MIT/LCS/TM-146

A MACHINE LANGUAGE INSTRUCTION SET
FOR A
DATA FLOW PROCESSOR

Donald J. Aoki

December 1979

A Machine Language Instruction Set for a Data Flow Processor

by

©Donald J. Aoki

Submitted to the Department of Electrical Engineering and Computer Science on 10 August 1979, in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

A data flow processor is a computer in which instructions are data driven and enabled for execution by the arrival of their operands. Data flow processors execute data flow programs, normally represented as program graphs, which represent the data dependencies between operations. This thesis presents a machine language instruction set for a Form I data flow machine based on the Dennis-Misunas design.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Key Words: ack-minimization, data-flow computer, program graph, data-flow instruction cell, safety, instruction cell configuration, instruction set

ACKNOWLEDGEMENTS

I would like to thank the members of the Computation Structures Group of the Laboratory for Computer Science and my advisor, Jack Dennis, for their guidance and counsel in preparing this thesis. I also gratefully acknowledge the support of Bell Telephone Laboratories, Incorporated, without whose support this thesis would not have been possible. I am much obliged to Arif Feridun for his comments, criticisms, and helpful suggestions concerning the format of the instruction cell. I thank the Laboratory for Computer Science for the facilities used in the preparation of this thesis and Alan Snyder for maintaining the R text justifier. I also thank my friends Kathleen Balcezak, James Bruce, Jim Leth, Mimi Mengis, and Marcos Ortiz for their friendship and helpfulness over the past year.

CONTENTS

FIGURES

1. Data Flow Concepts

1.1 Introduction

A *data flow computer* is a computer in which instructions are data driven and enabled for execution by the arrival of their operands. Data flow computers execute *data flow programs graphs*, also known as *data flow schemas*. The structure of a data flow program often permits many instructions to be simultaneously enabled, and the packet communication architecture of the machine is designed in a way which permits the concurrent execution of these enabled instructions, within certain limits. In this thesis we define a machine language instruction set for a Form I data flow computer as defined in this chapter.

1.2 Data Flow Program Graphs

Data flow program graphs explicitly represent the data dependencies within a program, and in so doing, identify program operations that may be executed independently (concurrently). Initial work developing the theory of program graphs was led by Rodriguez [13] and Dennis [6]. A data flow program graph may be formally viewed as a directed graph whose nodes are *actors* and *links*, and whose arcs represent data paths. A data flow *configuration* is the association of data values, or *tokens*, with certain arcs of the program graph. A program graph "executes" as tokens "flow" along the arcs, into and out of nodes according to the *firing rules* which determine the valid configuration sequences. The firing rules specify that a node may *fire* whenever it is enabled, *i.e.*, whenever tokens are present on each of its input arcs and no token is present on any of its output arcs. Enabled nodes may fire in any order. When an actor fires, tokens are removed from each of its input arcs, the function represented by the actor is computed using the absorbed data values, and the result is output as a token on the actor's output arc. When a link fires, the token is removed from its input arc, and its value is duplicated onto each of its output arcs. A link is essentially a copy operator. Figure 1 presents a summary of the basic data flow actors. An operator outputs the result of applying the function f to arguments x and y. A decider outputs a boolean as the result of applying the predicate p. A True (False) gate outputs the value received at a if b = *true* (*false*); otherwise, it absorbs both input tokens. A Merge gate absorbs and outputs the input found at x or y corresponding to the boolean value of b. The other token, if present, is unaffected. A Switch gate routes input x to either y or z, depending upon the boolean value of b.

An example would best serve to illustrate the representation of a simple computation in data flow program graph form. The data flow graph for computing

$$d = b^2 - 4ac$$

is shown in Figure 2. Link actors which distribute to only one destination are omitted. Let us assign the tokens with values 1, 3, and 2 to the graph input arcs a, b, and c respectively, and apply the firing rules to the resulting initial configuration. Tokens will be represented pictorially on program graphs as solid circles. The particular order in which the operators fire is irrelevant in computing the value of d correctly because Patil [11] has shown that a data flow graph is determinate if it is a composition of determinate data flow operators. In this paper we shall make three assumptions concerning the firing process. First, we assume that an operator firing requires one time unit, and that an operator always fires within one time unit after which it has become enabled. Second, we assume that link firings are instantaneous. Third, we assume that whenever two or more

Figure 1. Summary of Data Flow Actors

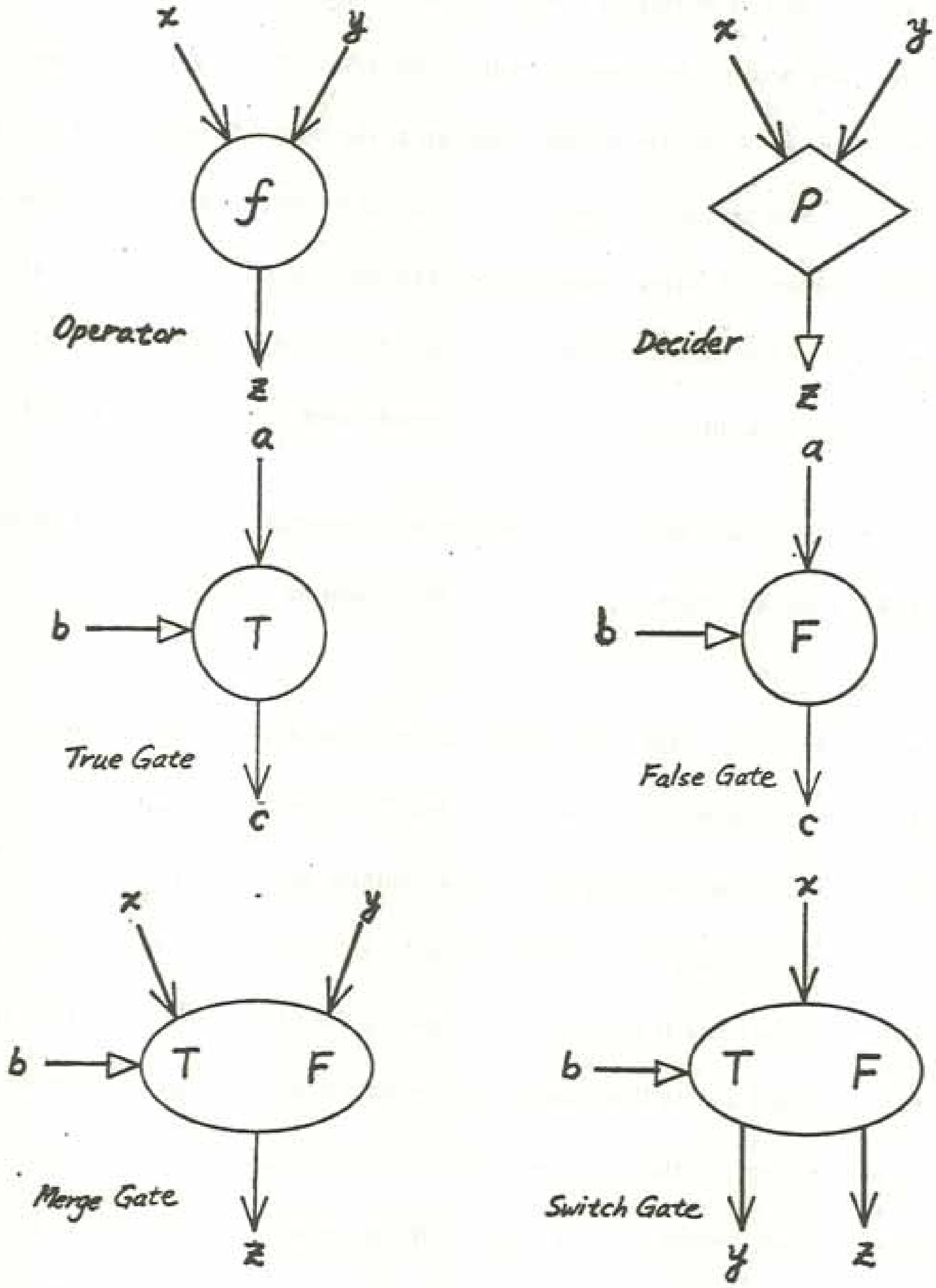
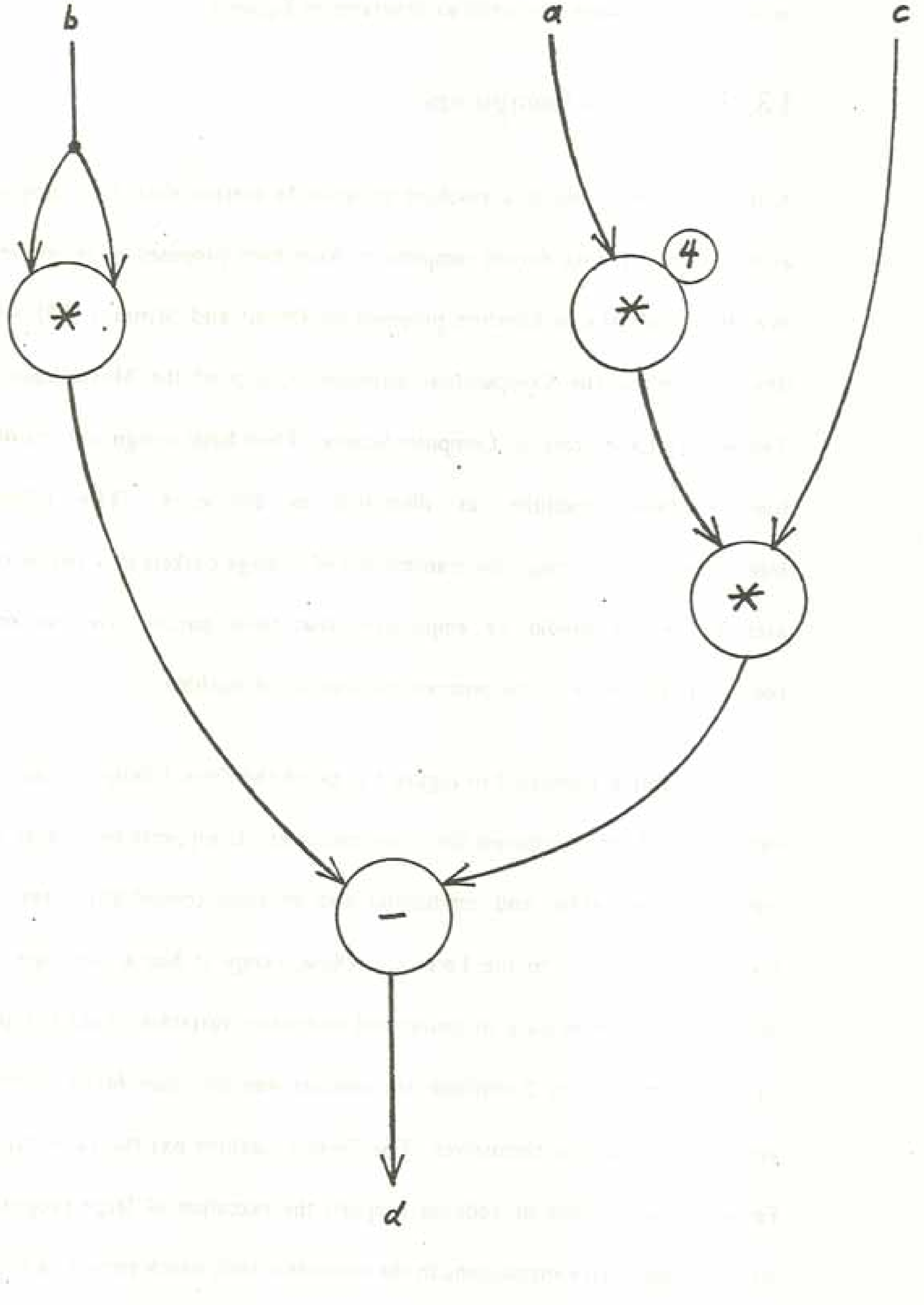


Figure 2. Program Graph for d=b^2-4ac



operators can be fired concurrently, they are. Using these assumptions, the value of d can be computed in three time units as illustrated in Figure 3.

1.3 Data Flow Computers

A data flow computer is a machine designed to execute data flow programs. Various architectures for data driven computation have been proposed in recent years [3,7,8,5,12]. We shall study the architecture proposed by Dennis and Misunas [7,8], which is under development by the Computation Structures Group of the Massachusetts Institute of Technology Laboratory for Computer Science. Their basic design consists of a network of four hardware modules, as illustrated in Figure 4. The different modules intercommunicate through the transmission of message packets in a packet communication architecture. It should be emphasized that these packets are the only means of communication between the separate modules of the machine.

The design illustrated in Figure 4 is called the *Form 1* design because it is the most elementary of four proposed data flow machines. It supports small data flow programs with scalar variables, and conditional and iteration control structures. The Form 2 machine is identical to the Form 1 machine, except it has a Structure Processor that permits the usage of data structures and non-scalar variables. Tokens representing data structures in a Form 2 machine are pointers into Structure Memory, which holds the actual data structures themselves. The Form 3 machine has the same capabilities of the Form 2 machine, but in addition supports the execution of large programs by holding only the most active instructions in the instruction cells, which serve as a cache.

Figure 3. The Computation of d=b^2-4ac

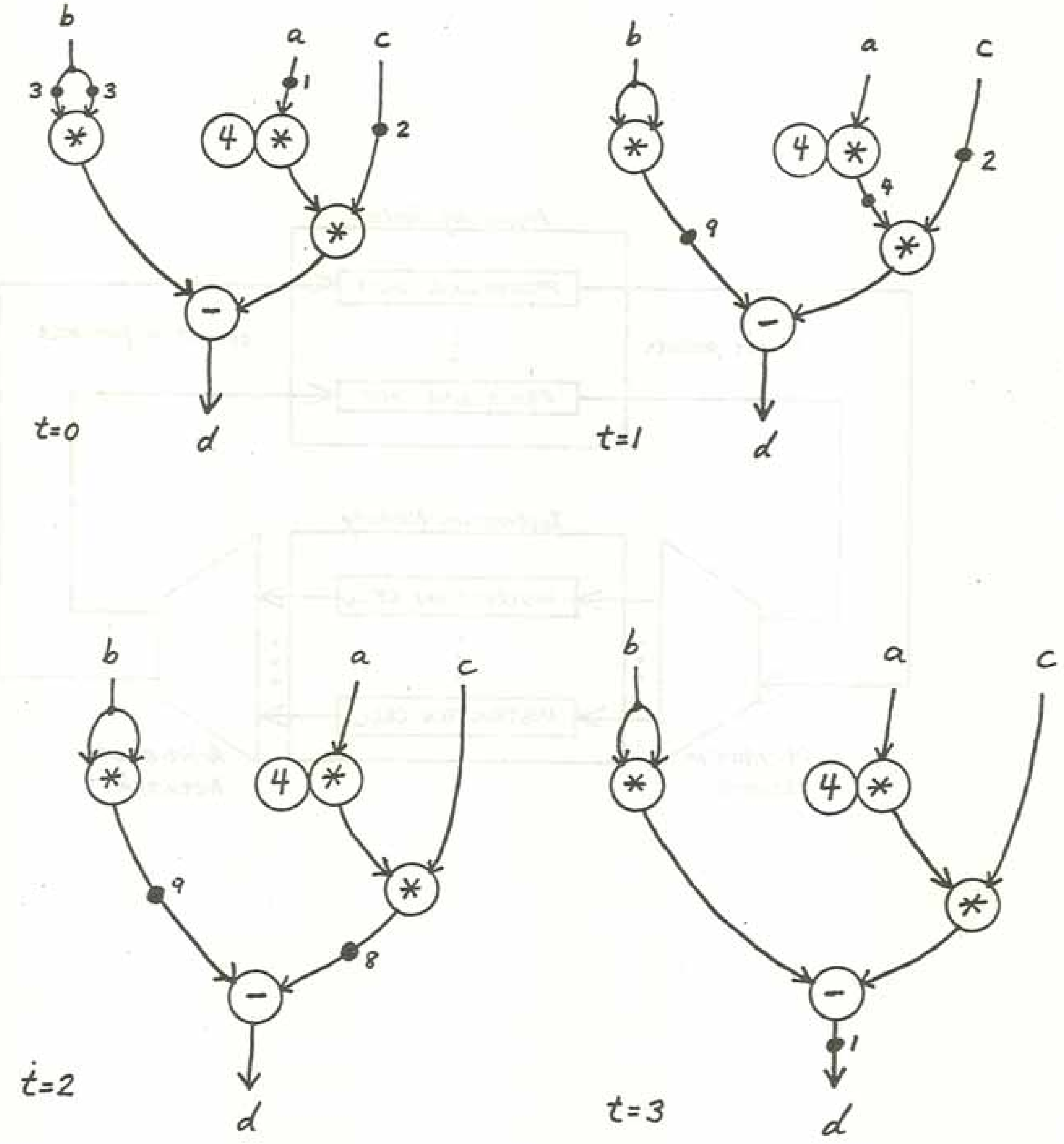
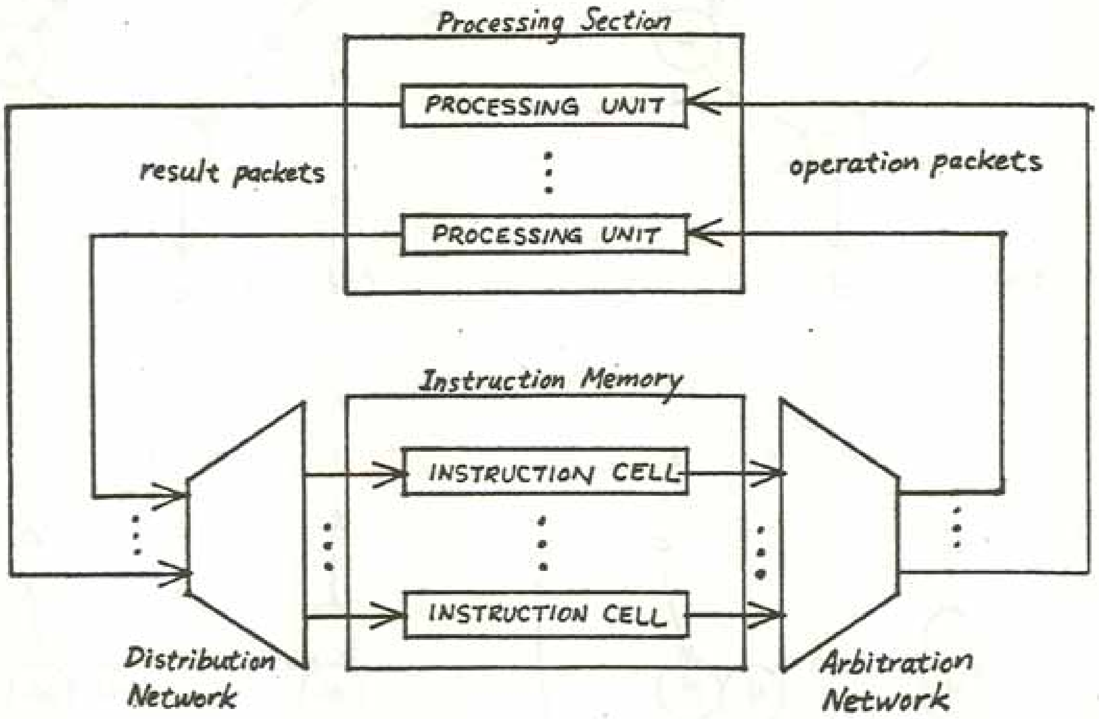


Figure 4. Form I Data Flow Machine



The Form 4 machine is envisioned as a general purpose computer with the capabilities of supporting all aspects of data driven computation.

The Form 1 machine operates as follows. A data flow program graph is encoded into Instruction Memory. Instruction Memory consists of *instruction cells*, each of which is loaded with an instruction operation code (opcode) corresponding to the function of the data flow actor, and a list of destination addresses which are pointers to other instruction cells which receive the computed results. A destination address consists of the address of an instruction cell, together with the address of a *receiver* within that cell. Each instruction cell has receivers available for the storage of operands obtained from the Distribution Network. An instruction cell becomes enabled by the arrival of all its operands and fires by sending an *operation packet* into the Arbitration Network, which sorts it, and sends it to an available Processing Unit. The operation packet contains the instruction operation code, the necessary operands, and the destination addresses to which the results are to be sent. The Processing Unit performs the operation requested and transmits the computed values in *result packets* to each destination, via the Distribution Network. The results arrive as the operands for other instruction cells which eventually become enabled and fire, repeating the cycle through the machine. Note that several instruction cells may be simultaneously enabled, and that they may all concurrently transmit operation packets to the Processing Section.

It is clear that the firing of an instruction cell in this machine model corresponds to the composite action of firing an operator along with the immediate firing of its output link in a data flow graph. Consequently, tokens in a data flow machine can be regarded as always residing on input arcs of operators, and never on output arcs. This is the justification for our assumption in the last section that link firings are conceptually instantaneous.

In summary, the Instruction Memory holds instruction cells which correspond to the actors in a data flow program, the Processing Section performs the computations associated with firing an actor, and the Arbitration and Distribution Networks serve as routing networks for packets between the Processing Section and Instruction Memory.

2. Data Flow Programming Languages

2.1 VAL - A Value-Oriented Algorithmic Language

VAL, a Value-Oriented Algorithmic Language, is a high level applicative data flow programming language proposed by Ackerman and Dennis [1]. As an applicative language, it is functional in nature and completely free of side effects. As a result, each VAL expression or module corresponds to a mathematical function, and the combination of such modules is equivalent to functional composition. Although its strength lies in its ability to easily express algorithms in the area of highly concurrent numerical computation, VAL is intended to be used as a general purpose language on future generations of general purpose data flow computers.

2.2 ADFL - An Applicative Data Flow Language

ADFL, an Applicative Data Flow Language, is a simplification of VAL. It is intended for simple applications on the Form 1 machine. It excludes type definitions and notions of modules which are found in VAL. A BNF syntax specification of ADFL follows:

```
exp ::= id | const | exp, exp | oper(exp) | let idlist = exp in exp |
        if exp then exp else exp | for idlist = exp do iteration

iteration ::= exp | iter exp | let idlist = exp in iteration |
              if exp then iteration else iteration

id ::= "programming language identifiers"

idlist ::= id {, id }

const ::= "programming language constants"

oper ::= "programming language operators"
```

Notes.

(1) The BNF grammar requires that all operator applications are specified in prefix form, *e.g.*, +(x,5). The infix form, "x+5", will be considered as an acceptable equivalent.

(2) Values are not assigned to identifiers, but rather locally bound to them. The expression

let *x*,*y* = 2,3 **in** *x*×*y*

evaluates the expression "*x*×*y*" with *x*=2 and *y*=3. The values of *x* and *y* remain unchanged outside the let expression.

(3) Iterations are expressed as tail recursions. The expression

```
for i,y = n,1 do
   if i>1 then iter i-1,y×i else y
```

computes the factorial of *n*. The iteration body is reexecuted if *i*>1, with the iteration identifiers bound to the corresponding iter expressions; otherwise, the iteration terminates with the value of *y*.

Brock has specified a translation algorithm, $\mathcal{T}$, which maps an ADFL expression into its program graph implementation [4]. We shall consider the graphs produced by his algorithm from syntactically correct ADFL programs to be *well-formed*. In the remaining chapters we will examine how to implement well-formed data flow graphs on a Form I computer.

3. Acknowledge Signal Generation

3.1 Safety and Acknowledgement

Definition. An *instruction cell configuration* for a program is an assignment of instructions and data (not necessarily unique) to instruction cells in Instruction Memory, such that the execution of the data flow machine simulates the execution of the program.

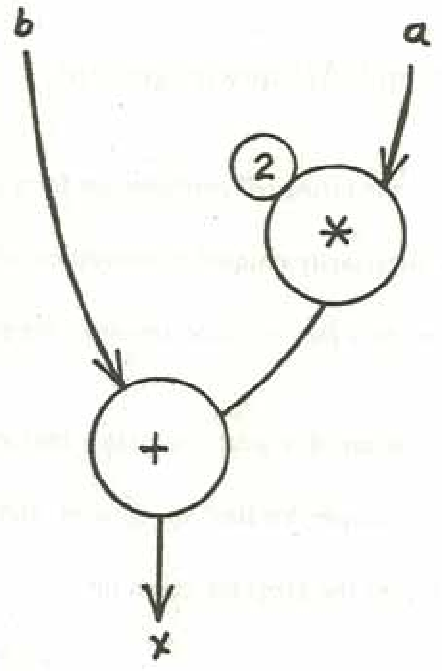
The translation of a program graph into an instruction cell configuration requires more than the simple loading of opcodes and destination addresses into Instruction Memory. Consider the program graph for

$$x = 2a + b$$

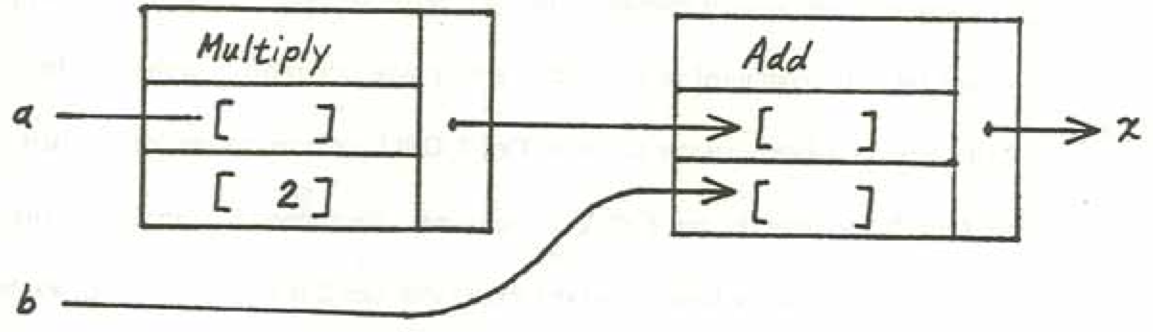
as shown in Figure 5(a) At first glance, it may appear that the corresponding instruction cell configuration is the arrangement of cells depicted in Figure 5(b). Upon closer analysis, however, we see that a problem exists.

Suppose we wish to compute x for two series of values for a and b. Suppose the values for b are computed or received at a very slow rate relative to the rate for a. Since Cell 1 fires at a much greater rate than Cell 2, Cell 1 produces values for the first receiver of Cell 2 too quickly for Cell 2 to consume. This does not endanger the proper computation of x if the flow of operand values into Cell 2 is controlled by the exchange of ready/acknowledge packets between Cell 2 and the Distribution Network. However, there is a problem in that if Cell 2 falls too far behind, the Distribution Network would become congested with result packets destined for the first receiver of Cell 2. This accumulation of packets waiting for acceptance would reduce the capacity of the network, and could produce a deadlock by preventing the further execution of Cell 2 if its second operand, b,

Figure 5. Program Graph and Instruction Cell Configuration for x=2a+b



(a)



(b)

becomes detained in the network due to blockage.

In general, the problem has to do with the inability of instruction cells to "look ahead" to see if space exists for their results. Instruction cells in the simplistic representation of Figure 5(b) fire blindly without considering the status of the instruction cells ahead. In order to ensure freedom from deadlock, we need to guarantee that our data flow programs are *safe*, where the concept of *safety* is defined as follows:

Definition. A configuration of tokens on operator input arcs of a data flow program is *safe* if each actor is enabled only if no tokens are present on any output arc of its output link. A data flow program is *safe* for a given initial configuration if each configuration reachable from the initial configuration through successive execution of actors is a safe configuration. An instruction cell configuration that is safe is ensured freedom from deadlock.

Definition. A $\mathcal{T}_{Q(1)}$ program graph is a well-formed data flow graph, each of whose data arcs acts as a single token queue, *i.e.*, each arc is either empty or carries one token at any instant in time. Each of the program graphs which have been presented up to this point may be viewed as a $\mathcal{T}_{Q(1)}$ graph. (Strictly speaking, the graphs produced by the translation algorithm $\mathcal{T}$ have data arcs that act as infinite queues, but Montz has demonstrated their equivalence to $\mathcal{T}_{Q(1)}$ graphs [10].)

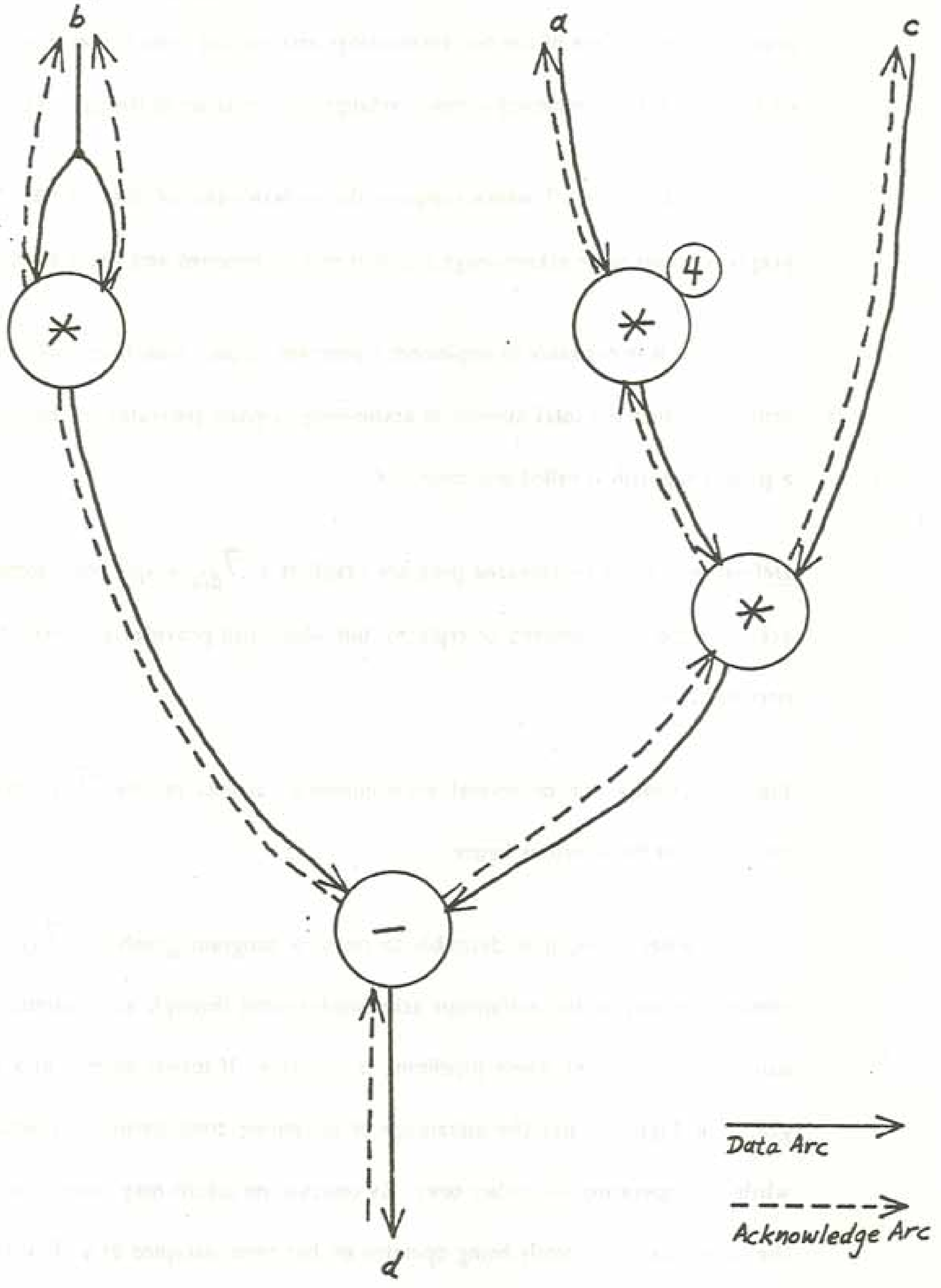
This problem of safety is absent from the $\mathcal{T}_{Q(1)}$ program graph representation because the data arcs function as single token queues and the firing rule prevents nodes from firing if their output arcs are not empty. However, we do need to consider this problem with respect to data flow machines and develop a mechanism for assuring the safe execution of instruction cell configurations, since instruction cell configurations do not identically correspond to $\mathcal{G}_{Q(1)}$ graphs.

We can assure safety in instruction cell configurations by providing a ready/acknowledge signalling mechanism between data flow operators. Upon firing, an instruction cell will send acknowledge signals to each operand source to indicate that its receivers are empty. Conceptually this involves adding an acknowledge signal arc in the reverse direction for each data arc present in the $\mathcal{G}_{Q(1)}$ graph. Such a transformed graph is called a $\mathcal{G}_{d/a}$ graph. By making each data arc a data/acknowledge arc pair, we can transform any $\mathcal{G}_{Q(1)}$ graph into a $\mathcal{G}_{d/a}$ graph.

Definition. A $\mathcal{G}_{d/a}$ graph is a data flow program graph with *all* data/acknowledge pairs present. Both the data and acknowledge arcs still continue to function as single token queues.

Figure 6 illustrates a simple $\mathcal{G}_{d/a}$ graph which computes b^2-4ac. It is functionally equivalent to the $\mathcal{G}_{Q(1)}$ graph shown in Figure 2. The acknowledge arcs are indicated in the figure by dashed lines. With the transformation of $\mathcal{G}_{Q(1)}$ graphs into $\mathcal{G}_{d/a}$ graphs we also need to modify the firing rule appropriately to read: An operator (except for a Merge gate) is enabled for firing whenever: (1) all data inputs are present, and (2) all necessary acknowledge inputs are present (occasionally all acknowledge arcs need not have tokens for an operator to fire; see Figure 14). A Merge gate is enabled for firing whenever: (1) the control (boolean) input is present, (2) the input at the True or False port corresponding to the value of the control token is present, and (3) all necessary acknowledge inputs are present.

Figure 6. $\mathcal{T}_{d/a}$ Graph for Computing b^2-4ac



Note that some of the acknowledge arcs in a $\mathcal{T}_{d/a}$ graph may be unnecessary in assuring safety. One of the two acknowledge arcs leading from the multiply operator to *b* in Figure 7 may be removed without endangering the safety of the program graph.

Consider Figure 7, which computes the boolean value of "$2*(x+1) < 0$". This program graph contains some acknowledge arcs that may be removed and replaced by longer ones.

Often it is desirable to implement a program graph with fewer acknowledge arcs in order to reduce the total number of acknowledge signals generated in the computer. Such a program graph is called *ack-minimized.*

Definition. An *ack-minimized* program graph is a $\mathcal{T}_{d/a}$ graph with some or all of its acknowledge arcs removed or replaced, but which still provides safe execution. It is not necessarily unique.

Figure 8 shows one of several ack-minimized versions of the $\mathcal{T}_{d/a}$ program graph illustrated in the previous figure.

At other times, it is desirable to retain a program graph in $\mathcal{T}_{d/a}$ form without eliminating any of the extraneous acknowledgements through ack-minimization. This is usually done in cases where pipelining is desirable. If tokens enter *x* at a high rate, the graph of Figure 8 has the advantage of beginning computations on additional tokens while still operating on earlier ones. By contrast, no tokens may enter *x* in Figure 9 until the token that is currently being operated on has been accepted at *y*. If it is important to have a high throughput for this section of code, then we would choose to retain this graph in $\mathcal{T}_{d/a}$ form.

Figure 7. $\mathcal{T}_{d/a}$ Graph with Unnecessary Acknowledge Arcs

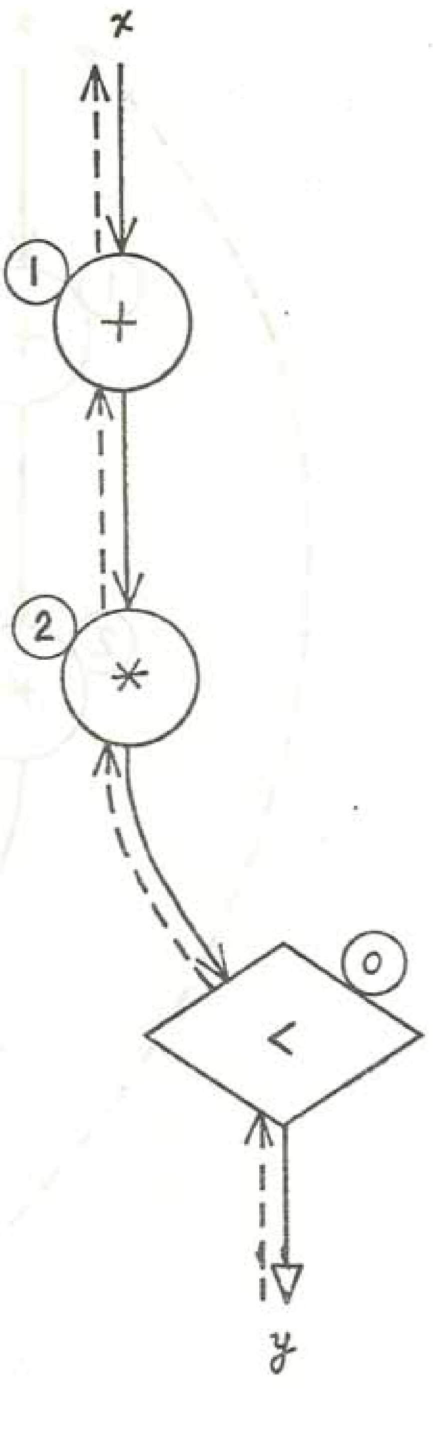
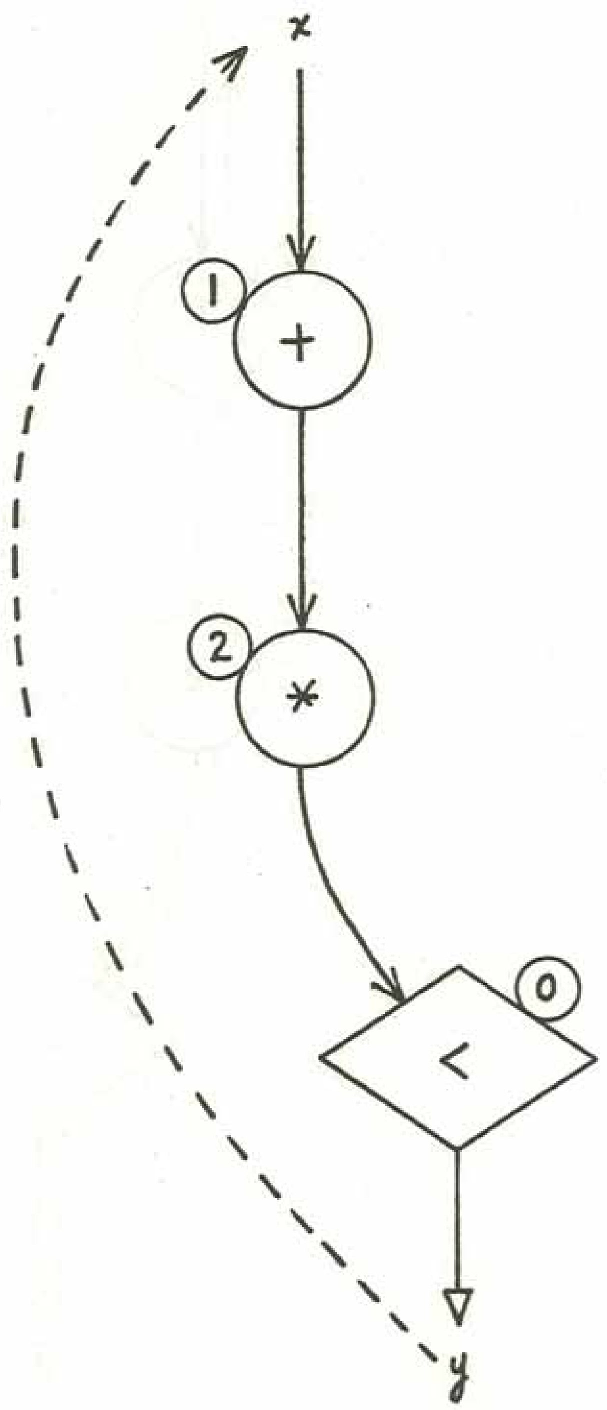


Figure 8. An Ack-minimized Program Graph



It is known that $\mathcal{T}_{d/a}$ graphs are safe because $\mathcal{T}_{Q(1)}$ graphs are safe, and Montz has proven the functional equivalence between the two [10]. She demonstrates that the presence of all acknowledge inputs is equivalent to the $\mathcal{T}_{Q(1)}$ state of all output arcs being empty. We can now derive a safe instruction cell configuration by first transforming a $\mathcal{T}_{Q(1)}$ graph into a $\mathcal{T}_{d/a}$ graph, and implementing the latter. There are two ways of implementing $\mathcal{T}_{d/a}$ graphs on a data flow computer. Both methods use different techniques for determining the acknowledgement addresses.

One approach, called *implicit* acknowledgement generation, models the acknowledge arcs by requiring that each instruction cell includes, upon firing, its memory address as a tag in the operation packet transmitted to the Arbitration Network. The firing instruction cell sends an acknowledge signal to each *operand source, i.e.*, each instruction cell which has sent it an operand. The cell is disabled until it receives an acknowledge signal from each of its destination cells, and all necessary operands for a new firing. Meanwhile, the Arbitration Network sends the packet to a Processing Unit, which reproduces the address tag and includes it with each result packet transmitted to the Distribution Network. The Distribution Network forwards each result to an instruction cell, which stores the operand and acknowledge address tag upon receipt. Each tag is later used by the receiving cell as an operand source acknowledge address.

The second approach is called *explicit* acknowledge generation. This method involves loading the acknowledge addresses directly into Instruction Memory along with the operation codes and destination addresses, prior to the execution of the program. When an instruction cell fires under the explicit acknowledgement method, it does not include its address in the operation packet because its destination cells already know where to acknowledge. Other than this, the machine functions identically to its operation under the implicit generation method.

3.2 Implicit Acknowledgement Generation

There are certain advantages and disadvantages associated with implicit acknowledgement generation. We shall first consider the advantages.

An instruction cell configuration which implicitly acknowledges the receipt of operands is assured to be *always safe*. This is so because it always implements a $\mathcal{T}_{d/a}$ graph. Since the machine never executes ack-minimized programs, the compiler never generates them, and it becomes unnecessary to be concerned with the problems of maintaining the safety of programs as they are optimized through ack-minimization. Programs which are explicitly acknowledged can also be guaranteed safe, but only if they are $\mathcal{T}_{d/a}$ graph implementations, or if the optimizations performed to derive the ack-minimized graph which is implemented are known to be preserve the safety of the original $\mathcal{T}_{d/a}$ form.

A second advantage of implicit acknowledgement generation is its ability to support programs which utilize instruction cells with multiple operand sources.

Definition. An instruction cell with *multiple operand sources* is any instruction cell with one or more receivers that has at least one operand that can be sent from two or more different sources. In such a situation, two or more instruction cells may send their results to the same destination during the execution of different parts of the program.

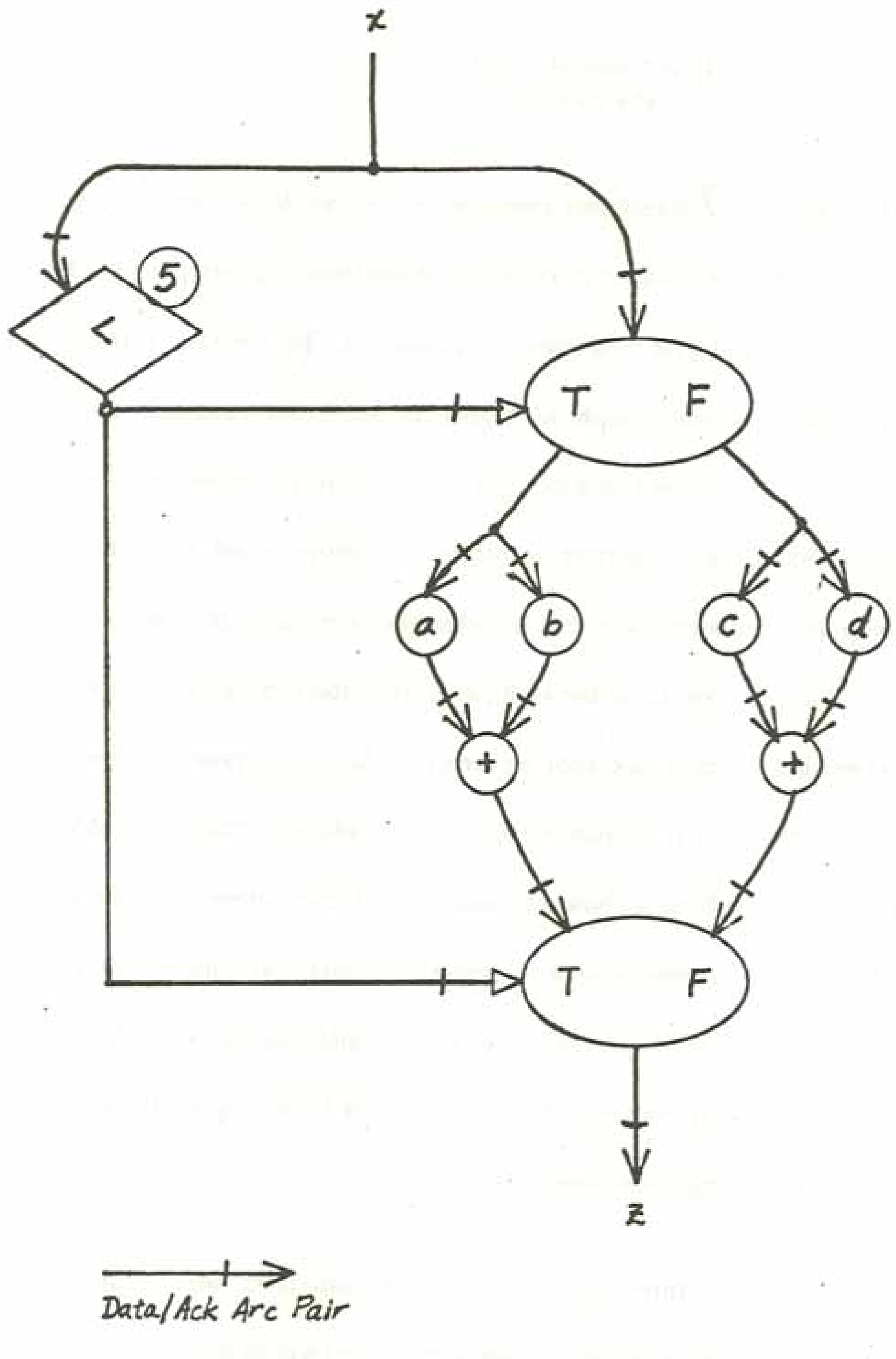
The use of multiple operand sources can optimize the space and time required by certain classes of data flow programs. Consider the following fragment of ADFL code:

```
z := if x<5 then a(x)+b(x)
              else c(x)+d(x)
```

Using Brock's $\mathcal{T}$ translation algorithm [4], we can derive the $\mathcal{T}_{d/a}$ graph of Figure 9. (All arcs may be assumed to represent data/acknowledge arc pairs.) This graph requires nine actors and five time units to compute z. By contrast, examine the functionally equivalent program graph of Figure 10 which uses multiple operand sources. An immediate savings of two actors and one time unit is realized. Suppose both graphs are converted to their respective instruction cell configurations and executed on a data flow computer. It does not matter if we acknowledge implicitly or explicitly in Figure 9, but we note that the structure of the multiple operand source graph (Figure 10) makes the explicit acknowledgement of operands between the addition operator and the cells *a*, *b*, *c*, and *d* impossible. This is because the acknowledge address is dynamic and it is not possible to know which cells to acknowledge unless it is known where each operand has originated. The implicit acknowledgement method provides this information in the address tag associated with each operand. The reader should note that explicit acknowledgement *can* be used for an ack-minimized version of Figure 10 (see Figure 11), but that the advantages of maximized pipelining would be lost.

There are three major disadvantages associated with implicit acknowledgement generation. First, note that the operation packets will be longer because of the inclusion of the firing cell's address. This may result in an increase in transit and processing times within the routing networks and the Processing Section. Second, many unnecessary and

Figure 9. Program Graph for a Conditional Statement



redundant acknowledge signals can be transmitted, cluttering up the Distribution Network and decreasing its performance. Third, it is expected that multiple operand source optimizations would apply to only a small fraction of data flow programs. This detracts from the argument that they would constitute a significant advantage.

3.3 Explicit Acknowledgement Generation

Explicit acknowledgement generation has the advantage of being more flexible than the implicit method, in the sense that it permits a greater variety of program optimizations. Explicit acknowledgement permits ack-minimization, while implicit acknowledgement does not, since the number of acknowledges can be controlled by the presence or absence of acknowledge addresses in each instruction cell.

Let us consider an example to see the advantages explicit acknowledgement offers us. Consider the following ADFL iteration:

```
out := for x := init do
          if x>1 then iter x÷3
                 else x
```

This program may be translated as shown in Figure 12. It uses four actors and eight data arcs. The numbers of the form *ae*/*ar* beside each actor indicate the number of acknowledge signals expected, *ae*, and the number of acknowledge signals received, *ar*. (If *ar* = *ae*, then the actor has all necessary acknowledge inputs.) Implicit acknowledgement generation would require that we use all eight acknowledge arcs; however, under the explicit method we can ack-minimize and use only four (see Figure 13). Furthermore, if we fully ack-minimize as shown in Figure 14, only three acknowledge arcs are required. Note

Figure 10. Program Graph with Multiple Operand Sources

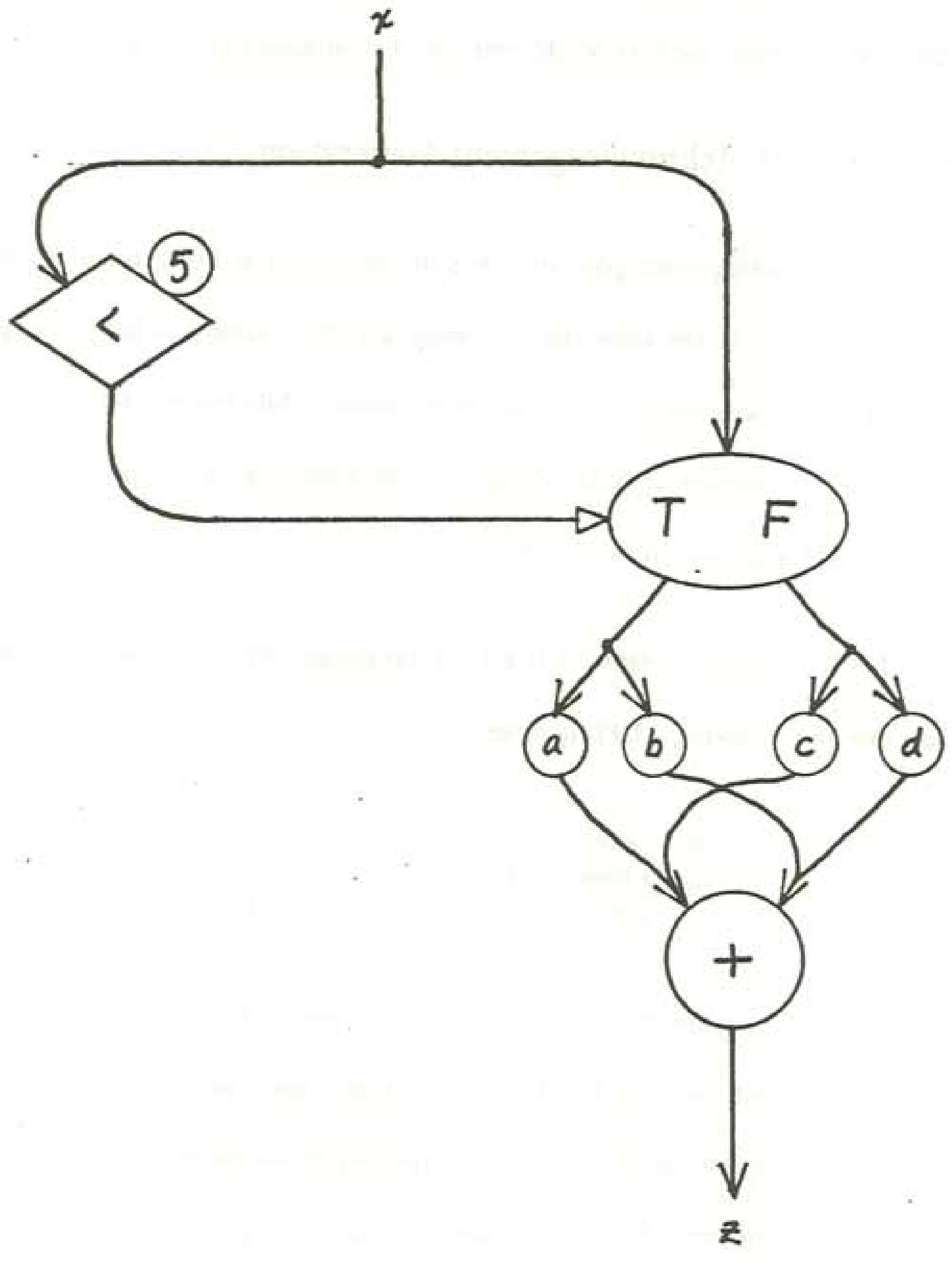


Figure 11. An Ack-minimized Version of Figure 10

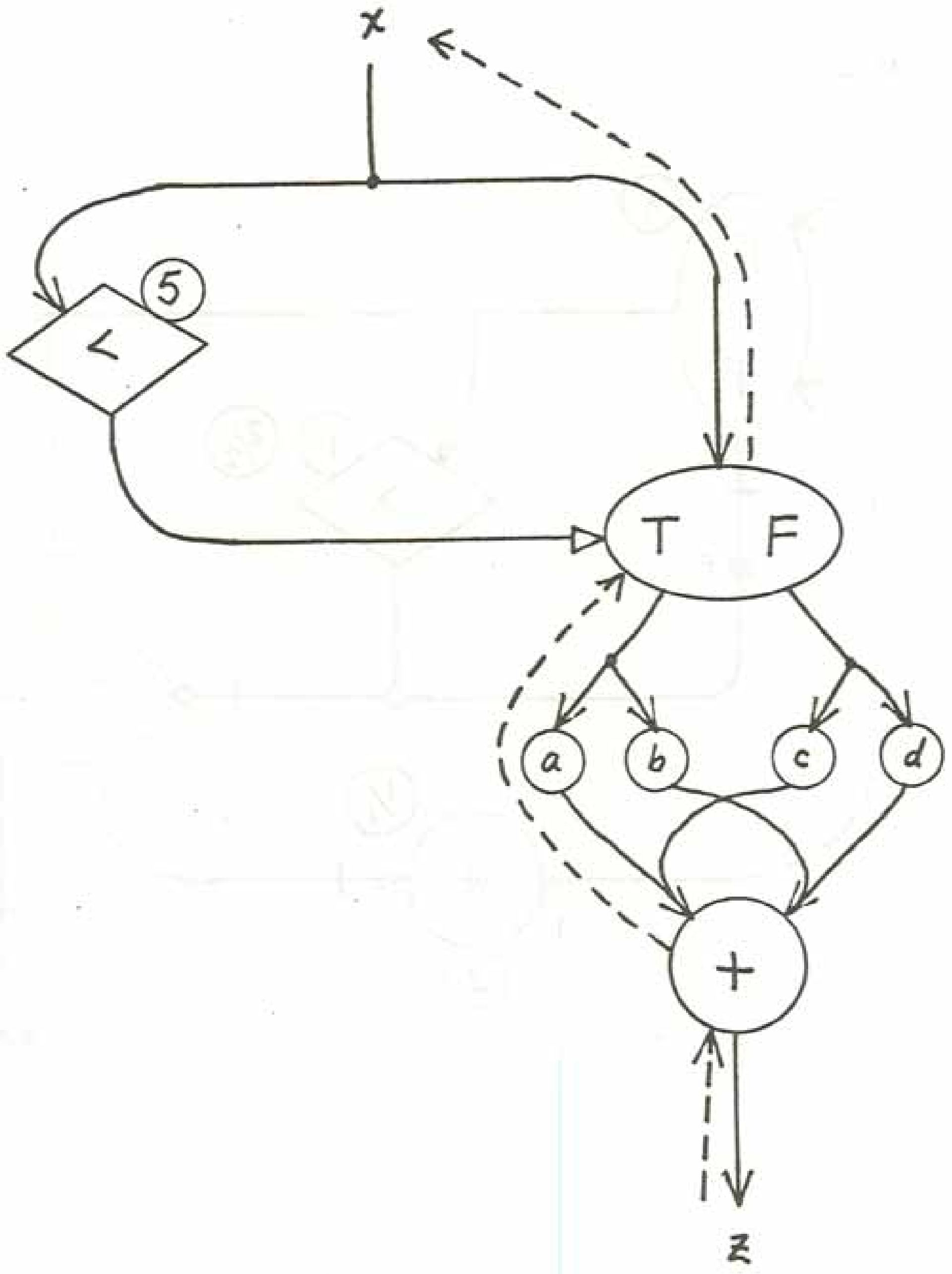


Figure 12. $\mathcal{T}_{d/a}$ Translation of an Iteration

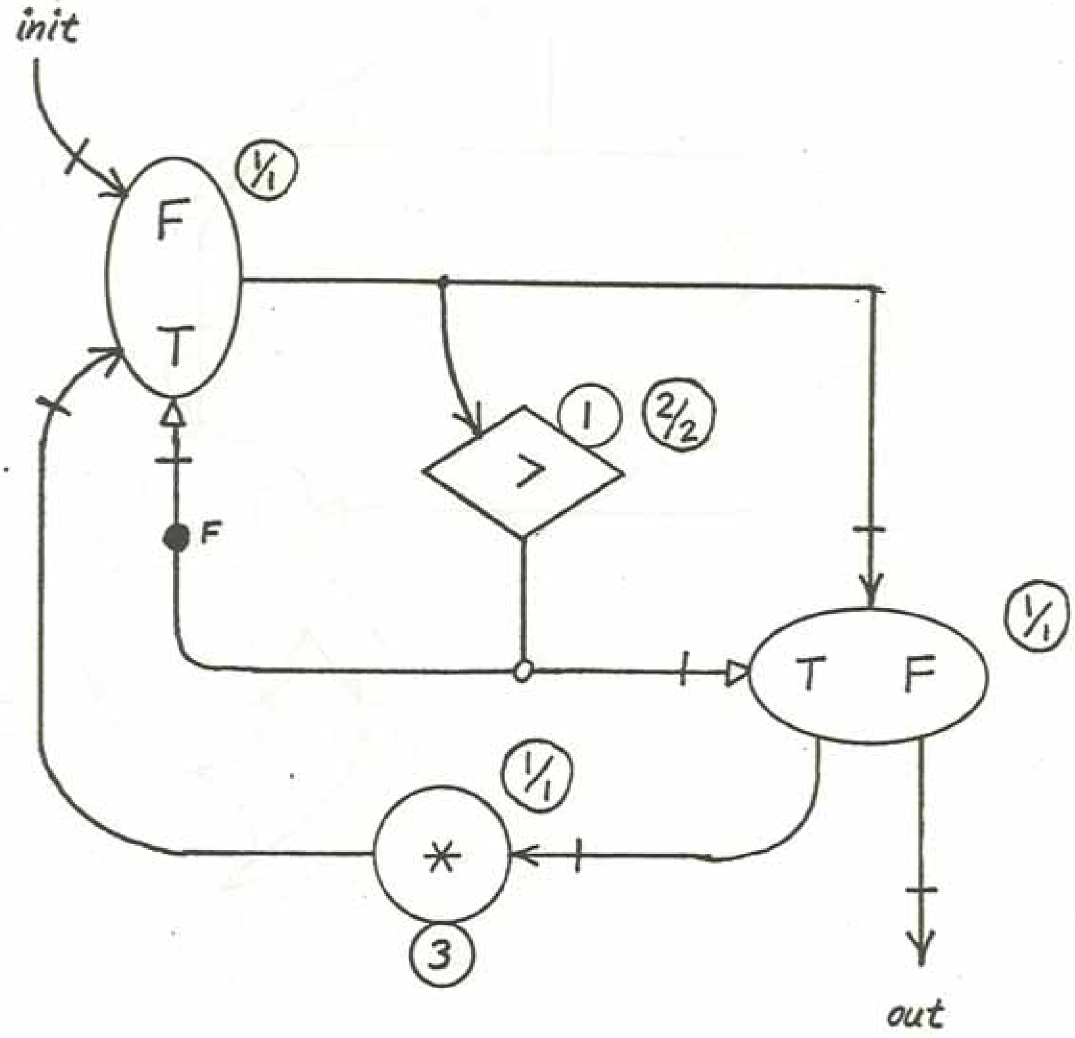


Figure 13. Explicit Acknowledgement of an Iteration

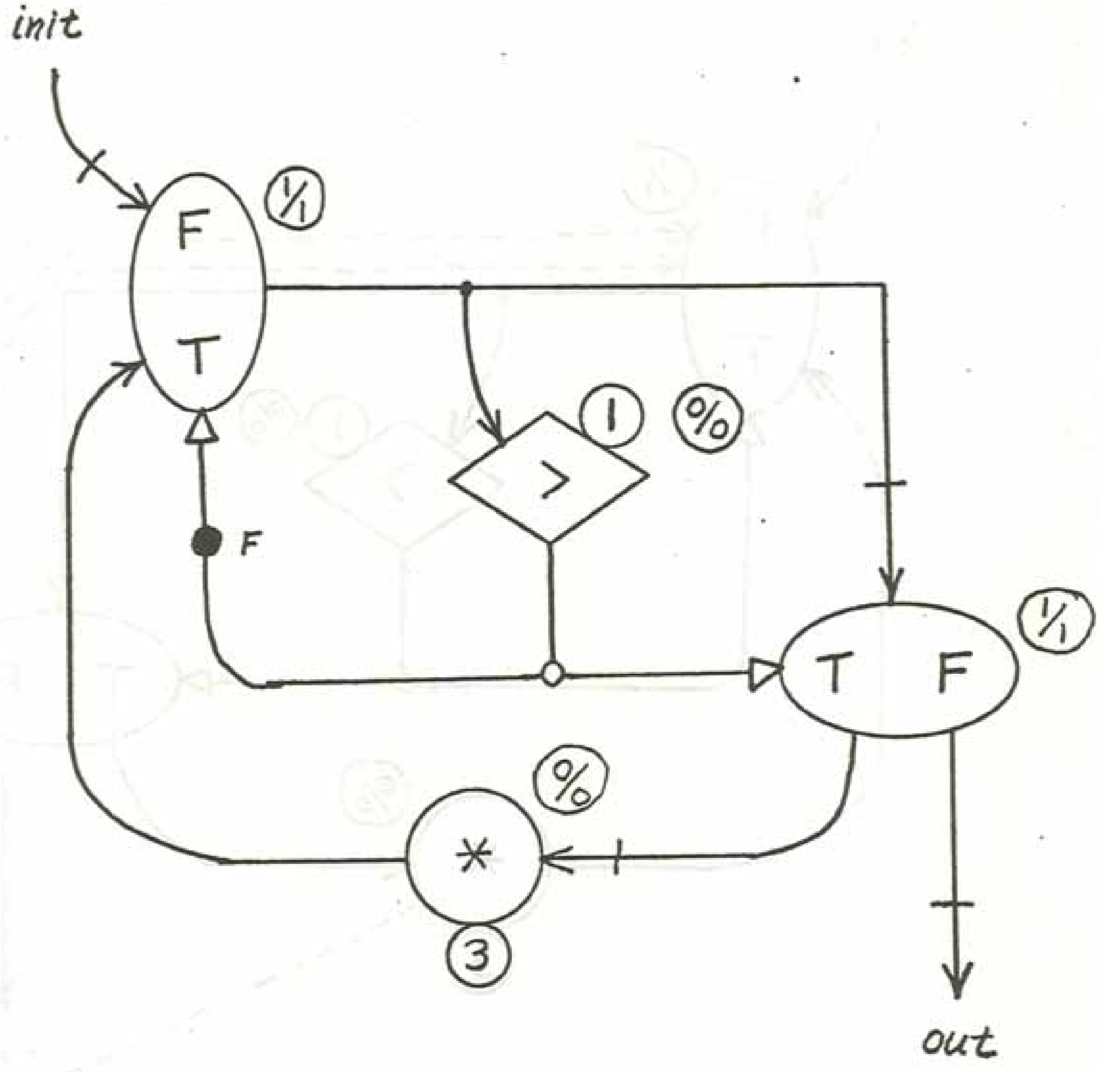
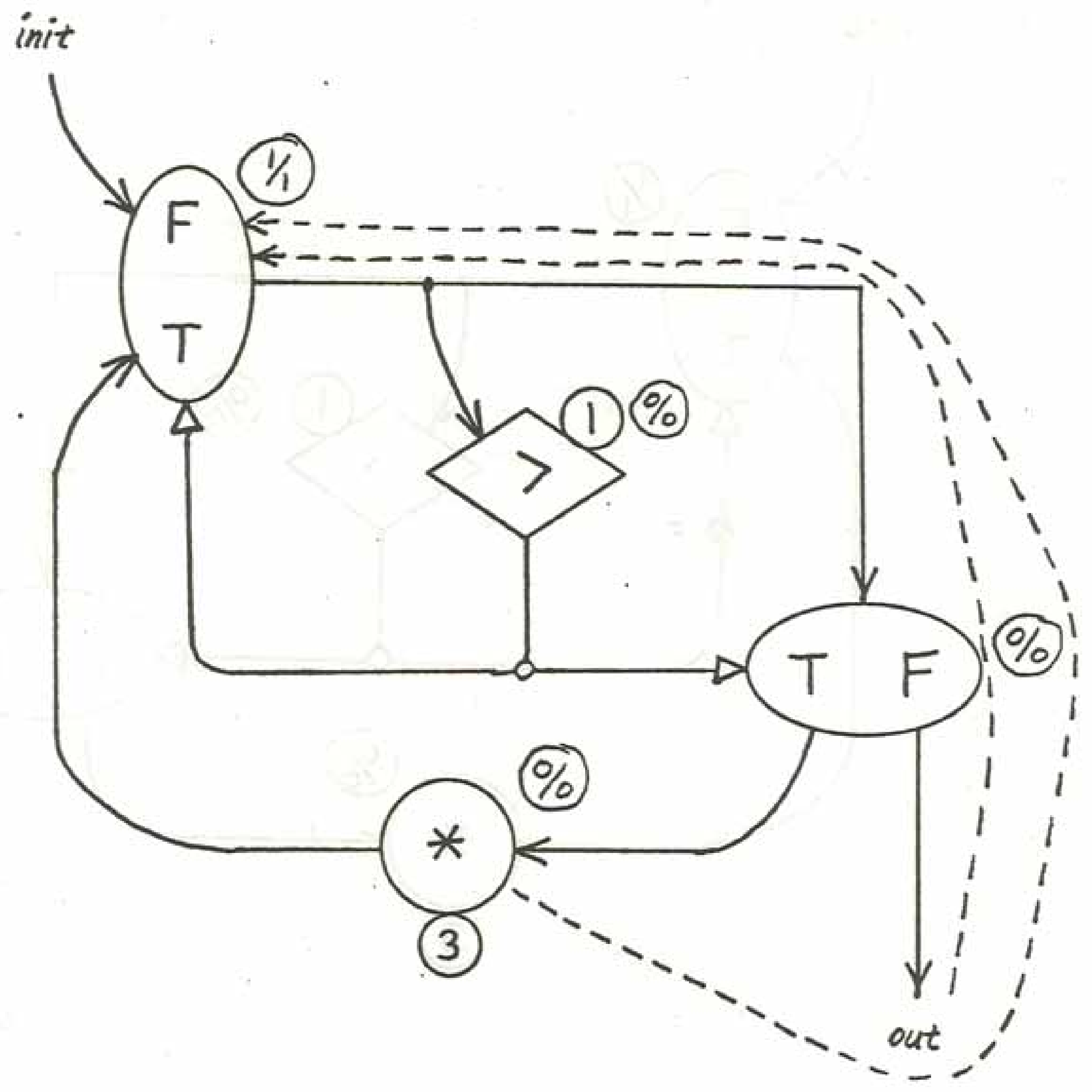


Figure 14. A Fully Ack-minimized Program Graph for an Iteration



that although fewer acknowledges are used in the ack-minimized versions, all three graphs require $4n+3$ time units to compute *out*, where n is the number of loop iterations.

We can optimize this same iteration with respect to speed by using multiple operand sources as shown in Figure 15. This program graph can compute *out* in n fewer steps than the earlier graphs. It requires only $3n+3$ time units. Despite the use of multiple operand sources, this graph is *unsuitable* for execution under implicit acknowledgement generation. The problem lies with the switch actor. Note that it gets its initial operand from the identity actor, I. When the switch fires, it acknowledges either the identity or multiplication actor, depending upon the value of the predicate. *The acknowledge address is not dependent upon the operand source, but rather upon the value of the boolean.* This makes this program graph impossible to implicitly acknowledge. However, explicit acknowledgement is indeed possible, if we ack-minimize first (see Figure 16).

There are no serious disadvantages in using explicit acknowledgement, other than the fact that it permits the execution of graphs which might not be in $\mathcal{J}_{d/a}$ form. This in itself is not a disadvantage, but since these graphs may lack some acknowledgement arcs, their safe behavior is not always guaranteed. However, if we assume that ack-minimization optimizations can be, and are, performed on $\mathcal{J}_{d/a}$ graphs in an algorithmic way which preserves their safety, then we need not be concerned about this issue. We can assume that a properly constructed compiler for a data flow target machine always generates safe code. The use of explicit acknowledgement transfers the responsibility for program safety from the hardware to the compiler and programmers writing machine code.

Figure 15. An Iteration Using Multiple Operand Sources

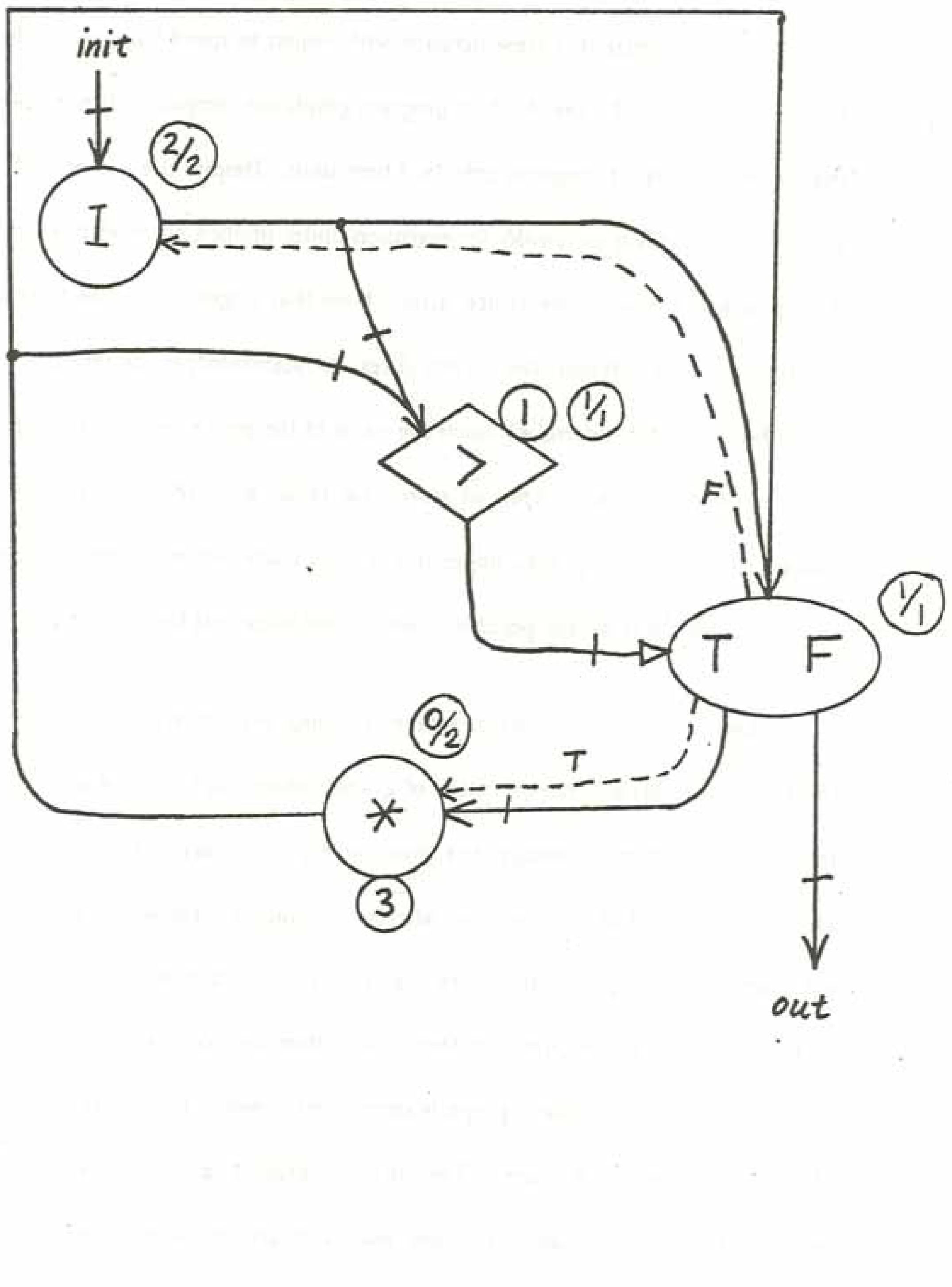
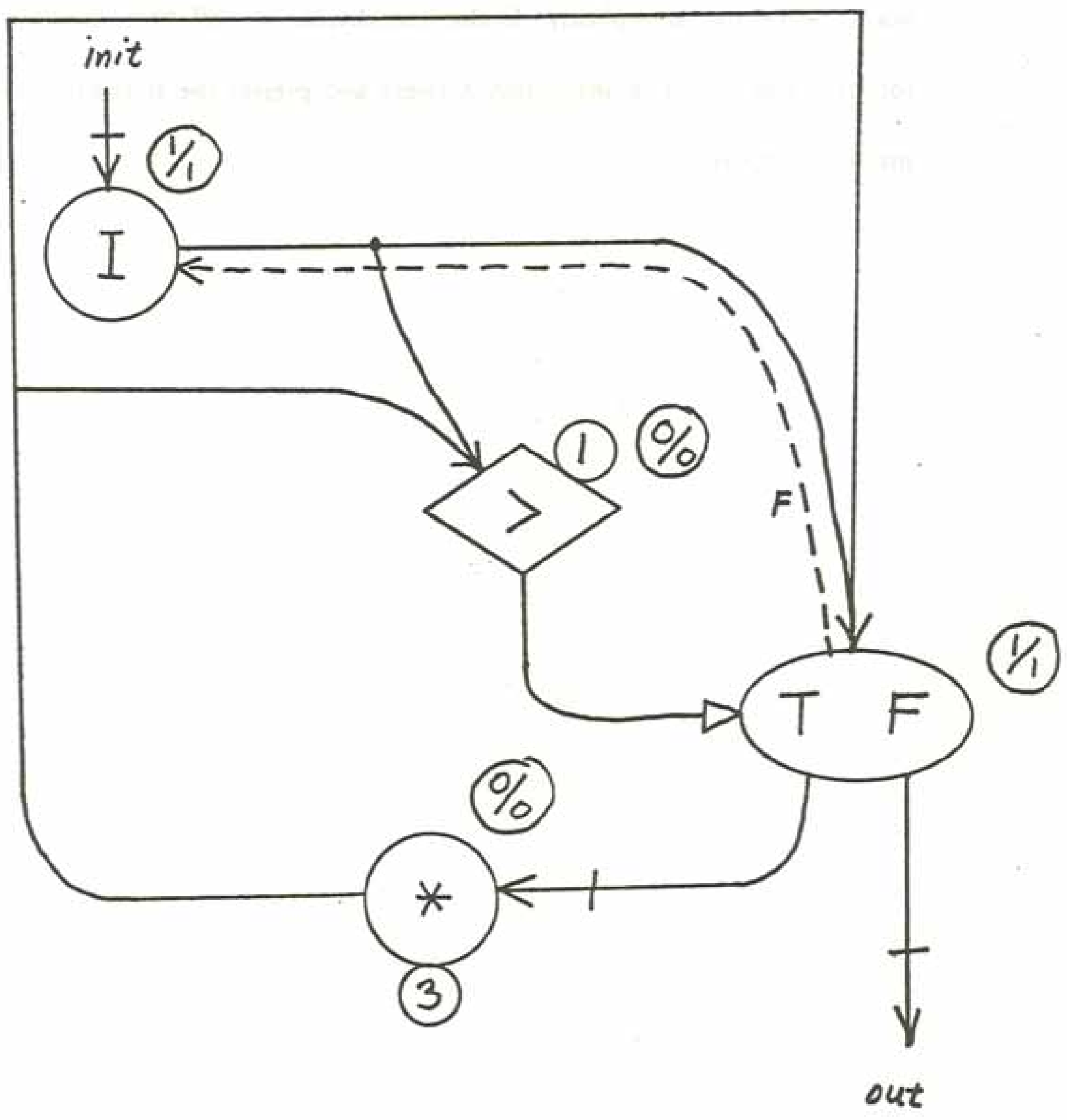


Figure 16. Ack-minimized Version of Figure 15



Due to the advantages offered by explicit acknowledgement over implicit acknowledgement in the area of program optimizations, the Form I machine will generate acknowledge signals explicitly. In the next chapter we will determine the representation for data flow actors in Instruction Memory and present the instruction formats for the machine language.

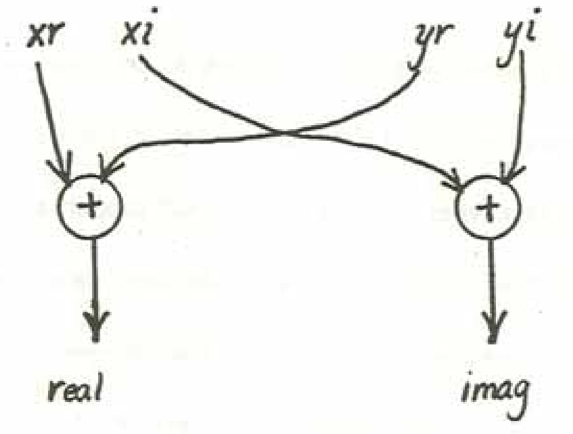
4. Operations and Representation

4.1 Data Type Specifications

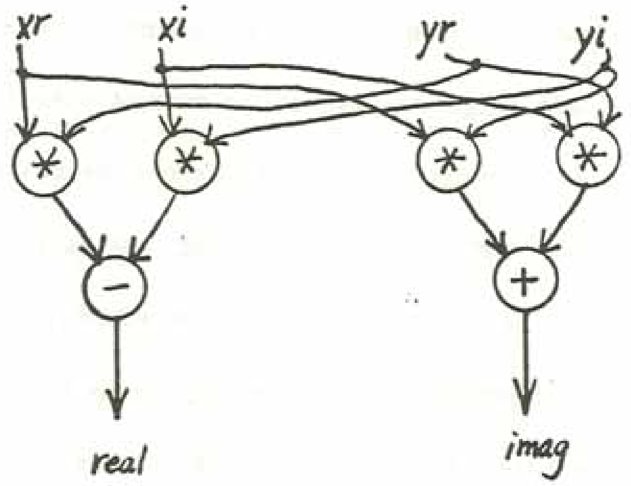
Since the Form I machine will serve primarily as an experimental prototype to assist the development of more advanced data flow models, it was designed to support only scalar operations. The data types that the machine will support are **boolean, integer,** and **real**. It is obvious why these types were chosen as the basic data types for the Form I machine. Boolean values are required for control, and both integer and real data types are needed for performing practical computations. What may need explanation are the reasons for excluding such scalar types as **multiple precision, complex,** and **character**.

Multiple precision and complex data types are not allowed because of storage limitations in the instruction cell, their infrequent use, and their requirements for a more complicated Processing Unit. In any case, complex arithmetic can be simulated with ordinary real and integer operands in the machine without adding much to the total processing time. Consider Figure 17, which shows the data flow graphs for performing complex addition and multiplication. Using real operands, complex addition and multiplication can be performed in only one and two time units, respectively. If we assume that a significant portion of the processing time is involved in transit through the computer, then it takes about the same amount of time to do complex addition with real numbers as with complex operands, and only about twice as long to do complex multiplication with reals. A program graph involving complex arithmetic is converted into real operations and analyzed in Section 4.2.3. Multiple precision and complex data types may be supported on future data flow machine implementations.

Figure 17. Data Flow Graphs for Complex Arithmetic



Complex Addition
$[xr+(xi)\hat{j}]+[yr+(yi)\hat{j}]$



Complex Multiplication
$[xr+(xi)\hat{j}]\times[yr+(yi)\hat{j}]$

Character operands are not permitted because they typically occur in character strings, which should be handled by a Structure Processor and kept in Structure Memory.

Since there is no control flow to interrupt in data flow programs, programming errors are handled by generating special error values. The Form I error values are:

> **boolean** - **undef[boolean]**
>
> **integer** - **undef[integer], pos over[integer], neg over[integer], unknown[integer], zero divide[integer]**
>
> **real** - **undef[real], pos over[real], neg_over[real], pos under[real], neg_under[real], unknown[real], zero_divide[real]**

The element undef[*type*] results when operand values are not in the domain of an operator. The elements pos_over[*type*] and neg_over[*type*] denote values, positive or negative, too large to be represented in the representation of *type*. The element unknown[*type*] indicates the result of a computation that has exceeded the capacity of the implementation, but whose true value is not known to be out of range. The element zero divide[*type*] results from division by zero. The elements pos under[*type*] and neg under[*type*] denote non-zero values, positive or negative, too small to be represented in the representation of *type*.

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

4.2 Representation

4.2.1 The Instruction Cell

An instruction cell consists of two components: an 8-byte (8 bits/byte) *cell state area (CSA)* and a 24-byte *operation packet area (OPA)* (see Figure 18). The instruction cells are collectively organized into *cell blocks*, each of which may hold several instruction cells and has a controlling mechanism that handles the receipt of operand and acknowledge packets from the Distribution Network, controls state changes in the cell state area, and assembles operation packets from the operation packet areas for transmission to the Arbitration Network.

The cell state area holds data concerning the status of the instruction cell. The data fields of the CSA are:

Cell Used (CU): Set to 1 if the instruction is non-null; otherwise 0.

All Acknowledgements Received (AAR): Set to 1 when AE=AR; otherwise 0; reset to 0 whenever an operation packet is transmitted.

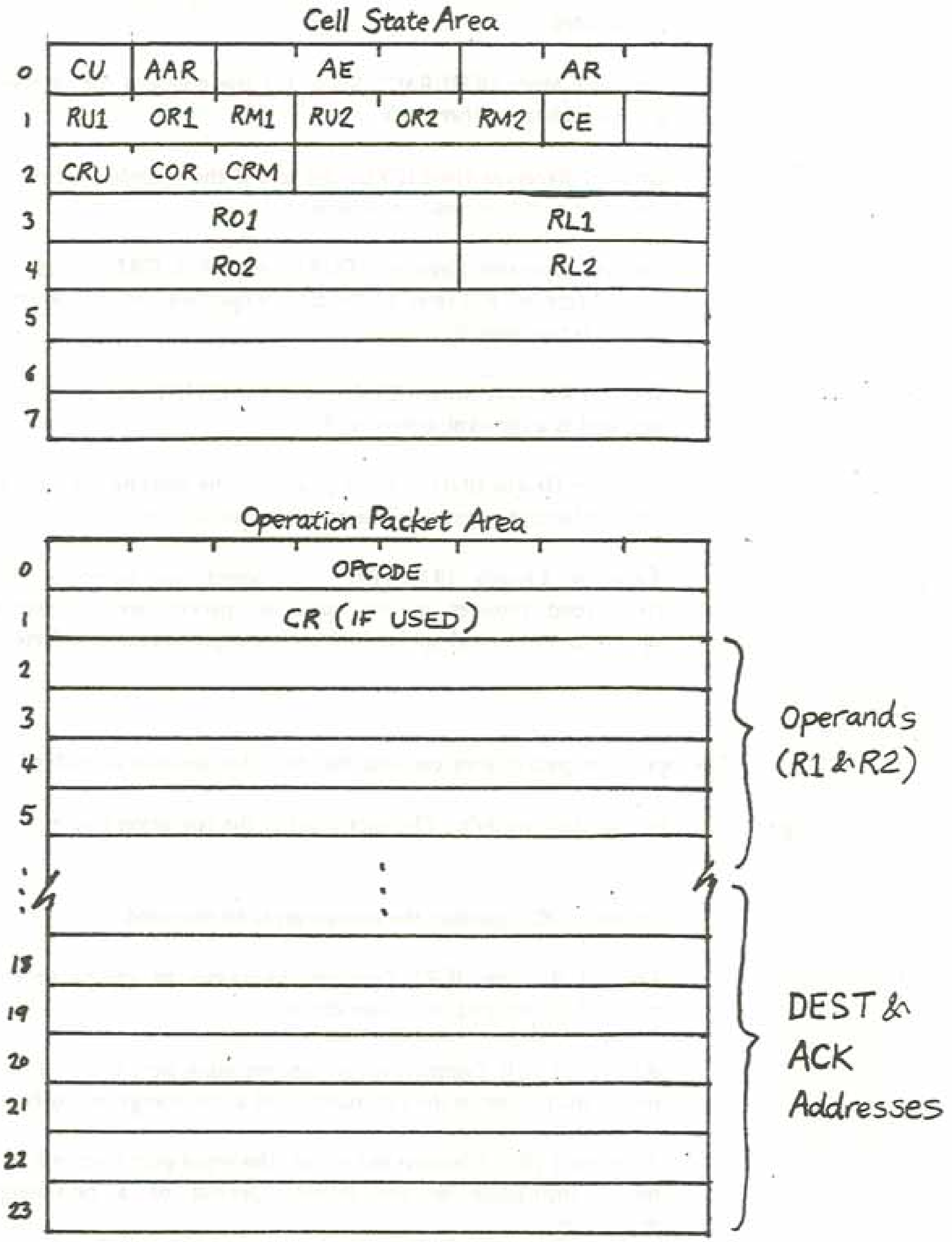
Acknowledgements Expected (AE): Number (0-7) of acknowledge packets expected.

Acknowledgements Received (AR): Number (0-AE) of acknowledge packets received; incremented each time an acknowledge packet arrives. Set to 0 when an operation packet is transmitted.

Cell Enabled (CE): Set to 1 if each receiver holds an operand and all acknowledgements have been received; otherwise 0. Set to 0 when the operation packet is transmitted.

Receiver Used (RU1/RU2): Set to 1 if the instruction requires an operand for receiver 1/2; otherwise 0.

Figure 18. Instruction Cell Format



Operand Received (OR1/OR2): Set to 1 if receiver 1/2 has received an operand; set to 0 when an operation packet is transmitted.

Receiver Mode (RM1/RM2): Set to 1 if the operand for receiver 1/2 is a constant; otherwise 0.

Control Receiver Used (CRU): Set to 1 if the opcode specifies a merge or switch instruction; otherwise 0.

Control Operand Received (COR): Set to 1 if CRU=1 and the control receiver has received the control operand; set to 0 when a packet is transmitted.

Control Receiver Mode (CRM): Set to 1 if CRU=1 and the control operand is a constant; otherwise 0.

Receiver Origin (RO1/RO2): A pointer to the starting location of the first/second receiver in the operation packet area.

Receiver Length (RL1/RL2): The length in bytes of the first/second receiver in the operation packet area. Boolean operands: 1 byte; real operands: 4 bytes; integer operands: 4 bytes.

The operation packet area contains the data that become assembled into operation packets by the cell block module. The data fields of the operation packet area are:

Opcode (OC): Specifies the instruction to be executed.

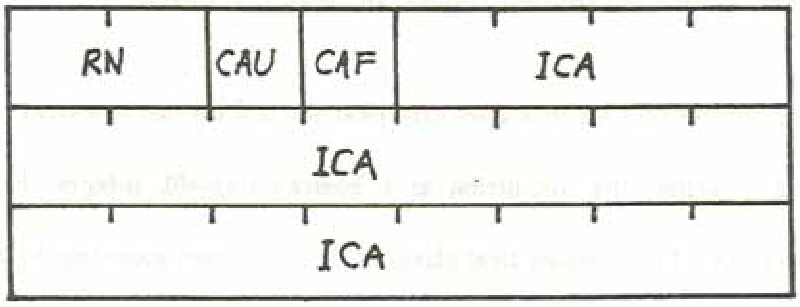
Control Receiver (CR): Contains the switch or merge control operand, if required; otherwise absent.

Receiver 1 (R1): Contains either the true-input-port operand of a merge instruction or the first operand of a non-merge instruction.

Receiver 2 (R2): Contains either the false-input-port operand of a merge instruction or the second operand of a non-merge instruction.

Destination and Acknowledge Addresses (DEST/ACK): Each destination/acknowledge address has the following format:
Receiver Number (RN): 0 - Not used (acknowledge addresses only); 1 - Receiver 1 (destination addresses only); 2 - Receiver 2 (destination addresses only); 3 - Control receiver (destination addresses only).
Conditional Address Used (CAU): 0 - Not used; 1 - Send an acknowledge/operand packet to this address if the control value matches the boolean value of the conditional address flag.
Conditional Address Flag (CAF): 0 - False; 1 - True.
Instruction Cell Address (ICA): 20-bit address.

Figure 19. Destination and Acknowledge Address Format



Data type information is not kept within the instruction cell. It is assumed that the operation code identifies the type of each operand, that the compiler guarantees that instruction cells are always loaded with operands that are compatible with their opcodes, and that the compiler always generates correct code.

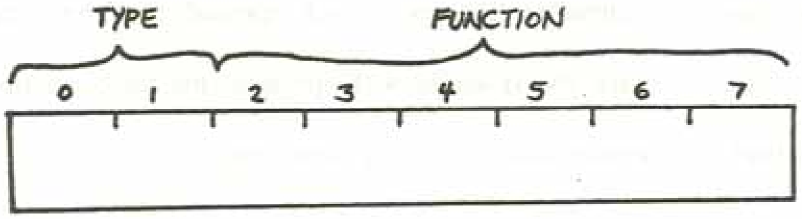
The values assigned to an instruction cell are initialized by a von Neumann host computer that sends initialization packets to the cell block holding that cell. The initialization packets can be sent only after the host computer has disabled the operation of the cell block with a *disable packet*. The host computer may also request state information for any instruction cell by transmitting a *dump packet* to the cell block in which it is held.

Whenever an instruction cell receives a result or acknowledge packet, the cell block must update the cell's state variables and check its enabling conditions. If an instruction cell becomes enabled, the cell block assembles an operation packet from the cell's OPA and sends a signal to the Arbitration Network indicating that it is ready to transmit a packet. If the Arbitration Network signals that it is ready to accept the packet, the packet is transmitted; if not, the cell block waits for a *ready signal* before transmitting.

4.2.2 Operations and Instruction Formats

The opcode consists of a 2-bit *type* field and a 6-bit *function* field. The value of the *type* field identifies the instruction as a boolean (*type*=0), integer (*type*=1), or real (*type*=2) operation. The *function* field classifies the instruction according to its functionality.

Figure 20. Opcode Format



Bits 0 and 1 comprise the type field and bit 7 indicates whether or not a control operand is required by the instruction, as in a merge or switch instruction. The Form I instruction set is presented in the tables that appear in Appendix II. For each instruction, the opcode is given first, followed by its name and two numbers. The first number is the number of arguments (operands) the instruction requires. The second number is the number of destination addresses that may be loaded, assuming that all operands are immediately acknowledged. The actual number of destination addresses may exceed this number if ack-minimization reduces the number of acknowledge addresses that get loaded. The maximum number of destination addresses, n, that may be loaded into an instruction cell is:

$$n = \mathrm{FLOOR}\left(\frac{23-(3a+c+op1+op2)}{3}\right)$$

where a is the number of acknowledge addresses loaded, c is the number of control operands used (0 or 1), and $op1$ and $op2$ are the lengths (in bytes) of the two non-control operands.

4.2.3 An Example: FFT Phase Factor Generation

Dennis, Leung, and Misunas present a data flow program for computing an 8-point Fast Fourier Transform in [9]. Their program makes use of complex data types and assumes that the machine on which it executes has processing units that are capable of handling complex operations. The Phase Factor Generation section of their program has been reproduced in Figure 21. The inputs to the Phase Factor Generation section are the variables *c*:complex, *n*:integer, and *b*:boolean. The output is the variable *w*:complex. The value *q* is an integer constant. If we wish to encode this graph into our Form 1 machine code, it is first necessary to convert it into a schema that uses only real, integer, and boolean data types. Such a schema is illustrated in Figure 22. The complex variables *c* and *w* are replaced by the real variables *cr*:real, *ci*:real, *wr*:real, and *wi*:real. Note that

Figure 21. Phase Factor Generation with Complex Types

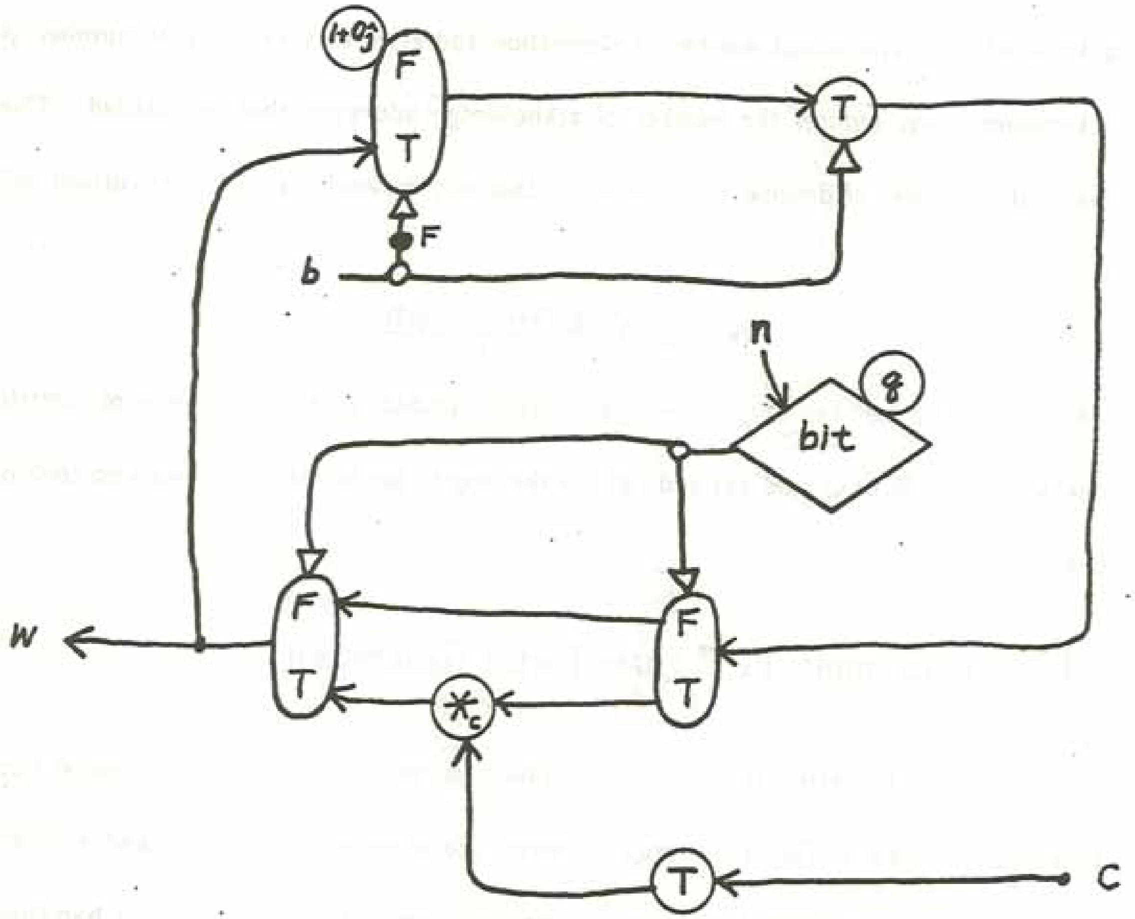
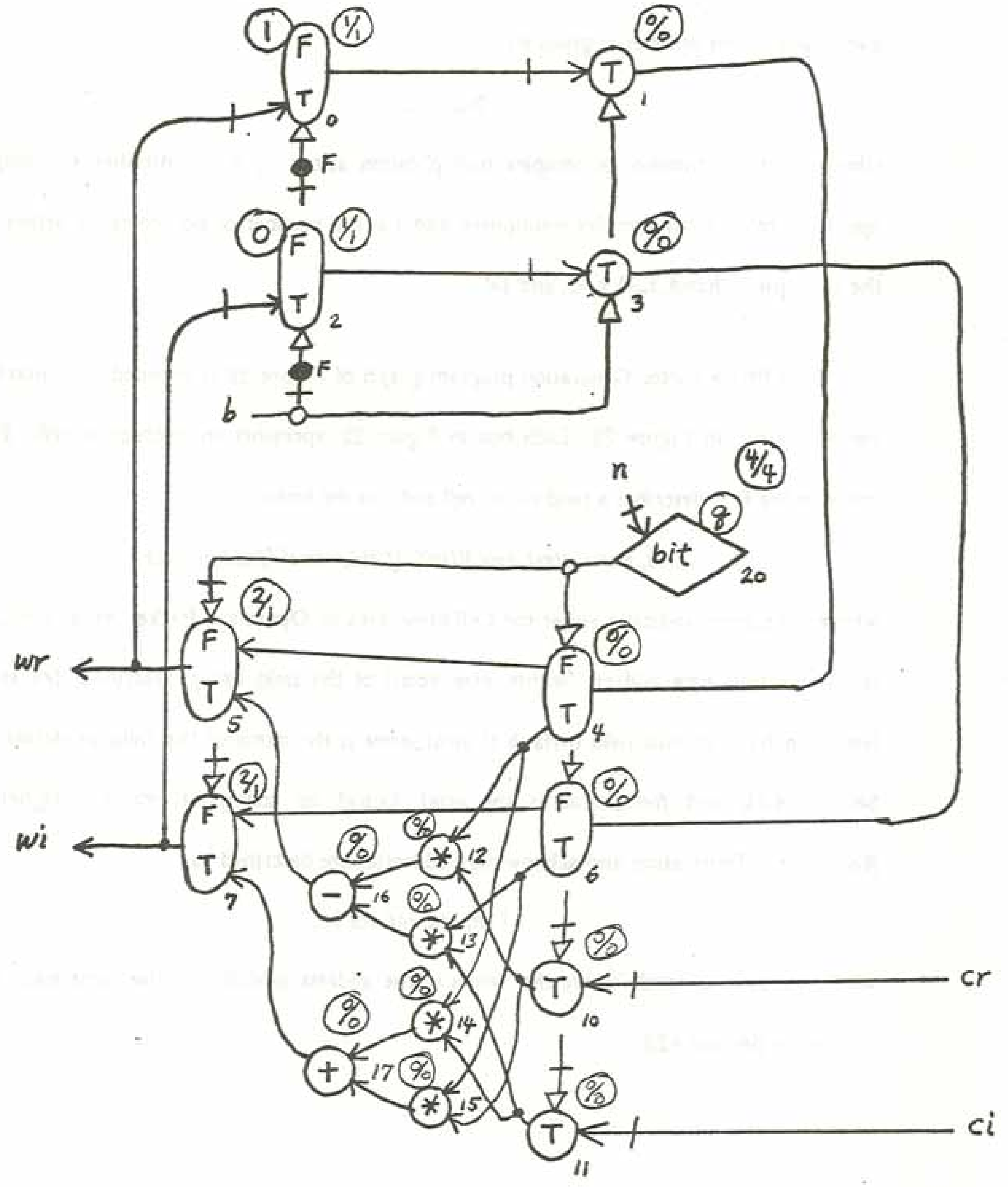


Figure 22. Phase Factor Generation without Complex Types



the number of actors increases from 7 to 17 with the elimination of complex types. The number of actors required by a program graph without complex data types is easily computable from the number of actors in the equivalent program graph with complex data types. This number is given by

$$6c_m + 2c_0 + s.$$

where c_m is the number of complex multiplication actors, c_0 is the number of complex operators other than complex multipliers, and s is the number of non-complex actors. In the example at hand, c_m=1, c_0=5, and s=1.

The Phase Factor Generation program graph of Figure 22 is encoded in a machine representation in Figure 23. Each box in Figure 23 represents an instruction cell. Each entry in the box describes a field of the cell and has the form

area_name: start_byte {(*len*)}: {*field_name*} [*field_value*]

where *area_name* indicates either the Cell State Area or Operation Packet Area, *start_byte* is the starting byte address (within *area_name*) of the field being described; *len* is the length in bytes of said field (default 1); *field_name* is the name of the field as defined in Section 4.2.1; and *field_value* is the octal, logical, or conceptual value assigned to *field_name*. Destination and acknowledge addresses are described as

[*rn*, *cau*, *caf*, ica]

where *rn*, *cau*, *caf*, and *ica* are the values of the address subfields of the same names, as defined in Section 4.2.1.

Figure 23. Phase Factor Generation: Machine Representation
Cell addresses correspond to actor numbers in Figure 22.
Unspecified bytes are padded with zeros.
All numbers are octal; "T" = true; "F" = false.

Cell 0
```
CSA:0: [3 1 1]
CSA:1: [2 3 6]
CSA:2: [2 4 0]
CSA:3: [0 2 4]
CSA:4: [0 6 4]
OPA:0: OC[R-Merge]
OPA:1: CR[F]
OPA:2(4): R1[ ]
OPA:6(4): R2[1]
OPA:12(3): DEST1[1,F,F,1]
OPA:15(3): ACK1[0,F,F,5]
OPA:20(3): ACK2[0,F,F,addr(b)]
```

Cell 2
```
CSA:0: [3 1 1]
CSA:1: [2 3 6]
CSA:2: [2 4 0]
CSA:3: [0 2 4]
CSA:4: [0 6 4]
OPA:0: OC[R-Merge]
OPA:1: CR[F]
OPA:2(4): R1[ ]
OPA:6(4): R2[0]
OPA:12(3): DEST1[1,F,F,3]
OPA:15(3): ACK1[0,F,F,7]
OPA:20(3): ACK2[0,F,F,addr(b)]
```

Cell 1
```
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): R1[ ]
OPA:6(3): DEST1[1,T,T,4]
OPA:11(3): ACK1[0,F,F,0]
```

Cell 3
```
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): R1[ ]
OPA:6(3): DEST1[1,T,T,6]
OPA:11(3): ACK1[0,F,F,2]
```

Cell 4
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(3): DEST1[2,T,F,5]
OPA:11(3): DEST2[1,T,T,12]
OPA:14(3): DEST3[1,T,T,14]

Cell 5
CSA:0: [2 2 1]
CSA:1: [2 2 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
CSA:4: [0 6 4]
OPA:0: OC[R-Merge]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(4): R2[ ]
OPA:12(3): DEST1[1,F,F,0]
OPA:15(3): DEST2[1,F,F,addr(*wr*)]
OPA:20(3): ACK1[0,F,F,20]

Cell 6
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(3): DEST1[2,T,F,7]
OPA:11(3): DEST2[1,T,T,13]
OPA:14(3): DEST3[1,T,T,15]

Cell 7
CSA:0: [2 2 1]
CSA:1: [2 2 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
CSA:4: [0 6 4]
OPA:0: OC[R-Merge]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(4): R2[ ]
OPA:12(3): DEST1[1,F,F,2]
OPA:15(3): DEST2[1,F,F,addr(*wi*)]
OPA:20(3): ACK1[0,F,F,20]

Cell 10
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
CSA:4: [0 0 0]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(3): DEST1[2,T,T,12]
OPA:11(3): DEST2[2,T,T,15]
OPA:14(3): ACK1[0,F,F,addr(*cr*)]
OPA:17(3): ACK2[0,F,F,20]

Cell 11
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [2 0 0]
CSA:3: [0 2 4]
OPA:0: OC[R-Switch]
OPA:1: CR[ ]
OPA:2(4): RI[ ]
OPA:6(3): DEST1[2,T,T,13]
OPA:11(3): DEST2[2,T,T,14]
OPA:14(3): ACK1[0,F,F,addr(*ci*)]
OPA:17(3): ACK2[0,F,F,20]

Cell 12

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Mult]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[1,F,F,16]
```

Cell 13

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Mult]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[2,F,F,16]
```

Cell 14

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Mult]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[1,F,F,17]
```

Cell 15

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Mult]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[2,F,F,17]
```

Cell 16

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Sub]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[1,F,F,5]
```

Cell 17

```
CSA:0: [3 0 0]
CSA:1: [2 2 0]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[R-Add]
OPA:1(4): R1[ ]
OPA:5(4): R2[ ]
OPA:11(3): DEST1[1,F,F,7]
```

Cell 20
CSA:0: [3 4 4]
CSA:1: [2 3 4]
CSA:2: [0 0 0]
CSA:3: [0 1 4]
CSA:4: [0 5 4]
OPA:0: OC[I-Bit]
OPA:1(4): R1[ ]
OPA:5(4): R2[q]
OPA:11(3): DEST1[1,F,F,21]
OPA:14(3): DEST2[3,F,F,4]
OPA:17(3): DEST3[3,F,F,6]
OPA:22(3): ACK1[1,F,F,addr(n)]

Cell 21
CSA:0: [3 0 0]
CSA:1: [2 0 0]
CSA:2: [0 0 0]
CSA:3: [0 1 1]
CSA:4: [0 0 0]
OPA:0: OC[B-Ident]
OPA:1: R1[ ]
OPA:2(3): DEST1[3,F,F,5]
OPA:5(3): DEST2[3,F,F,7]
OPA:10(3): DEST3[3,F,F,10]
OPA:13(3): DEST4[3,F,F,11]

Comments.

(1) This representation is ack-minimized.

(2) The bit predicate operates as follows:

bit(n,q) = if (the n-th bit of q=1) then true
else false

(3) The instruction cell configuration requires an additional identity actor (B-Ident) to distribute to cells 5, 7, 10, and 11 because cell 20 lacks the necessary space for all six destinations.

(4) If we assume every instruction cell requires one time unit to fire and that the computation begins with the firing of cells 0 and 2, the output values wr and wi are computable in

$$3m + 2F + 4T - 1 \text{ time units,}$$

where m is the number of consecutive times that b is true ($m \geq 1$), T is the number of times that the bit predicate is true, and F is the number of times that the bit predicate is false ($T + F = m$).

5. Conclusions and Suggestions for Future Research

An instruction set for a Form I data flow processor has been presented in this thesis. We have also discussed the merits and drawbacks of implicit and explicit acknowledgement generation, and have selected the explicit method as the technique for generating acknowledge signals. We have also characterized the storage utilization of instruction cells in terms of 8 byte cell state areas and 24 byte operation packet areas.

An attempt was made to make the instruction formats as upward compatible as possible for use with future generations of data flow machines. Many unused bit patterns are available for specifying the structure processing opcodes of the Form 2 machine. Although 20 bits may seem excessive for addressing small Form I instruction memories, the potential for addressing up to 1024K cells in future machines without modifying the format is viewed as desirable.

True and false gates were not included in the instruction set because the switch instruction eliminates their need. A true (false) gate is equivalent to a switch instruction with no false (true) destinations. A merge instruction was included in the instruction set despite the belief that it can be replaced, in the context of translated ADFL conditional and iterative expressions, by other instructions that use multiple operand sources. It was included to ensure that the use of merge gates in arbitrary data flow programs could be easily and directly encoded, maintaining a close correspondence between data flow actors and the machine instructions. If experience shows that merge instructions are unnecessary and have no practical value, they can be eliminated at a later time.

The FFT Phase Factor Generation graph was chosen for encoding and analysis because it appears to be a representative and practical computation suitable for the data flow approach. The analysis of the machine level representation (Figure 23) results in the following conclusions: (1) A single 24-byte OPA is adequate in size for holding destination and acknowledge address information for almost all the cells; and (2) Ack-minimization is important, not only from the standpoint of eliminating unnecessary acknowledge packets, but also so that more OPA space exists for those instructions which have a large number of destination addresses.

The size of the operation packet area should be reconsidered when more powerful data flow machines are under development. A larger operation packet area means more operands, more addresses, and a greater variety of instructions. A larger operation packet area would make it possible to combine instructions that frequently occur together into one, such as "add-and-switch," and to implement 3-or-more operand instructions, such as 3-operand multiplication and addition (among others). On the other hand, experience may indicate that larger operation packet areas waste an unacceptable amount of storage and do not cost justify the advantages they bring about.

The performance of programs written in Form 1 code needs to be carefully analyzed and quantified. Any problems in the design of the Form 1 instruction set should be corrected before a Form 2 instruction set specification is attempted. The issue of VAL **forall** translation still needs to be addressed, and the procedures for structure processing more carefully defined. The implementation of stream data types and stream actors on a Form 1 computer, as proposed by Weng [14], is another major area of future study.

Appendix I - Error Values

Table I. Error Values

Value	Name
10000010	unknown
10000011	undef
10011100	pos_over
10001100	neg_over
10010100	pos_under
10000100	neg_under
10000001	zero_divide

Appendix II - Operation Codes

Table II. Boolean Operations

Opcode	Name	No. of Args.	No. of Dests.
00000010	B-Ident	1	6
00000100	B-Not	1	6
00000110	B-Xor	2	5
00001000	B-And	2	5
00001010	B-Or	2	5
00000001	B-Switch	2	5
00000011	B-Merge	3	3
00100010	B-is-error	1	6
00100100	B-is-undef	1	6
00110110	B-equal	2	5
00111000	B-not-equal	2	5

Table III. Integer Operations

Opcode	Name	No. of Args.	No. of Dests.
01000010	I-Ident	1	5
01000100	I-Negate	1	5
01000110	I-Add	2	3
01001000	I-Mult	2	3
01001010	I-Sub	2	3
01001100	I-Div	2	3
01001110	I-Exp	2	3
01010000	I-Mag	1	5
01010010	I-Max	2	3
01010100	I-Min	2	3
01010110	I-Convert-to-Real	1	5
01011000	I-Mod	2	3
01011010	I-Bit	2	3
01011100	I-Shift	2	3
01000001	I-Switch	2	3
01000011	I-Merge	3	1
01100010	I-is-error	1	5
01100100	I-is-undef	1	5
01100110	I-is-posover	1	5
01101000	I-is-negover	1	5
01101010	I-is-over	1	5
01110010	I-is-arith-err	1	5
01110100	I-is-zero-div	1	5
01110110	I-equal	2	3
01111000	I-not-equal	2	3
01111010	I-greater-than	2	3
01111100	I-greater-than-or-eq	2	3

Table IV. Real Operations

Opcode	Name	No. of Args.	No. of Dests.
10000010	R-Ident	1	5
10000100	R-Negate	1	5
10000110	R-Add	2	3
10001000	R-Mult	2	3
10001010	R-Sub	2	3
10001100	R-Div	2	3
10001110	R-Exp	2	3
10010000	R-Mag	1	5
10010010	R-Max	2	3
10010100	R-Min	2	3
10010110	R-Truncate	1	5
10011110	R-Exp-w-Int	2	3
10000001	R-Switch	2	3
10000011	R-Merge	3	1
10100010	R-is-error	1	5
10100100	R-is-undef	1	5
10100110	R-is-posover	1	5
10101000	R-is-negover	1	5
10101010	R-is-over	1	5
10101100	R-is-posunder	1	5
10101110	R-is-negunder	1	5
10110000	R-is-under	1	5
10110010	R-is-arith-error	1	5
10110100	R-is-zero-div	1	5
10110110	R-equal	2	3
10111000	R-not-equal	2	3
10111010	R-greater-than	2	3
10111100	R-greater-than-or-eq	2	3

Bibliography

[1] Ackerman, W.B., and J.B. Dennis. "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual." Technical Report TR-218, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 13, 1979.

[2] Amikura, K. *A Logic Design for the Cell Block of a Data-Flow Processor.* Technical Memorandum TM-93, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1977.

[3] Arvind, and K.P. Gostelow. "A Computer Capable of Exchanging Processors for Time." *Information Processing '77*, North Holland, New York, 1977, pp. 849-854.

[4] Brock, J.D. *Operational Semantics of a Data Flow Language.* Technical Memorandum TM-120, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1978.

[5] Davis, A.A. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine." *Proceedings of the 5th Annual Symposium on Computer Architecture*, IEEE, April 1978, pp. 210-215.

[6] Dennis, J.B. "First Version of a Data Flow Procedure Language." *Programming Symposium Proceedings, Colloque sur la Programmation*, Paris, April 9-11, 1974. *Lecture Notes in Computer Science, Vol. 19*, Springer-Verlag, New York, 1974, pp. 362-376.

[7] Dennis, J.B., and D.P. Misunas. "A Computer Architecture for Highly Parallel Signal Processing." *Proceedings of the ACM 1974 National Conference*, 1974, pp. 402-409.

[8] Dennis, J.B., and D.P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor." *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, IEEE, New York, January 1975, pp. 126-132.

[9] Dennis, J.B., C. Leung, and D.P. Misunas. "Specification of the Instruction Cell Block for a Data Flow Processor." Data Flow Design Note 1, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 16, 1975.

[10] Montz, L. *Safety and Optimization Transformations for Data Flow Programs.* Deparatment of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, S.M. Thesis in preparation.

[11] Patil, S.S. "Closure Properties of Interconnections of Determinate Systems." *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970, pp.107-116.

[12] Plas, A., D. Conte, O. Gelly, and J.C. Syre. "Lan System Architecture: A Parallel Data-Driven Processor Based on Single Assignment." *Proceedings of the 1976 International Conference on Parallel Processing*, IEEE, August 1976, pp. 293-302.

[13] Rodriguez, J.E. *A Graph Model for Parallel Computation*. Technical Report MAC TR-64, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1979.

[14] Weng, K.-S. *Stream-Oriented Computation in Recursive Data Flow Schemas*. Technical Memorandum TM-68, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1975.