }
                 ENDBRANCH
           a:    x = { IF (z < 10) THEN
                           17
                      ELSE
                           y - 3
                      ENDIF }
                 ENDBRANCH
     ENDCASE
     WRITE ("case result: " : INT : x)
END
```

Both programs produce the following output:

```
  case result:     2
```

Figure 31

Both programs in Figure 31 generate the same data-flow graph (Figure 32).

DATA FLOW GRAPH FOR FIGURE 31

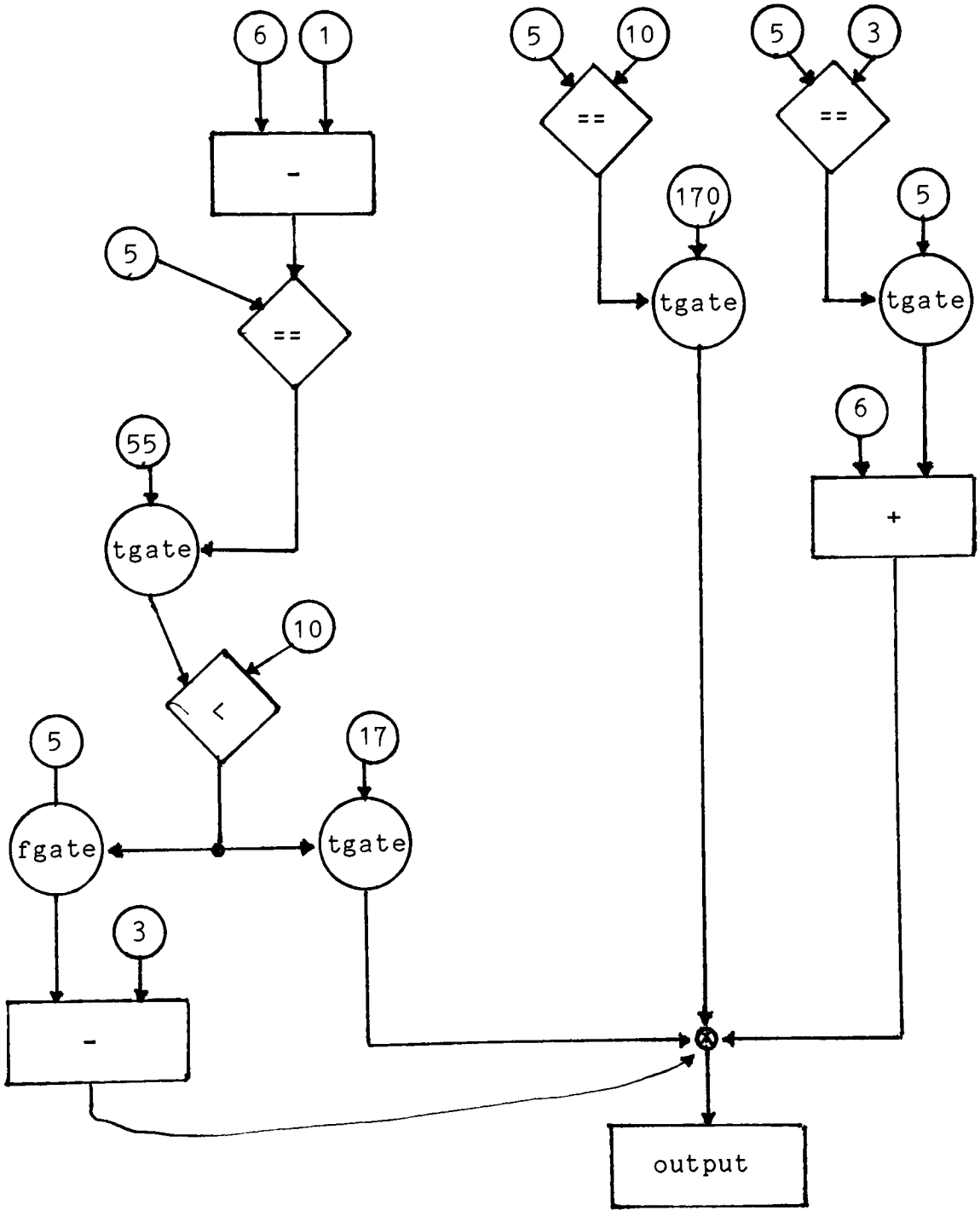


Figure 32

4.7.2.4 NESTING AND SINGLE ASSIGNMENT

Allowing conditionals to nest implies that violations of the single assignment will have to be detected at run time. There is no way to keep track of legitimate repeat assignments to the same label because of the ability to assign to multiple labels in one assignment statement. Also, values can be assigned with a mix of allowable expressions within any of the conditional branches. See Figure 33.

NESTED IFs IN ASSIGNMENT

```
x , y =  { if (test) then
               10,
               if (test) then
                  4
               else
                  6
               endif
           else
               if (test) then
                  1
               else
                  zoo
               endif,
               9
           endif }
```

Figure 33

4.7.2.5 LOOPS IN CONDITIONALS

A selector conditional can have a loop in one of its branches. If the loop doesn't return a value, it must come after all the values have been returned to the labels being assigned to.

```
surely = { if  y  <  8  then
                99,
                loop
                for  x = -10 to 1 by 2 do
                      haha = { x + 12 }
                      write ("haha:" : int : haha)
                endloop
           else
                101
           endif }
```

4.7.3 LET

The "LET" construct is used to temporarily expand the current block. It provides the ability to declare new identifiers for the life of the "LET" block. All identifiers not introduced in the "LET" block are inherited from the outside block. If an identifier name is defined outside the "LET" and also introduced in the "LET" block, then it is viewed as a new identifier inside the "LET".

The general form is:

```
LET
     declarations
ENDINIT
     assignments
IN
     body
ENDLET
```

The identifiers introduced in the "LET" block cannot be seen or accessed from the outside; upon completion of the "LET" block these identifiers cease to exist.

A "LET" block is formed as follows:

1. The word "LET" is followed by a declaration section introducing new identifiers and defining values for them if needed:

```
LET
     INT: x, y;
     x = 5;
ENDINIT
```

The word "ENDINIT" indicates the conclusion of this section.

2. "ENDINIT" is optionally followed by a section that permits assignment of values to inherited identifiers or newly introduced identifiers that have not previously been assigned to:

```
ENDINIT
     z =  { x * 4 }
     y =  { -4 }
```

3. The assignment section is followed by the word "IN" which introduces the evaluation section that returns one or more values as a result of its execution. If there is more than one expression returning a value, they are separated by commas. The end of the section terminates the "LET" block and is indicated by the word "ENDLET".

The following example returns a single value:

```
IN
     z * x - y % q
ENDLET
```

The following example returns two values:

```
IN
     z * x - y,
     IF ( z * x)  >  10  THEN
          y % q
     ELSE
          5
     ENDIF
ENDLET
```

If the "LET" expression returns a value or values, it must appear on the right-hand side of an assignment statement. There is a simple program example using the "LET" on the next page in Figure 34 and more complicated program examples in Figures 18-21 in Appendix C, pages 155-158.

LET PROGRAM EXAMPLE

```
main (notrace)

begin
        let
                int: x, y =  5, 8;
                int: z;
        endinit
        in
                z =  { x + y }
                write ("result is:" : int : z)
        endlet
end
```

Figure 34

4.7.4 FUNCTION APPLICATIONS AND DEFINITIONS

A program is composed of a collection of functions (if any) followed by the main block. Functions provide a means of temporarily expanding the environment.

Functions have access only to formal parameters and locally defined labels; there is no access to globally defined identifiers. Parameters are call-by-value and the function returns result(s). Functions can be recursive and can call any other function that has been previously defined.

```
FUNCTION name (argument list, return token types)
     declarations
BEGINFUN
     body
ENDFUN
```

In the data flow graph generated for a function, the first statement is the "begin" operator, with all input para-

meters to the subprogram arriving at ports 1 to n. The begin statement then sends the tokens to the appropriate operators in the function. All resulting output values from the function are directed to the end operator, from which point they are assumed to be directed back to the calling program. (Figure 35)

FUNCTION DATA FLOW GRAPH

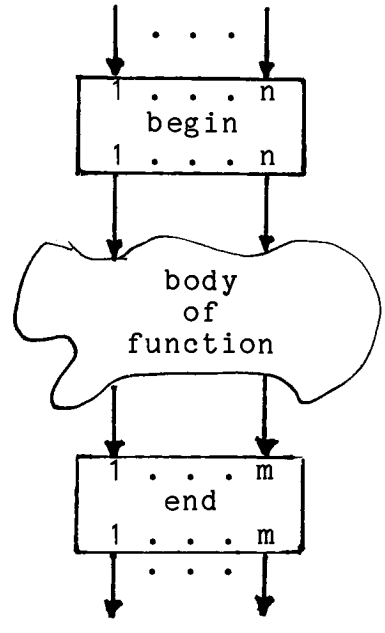


Figure 35

4.7.4.1 FUNCTION DEFINITIONS

A function is defined as follows:

1. The word "FUNCTION" followed by a header containing the name, number and type of arguments, and the number and type of values returned by the function:

```
function  factorial (int: i ; return int)
```

Factorial has one integer argument passed in from the calling block and returns one integer value when it terminates.

If there is more than one value passed in, the labels are separated by commas. If there is more than one type of value passed in, the definitions are separated by semicolons:

```
int: x , y ; boolean: gate ;
```

There must be at least one argument (input token) to a function in order to trigger its execution, and there is a maximum limit of 19.

If there is more than one value returned by a function, each one is listed by type and separated by a comma:

```
return  int , int , boolean
```

All arguments are call-by-value. Any manipulation of the tokens passed in is local to the function. All tokens passed in to a function must be used. If any of the argument tokens are not used, at run-time the program will run, produce no output, and conclude execution without any indication that there is a problem.

2. The declaration section specifies any additional labels needed for computation within the function block. See Section 4.3.3 for a more detailed discussion on declarations. The life of these identifiers and the tokens passed in terminate with the function.

3. "BEGINFUN" then signals the start of the body of the function definition. All DIVA expressions and statements are allowed in the function body. This includes invocations of previously defined functions as well as the one being defined.

4. The last statement in the body of the function is the "return" statement. It contains expressions indicating the values to be returned by the function. A function must return at least one value and no more than 19. If there is more than one value to be returned, they are separated from each other by commas:

```
return  x , y
```

5. "ENDFUN" signals the end of the function definition. See Figures 11 - 17, pages 147-154 in Appendix C for complete function definition illustrations.

4.7.4.2 FUNCTION APPLICATIONS

A function application is composed of the function name followed by a list of arguments (which must be expressions) within parentheses:

```
factorial (x)
```

in the header of the function definition. If they are not the same, it is detected at compile time, an error message is printed out, and compilation is terminated.

If there is more than one argument to a function, each one is separated by a comma:

```
two (a , b)
```

In general, a function can be applied anywhere value(s) are expected: on the right-hand side of the assignment statement or as an input token in an expression or statement.

```
x , y  =  { two (a , b) }
```

When the function application occurs on the right-hand side of the assignment statement, there must be sufficient identifiers to label the output tokens returned by the function.

A function that returns only one value may appear anywhere a single value is expected:

```
write ("message" : int : factorial (i))

i * factorial (i - 1)

if x  >  0 then
    somefunc (x + 1)
endif
```

A function can be an argument to another function:

```
two (a , factorial (i))
```

4.7.4.3 RESTRICTIONS AND SCOPE

At least one of the output tokens must be referenced in the program or a segmentation fault occurs at run time.

While functions can be recursive, the requirement that a function be defined before it is referenced precludes mutual recursion. In the program in Figure 14, page 150, in Appendix C, the scoping of function invocations is as follows:

in body of	can call
two	two
ano	ano, two
main	ano, two

A function does not inherit any token values from the calling block; therefore, only the actual parameters and locally defined identifiers are accessible to the function.

4.8.1 IMPLICIT SYNCHRONIZATION

All expressions and functions behave on a pass-by-value basis with no effect on any other expression or function currently ready to execute. If two operations do not depend on results of each other, they can execute concurrently. [VAL--Description and Analysis McGraw]. All operations follow the rule that all input values to the expression must be present in order to be able to execute. Therein lies the implicit identification of concurrency and synchronization.

IMPLICIT SYNCHRONIZATION WITHOUT CONCURRENCY

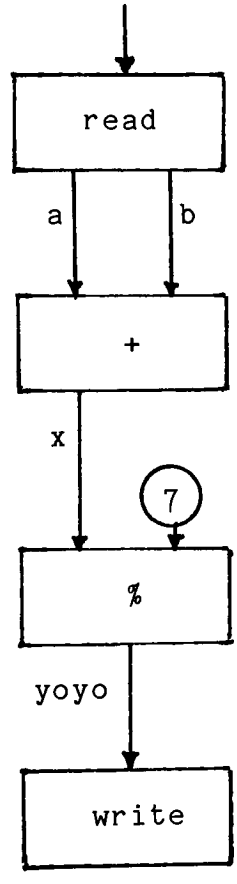


Figure 36

If line 4 were changed to:

yoyo = { x + 5 % 7 }

it now has two component operations identified by the parentheses (according to the rules of precedence) as follows:

yoyo = { x + (5 % 7) }

Let D temporarily represent 5 % 7 and we have an additional operation:

4. yoyo = { x + D }

5. D = { 5 % 7 }

If an operation is dependent on another, it will be missing an input value until after the execution of the other operation and thus not be able to execute until after the other.

```
1.  write ("result is:" : int : yoyo)
2.  x = { b + a }
3.  read (b, a : int, filename)
4.  yoyo = { x % 7 }
```

A directed graph (Figure 36) helps to trace the data dependencies between the operations and expose the order of execution. This program block would execute in a linear fashion as each operation is dependent on the outcome of another. The order of execution would be: 3, 2, 4, 1.

Now the order of execution would be: 3 and 5, 2, 4, 1. (See Figure 37 below.)

IMPLICIT SYNCHRONIZATION WITH CONCURRENCY

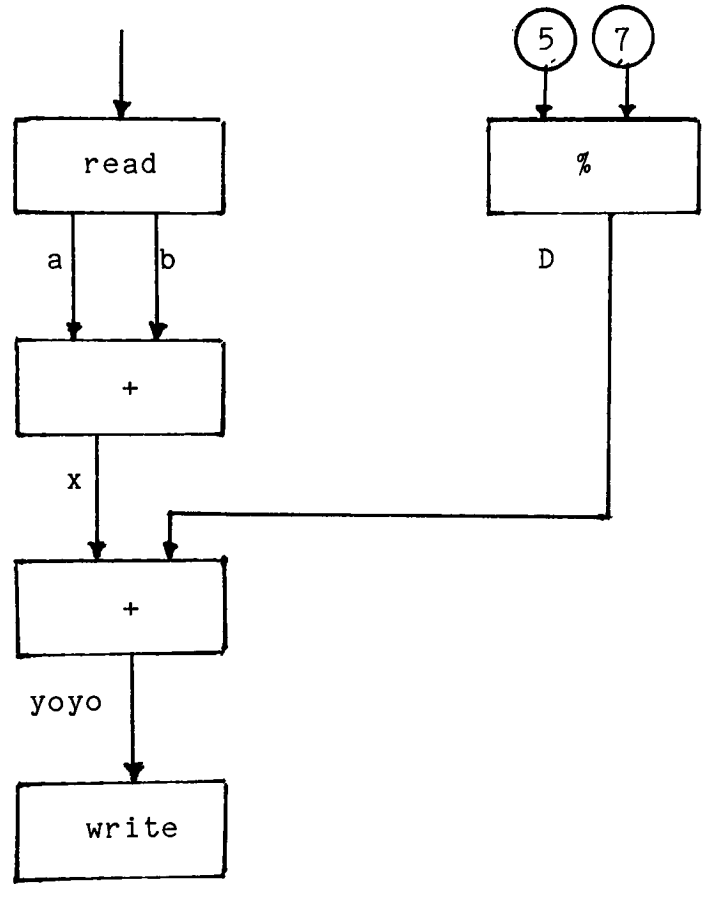


Figure 37

4.8.2 EXPLICIT SYNCHRONIZATION

To synchronize execution of unrelated code (in the sense that the output of one expression is not an input to the other), data dependencies must be created in the expressions concerned. The "WHEN" construct synchronizes the order of

execution of events by creating data dependencies that do not naturally exist.

The signal that an event has occurred in data flow can only be identified by the arrival of a token:

```
when  x  do
         read (y : int , filename)
         z  =  { y % 2 }
endwhen

x  =  { - (a * 50 % 3) }
```

This construct prevents evaluation of all code enclosed between the "do" and "endwhen" until a token arrives on "x".

This can be used as a means to determine the order in which data is read from a file if there is more than one "read" statement specifying that file. Otherwise, there is no guarantee as to the order in which data will be read in.

See Figures 22-25 in Appendix C, pages 159-161, for complete program examples and related output.

4.9 RUNNING A DIVA PROGRAM

DIVA is a compiled language. Source programs must pass through a compiler and an assembler to produce machine language that will run on the data flow simulator.

To run a program use the following command sequence:

```
% diva programname

% assem programname.d

% dfsim < programname.m datafile1 datafile2 . . .
```

Program names can be up to 17 characters long. The compiler (diva) takes the source code file (program name) and

produces a mnemonic representation of a directed data flow graph. It puts this mnemonic code in a file with the same name as the source file but with a ".d" appended to it. For example:

source file: prime
mnemonic file: prime.d

The new file is then processed by the assembler to produce the machine language code that will run on the simulator. The assembler puts this code in a file that is created by taking the name of the mnemonic code file, removing the ".d", and appending a ".m" to it. For example:

mnemonic file: prime.d
machine code: prime.m

The machine code file and any program argument files are listed after the simulator run command. The argument files are file names which are listed after the file name which contains the data flow program machine code. There can be up to five (5).

4.10 DIVA BNF

```
program ::=  { subprog }  main

main    ::=  MAIN  ( bug  [ , FILENAME { , FILENAME } ]  )
                 { decl }
             BEGIN
                 opers
                 { opers }
             END

bug     ::=  TRACE  |  NOTRACE

decl    ::=  vars = list ;  |  type : vars = list ;  |  type :
             vars ;

vars    ::=  LABEL  { , LABEL }

list    ::=  INTEGER  { , INTEGER }

type    ::=  INT  |  CHAR  |  BOOLEAN

subprog ::=
             FUNCTION NAME ( inparam  RETURN  type  { ,type } )
                  { decl }
             BEGINFUN
                  opers
                  { opers }
                  RETURN  exp6  { , exp6 }
             ENDFUN

inparam ::=  type : vars ;  { type : vars ; }

opers   ::=  exp6  {  exp6 }  |  asgn  |  io  |  specexp
             |  WHEN  primary  DO  opers  { opers }  ENDWHEN

asgn    ::=  LABEL  { , LABEL } = '{' exp6  { , exp6 } '}'
             |  LABEL = exp6

io      ::=  READ ( vars : type , FILENAME )  |
             WRITE ( " [ MESSAGE ] " : type : exp6 { , exp6 } )

specexp ::=  cond  |  iterate  |  casexp  |  letin

exp6    ::=  exp7  |  ! exp7

exp7    ::=  exp8  |  exp7 '||' exp8  |  exp7 '&&' exp8

exp8    ::=  exp9  |  exp8 < exp9  |  exp8 > exp9
             |  exp8 == exp9  |  exp8 != exp9
             |  exp8 <= exp9  |  exp8 >= exp9

exp9    ::=  exp10  |  exp9 + exp10  |  exp9 - exp10

exp10   ::=  exp11  |  exp10 * exp11  |  exp10 / exp11
```

```
              |  exp10 % exp11

exp11    ::=  primary  |  - primary

primary  ::=  INTEGER  |  BOOLEAN  |  LABEL  |  '|' exp6 '|'
              |  ( exp6 )  |  invoke  |  **NEW LABEL

invoke   ::=  LABEL  ( exp6  { , exp6 } )

iterate  ::=  LOOP
                  mid
              [  RETURN  exp6  { , exp6 }  ]
               ENDLOOP

mid      ::=  FOR  LABEL = exp9  TO  exp9  BY  exp9
                  DO  body1  { body1 }

              |  INITIAL  LABEL = exp9 ;
                    DO body2  { body2 }

body1    ::=  WRITE ( " [ MESSAGE ] " : type : exp6 { , exp6 } )
              |  asgn  |  specexp  |  WHEN  primary  DO  opers2
              {  opers2  }  ENDWHEN

body2*   ::=  body1  |  NEW LABEL = '{' exp6 '}'  |  WHILE exp6
```

When the conditional expressions or the **LET** expression appear in an assignment statement the following is their BNF:

```
cond     ::=  IF exp6 THEN opers2  { opers2 }   [ ELSE opers2
              { opers2 } ]  ENDIF

casexp   ::=  CASE  exp9  OF
                  exp9  :  opers2  { opers2 }  ENDBRANCH
                  { exp9  :  opers2  { opers2 }  ENDBRANCH }
              ENDCASE

letin    ::=  LET decl ENDINIT { asgn } IN opers2 { opers2 }
              ENDLET

opers2   ::=  exp6  { , exp6 }  |  specexp  |  io  |  WHEN
              primary  DO  opers2  { opers2 }  ENDWHEN
```

When the same expressions do not appear in an assignment statement, the following is their BNF:

```
cond     ::=  IF exp6 THEN opers  { opers }   [ ELSE opers
              { opers } ]  ENDIF

casexp   ::=  CASE  exp9  OF
                  exp9  :  opers  { opers }  ENDBRANCH
                  { exp9  :  opers  { opers }  ENDBRANCH }
              ENDCASE
```

```
letin   ::=  LET decl ENDINIT { asgn } IN opers { opers }
             ENDLET
```

*The WHILE expression can appear anywhere in the loop; however, the NEW LABEL assignment must appear after the WHILE expres-sion or it will create an endless loop situation at execution time. Both of these expressions appear only once in the body of the loop.

**This can only be used in loops when you want to reference the new value of the recurrence variable on the right-hand side of an assignment statement.

The brackets [] mean zero or one occurrence of the symbol(s) enclosed. The {} mean zero or more occurrences of the symbol(s) enclosed.

The '{' and '}' indicate that the characters enclosed are symbols of the language and not bnf occurrence representations.

5.0 CONCLUSION

5.1 LANGUAGE SUMMARY AND COMPARISON

DIVA includes many of the major programming tools found in traditional languages:

- conditionals
- loops
- functions
- assignment (single)
- arithmetic operators
- relational operators
- boolean operators
- recursion
- I/O operations
- simple data types

In addition, there is a unique feature, the "let-in" expression, that temporarily expands the current environment.

When programming in DIVA, the order of the instructions is irrelevant as flow of control is sequenced by the availability of data. Thus, maximum concurrency is obtained from any algorithm at all levels.

The current development of DIVA allows a reasonable range of programming applications to allow programmers to learn and experiment with the basic concepts of data flow. There are, however, some limitations imposed by the implementation of the simulator. Due to the fact that the simulator was written in Concurrent Euclid, reals are not implemented and integers have a maximum value of 32,767, which limits the range of program examples.

Like the two languages it is based on, VAL and ID, DIVA is a block Structured, single-assignment functional language. There is no updating of memory, no aliasing, and no sequential program control flow by a program counter.

While DIVA is intended to be a data-flow language based on VAL and ID, it does depart from them for the following reasons:

1. DIVA includes I/O operations.
2. DIVA has some different usage rules for some of the expressions.
3. Due to the implementation of the simulated data-flow computer DIVA is targeted for, only a subset of the data types and expressions could be implemented.
4. The simulator only allows integer constants.

Limitations due to the simulator also include:

1. no arrays or records, only simple data types
2. loops can only recirculate one value
3. language only has integer constants
4. relational expressions can only be performed on integer data

DIVA has the simple data types of VAL and ID (character, boolean, integer) but with fewer operations that can be performed on them. The absence of character constants makes it impossible to do anything other than a simple read and write of character data.

In VAL and ID if the arity of an expression is one, it may appear in an arithmetic expression as an operand. In DIVA

this is true only of the function expression, integer constants, and variable labels. The conditional, iteration, and "let-in" expressions can only return values to variables in an assignment statement.

DIVA also departs from VAL and ID by allowing these expressions to function as control flow statements without returning any values.

The "let-in" expression provided by both VAL and DIVA, which introduces one or more value names that can be used within the scope of the expression, is unnecessary in ID as there is no variable typing. Value names can be introduced at any time simply by assigning values to them.

ID has gone beyond VAL and the other applicative languages to include the capability to handle resource problems. It has included the concept of a resource and mechanisms for synchronization in order to write operating systems. DIVA has incorporated a version of the "when" construct found in ID to synchronize the order of execution of events.

VAL was designed with numerical computation as the primary application area and the hope that it would evolve into a general purpose language. Therefore, while it has included exception handling for all data types, it does not have facilities for resource management.

The comparison of the three languages can be summarized by the following table.

COMPARISON OF DIVA, VAL, AND ID

	DIVA	VAL	ID
Data Types			
Boolean	yes	yes	yes
Integer	yes	yes	yes
Character	yes	yes	yes
Real	no	yes	yes
Null	no	yes	yes
Error	no	yes	yes
Array	no	yes	yes
Record	no	yes	yes
Union	no	yes	no
String	no	no	yes
Programmer defined	no	yes	yes
Expressions			
Subroutines	yes	yes	yes
Conditionals			
If-then	yes	no	no
If-then-else	yes	yes	yes
Case	yes	only for unions	no
Iteration			
For	yes	yes	yes
Forall	no	yes	no
While	yes	no	yes
Let-In Expression	yes	yes	no
I/O operations	yes	no	no
Other Features			
Constants	integer only	yes	yes
Variable Typing	yes	yes	no
Type Conversion	no	no	yes
Recursion	yes	no	yes

Figure 38

5.2 RECOMMENDATIONS

Future development of this project includes plans to rewrite the simulator in a more portable language that will eliminate the difficulties encountered; and at the same time expand the simulator, and DIVA, to handle arrays and allow the loop construct to recirculate more than one value. With the addition of arrays, more mathematical features (e.g. vector operations) as well as character manipulation can be included in the language.

Because of multiple assignment and the allowable range of expressions on the right-hand side of the "=" sign in an assignment statement, YACC could not distinguish the left side from the right side without special treatment. Braces were used to enclose the right-hand side to solve this problem.

By moving the error checking that is done when the code is parsed to the final stage when the tuples are printed out, the complexity of the I/O statements could be reduced. It would not be necessary to indicate the data type of the parameters; simply check the type of the first parameter and make sure the rest are of the same type. Also, by changing the order of the components inside the parenthesis, commas could be used rather than colons to separate the major components thus reducing the complexity.

The "trace" option should be removed at this time also as it has no useful purpose at this level. The diagnostics generated by the simulator with this option are helpful to the assembly language programmer or to someone working on the compiler or assembler program.

Continuing research is being conducted to develop data flow languages into general purpose languages to run on future general purpose data flow computers. Future versions of VAL will include file update and input/output facilities with streams of values as a principle means of communicating between program modules. ID is working to integrate an error recovery model into the language and a possible solution to most of the security and protection problems found in the literature.

More work, also, needs to be done to make data flow languages available as teaching tools in computer science curriculums.

APPENDIX A

VAL SCOPING FOR FUNCTIONS

```
function F ( <header> )
external FF ( <header> );
type T = <type-spec>;
    function G ( <header> )
    type U = <type-spec>;
        function M ( <header> )
            function N ( <header> )

                body_N

                endfun

            body_M

            endfun

        body_G

        endfun
        function H ( <header> )
            function P ( <header> )

            BODY_P

            endfun

        BODY_H

        endfun

BODY_F

endfun
```

the body of	may invoke functions
F	FF (external), G, H (internal)
G	FF (external), M (internal)
M	FF (external), N (internal)
N	FF (external)
H	FF (external), P (internal)
P	FF (external)

BNF Description of DFL

<program> ::= program (input <inputlist>) yield <outputlist>;
{ <proceduredef> } <expression> end.

<proceduredef> ::= define<name> := procedure (input<inputlist>)
yield <outputlist>; { <proceduredef> }
<expression> end;

<inputlist> ::= <typedeclaration> { , <typedeclaration> }

<typedeclaration> ::= <name> { , <name> } : <type>

<outputlist> ::= <type> { , <type> }

<namelist> ::= <name> { , <name> }

<expression> ::= [<primitive expression> | <letblock expression> | <conditional expression> | <for expression> | <forall expression> | <application expression>] { , <expression> }

<letblock expression> ::= let <inputlist> := <restricted expression> {;<inputlist> := <restricted expression> } in <expression> end

<restricted expression> ::= [<primitive expression> | <application expression>] { , primitive expression> | <application expression> }

<conditional expression> ::= if <primitive expression> then <expression> else <expression> end

<for expression> ::= for <inputlist> := <expression> do if <primitive expression> then [[<iter expression> else <expression>] | [<expression> else <iter expression>]] end

<iter expression> ::= iter <namelist>:= <restricted expression> {; <namelist>:= <restricted expression> } end

<forall expression> ::= forall <name> in [<range>] <name> := <expression> construct <name> end

<application expression> ::= <name> (<expression>)

<primitive expression> ::= <simple expression> <relop> <simple expression> | <simple expression>

<simple expression> ::= <term> | <sign> <term> | <term> <addop> <simple expression>

```
<term> ::= <factor> | <factor> <mulop> <term>

<factor> ::= <name> | <const> | ( <primitive expression> ) |
             <name> [<expression>] | <application expression> |
             not <factor>

<sign> ::= -

<addop> ::= + | - | or

<mulop> ::= * | / | and

<relop> ::= < | > | = | <> | <= | >=

<type> ::=   <standard type> | array [<range> {, <range> } ] of
             <standard type>

<range> ::= <const>...<const>

<standard type> ::= integer | real | boolean | char

<const> ::= <integer const> | <real const>

<integer const> ::= <digit> { <digit> }

<real const> ::= <basic real const> | <basic real const> E
                 <integer const> | <basic real const> E <sign>
                 <integer const>

<basic real const> ::= <integer const> . . <integer const> | .
                       . <integer const>

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<name> ::= <letter> { <letter> | <digit> }

<letter> ::= A | B | C | . . . | X | Y | Z
```

{X} means zero or more occurrences of X
[X] means exactly one occurrence of X

DFL PROGRAM AND EXPRESSION EXAMPLES

The following procedure using a conditional expression returns the factorial of an integer:

```
define FACTORIAL := procedure (input n: integer) yield integer;

   define PRODUCT:=procedure(input n1,n2: integer)yield integer;
       if n2 <= n1 then
           n1
       else
           let MIDDLE: integer := (n1 + n2) / 2
           in
               PRODUCT (n1, MIDDLE) * PRODUCT (MIDDLE + 1, n2)
           end
       end
   end;
   if n < 0 then ERROR else PRODUCT (1, n)
end;
```

Example of the "for" expression which computes the nth Fibonacci number:

```
      for COUNT, FIB, PREFIB: integer := 0,0,1 do
          if COUNT <= n then
              iter PREFIB, FIB := FIB, PREFIB + FIB;
                   COUNT := COUNT + 1
              end
          else
              FIB
          end
```

Example of "forall" expression:

```
            forall I in [1 . . 10]
            MEAN: real := (a[I] + b[I] + c[I]) / 3.0
                 construct MEAN
                 end
```

DAISY PROGRAM EXAMPLES

Factorial program:

```
FACTORIAL:N = if:<eq?: <N 0> 1
     mpy: <N FACTORIAL:dcr:N>>
```

MEMBER specifies that the second element in the argument structure should be a nonempty list, and that the first of that list is to be implicitly bound to LA and the rest of that list to LD.

```
MEMBER:(A (LA ! LD)) = if:< same?:<A LA>  @true
     empty?:LD     []
     MEMBER:<A LD>     >
```

APPENDIX B

ASSEMBLER SYMBOL TABLES

Subprogram Addresses--the table is represented by an array of records with the following structure:

name	address of begin
factorial	12

Subprogram names can be up to 9 characters long. There can be up to 200 subprograms specified.

Program and Start Constants--there are 2 counters, one for the program constants and one for the start constants. These keep count of the number of unique constants in each category stored in their respective tables.

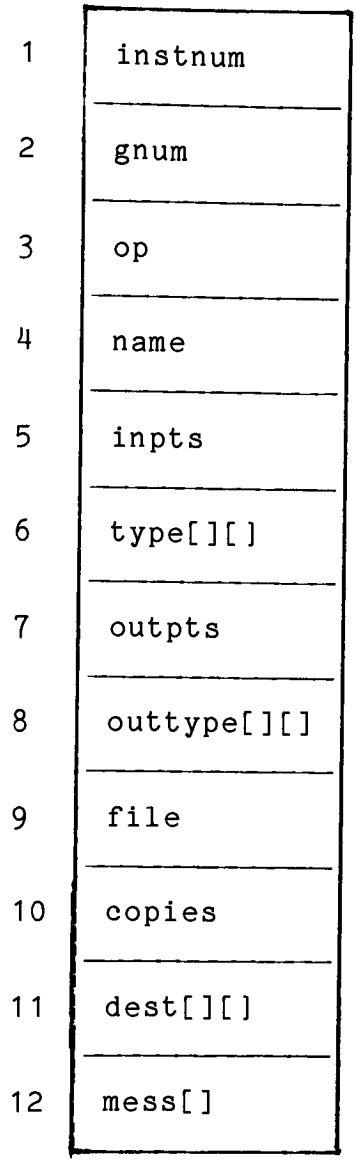
Each table is represented by an array of records with the following structure:

value
number of copies
array of numbers

instr #	arc	instr #	arc	. . .

There can be up to 200 unique program constants and 200 unique start constants.

INST RECORD



1. instnum--the instruction number of the command.

2. gnum--the instruction's group number.

3. op--the textual name of the command.

4. name--if the instruction read in is an "apply" or "subp" command, the textual name of the function is stored in this field.

5. inpts--the number of input tokens for the "input", "output", and "apply" commands.

6. type--for groups 2-4, the type for each input.

7. outpts--the number of ports producing output for commands generating output.

8. outtyp--the type for each output produced by a command.

9. file--an integer between 1-5 indicating the file to be read from by an "input" command.

10. copies--the total number of output tokens dispatched by a command.

11. dest--stores the output port, destination instruction number, and the destination port for each output copy.

12. mess--stores the message of the "output" command.

OPCODES AND DATA TYPES

LIST OF ALL OPCODES

abs
neg
input
not
halt
output
loop
loop1
do
do1
plus
minus
mul
div
mod
and
or
tgate
fgate
switch
less
leql
gre
geql
eql
neql
begin
end
subp (name of subprogram)
apply (name of subprogram)
stop
fin

DATA TYPES

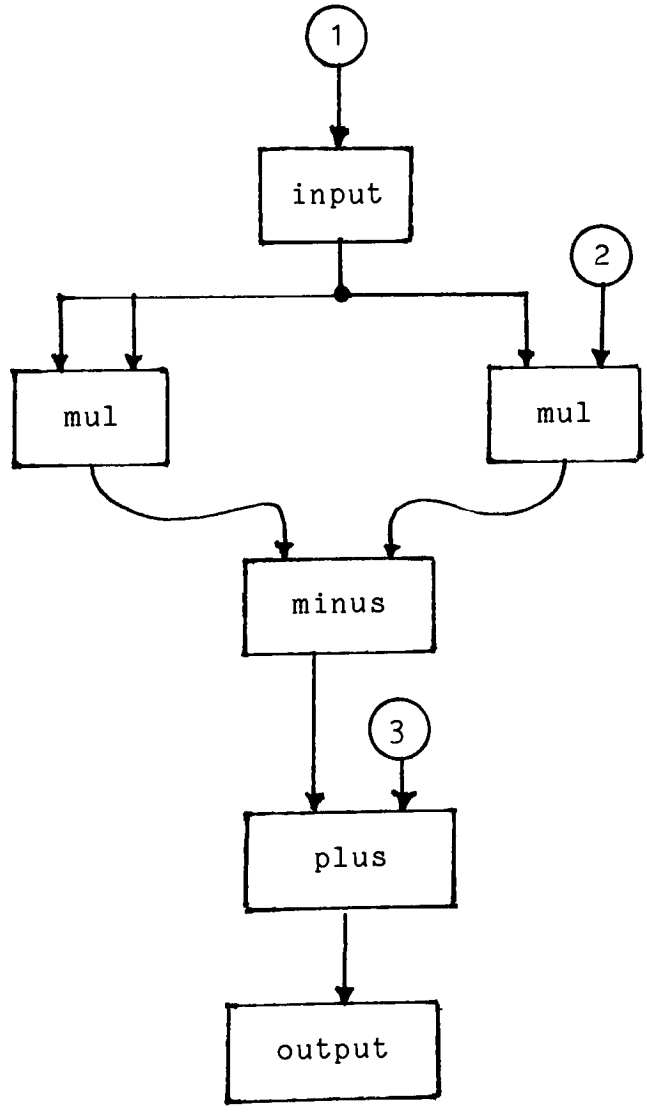
Data Type	Mneumonic
no data	ndata
boolean	bool
integer	int
character	char
instruction address	inadd
pointer	ptr
context control	cc

The following program computes the polynomial $x^2 - 2x + 3$ for any value of x read in from the specified file.

Source File

```
main (notrace,input)
        int: x;
begin
        read (x : int, input)
        write ("result is: " : int : (x * x) - 2 * x + 3)
end
```

Data-Flow Graph



Mnemonic Code File--Compiler Output

```
notrace
1  input  1  1  int  3
1  2  1
1  2  2
1  3  2
2  mul  1
1  4  1
3  mul  1
1  4  2
4  minus  1
1  5  1
5  plus  1
1  6  1
6  output  'result is: '  1  int
7  stop
2
2  1   3  1
3  1   5  2
1
1  1   1  1
8 fin
```

Machine Code File--Assembler Output

```
notrace
6
1 3 0  1     1     2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 000   3
  1  2     2    1
  1  2     2    2
  1  2     3    2
2  13  0  2        2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 00   1
  1  2     4    1
3  13  0  2         2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1
  1  2     4    2
4  12  0  2         2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1
  1  2     5    1
5  11  0  2         2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1
  1  2     6    1
6 6 0 1 'result is:' 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0
2
2  1  3  1
3  1  5  2
1
1  1  1  1
```

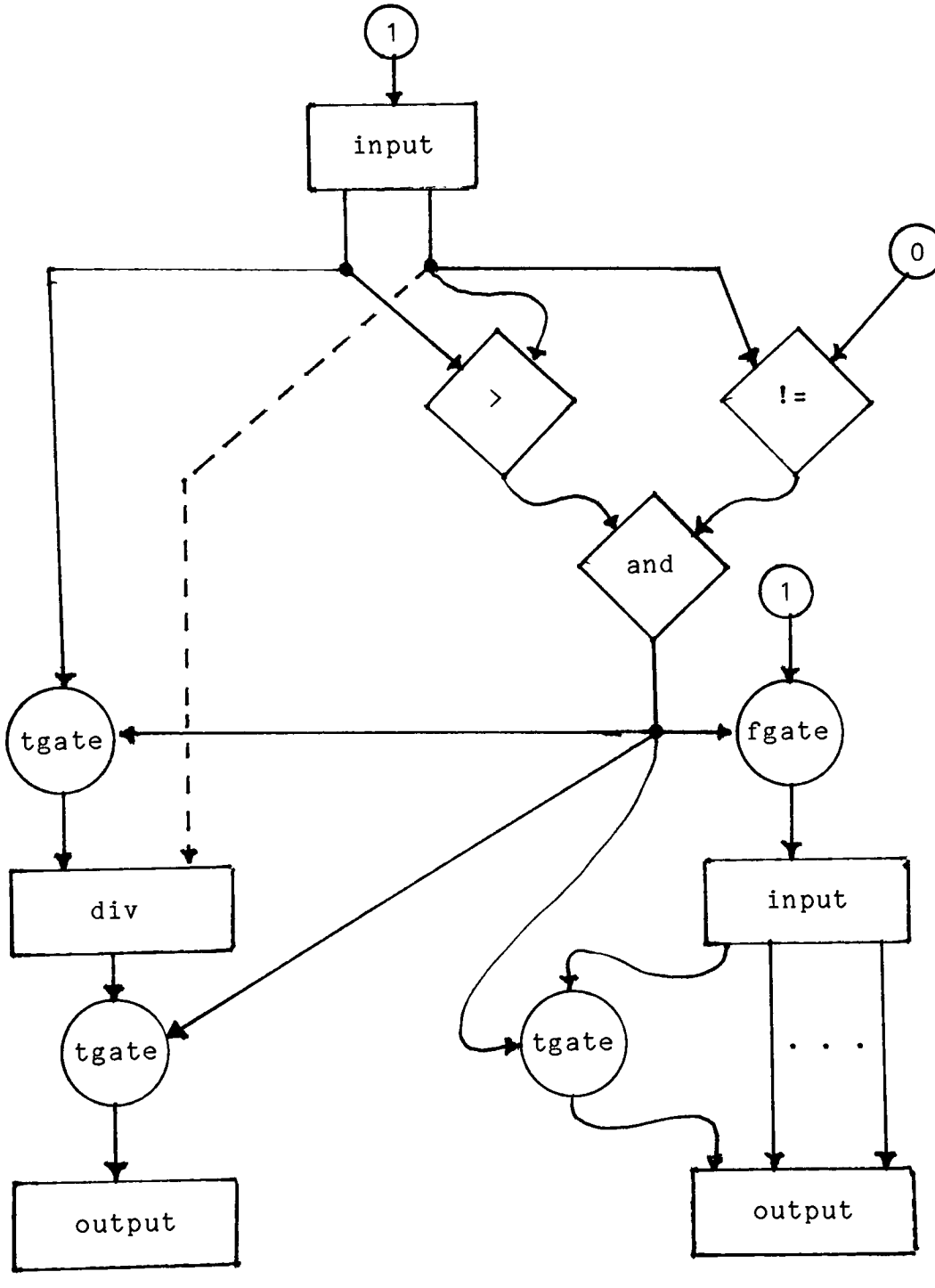
The following program reads in two numbers from the first file specified (input); if the first number is greater than the second number and the divisor is not 0, the first number is divided by the second and the result is printed out. If either test fails, an error message (illegal divisor) is read in from the second file specified (input2) and printed out.

Source File

```
main (notrace,input,input2)
   int: x, y;
   char: a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12,
         a13, a14, a15;
begin
   read (x, y : int, input)

   if (x > y) && (y != 0) then
      write ("result is: " : int : x / y)
   else
      read(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15
           : char, input2)
      write("" : char : a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,
            a13,a14,a15)
   endif
end
```

Data-Flow Graph



Mnemonic Code File--Compiler Output

```
notrace
1  input  1  2  int  5
1  2  1
1  5  1
2  2  2
2  3  1
2  6  2
2  gre  1
1  4  1
3  neql  1
1  4  2
4  and  4
1  5  2
1  7  2
1  9  2
1  11  2
5  tgate  int    1
1  6  1
6  div  1
1  7  1
7  tgate  int    1
1  8  1
8  output  'result is: '  1  int
9  fgate  int  1
1  10  1
10  input  2  15  char  15
1  11  1
2  12  2
3  12  3
4  12  4
5  12  5
6  12  6
7  12  7
8  12  8
9  12  9
10  12  10
11  12  11
12  12  12
13  12  13
14  12  14
15  12  15
11  fgate  char    1
1  12  1
12  output  ''  15  char
13  stop
2
0  1   3  2
1  1   9  1
1
1  1   1  1
14 fin
```

Machine Code File--Assembler Output

```
notrace
12
1   3   0   1      1    2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    5
   1   2      2    1
   1   2      5    1
   2   2      2    2
   2   2      3    1
   2   2      6    2
2   23   0   2          2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   1      4    1
3   26   0   2          2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   1      4    2
4   16   0   2          1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    4
   1   1      5    2
   1   1      7    2
   1   1      9    2
   1   1     11    2
5   18   0   2          2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   2      6    1
6   14   0   2          2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   2      7    1
7   18   0   2          2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   2      8    1
86  0 1 'result is:' 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    0
9   19   0   2          2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   2     10    1
10   3   0   1      2    3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0   15
   1   3     11    1
   2   3     12    2
   3   3     12    3
   4   3     12    4
   5   3     12    5
   6   3     12    6
   7   3     12    7
   8   3     12    8
   9   3     12    9
  10   3     12   10
  11   3     12   11
  12   3     12   12
  13   3     12   13
  14   3     12   14
  15   3     12   15
11   19   0   2          3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0    1
   1   3     12    1
12   6   0   15     '' 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0    0
2
0   1   3   2
1   1   9   1
1
1   1   1   1
```

MACHINE LANGUAGE STATEMENTS

instr #	opcode	Enab. Cnt CC	Enab. Cnt data	File nmbr or print string	input types	nmbr of dests

File nmbr or print string applies to input and output statements

DESTINATIONS OF OUTPUTS

output port	type of data	instr. nmbr.	input port

PROGRAM AND START CONSTANTS

value of constant	nmbr of destinations	instruction	arc

MNEUMONIC CODES & MACHINE LANGUAGE INTERPRETATIONS

operator	opcode	Enab. cnt CC	Enab. cnt data
abs	1	0	1
neg	2	0	1
input	3	0	1
not	4	0	1
halt	5	0	1
output	6	0	i
L	7	0	1
L1	8	1	1
D	9	0	1
D1	10	0	1
+	11	0	2
-	12	0	2
*	13	0	2
/	14	0	2
mod	15	0	2
and	16	0	2
or	17	0	2
Tgate	18	0	2
Fgate	19	0	2
switch	20	0	2
if <	21	0	2
if <=	22	0	2
if >	23	0	2
if >=	24	0	2
if =	25	0	2
if /=	26	0	2
begin	27	0	1
end	28	1	m
activate	29	0	k
terminate	30	0	1

Data Type	Code
no data	0
boolean	1
integer	2
character	3
instruction address	4
pointer	5
context control	7

i = number of values to output
k = number of parameters going to subprogram
m = number of parameters returned from subprogram

APPENDIX C

```
main (notrace)
        int: x, y;
begin
        x = { 3 + -5 * 4 }
        y = { | x | }
        write ("y = " : int : y)
end
```

```
Program Output:       y =         17
```

(Figure 1)

```
main (notrace,fill)
        int: x;
begin
        read(x: int, fill)
        write ("result is" : int : ((x * x) - (2 * x) + x))
end
```

```
Program Output:       result is           12
```

(Figure 2)

```
main (notrace,file)
        int: x, y;
        int: c, d, z;
begin
        d, c = { 5, 10 }
        z = { if  c < 10  then
                read (x, y : int, file)
                d
            endif }
        write ("result is:" : int : x, z)
        write ("result is:" : int : y)
        write ("result is:" : int : d)
end
```

```
Program Output:       result is:          5
```

(Figure 3)

IF, CASE, AND LOOP EXAMPLES

```
main (notrace)
        int: x, y;
        int: a = 5;
begin
        y = { 5 }
        x = { case  y  of
                10:     17
                        endbranch
                 3:     y + a
                        endbranch
                 a:     y - 3
                        endbranch
              endcase }
        write ("case result: " : int : x)
end


Program Output:       case result:        2
```

(Figure 4)

```
main (notrace)
        int: zab, zoo;
        int: xxx, yak = 6, 2;
begin
        zab =  { if yak < xxx then
                        - (xxx / yak)
                 else
                        7
                 endif }
        zoo = { zab + 3 }
        write ("result is:" : int : zoo)
end


Program Output:       result is:          0
```

(Figure 5)

```
main (notrace)
        int: cat, dog;
begin
        dog = { loop
                    initial cat = 1;
              do
                    while cat <= 10
                         new cat = { cat + 1 }
              return cat
              endloop }
        write ("result is:" : int : dog)
end
```

```
Program Output:     result is:     11
```

(Figure 6)

```
main (notrace,goat)
        int: bang, bong, ding, dong;
begin
        bang = { 2 }
        bong = { loop
                    for ding = bang  to  20  by  2
                 do
                    dong = { if ding < 10 then
                                  5
                             else
                                  7
                             endif }
                    write ("dong: " : int : dong)
                 return ding + 1
                 endloop }
        write ("bong: " : int : bong)
end
```

```
Program Output:      dong:          5d
                     ong:          5
                     dong:          5
                     dong:          5
                     dong:          7
                     dong:          7
                     dong:          7
                     dong:          7
                     dong:          7
                     dong:          7
                     bong:         23
```

***Output reflects the asynchronous nature of dataflow concurrency.**

(Figure 7)

```
main (notrace,goat)
        int: bang, bong, ding, dong;
begin
        bang = { 2 }
        bong = { if bang < 10 then
                        loop
                            for ding = bang  to  20  by  2
                        do
                            write ("ding: " : int : ding)
                        return ding
                        endloop
                    else
                        7
                    endif }

        write ("bong: " : int : bong)
end
```

```
Program Output:       ding:                2
                      ding:                4
                      ding:                6
                      ding:                8
                      ding:               10
                      ding:               12
                      ding:               14
                      ding:               16
                      ding:               18
                      ding:               20
                      ding:               22
```

(Figure 8)

```
main (notrace)
     int: x, y;
     int: a, wong;
     int: z = 8;
begin
     y = { 5 }
     a = { 3 }
     x = { case y of
                10:   17
                      endbranch
                 3:   y + a
                      endbranch
           a + 2:   if  (z < 8)  then
                           y - 3
                      else
                           loop
                                for wong = 1  to  19  by  2
                           do
                                write ("wong: " : int : wong)
                           return wong
                           endloop
                      endif
                      endbranch
          endcase }
     write ("case result: " : int : x)
end
```

```
Program Output:      wong:              1
                     wong:              3
                     wong:              5
                     wong:              7
                     wong:              9
                     wong:             11
                     wong:             13
                     wong:             15
                     wong:             17
                     wong:             19
                     case result:              21
```

(Figure 9)

```
main (notrace)
   int: x, y;
   int: a, wong;
   int: z = 8;
begin
   y = { 5 }
   a = { 3 }
   x = { case y of

            10:  17
                 endbranch
             3:  y + a
                 endbranch
         a + 2:  if  (z < 8)  then
                     y - 3
                 else
                     loop
                        initial wong = 6;
                     do
                        write ("contol wong: " : int : wong)
                        while wong <= 500
                        new wong = { wong * 5 - 20 }
                        write ("new wong: " :int : new wong)
                     return wong
                     endloop
                 endif
                 endbranch
       endcase }

   write ("case result: " : int : x)
end
```

```
Program  Output:  contol  wong:       6
                  contol nweong: w          10wong:        10

                  nceonwtol wo wongng: :          30        30

                  new wong:         130contol wong:        130

                  new wong:         630contol wong:        630

                  case result:        630
```

*Output illustrates the asynchronous nature of dataflow concurrency.

(Figure 10)

PROGRAMS WITH FUNCTION CALLS

The following program computes the factorial values for the integer numbers 1 through 7 and prints them to the standard output.

```
function factorial (int: i; return int)
     int: x;
beginfun
     x = { if  (i == 0)  then
               i + 1
           else
               i * factorial (i - 1)
           endif }
     return x
endfun


main (notrace)
    int: i;
begin
    loop
        initial i = 1;
    do
        while i <= 7
           write ("loop# factorial:" : int : i, factorial (i))
           new i = { i + 1 }
   endloop
end
```

Program Output:

```
loop# factorial:        1          1
loop# factorial:        2          2
loop# factorial:        3          6
loop# factorial:        4         24
loop# factorial:        5        120
loop# factorial:        6        720
loop# factorial:        7       5040
```

(Figure 11)

This program finds the greatest common divisor for the two numbers passed in the function call and prints it to the standard output.

```
function euclid (int: x, y; return int)
        int: good;
beginfun

        good = { if  (x == y)  then
                        y
                 else
                        if  (x > y)  then
                            euclid (y,x - y)
                        else
                            euclid (x,y - x)
                        endif
                 endif }
        return good
endfun


main (notrace)
        int: ans, ina, inb;
begin
        ina, inb = { 55, 35 }
        write ("divisor: " : int : ans)
        ans = { euclid (ina, inb) }
end
```

Program Output: divisor 5

(Figure 12)

This version of the common divisor program illustrates the nonsequential nature of program flow. Instructions execute as soon as all their tokens arrive. As can be seen below, the output did not occur in the same order as the instructions were listed.

```
function euclid (int: x, y; return int)
        int: good;
beginfun

        good = { if (x == y) then
                        y
                 else
                        if (x > y) then
                            euclid (y, x - y)
                        else
                            euclid (x, y - x)
                        endif
                 endif }
        return good
endfun


main (notrace)
        int: ans, ina, inb;
        int: sugar;
begin
        ina, inb = { 55, 35 }
        write ("divisor: " : int : ans)
        ans = { euclid (ina, inb) }

        write ("sugar: " : int : sugar)
        sugar = { euclid (33,66) }
end
```

```
Program Output:      sugar:          33
                     divisor:         5
```

(Figure 13)

```
function two (int: a, b; return int, int)
     int: x, y;
beginfun
     x = { if  (a < b)  then
                b
           endif }

     y = { a + b % x }
     return x, y
endfun


function ano(int: ab; return int, int)
    int: x, y, wambatt;
beginfun
    y = { loop
              initial x = 5 + ab;
          do
              while x < 15
              new x = { x * 2 }
              write ("x in ano:" : int : x)
          return x
          endloop }

    wambatt = { y + 16 }
    return wambatt, y
endfun


main (notrace)
    int: fun, bun, sun, honey, cecil;
    int: moon = 3;
    boolean: gate;
begin

    gate = { moon > 9 }
    fun = { if ! gate then
                 moon + 5
            endif }
    bun, sun = { two (moon,fun) }
    honey, cecil = { ano (bun) }
    write ("fn bn sn hy cecl:" : int : fun,bun,sun,honey,cecil)
end
```

```
Program Output:  x in ano:          13
                 fn bn sn hy cecl:   8    8     3     42    26
```

(Figure 14)

```
function two (int: a, b; return int, int)
     int: x, y;
beginfun
     x = { if  (a < b)  then
               b
           endif }

     y = { a + b % x }
     return x, y
endfun



function ano(int: ab; return int, int)
     int: x, y, wambatt;
beginfun
     y = { loop
               initial x = 5 + ab;
           do
               while x < 15
               new x = { x * 2 }
               write ("x in ano:" : int : x)
           return x
           endloop }
     wambatt = { y + 16 }
     return wambatt, y
endfun



main (notrace)
    int: fun, bun, sun, honey, cecil;
    int: moon = 3;
    boolean: gate;
begin
    gate = { moon > 9 }
    fun = { if  (! gate  &&  moon > 1)  then
                moon + 5
            endif }
    bun, sun = { two (moon,fun) }
    honey, cecil = { ano (bun) }

    write ("fn bn sn hy cecl:" : int : fun,bun,sun,honey,cecil)
end
```

```
Program Output:    x in ano:          13
                   fn bn sn hy cecl:   8    8    3    42    26
```

(Figure 15)

PRIME PROGRAM

For numbers 1 to 55 the PRIME program determines if they are prime numbers. It prints out each number preceded by the message "prime" or "not prime".

```
function prime (int: n, divisor; return boolean)
        boolean: x;
beginfun
        x = { if (divisor >= n)  then
                    divisor >= n
              else
                    if (n % divisor  ==  0)  then
                         n > n
                    else
                         prime (n, divisor + 1)
                    endif
              endif }

        return x
endfun

main (notrace)
        int: n;
        boolean: boo;
begin
        loop
             initial n = 1;
        do
             while  n  <=  55
             new n = { n + 1 }
             boo = { prime (n,2) }
             if  (boo)  then
                  write ("prime: " : int : n)
             else
                  write ("notprime: " : int : n)
             endif
        endloop
end
```

OUTPUT ON NEXT PAGE

(Figure 16)

PRIME OUTPUT

```
Read the program.

Start execution.
prime:            1
prime:            2
notprime:            4
prime:            3
notprime:            6
notprime:            8
notprime:           10
notprime:           12
notprime:            9
prime:            5
notprime:           14
notprime:           16
notprime:           18
notprime:           15
notprime:           20
prime:            7
notprime:           22
notprime:           21
notprime:           24
notprime:           26
notprime:           28
notprime:           27
notprime:           30
notprime:           32
notprime:           25
notprime:           34
prime:           11
notprime:           33notprime
:          36
notprime:           38
notprime:           40
notprime:           42
notprime:           39
notprime:           35
prime:           13
notprime:           44
notprime:           46
notprime:           45
notprime:           48
notprime:           50
notprime:           52
notprime:           51
notprime:           54
prime:           17
notprime:           49
notprime:           55
prime:           19
prime:           23
prime:           29
prime:           31
prime:           37
prime:           41
prime:           43
prime:           47
prime:           53
```

FIBONACCI PROGRAM

This program computes the Fibonacci value of any number read in from the specified file. The first 2 numbers in the file must be 0 and 1 to initialize the other argument tokens in the function call. The third number specified in the file is the one the Fibonacci value is calculated for. The intermediate calculations are printed out as well as the final Fibonacci value.

```
function fibonac (int: count, fib, prefib, n; return int)
  int: x;
beginfun
   x = { if  (count <= n)  then
             fibonac (count + 1, prefib + fib, fib, n)
             write ("cnt fib prefib: " : int : count,fib,prefib)
         else
             fib
         endif }

   return x
endfun


main (notrace, fibnum)
    int: x, y, a, b;
begin
    read (a, b, x: int, fibnum)
    write ("fibonacci value: " : int : y)
    y = { fibonac(a,a,b,x) }
end
```

FIBONACCI OUTPUT

```
Read the program.

Start execution.
cnt fib prefib:      0       0       1
cnt fib prefib:      1       1       0
cnt fib prefib:      2       1       1
cnt fib prefib:      3       2       1
cnt fib prefib:      4       3       2
cnt fib prefib:      5       5       3
cnt fib prefib:      6       8       5
cnt fib prefib:      7      13       8
cnt fib prefib:      8      21      13
cnt fib prefib:      9      34      21
cnt fib prefib:     10      55      34
cnt fib prefib:     11      89      55
cnt fib prefib:     12     144      89
fibonacci value:   233
```

(Figure 17)

LET EXAMPLES

```
main (notrace)
    int:  zoo, flak, drago, dogg, xxx;
    int: yak = 2;
    int: tester = -3;
    int: x, haha;
begin
    zoo = { drago + dogg }
    drago, dogg = { let
                        int: dog, cat;
                        dog = 5;
                        cat = -4;
                    endinit
                        xxx = { 6 }
                    in
                        if  (xxx >= yak)  then
                             - (xxx / yak)
                        else
                             xxx * yak
                        endif,

                          let
                             int: zab, fun;
                        endinit
                             zab = { 10 * cat }
                             fun = { dog * 10 }
                        in
                             loop
                                  for x = -10 to 1 by 2
                             do
                                  haha = { x + 12 }
                                  write ("haha:" : int : haha)
                             endloop

                             zab + fun
                         endlet
                     endlet }
     write ("flak:" : int : flak)
     flak = { tester * 7 }
     write ("drago, dogg, zoo:" : int : drago, dogg, zoo)
 end
```

```
Program Output:   flak:                          -21
                  drago, dogg, zoo: haha    -3     10     7:     2

                  haha:                      4
                  haha:                      6
                  haha:                      8
                  haha:                     10
                  haha:                     12
```

(Figure 18)

```
main (notrace)
        int:  zoo, flak, drago, dogg, xxx;
        int: yak = 2;
        int: tester = -3;
begin
        zoo = { drago + dogg }
        drago, dogg = { let
                              int: dog, cat;
                              dog = 5;
                          endinit
                              xxx = { 6 }
                              cat = { -4 * 2 }
                          in
                              if  (xxx >= yak)  then
                                   - (xxx / yak)
                              else
                                   xxx * yak
                              endif,

                              let
                                   int: zab, fun;
                              endinit
                                   zab = { 10 * cat }
                                   fun = { dog * 10 }
                              in
                                   zab + fun
                              endlet
                          endlet }

        write ("flak:" : int : flak)
        flak = { tester * 7 }
        write ("drago, dogg, zoo:" : int : drago, dogg, zoo)
end
```

```
Program Output:      flak:                -21
                     drago, dogg, zoo:     -3     -30     -33
```

(Figure 19)

```
main (notrace)
        int: zab, zoo,  rrr, flak, out, drago, dogg;
        int: xxx, yak = 6, 2;
        int: tester = -3;
begin
         zoo = { zab + rrr }
         rrr, zab =  { if  (xxx >= yak)  then
                           - (xxx / yak),
                           xxx - yak
                       else
                           xxx + yak,
                           xxx * yak
                       endif }

        write ("result is:" : int : zoo, rrr)

        drago, dogg =  { let
                               int: dog, cat;
                               dog = 5;
                               cat = -4;
                         endinit
                               out = { dog + 7 }
                         in
                               9 * cat % 3 + out,

                               let
                                     int: zab, fun;
                               endinit
                                     zab = { 10 * cat }
                                     fun = { dog * 10 }
                               in
                                     zab + fun
                               endlet
                         endlet }

        write ("flak:" : int : flak)
        flak = { tester * 7 }
        write ("drago, dogg:" : int : drago, dogg)
end
```

```
Program Output:      flak:                -21
                     drago, dogg:          12          10resul
                     t is:                 1     -3
```

(Figure 20)

```
main (notrace)
        int: zab, zoo,  rrr, flak, out, drago;
        int: xxx, yak = 6, 2;
        int: tester = -3;
begin
         zoo = { zab + rrr }
         rrr, zab =  { if  (xxx >= yak)  then
                             - (xxx / yak),
                             xxx - yak
                       else
                             xxx + yak,
                             xxx * yak
                       endif }

        write ("result is:" : int : zoo, rrr)

        drago = { let
                        int: dog, cat;
                        dog = 5;
                        cat = -4;
                  endinit

                        out = { dog + 7 }
                  in
                        9 * cat % 3 + out
                  endlet }

        write ("flak:" : int : flak)
        flak = { tester * 7 }
        write ("drago:" : int : drago)
end
```

```
Program Output:     flak:              -21
                    drago:        12result is:        1       -3
```

(Figure 21)

SYNCHRONIZATION

In this section two programs are shown twice: one version of each uses the synchronization mechanism the other does not. In Figure 20 two **READ** statements are synchronized to control the order that data is read in from the same file. Compare the output in Figure 21 to that in 22 to see how the difference in values is affected. In Figures 22 and 23 the order of the output lines is affected.

```
main (notrace,data)
        int: z, x, y;
begin
        when x do
                read(y : int, data)
                write("y: " : int : y)
        endwhen
        z = { - (x % 6) }
        write ("result is:" : int : z)
        read (x: int, data)
end
```

```
Program Output:      result is:          -4
                     y:                  589
```

(Figure 22)

```
main (notrace,data)
        int: z, x, y;
begin
        read(y : int, data)
        write("y: " : int : y)
        z = { - (x % 6) }
        write ("result is:" : int : z)
        read (x: int, data)
end
```

```
Program Output:      y:                  4
                     result is:          -1
```

(Figure 23)

```
function two (int: a, b; return int,int)
     int: x, y;
beginfun
     x = { if  (a < b)  then
               b
           endif }
     y = { a + b % x }
     return x, y
endfun


function ano(int: ab; return int, int)
     int: x, y, wambatt;
beginfun
     y = { loop
               initial x = 5 + ab;
           do
               while x < 15
               new x = { x * 2 }
               write ("x in ano:" : int : x )
           return x
           endloop }

     wambatt = { y + 16 }
     return wambatt, y
endfun


main (notrace)
    int: fun, bun, sun, honey, cecil, glow;
    int: moon = 3;
    boolean: gate;
begin
    glow = { 788 + 2 }

    when honey do
          write("when honey: " : int : glow)
    endwhen

    gate = { moon > 9 }
    fun = { if  (! gate)  then
                 moon + 5
            endif }

    bun, sun = { two (moon,fun) }
    honey, cecil = { ano (bun) }
    write("fn bn sn hy cecl:" : int : fun,bun,sun,honey,cecil)
end
```

```
Program Output:  x in ano:           13
                 fn bn sn hy cecl:    8    8    3    42    26
                 when honey:         790
```

(Figure 24)

```
function two (int: a, b; return int, int)
     int: x, y;
beginfun
     x = { if  (a < b)  then
                b
           endif }
     y = { a + b % x }
     return x, y
endfun


function ano(int: ab; return int, int)
     int: x, y, wambatt;
beginfun
     y = { loop
                initial x = 5 + ab;
           do
                while x < 15
                new x = { x * 2 }
                write ("x in ano:" : int : x)
           return x
           endloop }

     wambatt = { y + 16 }
     return wambatt, y
endfun


main (notrace)
     int: fun, bun, sun, honey, cecil, glow;
     int: moon = 3;
     boolean: gate;
begin
     glow = { 788 + 2 }
     write("when honey: " : int : glow)

     gate = { moon > 9 }
     fun = { if  (! gate)  then
                  moon + 5
             endif }

     bun, sun = { two (moon,fun) }
     honey, cecil = { ano (bun) }

     write("fn bn sn hy cecl:" : int : fun,bun,sun,honey,cecil)
end
```

```
Program Output:  when honey:          790
                 x in ano:             13
                 fn bn sn hy cecl:      8     8     3     42     26
```

(Figure 25)

BIBLIOGRAPHY

[Ackerman 1982]

Ackerman, W. B. "Data Flow Languages," Computer 15 February 1982, pp. 15-25.

[Anderson 1978]

Anderson, Karen. "CONCUR: A High-Level Language for Concurrency Programming," Masters Thesis, RIT, Rochester, New York, 1978.

[Ackerman and Dennis 1979]

Ackerman, William B. and Jack B. Dennis. "VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual," Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1979.

[Amamiya and Hasegawa]

Amamiya, M. and Ono S. Hasegawa. "VALID: High-Level Functional Programming Language for Data Flow Machines," REV ELECTR Communication Lab (Japan), Vol. 32, No. 5, pp. 793-802.

[Arvind and Gostelow 1975]

Arvind and K. P. Gostelow. "A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture," Technical Report 72, Department of Information and Computer Science, University of California-Irvine, Irvine, California, October 1975.

[Arvind and Gostelow 1982]

Arvind and K. P. Gostelow. "The U-Interpreter," Computer 15, February 1982, pp. 42-49.

[Arvind, Gostelow, and Plouffe 1978]

Arvind, K. P. Gostelow and W. Plouffe. "The (Preliminary) Id Report," Department of Information and Computer Science (TR 114), University of California-Irvine, Irvine, California, May 1978.

[Arvind and Kathail 1981]

Arvind and Vinod Kathail. "A Multiple Processor Data Flow Machine that Supports Generalized Procedures," Eighth Annual Symposium on Computer Architecture, Minneapolis, Mn., 12-14 May 1981 (IEEE 1981), pp. 291-302.

[Ashcroft and Wadge 1976]

Ashcroft, E. A. and W. W. Wadge. "Lucid--A Formal System for Writing and Proving Programs," SIAM J. Computing, Vol. 5, No. 3, September 1976, pp. 336-354.

[Ashcroft and Wadge 1977]

Ashcroft, E. A. and W. W. Wadge. "Lucid, a Nonprocedural Language with Iteration," Communications of the ACM, Vol. 20, No. 7, July 1977.

[Backus 1978]

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613-641.

[Brock and Montz 1979]

Brock, J. Dean and Lynn B. Montz. "Translation and Optimization of Data Flow Programs," ACM Proceedings of the 1979 International Conference on Parallel Processing, August 1979 (IEEE 1979),pp. 46-54.

[Comte, Durrieu, Gelly, Plas and Syre 1978]

Comte, D., G. Durrieu, O. Gelly, A. Plas and J. C. Syre. "Parallelism, Control and Synchronization Expression in a Single Assignment Language," SIGPLAN Notices, Vol. 13, No. 1, January 1978, pp. 25-33.

[Davis and Keller 1982]

Davis, Alan L. and Robert M. Keller. "Data Flow Program Graphs," Computer, February 1982, pp. 26-41.

[De Francesco, Perego, Vaglini and Vanneschi 1980]

De Francesco, N., G. Perego, G. Vaglini and M. Vanneschi. "MAL Data Flow Programming Language," Annual Conference AICA 1980, Part II, October 1980, Technoprint, Bologna, Italy.

[Dennis 1974]

Dennis, J. B. "On Storage Management for Advanced Programming Languages," Computation Structures Group Memo 109-1, Project MAC, MIT, Cambridge, Mass., October 1974.

[Dennis 1975]

Dennis, J. B. "First Version of a Data Flow Procedure Language," MAC Technical Memorandum 61, Project MAC, MIT, Cambridge, Massachusetts, May 1975.

[Dennis 1977]

Dennis, J. B. "A Language Design for Structured Concurrency," Computation Structures Note 28-1, Laboratory for Computer Science, MIT, Cambridge, Mass., February 1977.

[Dennis, Misunas and Leung 1977]

Dennis, J. B., D. P. Misunas and C. K. Leung. "A Highly Parallel Processor Using a Data Flow Machine Language," Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, Cambridge, Mass., January 1977.

[Evans 1968]

Evans, A., Jr. "PAL--A Language designed for Teaching Programming Linguistics." Proceedings of the 23rd ACM National Conference, 1968, pp. 395-403.

[Farrow 1983]

Farrow, Rodney. "Attribute Grammars and Data-Flow Languages," ACM SIGPLAN Notices, Vol. 18, No. 6, June 1983, pp. 28-40.

[Filman and Friedman 1982]

Filman, Robert E. and Daniel P. Friedman. "Models, languages, and heuristics for distributed computing," AFIPS Conference Proceedings, Vol. 51, National Computer Conference, 1982, pp. 671-677.

[Friedman and Wise 1976]

Friedman, Daniel P. and David S. Wise. "The Impact of Applicative Programming on Multiprocessing," Technical Report No. 52, Computer Science Department, Indiana University, Bloomington, Indiana, July 1976.

[Friedman and Wise 1979]

Friedman, D. P. and D. S. Wise. "Aspects of Applicative Programming for Parallel Processing," IEEE Transactions on Computers C27, April, 1978, pp. 289-296.

[Gajski, Padua and Kuck 1982]

Gajski, D. D., D. A. Padua and D. J. Kuck. "A Second Opinion on Data Flow Machines and Languages," Computer, February 1982, pp. 58-69.

[Gaudiot and Ercegovac 1982]

Gaudiot, J-L and M. D. Ercegovac. "A Scheme for Handling Arrays in Data-Flow Systems," International Conference on Distributed Computing Systems, IEEE 1982, pp. 724-729.

[Gaudiot and Ercegovac 1984]

Gaudiot, J. L. and M. D. Ercegovac. "Performance Analysis of a Data Flow Computer with Variable Resolution Actors," IEEE Proceedings of the Fourth International Conference on Distributed Computing Systems, May 14-18, 1984, IEEE Computer Society, pp. 2-9.

[Gostelow and Thomas 1979]

Gostelow, Kim P. and Robert E. Thomas. "A View of Dataflow," AFIPS Conference Proceedings, 1979 National Computing Conference, AFIPS Press, New Jersey, Vol. 48, pp. 629- 636.

[Gurd and Watson 1980]

Gurd, John and Ian Watson. "Data Driven System for High Speed Parallel Computing--Part 1: Structuring Software for Parallel Execution," Computer Design, June 1980, pp. 91-100.

[Hankin and Glaser 1981]

Hankin, C. L. and H. W. Glaser. "The Data Flow Programming Language CAJOLE--An Informal Introduction," SIGPLAN Notices, Vol. 16, No. 7, July 1981, pp. 35-43.

[Jagannathan and Ashcroft 1984]

Jagannathan, R. and E. A. Ashcroft. "Eazyflow: A Hybrid Model for Parallel Processing," IEEE Parallel Processing, 1984, pp. 514-523.

[Johnson and Kohlstaedt 1981]

Johnson, S. D., and A. T. Kohlstaedt. "DSI Program Description," Technical Report No. 120, Department of Computer Science, Indiana University, Bloomington, Indiana, November 1981.

[Kohlstaedt 1981]

Kohlstaedt, Anne T. "Daisy 1.0 Reference Manual," Technical Report No. 119, Department of Computer Science, Indiana University, Bloomington, Indiana, November 1981.

[Kosinski 1973]

Kosinski, P. R. "A Data Flow Language for Operating Systems Programming," SIGPLAN Notices, Vol. 8, No. 8, (September 1973), pp. 89-94.

[Kubo, Kohmato and Ohno 1984]

Kubo, Masatoshi, Tatsuya Kohmato and Ytaka Ohno. "A Data Flow Machine with Optimization Driven Graph Reduction Mechanism," IEEE Proceedings of the Fourth International Conference on Distributed Computing Systems, May 14-18, 1984, IEEE Computer Society, pp. 10-14.

[Maurer and Oldehoeft 1983]

Maurer, Peter M. and Arthur E. Oldehoeft. "The Use of Combinators in translating a Purely Functional Language to Low-Level Data-Flow Graphs," Computing Languages, Vol. 8, No. 1, pp. 27-45.

[McGraw 1982]

McGraw, James R. "The VAL Language: Description and Analysis," ACM Transactions Programming Languages and Systems, Vol. 4, No. 1, January 1982, pp. 44-82.

[Miranker]

Miranker, Glen Seth. "Implementation of Procedures on a Class of Data Flow Processors," Laboratory for Computer Science, MIT, Cambridge, Mass.

[Pingali and Arvind 1985]

Pingali, Keshav and Arvind. "Efficient Demand-Driven Evaluation. Part 1," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 2, April 1985, pp. 311-333.

[Plas, Comte, Gelly and Syre 1976]

Plas, A., D. Comte, O. Gelly and J. C. Syre. "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment," Proceedings of the 1976 International Conference on Parallel Processing, August 1976, pp. 293-302.

[Rumbaugh 1977]

Rumbaugh, James. "A Data Flow Multiprocessor," IEEE Transactions on Computers, Vol. C-26, No. 2, February 1977, pp. 138-146.

[Salter, Brennan and Friedman 1980]

Salter, Brennan and Friedman. "CONCUR: A Language for Continuous Concurrent Processes," Computer Languages, Vol. 5, 1980, pp. 163-189.

[Srini 1981]

Srini, Vason P. "An Architecture for Extended Abstract Data Flow," Eighth Annual Symposium on Computer Architecture, Minneapolis, Mn., May 12-14, 1981 (New York: IEEE 1981), pp. 303-325.

[Srini 1984]

Srini, Vason P. "Node Reassignment in a Data Flow System," IEEE Proceedings of the Fourth International Conference on Distributed Computing Systems, May 14-18, 1984, IEEE Computer Society, pp. 15-

[Suzuki, Kurihara, Tanaka and Moto-Oka 1982]

Suzuki, Tatsuo, Ken Kurihara, Hidehiko Tanaka and Tohru Moto-Oka. "Procedure Level Data Flow Processing on Dynamic Structure Multimicroprocessors," Journal of Information Processing, Vol. 5, No. 1, 1982, pp. 11-16.

[Syre, Comte and Hifdi 1977]

Syre, J. C., D. Comte and N. Hifdi. "Pipelining, Parallelism, and Asynchronism in the LAU System," Proceedings of the 1977 International Conference on Parallel Processing, August 1977, pp. 87-92.

[Torsone 1984]

Torsone, Carol M. "Simulation of a Data Flow Computer," Masters Thesis, RIT, Rochester, New York, 1984.

[Treleaven 1978]

Treleaven, Philip C. "Principal Components of a Data Flow Computer," Fourth EUROMICRO Symposium on Microprocessing and Microprogramming, October 17-19, IEEE Computer Society, 1978, pp. 366-374.

[Treleaven 1979]

Treleaven, Philip C. "Exploiting Program Concurrency in Computing Systems," Computer 12, January 1979), pp. 42-49.

[Treleaven 1980]

Treleaven, Philip C. "VLSI: Machine Architecture and Very High Level Languages," SIGARCH Computer Architecture News, Vol. 8, December 15, 1980, pp. 27-38.

[Turner 1979]

Turner, D. A. "A New Implementation Technique for Applicative Languages," Software-Practice and Experience, Vol. 9, January 1979.

[Watson and Gurd 1979]

Watson, Ian and John Gurd. "A Prototype Data Flow Computer with Token Labelling," AFIPS Conference Proceedings, 1979 National Computer Conference, pp. 623-628.

[Weng 1975]

Weng, Kung-Song. "Stream-Oriented Computation in Recursive Data Flow Schemas," Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.

[Wetherell 1982]

Wetherell, C. S. "Error Data Values in the Data-Flow Language VAL," ACM Transactions Programming Languages and Systems, Vol. 4, No. 2, April 1982, pp. 44-82.

[Wise 1982]

Wise, Michael J. "Epilog = Prolog + Data Flow: Arguments for Combining Prolog with a Data Driven Mechanism," SIGPLAN Notices, Vol. 17, No. 12, December 1982, pp. 80-86.

[Yamaguchi, Inamoto and Kunii 1985]

Yamaguchi, Kazunori, Naota Inamota and Tosiyasu L. Kunii. "A Data Flow Language for Controlling Multiple Interactive Devices," IEEE CG&A, March 1985, pp. 48-60.