MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 134 -1

A Highly Parallel Processor Using a Data Flow Machine Language

by

Jack B. Dennis
Clement K. Leung
David P. Misunas

This work was supported by grant DCR75-04060 from the National Science Foundation for research on data flow computer architecture.

January 1977

(Revised June 1979)

A Highly Parallel Processor Using a Data Flow Machine Language*

by

Jack B. Dennis
Clement K. Leung
David P. Misunas

Abstract: A computer based on the data flow principle executes instructions in response to the arrival of their operands -- there is no notion of sequential control flow. Because programs expressed in data flow form are free of sequencing constraints other than those imposed by the flow of operands between instructions, a processor using a data flow program representation can be designed to achieve highly parallel operation through concurrent execution of program parts which have no data dependencies. A graphical data flow language and a corresponding architecture for a highly parallel processor are presented in this paper. The language is illustrated by expressing a Fast Fourier Transform algorithm in data flow form. A machine language program suitable for executing the FFT algorithm on the data flow processor is derived, and the potential performance of the processor for this computation is discussed.

*This work was supported by grant DCR75-04060 from the National Science Foundation for research on data flow computer architecture.

I. Introduction

Most efforts to devise computer architectures for highly parallel computation have retained the traditional concept of sequential control flow in their machine level program representations. On one hand a very limited level of parallelism is achieved through use of multiprocessor organizations. On the other hand, parallel operation is achieved in single instruction stream machines either through analysis of the instruction stream as in the IBM 360/195 [5], or through use of specialized data formats and operations as in machines like the Texas Instruments Advanced Scientific Computer [35] which have pipelined processing units, and in array machines such as the Illiac IV [9] and Staran [10].

An alternate approach is to use an architecture able to exploit parallelism on a global basis through use of a machine level program representation based on the concept of data flow. In such a machine, the execution of instructions is driven by the flow of data, and any instructions that are not data-dependent may be executed concurrently.

The concept of data-directed instruction execution has appeared several times in the literature; in particular, see the reports of Shapiro, Saint and Presberg [33], Seeber and Lindquist [32] and Miller and Cocke [25]. However, these early papers left many questions unanswered and failed to relate proposed architecture to a well-defined level of programming language expressive power.

Later, work in the area of program schemata, especially that of Karp and Miller [21, 22] led to development of the data flow program graphs [14, 29] and related abstract models for data-driven programs [4, 8, 24]. This work led to the use of data flow program graphs as the basis for the conception and development of several computer architectures having potential for extensive exploitation of parallelism in computation. These early projects include the work of Davis at the University of Utah [13], Arvind and Gostelow at the University of California at Irvine [6], and a closely related project at Toulouse, France [28], based on the single assignment idea of Tesler and Enea [34]. Interest in the data driven approach to computer architecture has spread recently with projects at the Texas Instruments Co., at Manchester and Newcastle Universities in England, and a second project at the University of Utah.

In the Computation Structures Group of the MIT Laboratory for Computer Science, two of the authors have designed several hypothetical machines that implement specific data flow languages [16, 17], and the architecture of a data flow multiprocessor that implements a high order data flow language has been presented in the doctoral thesis of Rumbaugh [30, 31]. The subject of the present paper is a comprehensive treatment of a data flow processor in which the machine language programs are a direct encoding of programs expressed as data flow program graphs, and an analysis of its performance potential for a particular computation of considerable importance -- the Fast Fourier Transform.

Data flow program graphs as a representation for computation that exposes concurrency are developed in Section II. In illustration, a data flow program for the Fast Fourier Transform algorithm is developed. The overall architecture and principles of operation of the data flow processor are presented in Sections III and IV. The coding of the FFT algorithm as a machine level program for the data flow processor is developed in Section V. The structure of the routing networks that convey information between sections of the processor is discussed in Section VI, where we also estimate the performance potential of our hypothetical processor for the Fast Fourier Transform. In the concluding section we discuss improvements and extensions of our architectural concepts.

II. Data Flow Program Graphs

We envision that a user of a data flow processor would express his programs in a textual language and a translation program would be used to generate the machine level program representation directly. The design of a textual source language and a translator to match the qualities of a data flow computer is an interesting problem in itself and has been addressed in a thesis report by Weng [36] and in [3], but further discussion is beyond the scope of this paper. To illustrate the concepts of data-driven computation, we have chosen for presentation in this paper a graphical representation of data flow programs, an example of which is given in Figure 1. These data flow program graphs are convenient for representing the structure of programs prepared for execution on the data flow processor and for studying their properties. In later sections, we develop a program graph to represent a data flow program for an FFT algorithm and also derive from it a machine language representation of the algorithm.

The nodes of a data flow program graph are of two kinds called *actors* and *links*. Informally, actors perform elementary computational steps and pass results to succeeding actors by way of the links.

For a formal discussion of the behavior of data flow program graphs we consider *configurations* of the program graph in which tokens, represented by large solid dots, are associated with certain arcs of the graph. Each token carries a value which is an element of some data type. Several configurations of the program graph in Fig. 1 are shown in Fig. 2. The behavior of a data flow program graph is specified by *firing rules* which specify the possible sequences of configurations that may describe computations by the program graph. For all links and actors of data flow program graphs (with an exception noted later), the firing rule is the following: (1) A node (an actor or a link) is said to be *enabled* if a token is present on each of its input arcs and no token is present on any of its output arcs; (2) any enabled node may be chosen to "fire"; (3) firing a link means removing the token from the input arc and associating its value with tokens placed on each of its output arcs; (4) firing an actor means that tokens are removed from each of the actor's input arcs, the values associated with the input tokens are used to determine a result, and a token

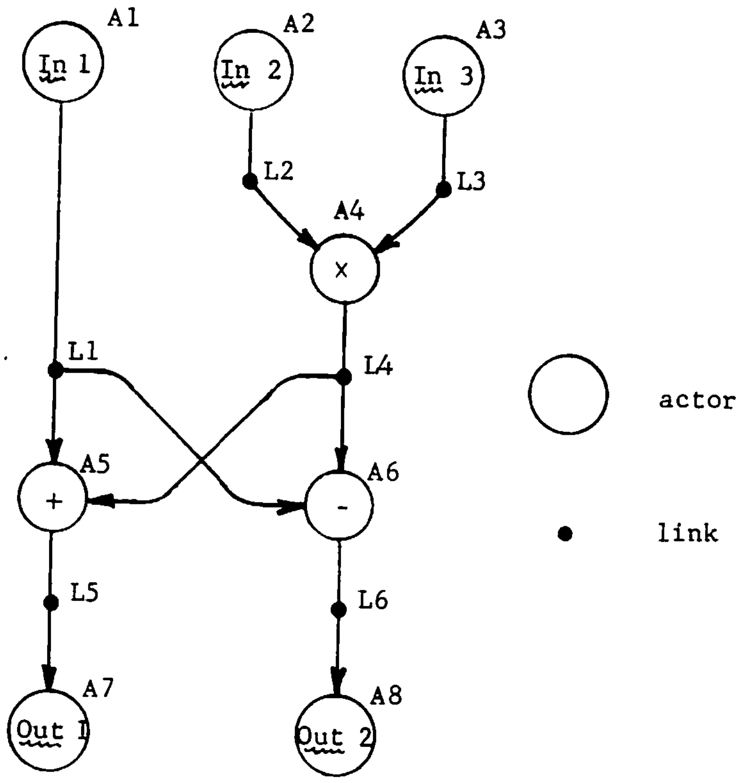


Figure 1. A data flow program graph.

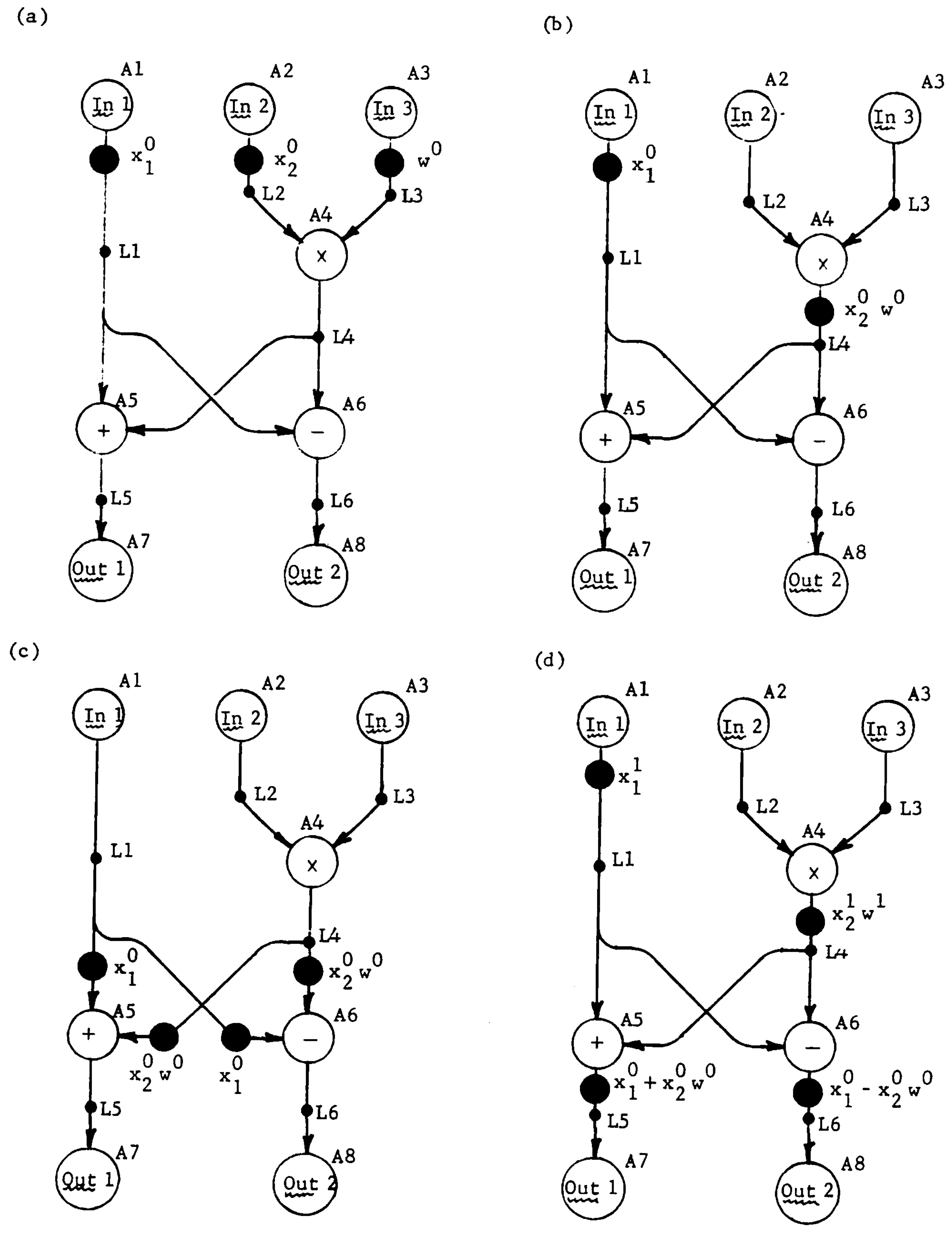


Figure 2. Snapshots of a data flow program in execution.

carrying this result value is placed on the actor's output arc.

Figure 2 illustrates the behavior of the program graph of Fig. 1, which consists of the eight actors A1 - A8 and the six links L1 - L6. Actors A1, A2 and A3 are input operators which place values from the environment on their output arcs whenever they are available and the arcs are not occupied. In Fig. 2a the values x_1^0, x_2^0 and w^0 have been received. In this configuration of the program graph, links L1, L2 and L3 are enabled. Suppose links L2 and L3 fire, placing tokens on the input arcs of the multiplication actor A4. Then A4 may fire, placing a token carrying the value $x_2^0w^0$ on its output arc, yielding configuration (b). Links L1 and L4 are now enabled and may fire in either order. Once both fire, configuration (c) is reached. Actors A5 and A6 are now enabled and their firings deliver the result values $x_1^0 + x_2^0w^0$ and $x_1^0 - x_2^0w^0$ to output actors A7 and A8 through links L5 and L6 as in configuration (d). Whenever the environment is ready to accept an available output, an output actor removes the token carrying the output value from its input arc and delivers it to the environment. In the meantime, more input values x_1^1, x_2^1 and w^1 may have been received and a second instance of computation by the program graph may follow the first through the graph, as indicated in the figure. Thus this data flow program graph allows the execution of computations in pipeline fashion.

The links and actors used in the data flow program graphs of the present paper are shown in Figs. 3 and 4. The two kinds of links transmit data values (values of type **integer**, **real** or **complex**, for example) and **boolean** values, respectively. The behavior of operator actors (Fig. 4a) has already been explained; the function letter f may denote any primitive operation on data values. The actors in the program graph of Fig. 1 are all operators. An identity operator (Fig. 4b) is a special kind of operator that has one input arc and transmits its input value unchanged.

Deciders, gates and merge actors are used in the representation of conditional or iterative computation in data flow program graphs. A decider (Fig. 4c) requires a value from each input arc

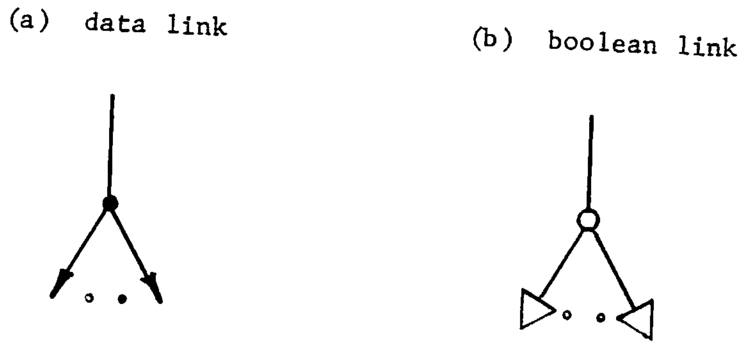


Figure 3. Links of the data flow language.

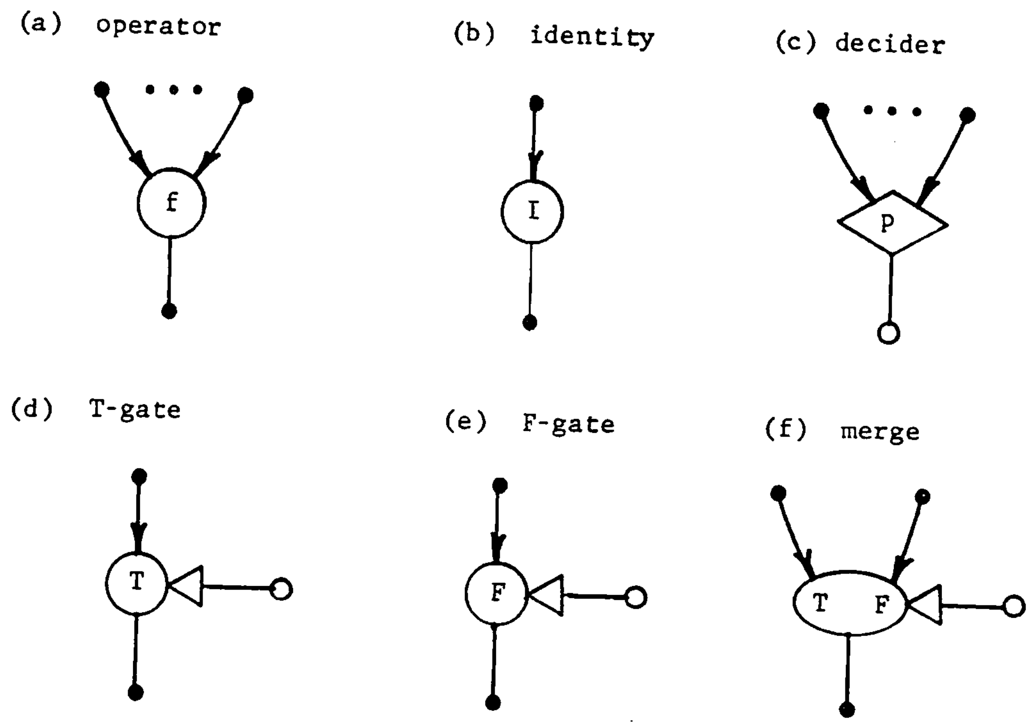


Figure 4. Actors of the data flow language.

and produces the truth value resulting from applying the predicate p to the values received. Tokens bearing truth values control the flow of data tokens by means of T-gates, F-gates and merge actors (Fig. 4d, e, f). A T-gate passes a data token from its data input arc to its output arc when it receives the value **true** on its control input arc. It will absorb a data token from its data input arc and place nothing on its output arc if it receives the truth value **false**. An F-gate has similar behavior, but with the sense of the truth value reversed. A merge actor has T- and F-data input arcs, and a truth value input arc. When a truth value is received, the merge actor places a token on its output arc bearing the next data value received on the corresponding data input arc. A token on the other data input arc is unaffected.

The data flow program graph in Fig. 5 illustrates use of the decider, gate and merge actors. It represents the computation.of $z = x^n$ specified by the following conventional program:

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

The successive values assumed by the loop variables y and i pass through the links they label in the program graph. The decider emits a token caarrying the value **true** each time execution of the loop body is required. (This routes the current values of loop variables through the body operators.) When firing of the decider yields **false**, the value of y is routed to the output link z.

Note the presence of tokens carrying **false** values on the truth value input arcs of the merge actors. These tokens allow the merge actors to initiate execution of the loop by passing in initial values for the loop variables.

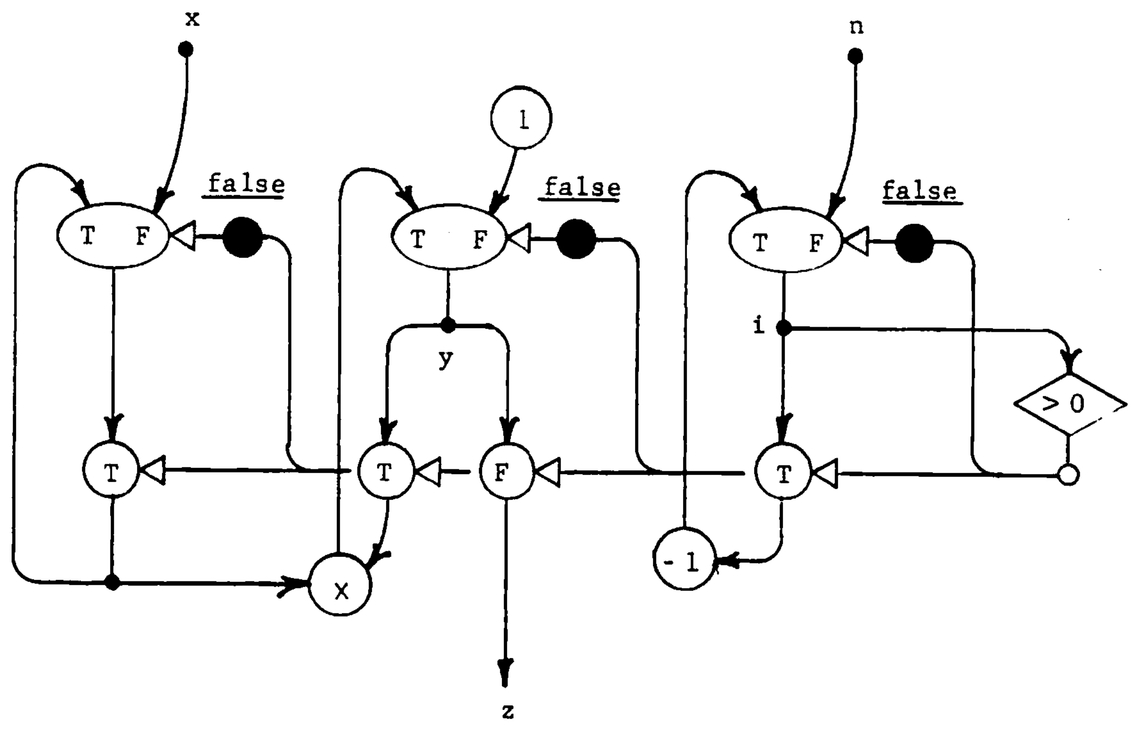


Figure 5. An iterative data flow program.

The Fast Fourier Transform

Now we are ready to construct a data flow program graph for the Fast Fourier Transform algorithm. The discrete Fourier transform of a sequence of $N = 2^n$ input samples $x_0, \ldots, x_{N-1}$ is the sequence of values $f_0, \ldots, f_{N-1}$ where

$$f_k = \sum_{i=0}^{N-1} x_i W^{ik} \tag{1}$$

and

$$W = e^{-j(2\pi/N)}$$

The direct computation of these values involves the accumulation of N^2 product terms; the Fast Fourier Transform (FFT) is based on the observation that the transform on 2^p data samples can be simply expressed in terms of two transformations on 2^{p-1} samples. Continuing recursively, one discovers that the transform on 2^n points can be expressed in terms of $n \cdot 2^{n-1}$ transformations of two points each. Figure 6 shows the flow of values in one arrangement of the FFT computation for eight data points ($n = 3$). This arrangement, in which the computation consists of n stages (the columns of the figure) having identical form, is known as the time decimated, constant geometry FFT [19]. Each stage of the computation consists of N/2 units of similar form, known as "butterflies," which compute two-point transforms.

The general form of this FFT algorithm may be described as follows: Let $u_{p,k}$ be the kth component of the vector of values computed by the pth stage of the computation. Then $B_{p,q}$, the qth butterfly of stage p computes

$$u_{p,q} = u_{p-1,2q} + u_{p-1,2q+1} W^{e_{p,q}} \tag{2}$$

$$u_{p,q+2^{n-1}} = u_{p-1,2q} - u_{p-1,2q+1} W^{e_{p,q}} \tag{3}$$

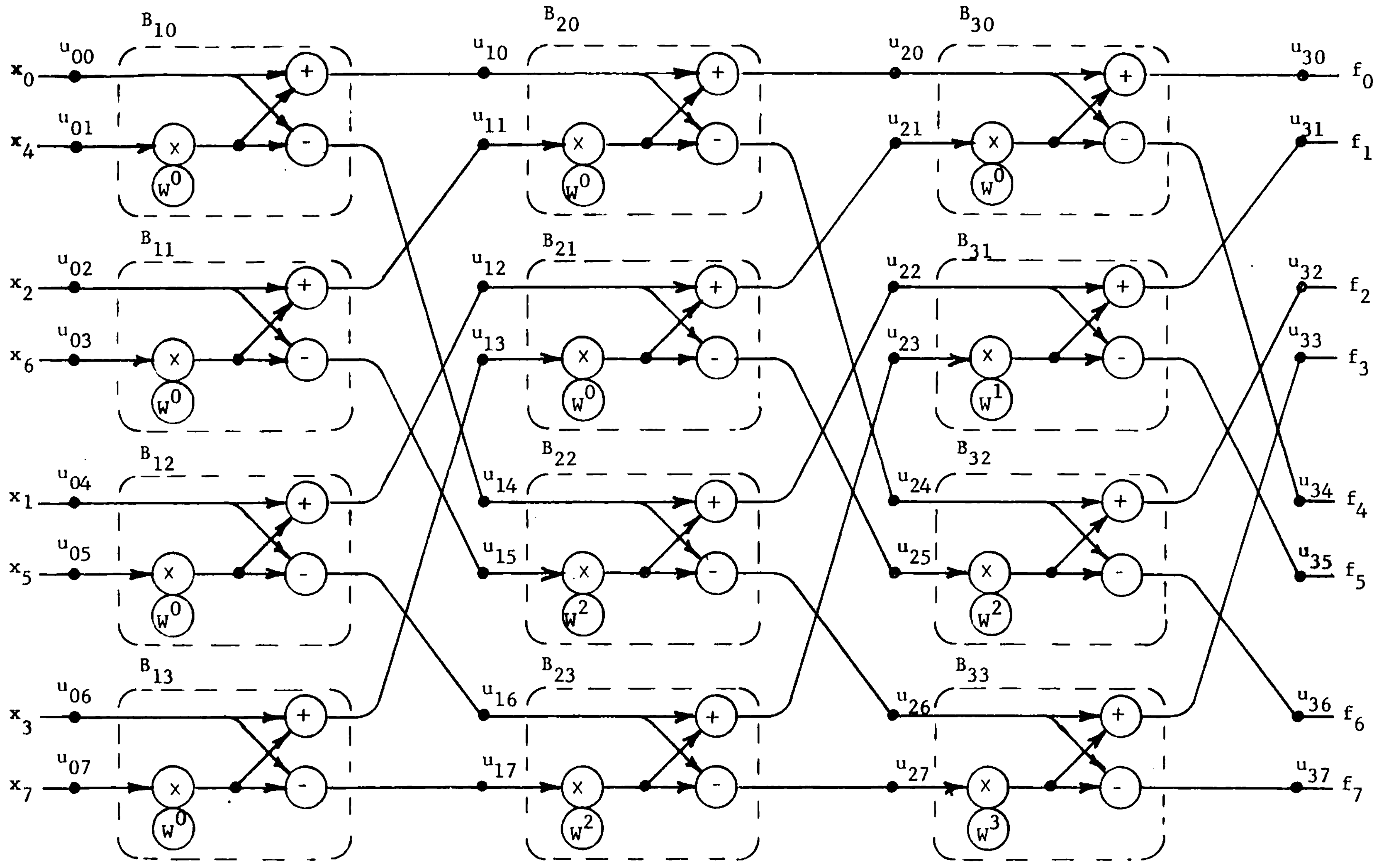

B_{10}
B_{11}
B_{12}
B_{13}
B_{20}
B_{21}
B_{22}
B_{23}
B_{30}
B_{31}
B_{32}
B_{33}
x_0 u_{00} u_{10} u_{20} u_{30} f_0
x_4 u_{01} u_{11} u_{21} u_{31} f_1
x_2 u_{02} u_{12} u_{22} u_{32} f_2
x_6 u_{03} u_{13} u_{23} u_{33} f_3
x_1 u_{04} u_{14} u_{24} u_{34} f_4
x_5 u_{05} u_{15} u_{25} u_{35} f_5
x_3 u_{06} u_{16} u_{26} u_{36} f_6
x_7 u_{07} u_{17} u_{27} u_{37} f_7
W^0
W^1
W^2
W^3
+
-
x

Figure 6. The eight-point, constant geometry, time decimated FFT.

where the exponent $e_{p,q}$ of each phase factor $w_{p,q} = W^{e_{p,q}}$ is given by

$$e_{p,q} = 2^{n-p}\ \mathrm{quo}(q, 2^{n-p}) \tag{4}$$

and

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

$$0 < p \leq n$$

The function quo(m,n) yields the integer quotient of m divided by n. The input values for stage one are related to the data samples by

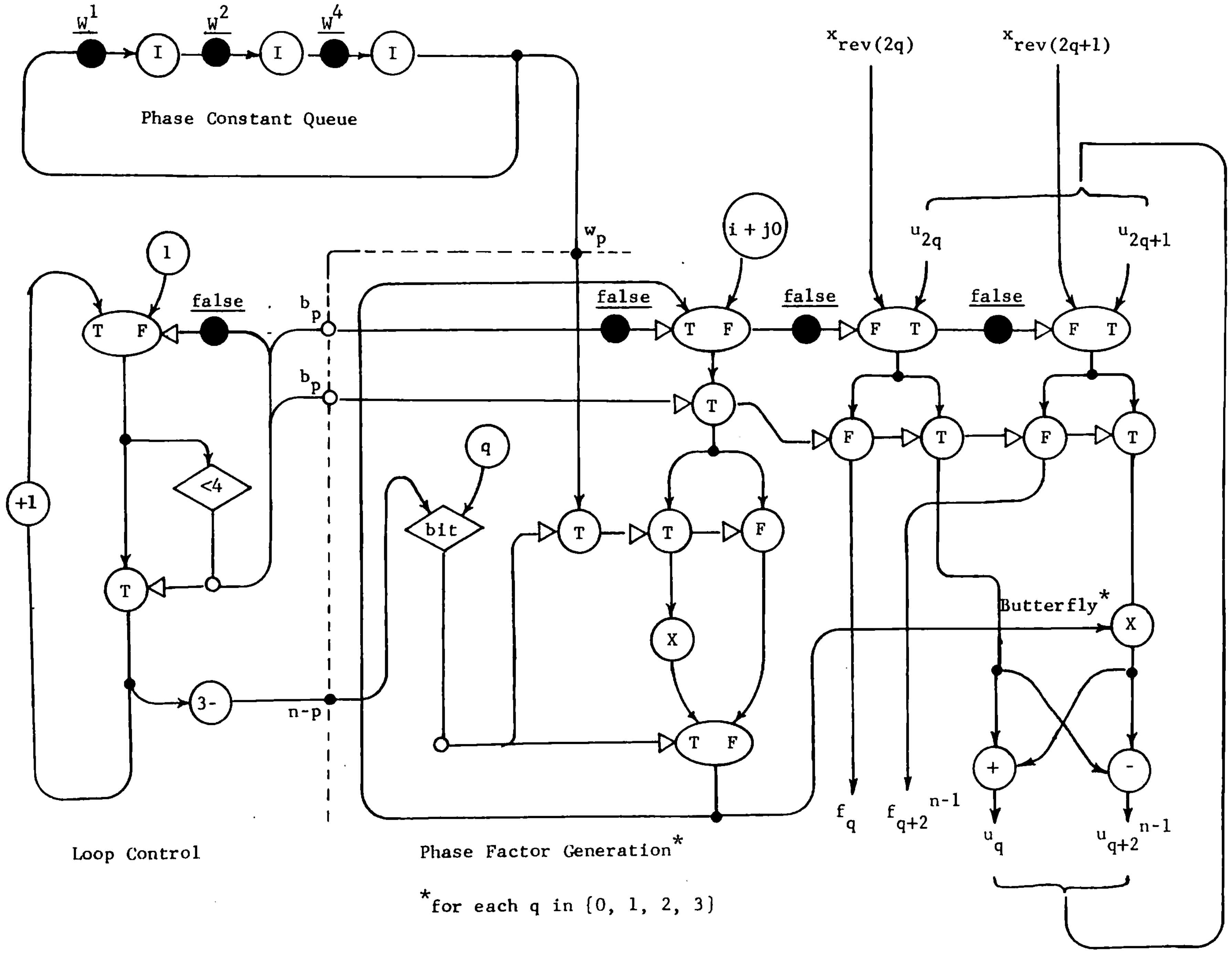
$$u_{0,k} = x_i \quad \text{where} \quad i = \mathbf{rev}(k)$$

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

$$f_k = u_{n,k}, \qquad 0 \leq k < 2^n$$

We wish to take maximum advantage of parallelism in representing the FFT as a data flow program graph, but since each actor will take space in the machine representation, we do not want to use a larger program graph than necessary to exploit concurrency. Since each stage of the computation uses values computed by the preceding stage, it is appropriate to construct the program graph as an n-cycle iteration in which the body consists of the 2^{n-1} butterflies comprising one stage of computation written out explicitly. The form of the corresponding data flow program graph is shown in Figure 7 for the eight-point case. This is fairly easy because the constant geometry of the computation over all stages makes it possible to use a fixed routing of values from the outputs of the butterflies to their inputs where they become operands for the next cycle. Generating the phase factors for each butterfly, however, presents a problem. The usual technique is to use a table lookup in a table of powers of W, but our program graph notation includes no suitable mechanism. Instead, the factor $w_{p,q}$ used for butterfly q in stage p may be computed from

Figure 7. Iteration data flow program graph for the eight point FFT.



*for each q in {0, 1, 2, 3}

the factor $w_{p-1,q}$ used for the previous stage by a simple rule derived as follows: The exponents of W for $w_{p,q}$ and $w_{p-1,q}$ are (from (4)):

$$e_{p,q} = 2^{n-p}\,\mathbf{quo}(q, 2^{n-p})$$

$$e_{p-1,q} = 2^{n-p+1}\,\mathbf{quo}(q, 2^{n-p+1})$$

Then

$$\begin{aligned} e_{p,q} &= e_{p-1,q} + (e_{p,q} - e_{p-1,q}) \\ &= e_{p-1,q} + 2^{n-p}\,\underbrace{(\mathbf{quo}(q, 2^{n-p}) - 2\,\mathbf{quo}(q, 2^{n-p+1}))}_{T_{p,q}} \end{aligned}$$

Careful study of the factor $T_{p,q}$ reveals that

$$T_{p,q} = \begin{cases} 0 & \text{if } \mathbf{quo}(q, 2^{n-p}) \text{ is even} \\ 1 & \text{if } \mathbf{quo}(q, 2^{n-p}) \text{ is odd} \end{cases}$$

Thus $T_{p,q}$ is the $(n - p)^{th}$ bit in $q_{n-1} \dots q_0$, the n-bit binary representation of q. Let **bit**(r, q) be a primitive function that yields the r^{th} bit of q. Then we have

$$w_{p,q} = w_{p-1,q} \times (\text{if } \mathbf{bit}(n - p, q) = 1 \text{ then } W^{2^{n-p}} \text{ else } 1)$$

The initial value of the phase factor for the q^{th} butterfly is

$$\begin{aligned} w_{1,q} &= W^{e_{1,q}} \quad \text{where } e_{1,q} = 2^{n-1}\;\mathbf{quo}(q, 2^{n-1}) \\ &= W^0 = (1 + j0) \end{aligned}$$

The computation of the phase factors $w_{p,q}$ is performed by the sections of Figure 7 labelled "Phase Factor Generation" and "Phase Constant Queue."

We suppose that the signal values are delivered to the program graph as a continuous

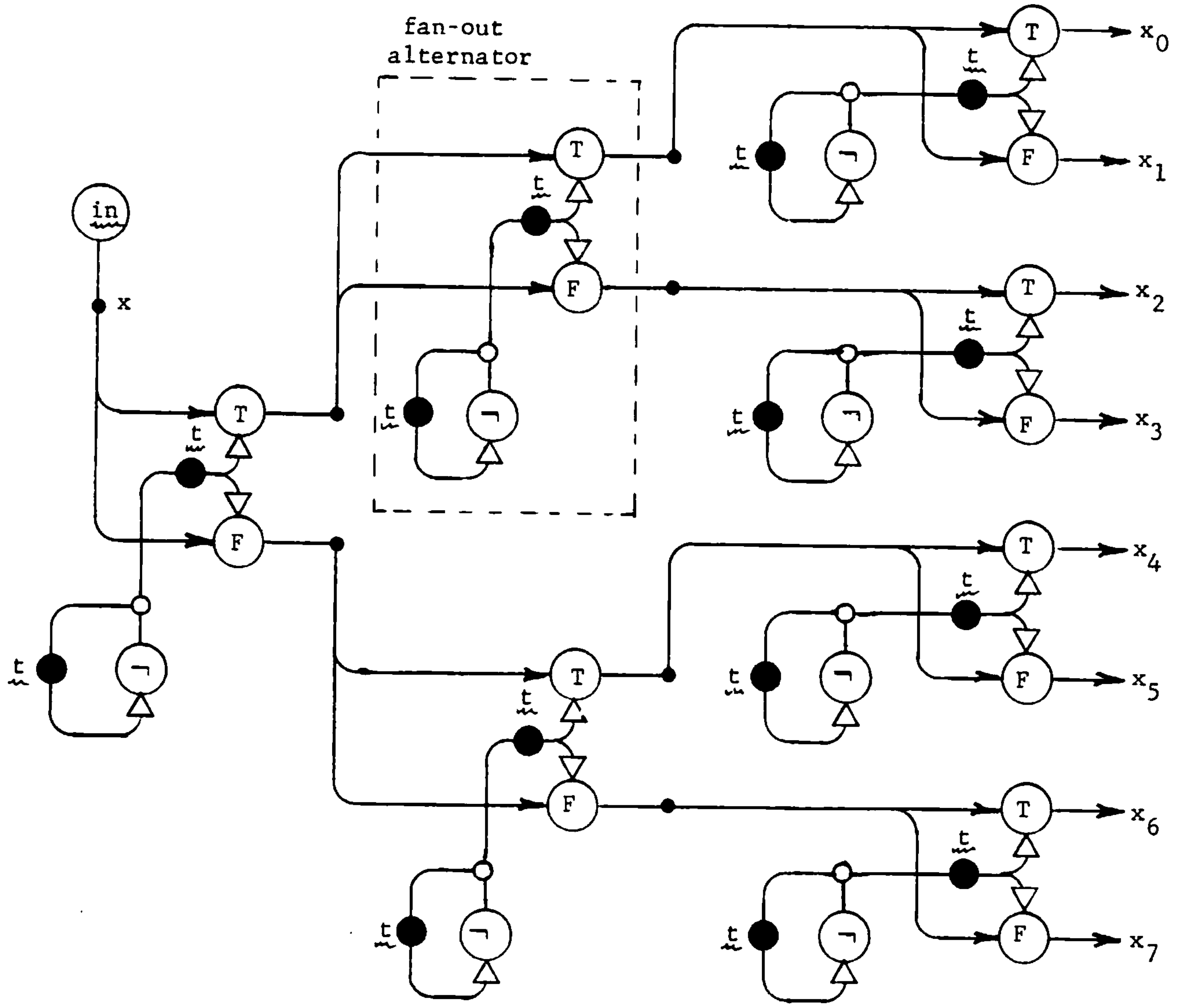


Figure 8. Tree of fan-out alternators for sample distribution.

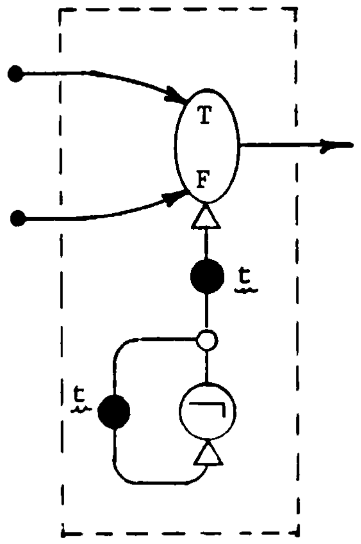


Figure 9. Fan-in alternator.

stream through a single input operator, and must be distributed among the 2^n input links of the FFT program. This may be done by means of a binary tree of program graph fragments, which we may call fan-out *alternators*, connected as in Fig. 8. A similar binary tree of fan-in alternators (Fig. 9) can be used to form the transform values $f_0, \ldots, f_{N-1}$ into a stream.

III. The Data Flow Processor: An Overview

The data flow processor is a stored program computer designed to exploit the concurrency of action represented by data flow program graphs such as we have illustrated for the Fast Fourier Transform. The overall structure of this processor is shown in Fig. 10; it consists of five major sections connected by channels through which information is sent in the form of discrete packets. The five sections are:

Memory Section -- consists of Instruction Cells which hold instructions and their operands.

Processing Section -- consists of Processing Units that perform the basic operations on data values

Arbitration Network -- delivers *operation packets* from the Memory Section to the Processing Section.

Control Network -- delivers *control packets* from the Processing Section to the Memory Section.

Distribution Network -- delivers *data packets* from the Processing Section to the Memory Section.

Briefly, instructions held in the Memory Section are enabled for execution by the arrival of their operands in data packets from the Distribution Network and control packets from the Control Network. Enabled instructions, together with their operands, are sent as operation packets to the Processing Section through the Arbitration Network. The results of instruction execution are sent through the Distribution and Control Networks to the Memory Section where they become

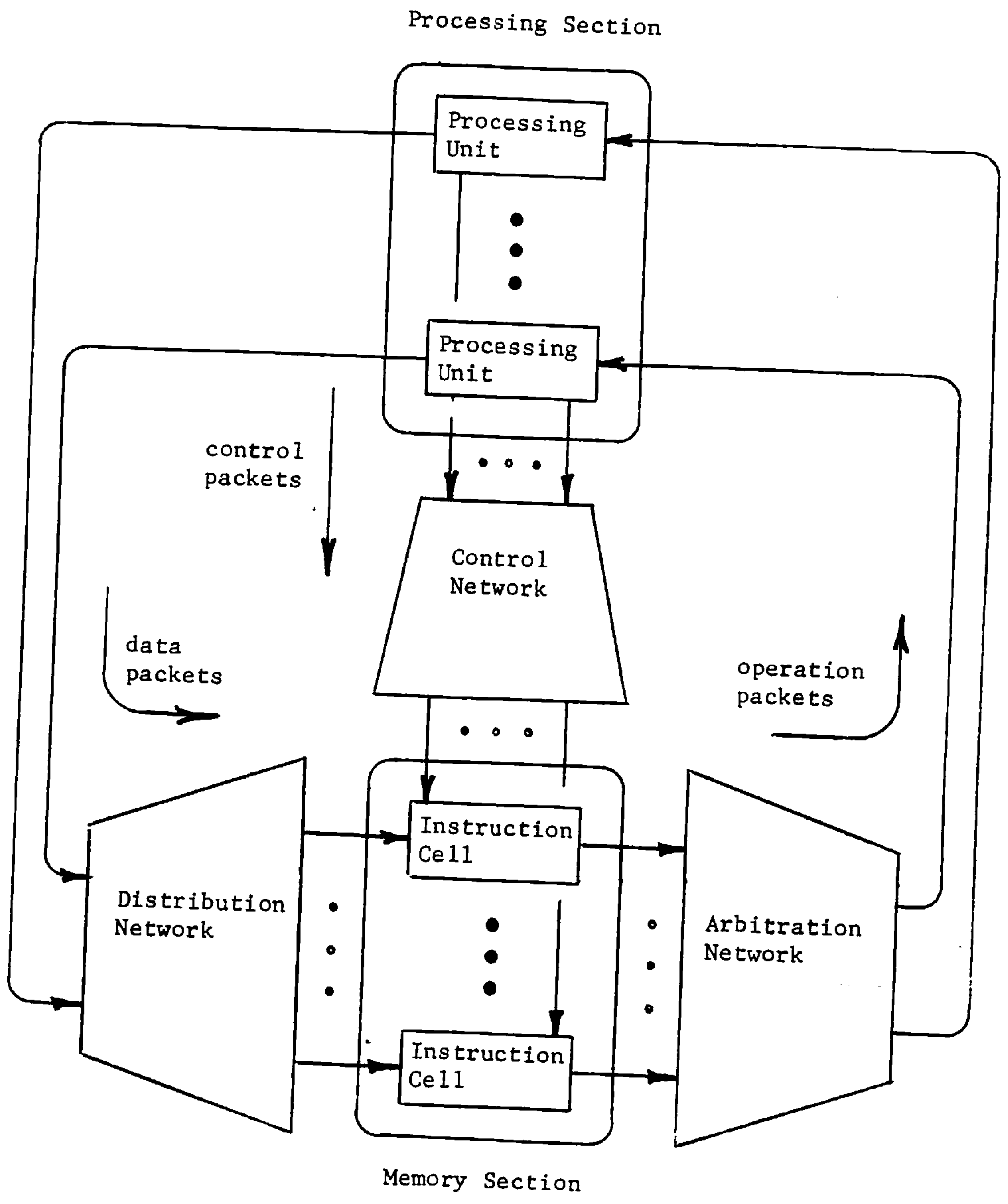


Figure 10. General structure of the data flow processor.

operands of other instructions. We next consider the operation of each section in more detail.

The Memory Section of the processor is a collection of Instruction Cells. Each Instruction Cell has a unique identifying address, the *cell identifier*. An occupied Cell holds an *instruction* consisting of an *operation code* and several *destinations*. Each *destination* contains a *destination address*, which is a cell identifier, and additional control information used by processing units to generate result packets. An instruction represents one or more actors of the program graph together with their output links. Instructions are linked together through destination addresses stored in their destination fields.

Each Cell also contains three Receivers (Figure 12) which await the arrival of values for use as operands by the instruction. Once an Instruction Cell has received the necessary operand values and acknowledge signals,[1] the Cell becomes enabled and sends an operation packet, consisting of the instruction and the operand values, to the appropriate Processing Unit through the Arbitration Network.

The Arbitration Network provides a path from each Instruction Cell to each Processing Unit, and sorts the operation packets among its output ports according to the operation codes of the instructions they contain. For each operation packet received, a Processing Unit performs the operation specified by the instruction using the operand values in the packet, and produces one or more *result packets* which are sent to Instruction Cells through the Control Network and Distribution Network. Each result packet consists of a result value and a destination address derived from the instruction being processed by the Processing Unit. There are two kinds of result packets: *control packets* containing boolean values or acknowledge signals, which are sent through the Control Network; and *data packets* containing integer or complex values, which are sent

1. Acknowledge packets at the machine level are needed to correctly implement the firing rule for program graphs. Their use is explained fully in Section V.

through the Distribution Network. The two networks deliver result packets to Receivers of Instruction Cells as specified by their destination address fields; that is, result packets are routed according to their destination address.

Arrival of a result packet at an Instruction Cell either provides one of the Receivers of the Cell with an operand value or delivers an acknowledge signal; if all result packets required by the instruction in the Cell have been received, the Instruction Cell becomes enabled and dispatches its contents to the Arbitration Network as a new operation packet.

Note that the functions performed by the processing unit of a conventional machine are distributed among several sections of the data flow processor. The operations specified by instructions are carried out in the Processing Section, but control of instruction sequencing is a function of the Instruction Cells of the Memory Section, and the decoding of operation codes is partially done within the Arbitration Network. Also unusual is the fact that address fields (destination addresses) of instructions only specify where results are to go, and are not used to access operand values. Instead of instructions having to ask for their operands, the operand values are sent to the instructions.

We emphasize that all communication between parts of the data flow processor is by packet transmission over the channels shown explicitly in Fig. 10; there are no connections other than those shown in the figure. Furthermore, transmission of packets over each channel is done using an asynchronous protocol so the five sections of the processor may operate independently without need for a clock or other central source of timing signals. Systems organized to operate in this manner are said to have *packet communication architecture* [15].

In particular, note that the Instruction Cells are assumed to be physically independent, so at any time many of them may be enabled. Later, we will show how the Arbitration Network can be designed so many instruction packets may flow into it concurrently and be funneled into dense streams of packets directed to the processing units. Similarly, the Control Network and the Distribution Network can be designed to distribute dense streams of control and data packets efficiently to the Instruction Cells through highly concurrent operation. In this way, highly parallel operation of the entire processor is achieved, and the appetites of pipelined processing units can be satisfied.

IV. The Data Flow Processor: Principles of Operations

We have described the major modules of a data flow processor and a graphical representation of data flow programs. The architectural implications of data flow concepts is further studied in this paper by deriving a machine level program representation of the FFT algorithm from the program graph of Figure 7 and estimating the performance of a data flow processor in executing this program. In this section we define the data flow processor modules in sufficient detail to support these developments. An instruction set is specified, and its capabilities are explained by detailing the principles of operation for an instruction cell and for two functional units.

The Processing Section consists of five processing units having characteristics appropriate for the FFT algorithm:

1. Multiplier -- multiplication of complex operands.
2. Adder -- addition and substraction of complex operands.
3. Distributor -- replication and distribution of data and control values.
4. Int-Processor -- integer arithmetic and test operations.
5. Cntl-Processor -- replication and routing of data and control values.

The formats of packets transmitted between major sections of the data flow processor are given in Figure 11. Each instruction consists of an operation code from opcode-set and an array of up to five destinations which specify the data and control packets to be generated by instruction

```
definitions:

type instruction = record
        opcode: opcode-set;
        dest: array[1..5] of destination
        end;

type destination = record
        used: boolean;
        send-ack-signal: boolean;
        switch: {null, boolean};
        addr: address
        end;

type address = record
        cell-id: 1.. n-of-cell;
        rec-id: 1..3
        end;

type operand = {null, boolean, integer, complex};

type operation-pkt = packet
        inst: instruction;
        opd: array[1..3] of operand
        end;

type cntl-pkt-c = packet
        ctype: (ACK, BOOL);
        value: {null, boolean};
        addr: address
        end;

type cntl-pkt-r = packet
        ctype: (ACK, BOOL);
        value: {null, boolean};
        rec: 1..3
        end;

type data-pkt-c = packet
        dtype: (INT, CPLX);
        value: {integer, complex};
        addr: address
        end;

type data-pkt-r = packet
        dtype: (INT, CPLX);
        value: {integer, complex};
        rec: 1..3
        end;
```

Figure 11. Packet Definitions for a Data Flow Processor.

execution in the Processing Section and where the packets are to be sent. A destination consists of an *address* and other information to be explained later, which is interpreted by the Processing Units. An address consists of an integer that designates an Instruction Cell and an integer that specifies the Receiver of the Instruction Cell which is to receive a control or data packet. An operation packet consists of an instruction and an array of three operands, each of which may be **null** in case the operand is not required. Two forms of control packets and two forms of data packet are declared; this is because the cell-id component of the address field is irrelevant once the packet has been routed to the correct Instruction Cell by the Control or Distribution Network. Packets generated by the processing units are of type *data-pkt-c* or *cntl-pkt-c*. Packets delivered to instruction cells are of type *data-pkt-r* or *cntl-pkt-r*.

The Arbitration Network delivers an operation packet from an Instruction Cell to the processing unit specified by its *opcode*. The Distribution Network delivers a data packet from a processing unit to the Instruction Cell specified in its *cell-id*. Similarly the Control Network delivers control packets from processing units to Instruction Cells. The structure of the routing networks will be discussed later where we relate their characteristics to the performance potential of the data flow processor for the FFT algorithm.

Instruction Cell Operation

The function of each Instruction Cell is to receive control and data packets, and to transmit an operation packet when all needed operands have arrived and the enabling condition for its instruction is satisfied. In a machine language program which contains an iteration construct, or which is intended to process data streams via pipelining, it is necessary to condition instruction execution on receipt of a specified number of acknowledge signals (as explained in Section V) in order to implement the firing rule correctly. Thus the general enabling condition for an Instruction Cell is that the required data and control packets have arrived and that the Cell has also received a specified number of acknowledge packets.

As shown in Figure 12, an Instruction Cell consists of an input interface module Cell-Input-Cntl, three Receiver modules and an output interface module Cell-Output-Cntl. The Cell-Input-Cntl unit processes control and data packets, sending acknowledge signals on to Cell-Output-Cntl and distributing operand values to the Receivers according to the receiver number in the packet. The format of receiver packets, sent from Cell-Input-Cntl to the Receivers, is:

```
type receiver-pkt = packet rtype: (BOOL, INT, CPLX);
                           value: {boolean, integer, complex} end;
```

A behavior description of the Cell-Input-Cntl module is given in Figure 13.[2] Behavior descriptions for the Receiver module and the Cell-Output-Cntl module are given in Figures 14 and 15.

Each Receiver (Figure 14) may be set (by initializing *receiver-type* and *receiver-mode*) to provide one of three different types of operand values, **boolean, integer** or **complex**, and to operate in one of two modes: constant and variable. The behavior of a Receiver for each allowed

---

2. The constructs used in these descriptions are familiar programming language constructs except for the **receive** and **send** statements. We envision that operations specified in the separate boxes of a behavioral description are carried out concurrently, and are synchronized by passing signals between them explicitly. A statement

receive x **at** P

means that the next packet to arrive at input port P is made the value denoted by identifier x. A statement

send y **at** Q

means that the value denoted by identifier y is transmitted at output port Q. Execution of a **receive** statement cannot be completed until an input packet is available at the named input port. Likewise execution of a **send** statement is not completed until the unit connected to the output port is prepared to accept a packet. It is also assumed that proper arbitration is performed when **send** statements having the same output port are executed concurrently.

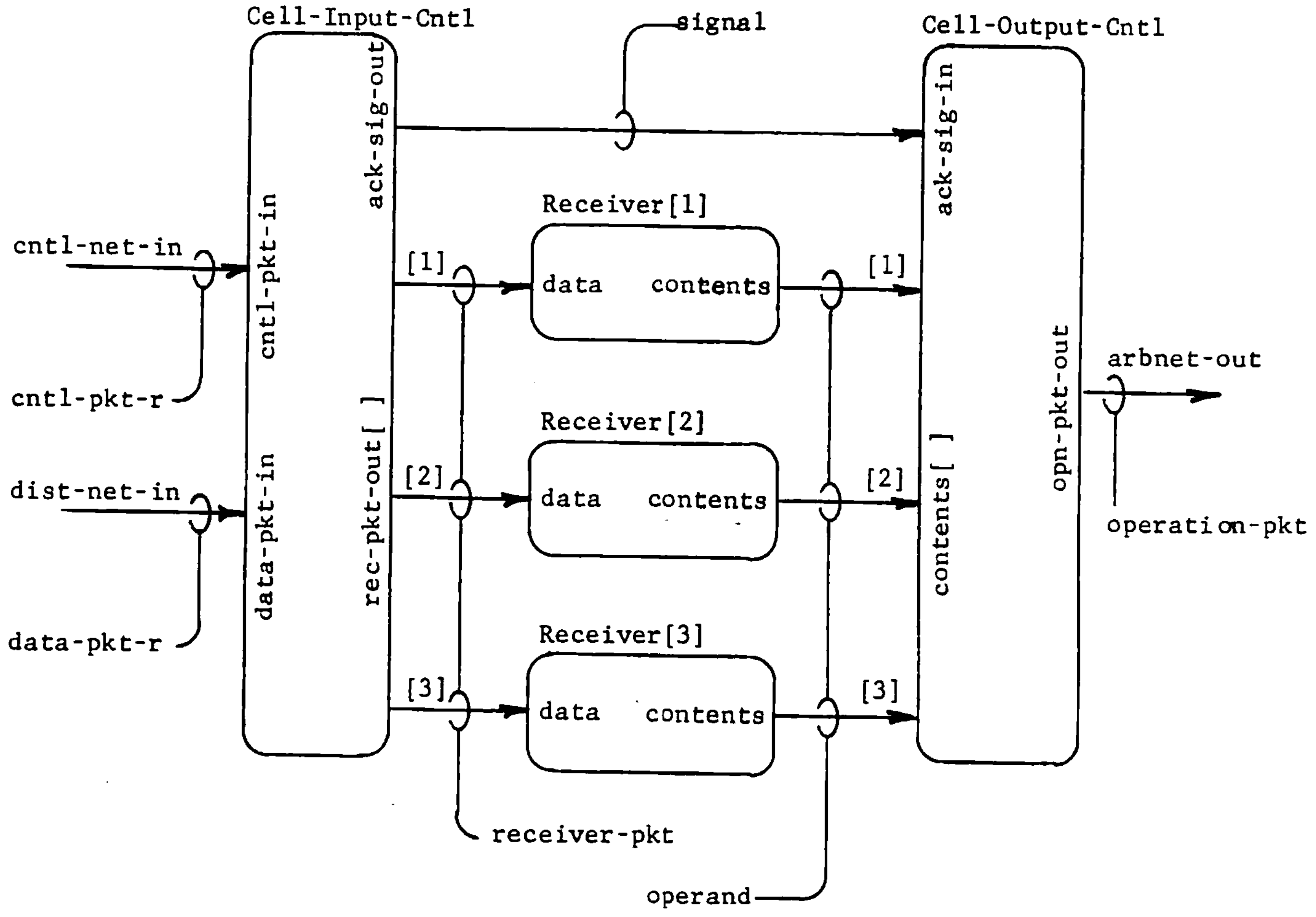
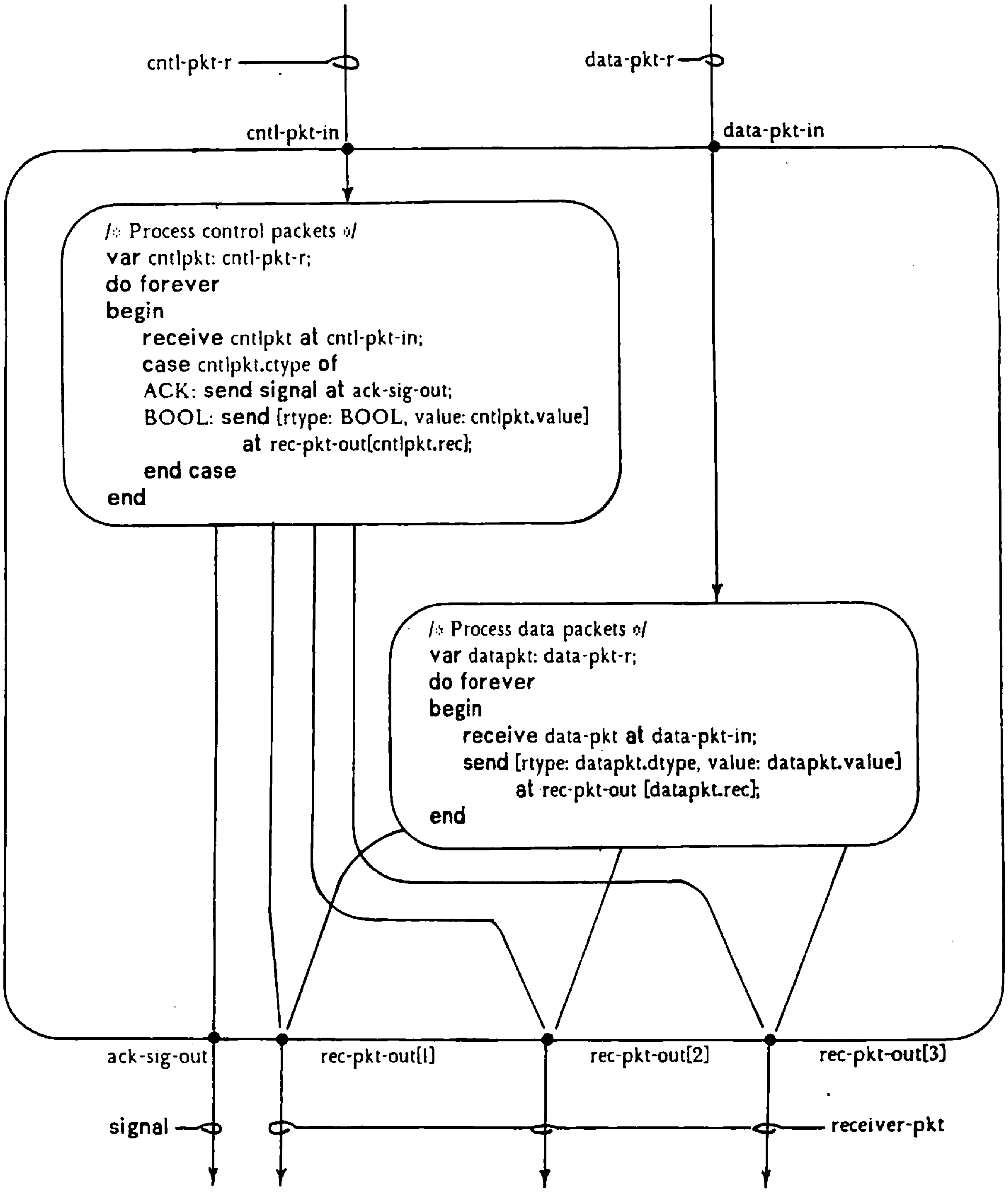


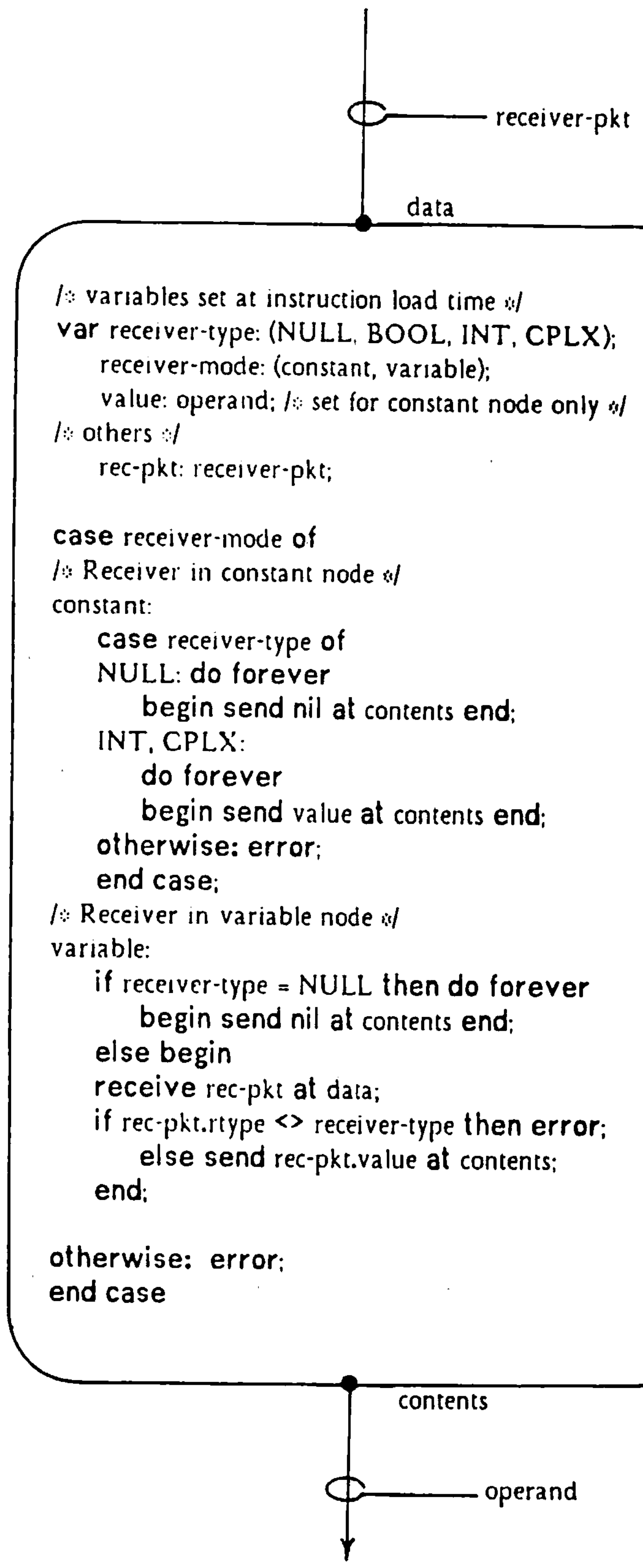
Figure 12. Structure of an Instruction Cell.



```
/* Process control packets */
var cntlpkt: cntl-pkt-r;
do forever
begin
    receive cntlpkt at cntl-pkt-in;
    case cntlpkt.ctype of
    ACK: send signal at ack-sig-out;
    BOOL: send [rtype: BOOL, value: cntlpkt.value]
              at rec-pkt-out[cntlpkt.rec];
    end case
end
```

```
/* Process data packets */
var datapkt: data-pkt-r;
do forever
begin
    receive data-pkt at data-pkt-in;
    send [rtype: datapkt.dtype, value: datapkt.value]
          at rec-pkt-out [datapkt.rec];
end
```

Figure 13. Behavior description of Cell-Input-Cntl.

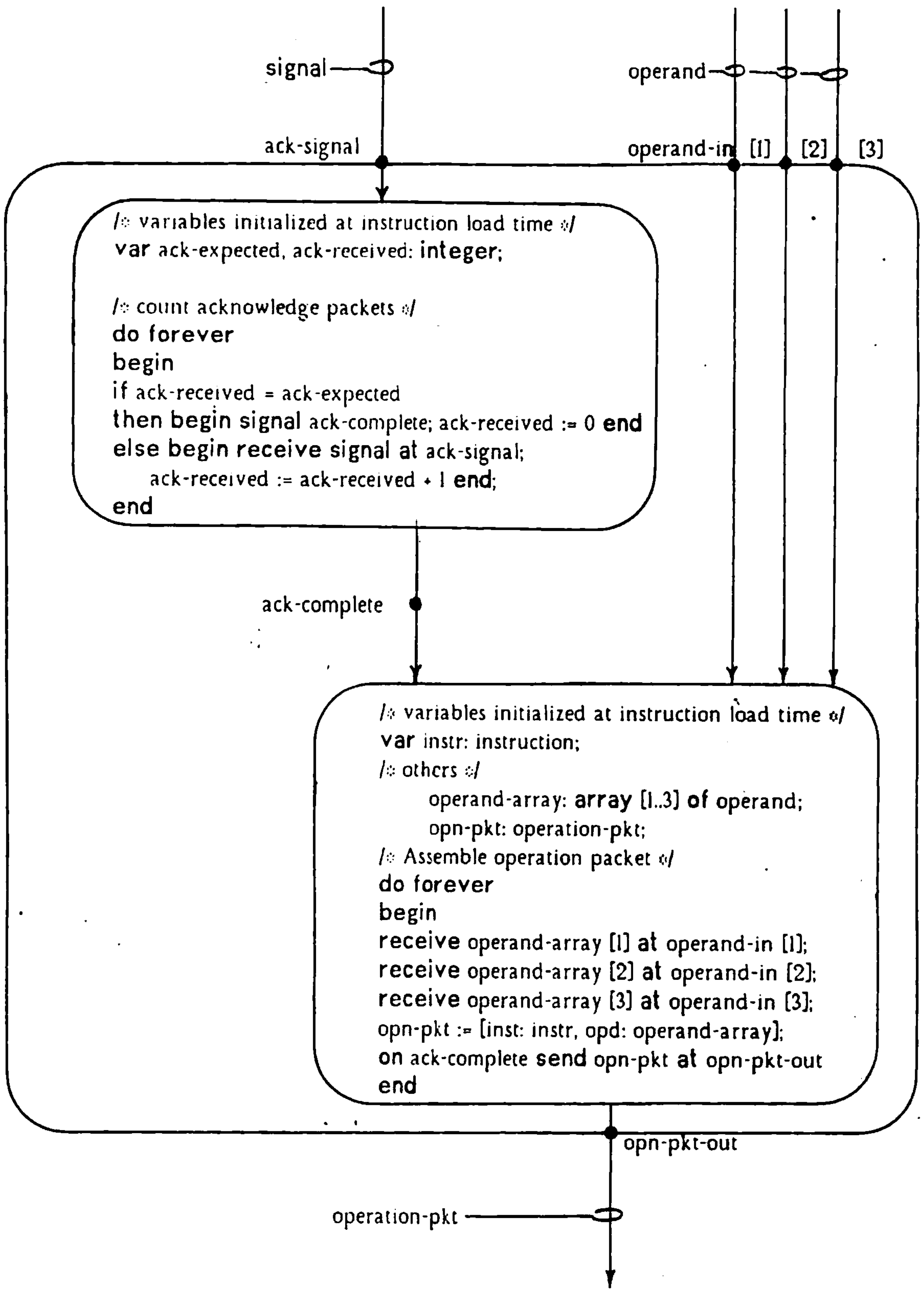


```
/* variables set at instruction load time */
var receiver-type: (NULL, BOOL, INT, CPLX);
    receiver-mode: (constant, variable);
    value: operand; /* set for constant node only */
/* others */
    rec-pkt: receiver-pkt;

case receiver-mode of
/* Receiver in constant node */
constant:
    case receiver-type of
    NULL: do forever
        begin send nil at contents end;
    INT, CPLX:
        do forever
        begin send value at contents end;
    otherwise: error;
    end case;
/* Receiver in variable node */
variable:
    if receiver-type = NULL then do forever
        begin send nil at contents end;
    else begin
    receive rec-pkt at data;
    if rec-pkt.rtype <> receiver-type then error;
        else send rec-pkt.value at contents;
    end;

otherwise: error;
end case
```

Figure 14. Behavior description of a Receiver module.



```
/* variables initialized at instruction load time */
var ack-expected, ack-received: integer;

/* count acknowledge packets */
do forever
begin
if ack-received = ack-expected
then begin signal ack-complete; ack-received := 0 end
else begin receive signal at ack-signal;
    ack-received := ack-received + 1 end;
end
```

```
/* variables initialized at instruction load time */
var instr: instruction;
/* others */
      operand-array: array [1..3] of operand;
      opn-pkt: operation-pkt;
/* Assemble operation packet */
do forever
begin
receive operand-array [1] at operand-in [1];
receive operand-array [2] at operand-in [2];
receive operand-array [3] at operand-in [3];
opn-pkt := [inst: instr, opd: operand-array];
on ack-complete send opn-pkt at opn-pkt-out
end
```

Figure 15. Behavioral Description of the Cell-Output_Cntl unit

combination of type and mode is summarized in Table 1. If the instruction held by an Instruction Cell requires fewer than three operands, then one or more of the Receivers are set to type NULL; in this case the Receiver is always enabled and delivers the value nil repeatedly, since Cell-Output-Cntl requires an operand packet from each Receiver to form each operation packet.

The instruction held by an Instruction Cell is represented by the value of *instr* in Cell-Output-Cntl (Figure 15). The number of acknowledge packets required to enable the Instruction cell and the number of acknowledge packets received since the previous firing of the Instruction Cell are stored in *ack-expected* and *ack-received*. The upper box of the Cell-Output-Cntl module waits for arrival of the expected number of acknowledge packets; it then resets its count and transmits the signal *ack-complete*. The lower box waits for this signal, and then transmits an operation packet containing one operand value (possibly nil) from each of the three receivers.

The variables *instr*, *ack-expected* and *ack-received* in Cell-Output-Cntl and *receiver-type*, *receiver-mode* and *value* in each Receiver must be initialized with values derived from a machine language program for its proper execution on the data flow processor. The corresponding "loading" mechanisms for setting these values will not be discussed in this paper.

Table 2 presents the instruction types which will be used to code the FFT program for the data flow processor. These instructions are defined so several actors in a data flow graph may be encoded by a single instruction. Although these instructions have been chosen to illustrate the performance achievable in the FFT computation, they are nevertheless representative of the sort of instructions that might be included in a complete instruction code. For each type of instruction, the letter (M, A, D, I or C) under each instruction name indicates the Processing Unit that executes instructions of that type. The format given in the table specifies the type of each Receiver and whether a value is required from it. The format also indicates the kinds of destinations that make sense for the instruction. This requires a bit more explanation. Destinations have the form

Table 1. Type and Mode Settings for Receivers

Symbol	Receiver Type	Receiver Mode	Packet(s) Required	Enabling Condition
N	NULL		none	always enabled
B	BOOL	variable	BOOL	receipt of a boolean packet
I	INT	variable	INT	receipt of an integer packet
C	CPLX	variable	CPLX	receipt of a complex packet
Ic	INT	constant	none	always enabled
Cc	CPLX	constant	none	always enabled

Name	Format	Results	Name	Format	Results
Integer Add (I)	i-add I [p] I [q] N	● → {p + q} ✱ → a	Complex Switch (C)	c-sw C [z] B [b] N	● $\xrightarrow{F}$ {z} if b = *false* ● $\xrightarrow{T}$ {z} if b = *true* ✱ → a
Integer Subtract (I)	i-sub I [p] I [q] N	● → {p - q} ✱ → a	Complex Add and Switch (A)	c-add C [x] C [y] B [b]	● $\xrightarrow{F}$ {x + y} if b = *false* ● $\xrightarrow{T}$ {x + y} if b = *true* ✱ → a
Integer Distribute (D)	i-dist I [r] N N	● → {r} ✱ → a	Complex Subtract and Switch (A)	c-sub C [x] C [y] B [b]	● $\xrightarrow{F}$ {x - y} if b = *false* ● $\xrightarrow{T}$ {x - y} if b = *true* ✱ → a
Complex Multiply (M)	c-mul C [x] C [y] N	● → {x x y} ✱ → a	Integer Compare (I)	i-less I [p] I [q] N	○ → ▷ {b} where b = *if* p < q *then true else false* ✱ → a
Complex Distribute (D)	c-dist C [z] N N	● → {z} ✱ → a	Bit Test (I)	i-bit I [p] I [q] N	○ → ▷ {b} where b = *if bit*(p,q) = 1 *then true else false* ✱ → a
Boolean Distribute (D)	b-dist B [b] N N	○ → ▷ {b} ✱ → a	Integer Switch (C)	i-sw I [p] B [b] N	• $\xrightarrow{F}$ p if b = false • $\xrightarrow{T}$ p if b = true × $\xrightarrow{a}$

Receiver types: N-NULL; B-BOOL; I-INT; C-CPLX   Values: p,q,r-integer; x,y,z-complex; b-boolean

Table 2. Instruction types.

```
type destination = record
        used: boolean;
        send-ack-signal: boolean;
        switch: {null, boolean};
        addr: address
        end;
```

and are used by Processing Units to determine the control and data packets they generate. The *used* field is set to false if this destination is not needed in the program. If an acknowledge packet is to be sent to the Instruction Cell specified by a destination address, then its *send-ack-signal* field is set to true. Otherwise a result packet generated according to the instruction type is sent.

An instruction such as *i-add* that represents a data flow operator, or such as *c-dist* that provides necessary fan-out, would typically use destinations for both purposes. The switch fields of destinations, while ignored in the execution of these instructions, are used in the several switch instructions provided. A switch instruction is convenient for coding the commonly occurring program graph fragment shown in Figure 16. For these instructions, the switch field of a destination indicates whether a result packet should be sent to the Instruction Cell specified by the destination address in the event of a false outcome (F), a true outcome (T), or both (nil). In our figures, destination arcs of switch instructions with non-null switch fields are always labelled with the corresponding boolean values (Figure 16).

The memory space required in an Instruction Cell depends on the number and type of operands and the number of destinations used, and the possibilities permitted by our specification of the Instruction Cell span a wide range. Clearly, the instructions for complex arithmetic are the most demanding of memory for operands, so the number of used destinations should be limited. In our programs, we have permitted instructions to have up to five destinations. The manner in which operands and destination addresses are efficiently coded in Instruction Cells and in operation packets is a matter that would be dealt with in the detailed design of a complete instruction code.

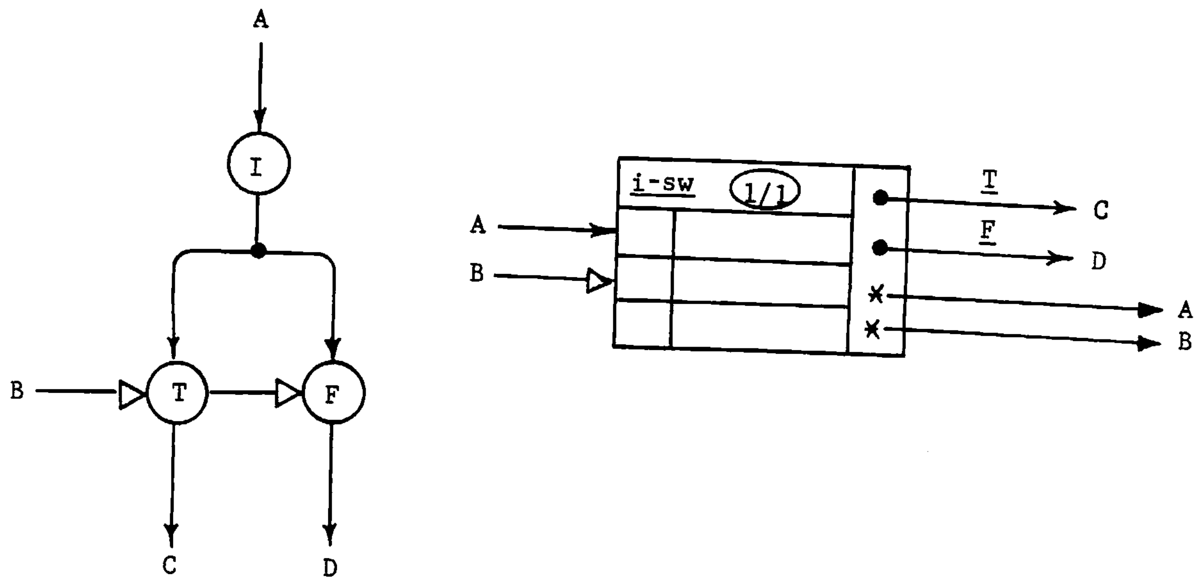


Figure 16. The switch instruction.

Processing Unit Operation

To illustrate the operation of the processing units, we consider the Int-Processor unit and the Cntl-Processor unit in more detail. Operation of the remaining units is similar.

As shown in Fig. 17, the Int-Processor unit consists of an Int-ALU module and an Int-Cntl module. The Int-Cntl submodule receives operation packets of the form

```
[inst: [opcode: (i-add, i-sub, bit, less), dest: array[1..5] of destination],
  opd: array[1: integer, 2: integer, 3: null] ]
```

On receipt of an operation packet, Int-Cntl determines whether addition, subtraction, bit test or comparison is required and sends a *command-pkt* of the form

```
[op: (i-add, i-sub, bit, less), op1, op2: integer]
```

to Int-ALU to request that the appropriate operation be performed. Upon receiving from Int-ALU a result-pkt of the form

```
[rtype: (BOOL, INT), value: {boolean, integer}]
```

Int-Cntl constructs control and data packets for transmission through the Control and Distribution Networks according to the destination fields of the operation packet.

Behavioral descriptions of the integer processor control unit and of the integer arithmetic logical unit are given in Figs. 18 and 19. Both descriptions are straightforward and should be easily understood.

The Int-Cntl module consists of two parts which are activated alternately. The first part waits for an operation packet to arrive; then it fetches the operands from the packet, determines if the operation code is either of the four allowed values, and dispatches to the Int-ALU module the two integer operands and a scalar value indicating whether addition, subtraction, bit test or integer

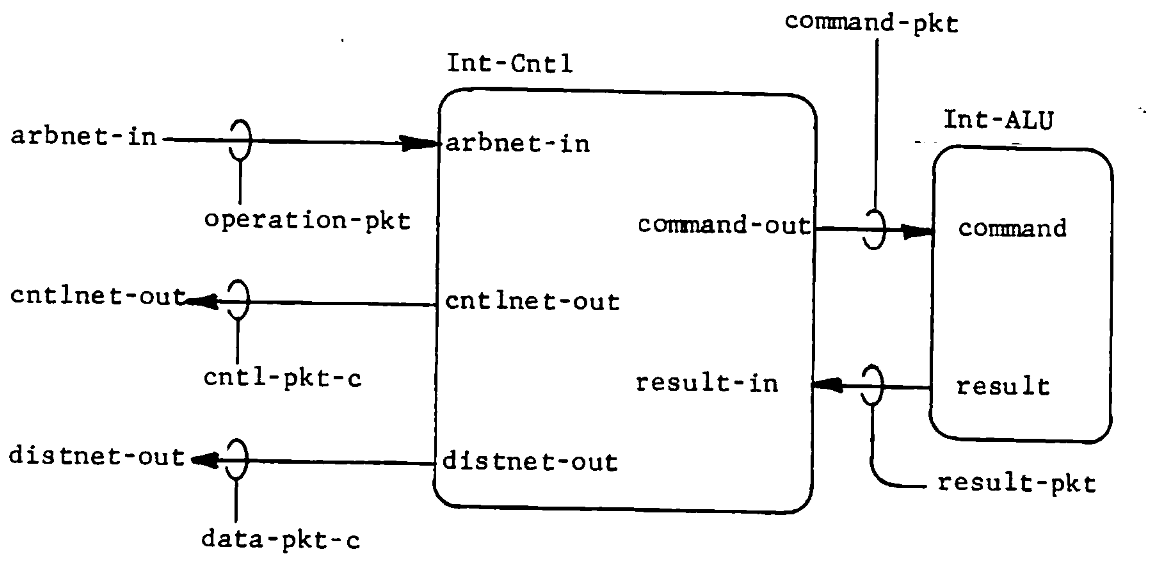
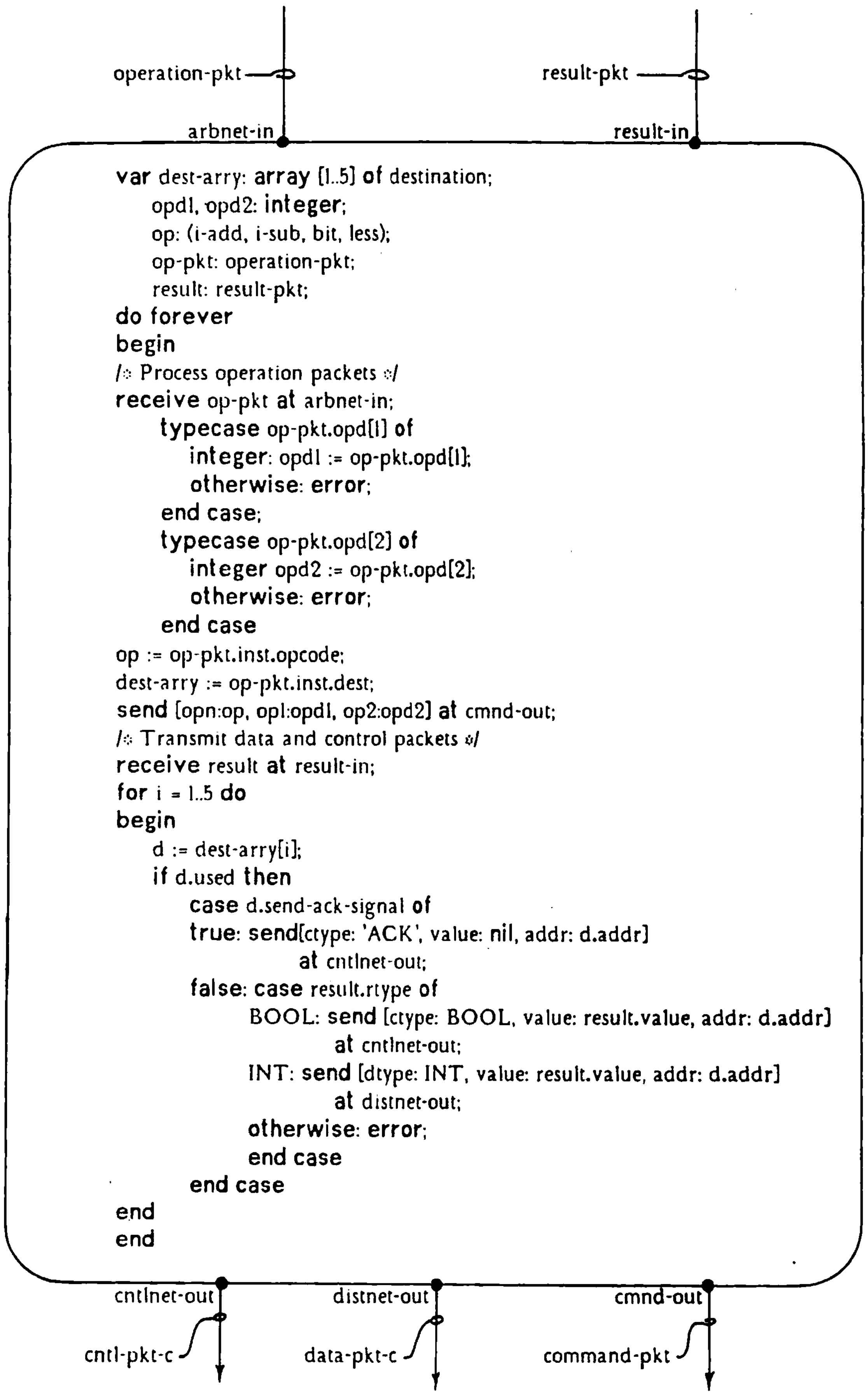
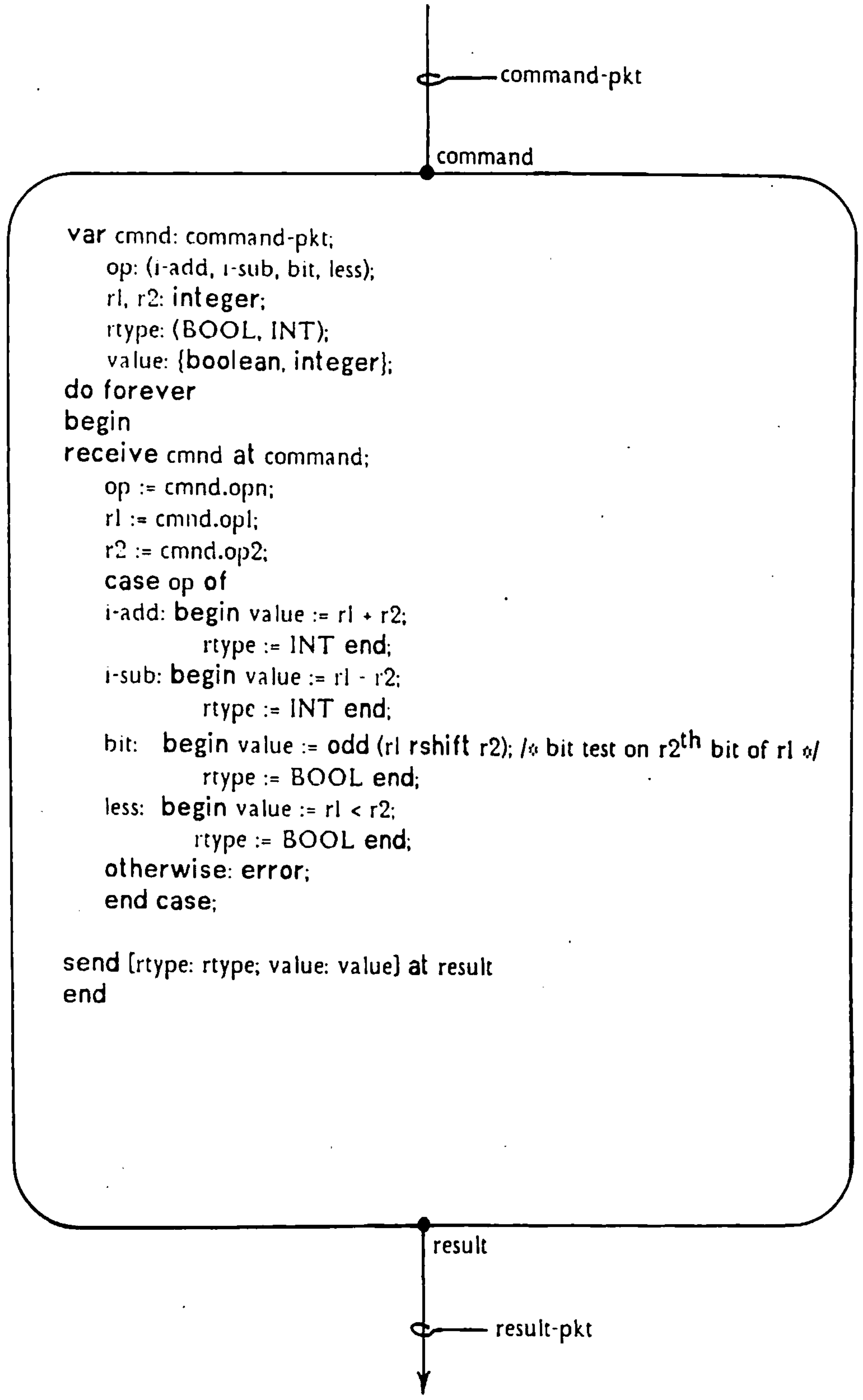


Figure 17. Structure of the Int-Processor processing unit.



```
var dest-arry: array [1..5] of destination;
    opd1, opd2: integer;
    op: (i-add, i-sub, bit, less);
    op-pkt: operation-pkt;
    result: result-pkt;
do forever
begin
/* Process operation packets */
receive op-pkt at arbnet-in;
     typecase op-pkt.opd[1] of
        integer: opd1 := op-pkt.opd[1];
        otherwise: error;
     end case;
     typecase op-pkt.opd[2] of
        integer opd2 := op-pkt.opd[2];
        otherwise: error;
     end case
op := op-pkt.inst.opcode;
dest-arry := op-pkt.inst.dest;
send [opn:op, op1:opd1, op2:opd2] at cmnd-out;
/* Transmit data and control packets */
receive result at result-in;
for i = 1..5 do
begin
    d := dest-arry[i];
    if d.used then
        case d.send-ack-signal of
        true: send[ctype: 'ACK', value: nil, addr: d.addr]
                  at cntlnet-out;
        false: case result.rtype of
               BOOL: send [ctype: BOOL, value: result.value, addr: d.addr]
                     at cntlnet-out;
               INT: send [dtype: INT, value: result.value, addr: d.addr]
                     at distnet-out;
               otherwise: error;
               end case
        end case
end
end
```

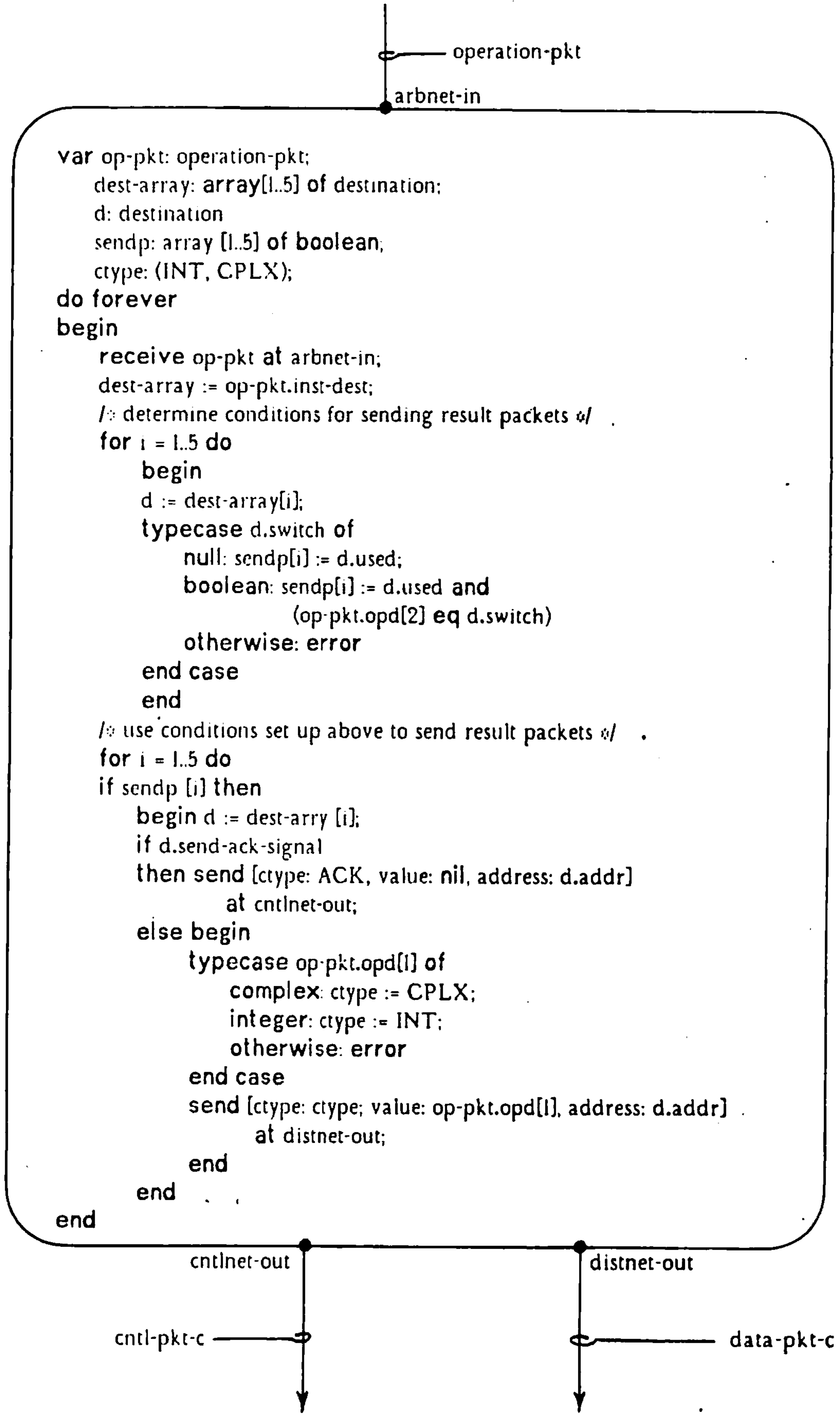
Figure 18. Behavioral description of the integer processor control unit.



```
var cmnd: command-pkt;
    op: (i-add, i-sub, bit, less);
    r1, r2: integer;
    rtype: (BOOL, INT);
    value: {boolean, integer};
do forever
begin
receive cmnd at command;
    op := cmnd.opn;
    r1 := cmnd.op1;
    r2 := cmnd.op2;
    case op of
    i-add: begin value := r1 + r2;
               rtype := INT end;
    i-sub: begin value := r1 - r2;
               rtype := INT end;
    bit:   begin value := odd (r1 rshift r2); /* bit test on r2th bit of r1 */
               rtype := BOOL end;
    less:  begin value := r1 < r2;
               rtype := BOOL end;
    otherwise: error;
    end case;

send [rtype: rtype; value: value] at result
end
```

Figure 19. Behavioral description of the Int-ALU unit.



```
var op-pkt: operation-pkt;
    dest-array: array[1..5] of destination;
    d: destination
    sendp: array [1..5] of boolean;
    ctype: (INT, CPLX);
do forever
begin
    receive op-pkt at arbnet-in;
    dest-array := op-pkt.inst-dest;
    /* determine conditions for sending result packets */
    for i = 1..5 do
        begin
        d := dest-array[i];
        typecase d.switch of
            null: sendp[i] := d.used;
            boolean: sendp[i] := d.used and
                        (op-pkt.opd[2] eq d.switch)
            otherwise: error
        end case
        end
    /* use conditions set up above to send result packets */
    for i = 1..5 do
    if sendp [i] then
        begin d := dest-arry [i];
        if d.send-ack-signal
        then send [ctype: ACK, value: nil, address: d.addr]
                at cntlnet-out;
        else begin
            typecase op-pkt.opd[1] of
                complex: ctype := CPLX;
                integer: ctype := INT;
                otherwise: error
            end case
            send [ctype: ctype; value: op-pkt.opd[1], address: d.addr]
                at distnet-out;
            end
        end
end
```

Figure 20. Behavioral description of the Cntl-Processor.

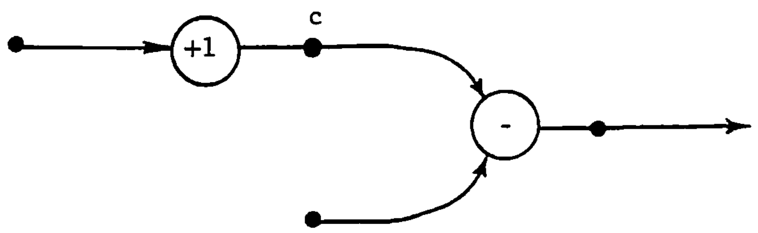
comparison is required. The components of the destination array of the operation packet specify what is done with the result value when it is returned by Int-ALU. For each destination, Int-Cntl sends an acknowledge packet, a boolean packet, a data packet, or nothing if the destination is marked as unused.

Use of the *switch* field is illustrated in the operation of the Cntl-Processor unit (Figure 20). For each operation packet received, the Cntl-Processor unit first of all decides which Instruction Cells should receive result packets. Each such cell is addressed by a destination marked as used; furthermore, if the *switch* field of the destination is set to either **true** or **false**, its value must match that of the boolean operand carried along in the operation packet. To each such cell the Cntl-Processor unit sends a result packet, just like the Int-Cntl module.

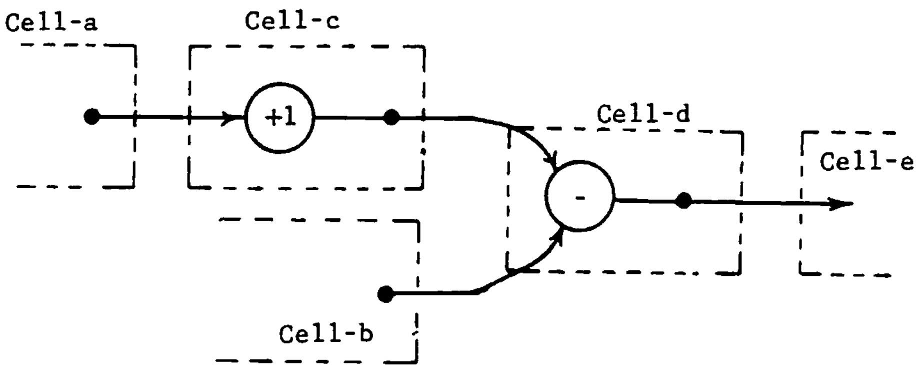
V. The Fast Fourier Transform Program

In this section we develop a complete machine level program for the FFT algorithm expressed as a data flow program graph in Figure 7. Groups of nodes in the program graph are encoded into machine instructions. For execution on the data flow processor, these machine instructions are loaded into Instruction Cells and linked together through cell identifiers stored in the destination fields of these instructions.

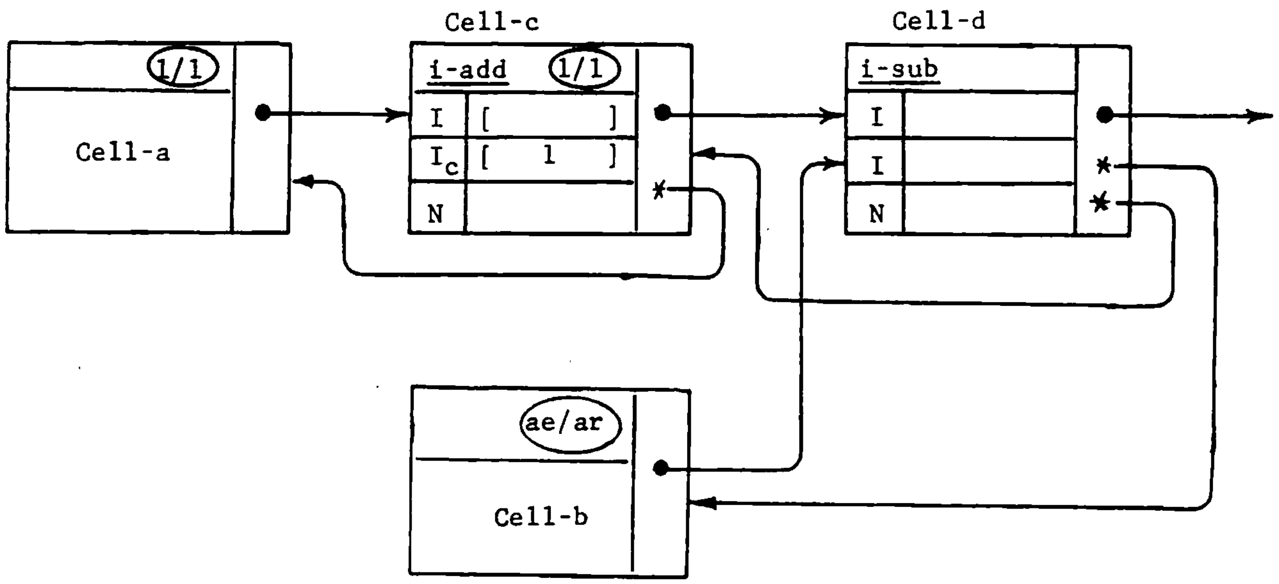
To illustrate the general procedure, let us consider the pair of data flow operators shown in Fig. 21a. First the program graph is partitioned (as in Fig. 21b, for example), where it is intended that each block be able to compute concurrently with others in pipeline fashion. Then the program graph is encoded into Instruction Cells using acknowldge signals so the machine level program correctly simulates the program graph firing rules. Partitioning of this program segment and the corresponding machine instruction encoding are shown in Figs. 21b and 21c, respectively. Each execution of the instructions in Cell-c delivers a data packet to Cell-d. Each execution of the instruction in Cell-d returns an acknowledge packet to Cell-c, and the re-enabling of Cell-c is



(a) program graph



(b) partition



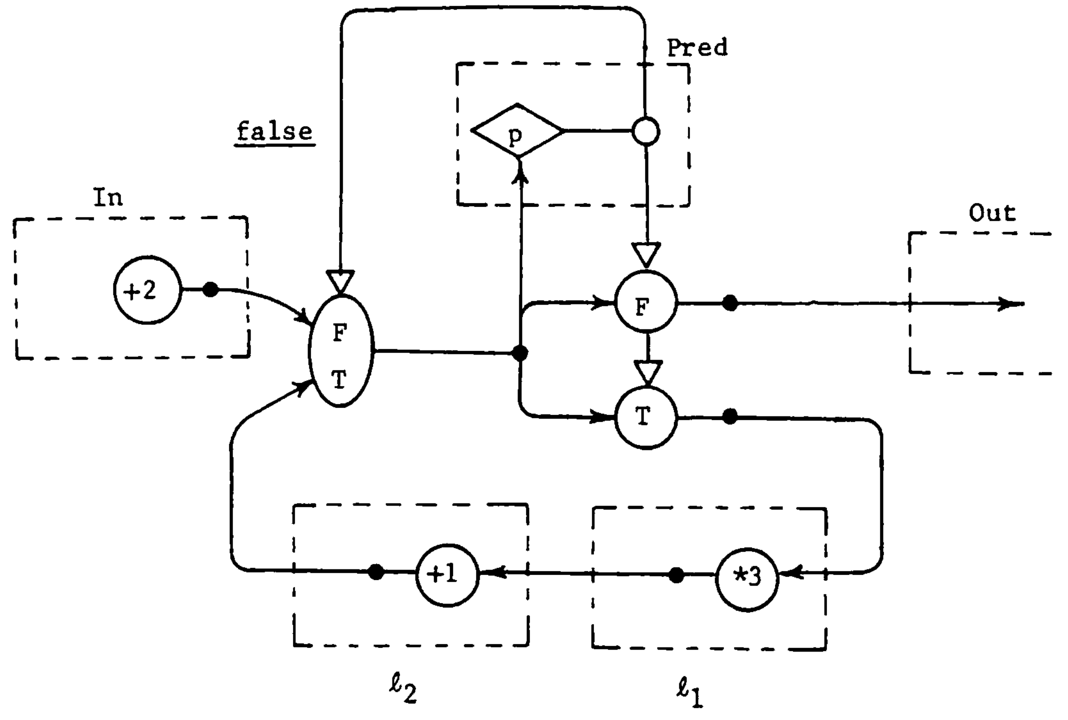
(c) machine level representation

Figure 21. Machine level representation of program graphs.

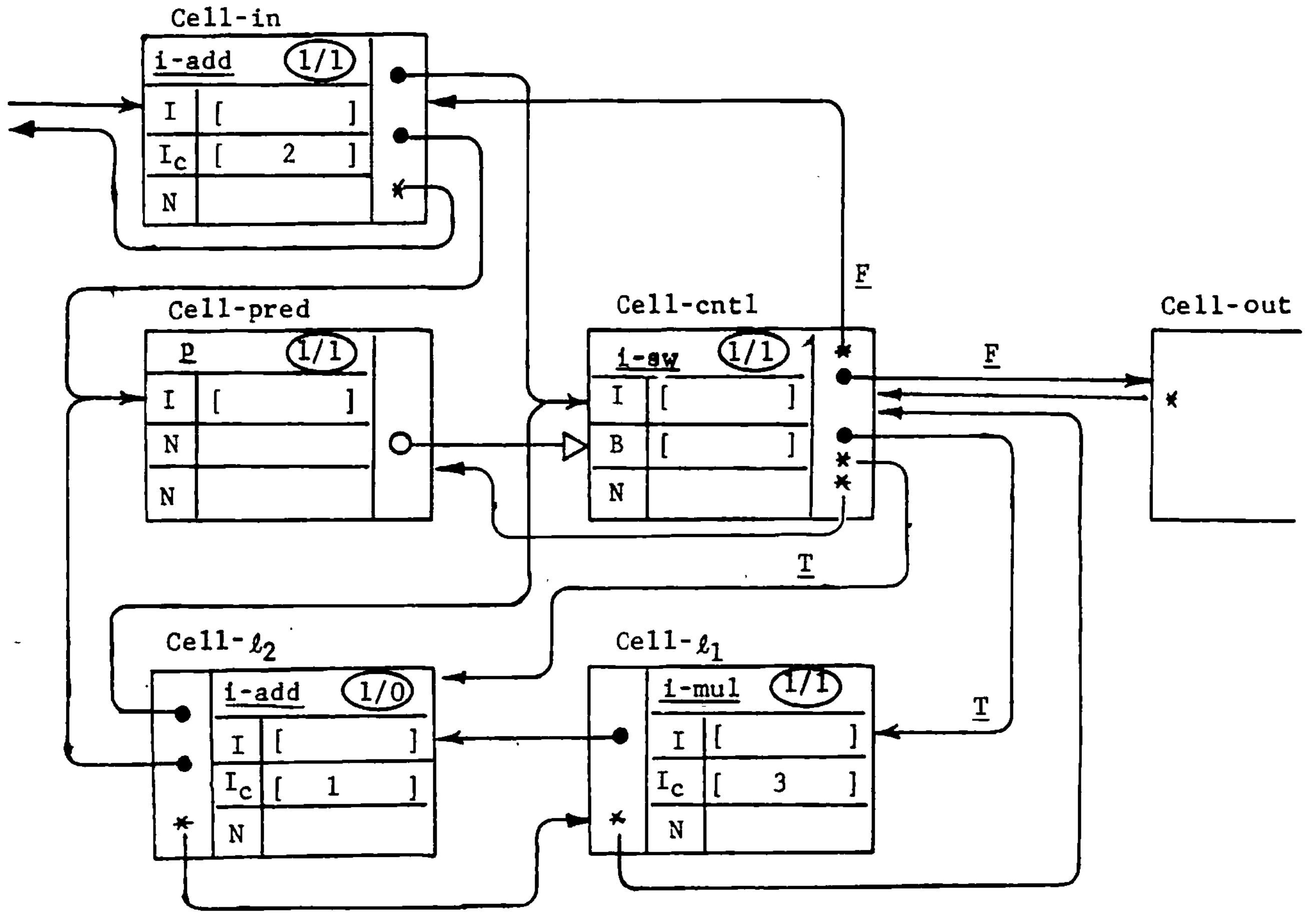
predicated upon the receipt of this acknowledge packet (c.f. Instruction Cell Operation, Section IV). According to the semantics of data flow graphs under the firing rule (Section II), data link c encoded in Cell-c cannot fire until its output arc is empty. This condition is signalled during the execution of a machine language program by sending an acknowledge packet from Cell-d to Cell-c. To indicate the necessary synchronization, the pair of numbers within the ellipse in an Instruction Cell (Fig. 21c) specifies the number of acknowledge signals required to enable the cell, ae, and the number of signals presumed to have been received in the specified configuration, ar. These variables are initialized to their proper values for each instruction in a machine language program. It should be noted that when a machine instruction encodes several actors and their output links, each execution of the machine instruction corresponds to the firing of the actors followed immediately by the firing of the output links. Hence such a machine instruction must not be enabled until the instruction cells implementing the output arcs of these data links are all free to receive the next set of data.

A detailed discussion on the deadlock problems that may arise in a data flow processor supporting iteration if acknowledgements are not provided in a machine language program can be found in [17].

The main body of the FFT program graph (Fig. 7) is an iteration construct. To facilitate the subsequent presentation, an example of an iteration construct and its machine level implementation are shown in Fig. 22. Initially Cell-in (Fig.22)is enabled upon receiving a data packet. Each time the predicate p in Cell-pred evaluates to **true**, execution of the instruction in Cell-cntl delivers a data packet to Cell-l_1 and an acknowledge packet to Cell-l_2, allowing a new iteration. If p evaluates to **false**, the output of the iteration construct is sent to Cell-out and an acknowledge packet is sent to Cell-in, allowing the iteration construct to be re-entered. Cell-cntl can be re-enabled upon receiving an acknowledgement from either Cell-l_1 or Cell-out. No



(a) program graph



(b) machine level implementation

Figure 22. Machine level representation of an iteration construct.

acknowledgement is needed from Cell-pred to Cell-in and Cell-I_I. The reader should convince himself that the cell configuration in Fig. 22 implements an iteration construct correctly, according to the firing rule for program graphs.

The data flow program graph for the FFT algorithm (Fig. 7) consists of five major sections: Phase Constant Generation, Loop Control, Phase Factor Generation, Butterfly and Distribution Trees. We consider each section in turn.

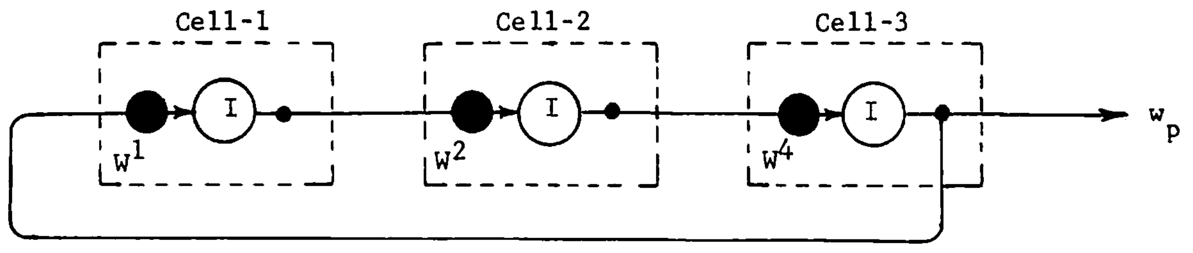
Phase Constant Generation

The Phase Constant Queue section of the FFT program graph (reproduced in Fig. 23)[3] might be partitioned into three blocks (Fig. 23a) for machine level encoding. This yields the instruction cell configuration in Fig. 23b. However, we have already noted that this partition should not be used naively to derive the machine language representation in Fig. 23b. The difficulty is that executing any instruction in Fig. 23b results in delivering a data packet to an occupied cell receiver, in violation of the firing rule. To avoid this problem, we use the partition in Fig. 23c, after adding an identity operator, to derive the machine language representation in Fig. 23d.

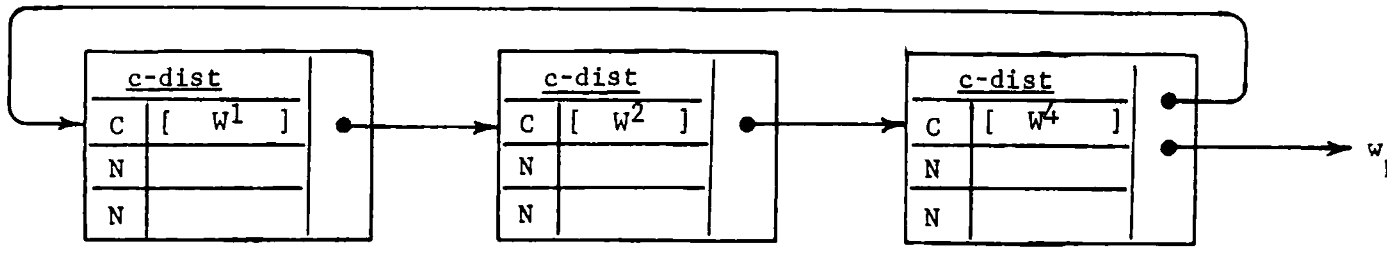
Loop Control

The data flow graph of the Loop Control section is reproduced in Fig. 24. Partitioning and implementation of this program graph (Figs. 24a and b) follows closely the strategy illustrated in Fig. 22. Note that each execution of the instruction held in Cell-cntl delivers two acknowledge packets back to Cell-cntl, so that the enabling condition of Cell-cntl does not depend on the decision

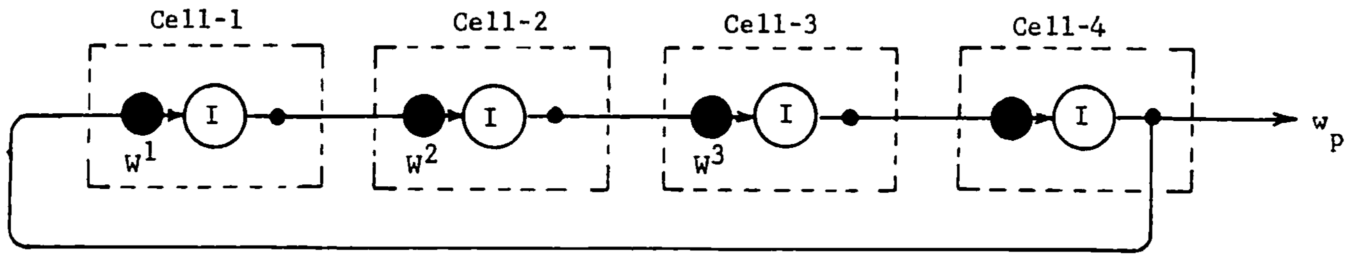
3. To avoid clustering up the figures, we have introduced a new arc type $\relbar\!\!\!+\!\!\!\rightarrow$ to denote a pair of arcs delivering a data packet in the forward direction and an acknowledge packet in the opposite direction.



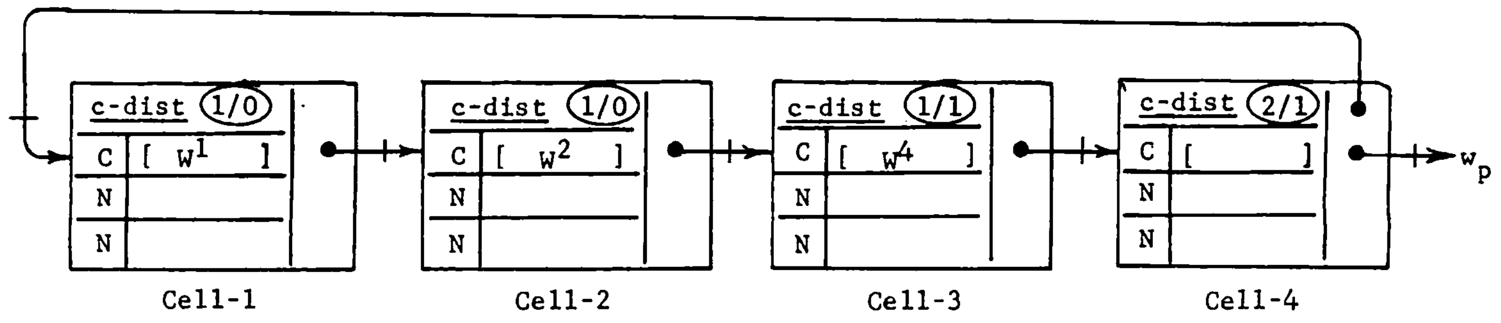
(a) phase constant queue with partitioning.



(b) machine language program from (a)

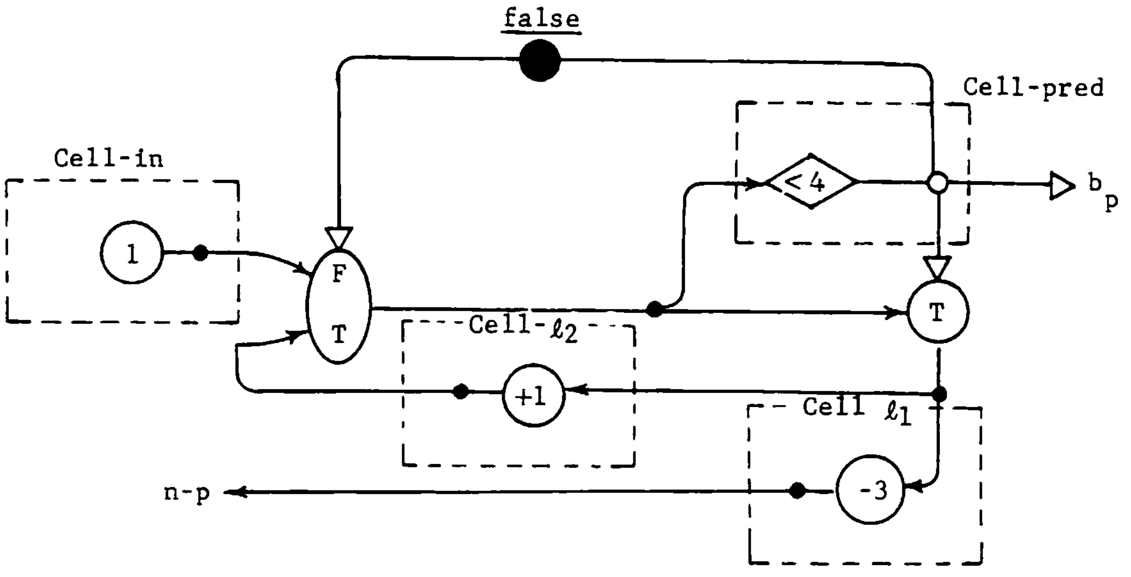


(c) adding an identity operator

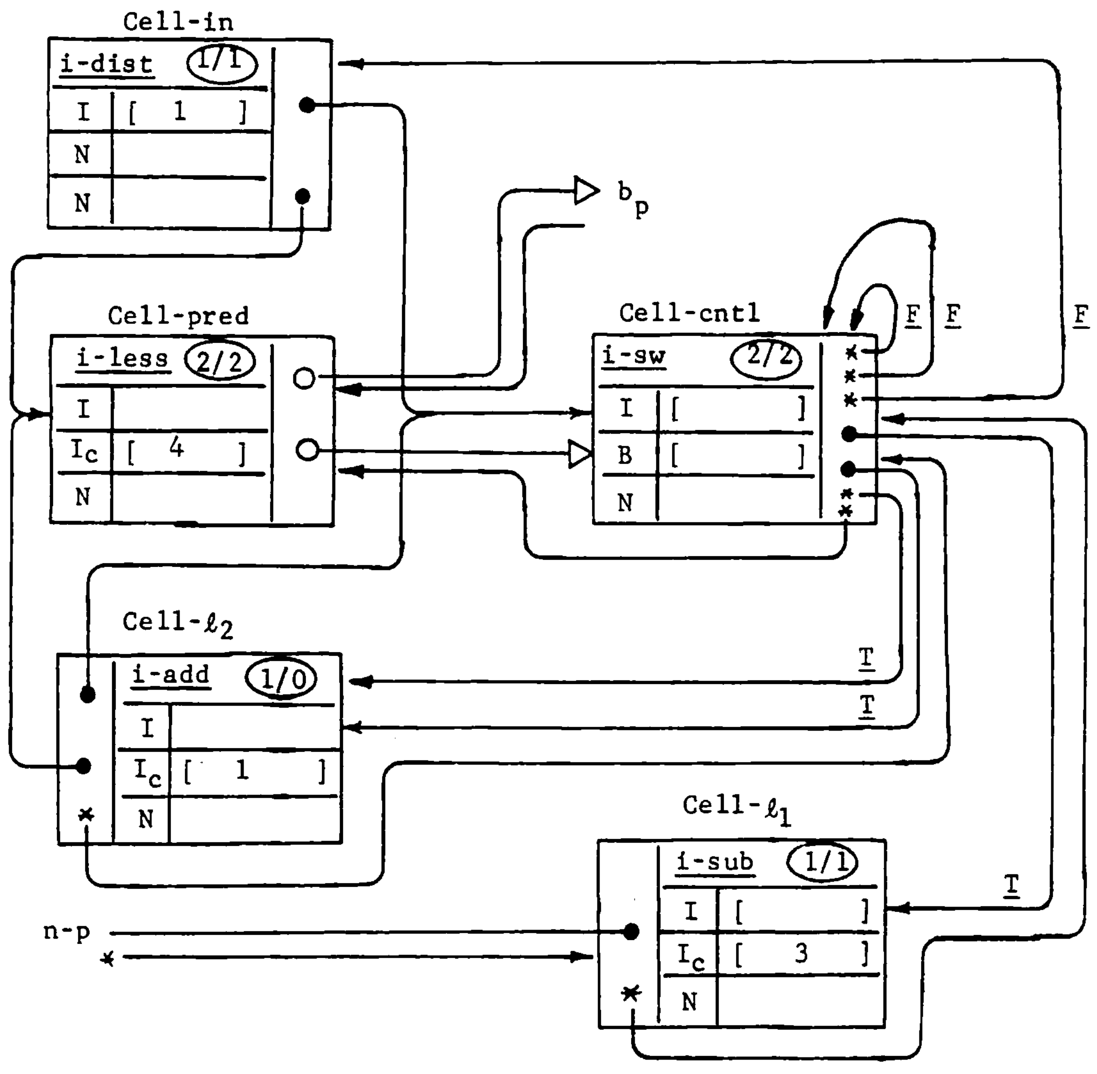


(d) machine language program from (c)

Figure 23. Machine level representation of the phase constant queue.



(a) partitioning the program graph



(b) machine level representation

Figure 24. Machine level representation of the loop control section.

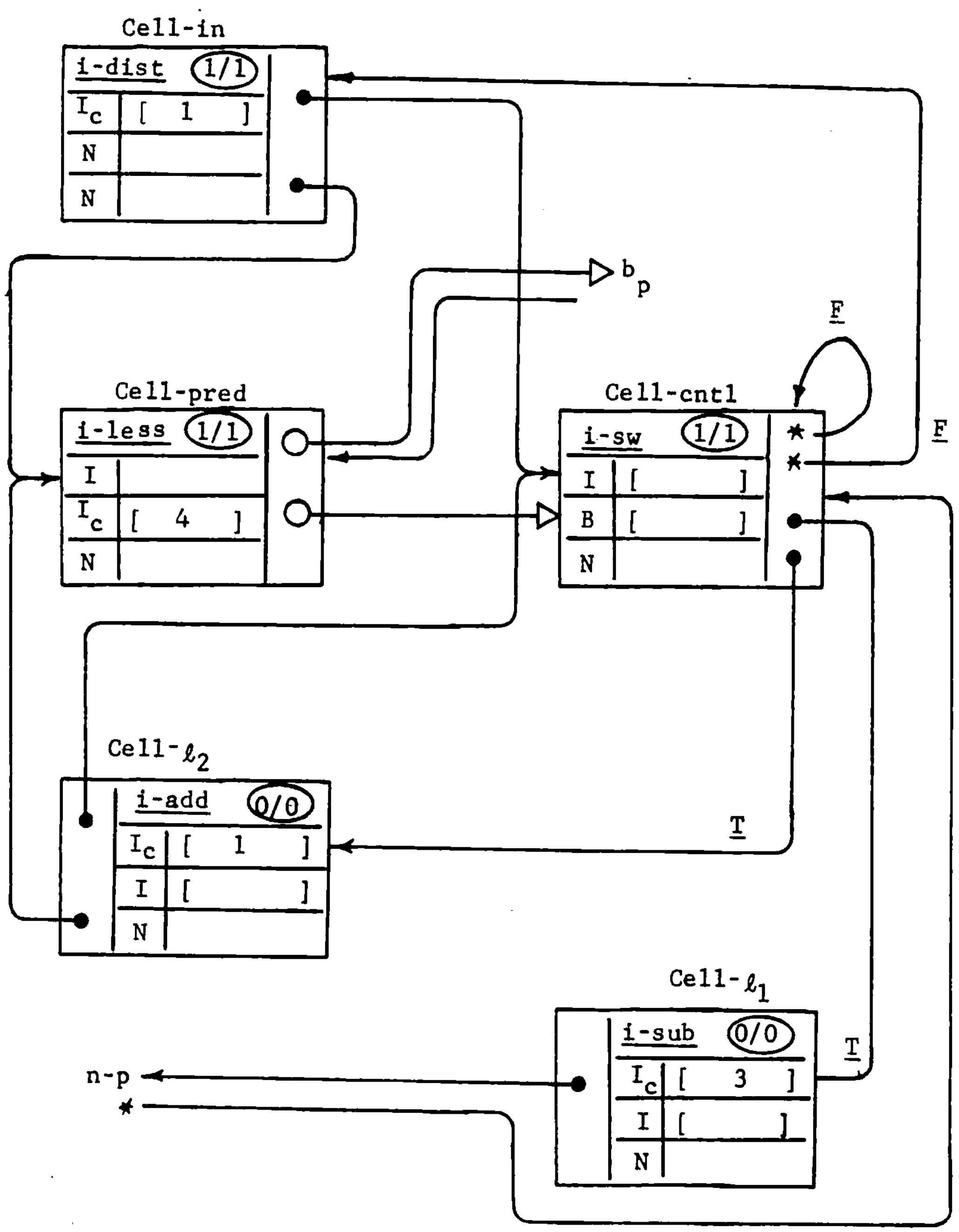


Figure 25. An optimized machine level implementation of the loop control section.

outcome (a boolean operand) it receives from Cell-pred. A slightly optimized implementation of the Loop Control section is given in Fig. 25, obtained by applying the following transformations to the machine level representation of Fig. 24b:

i. Whenever execution of the i-sw instruction held in Cell-cntl delivers a data packet to Cell-l_2, an acknowledge packet is also delivered to Cell-l_2. The acknowledgement can be omitted and the enabling condition of Cell-l_2 adjusted accordingly. Similarly acknowledgement from Cell-l_2 to Cell-cntl can be omitted.

ii. Each time Cell-l_1 receives an acknowledgement, it will send one in turn to Cell-cntl. The latter can be eliminated by sending the acknowledgement intended for Cell-l_1 to Cell-cntl directly (Fig. 25).

iii. Every execution of the instruction in Cell-cntl leads indirectly, either through Cell-in or Cell-l_1, to delivery of a data packet to Cell-pred. Acknowledgement from Cell-cntl to Cell-pred can be omitted and the enabling condition of Cell-pred adjusted accordingly.

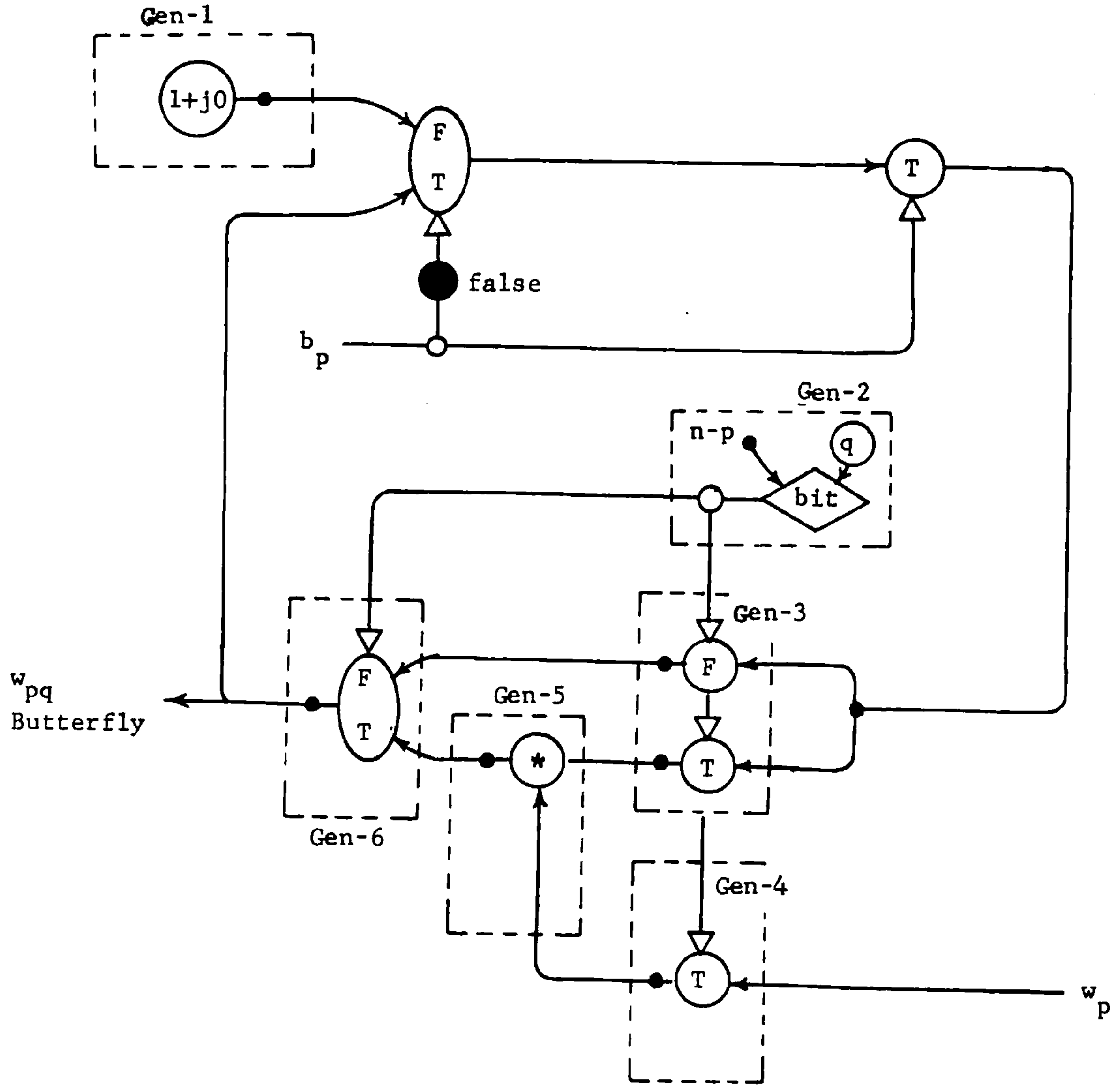
These optimizing transformations illustrate the opportunities for reducing the number of acknowledgements needed to implement the firing rule correctly.

Phase Factor Generation

For each iteration of the FFT algorithm, each butterfly section receives a phase factor generated by the program graph shown in Fig. 26. To derive the machine level implementation, we partition the program graph as in Fig. 26a. The corresponding machine language program is given in Fig. 26b. Noting that several acknowledgements have been eliminated through optimization, the reader should again convince himself that this machine level program correctly implements the program graph.

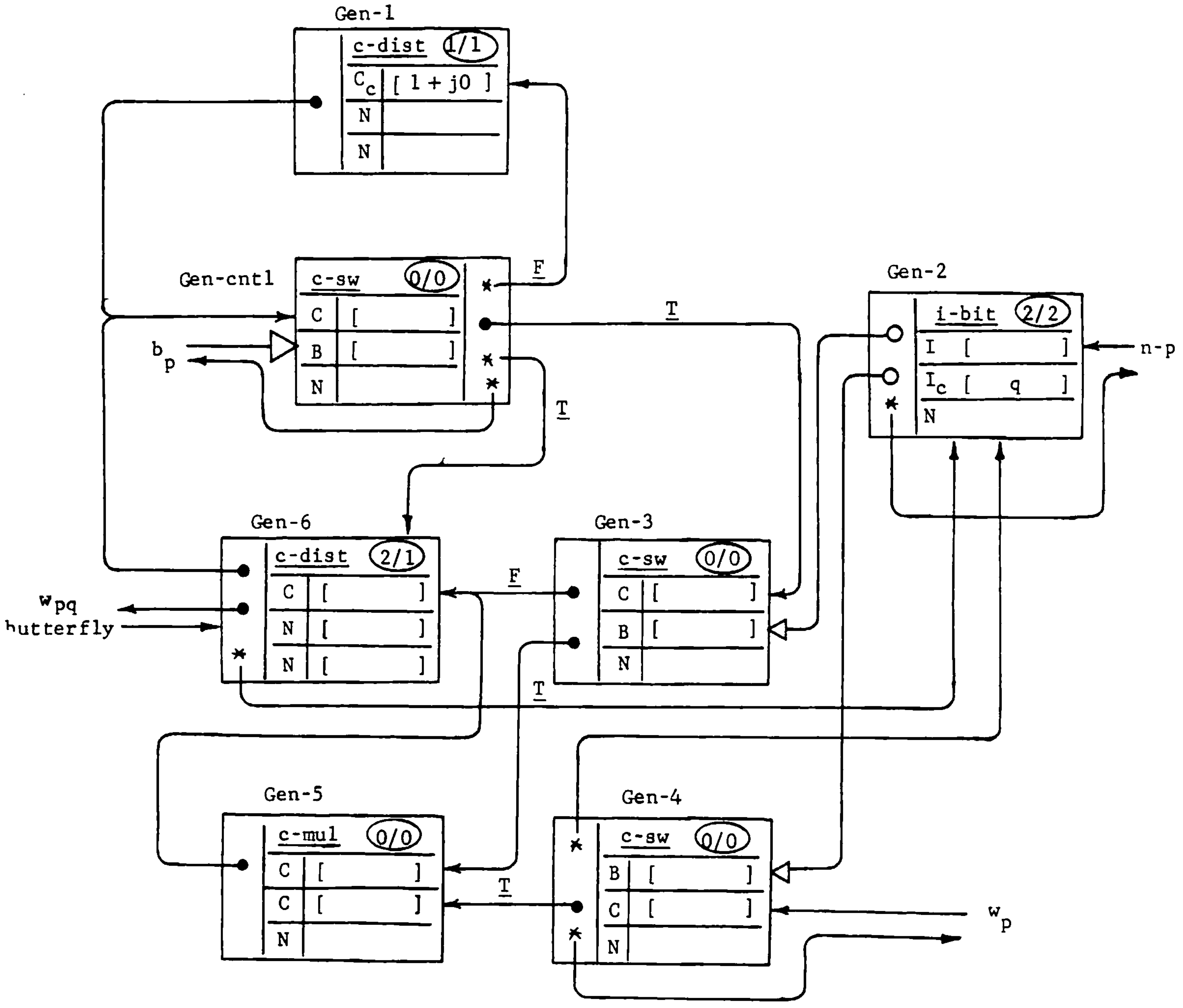
Butterfly Section

Derivation of a machine level program from the program graph of a Butterfly section is

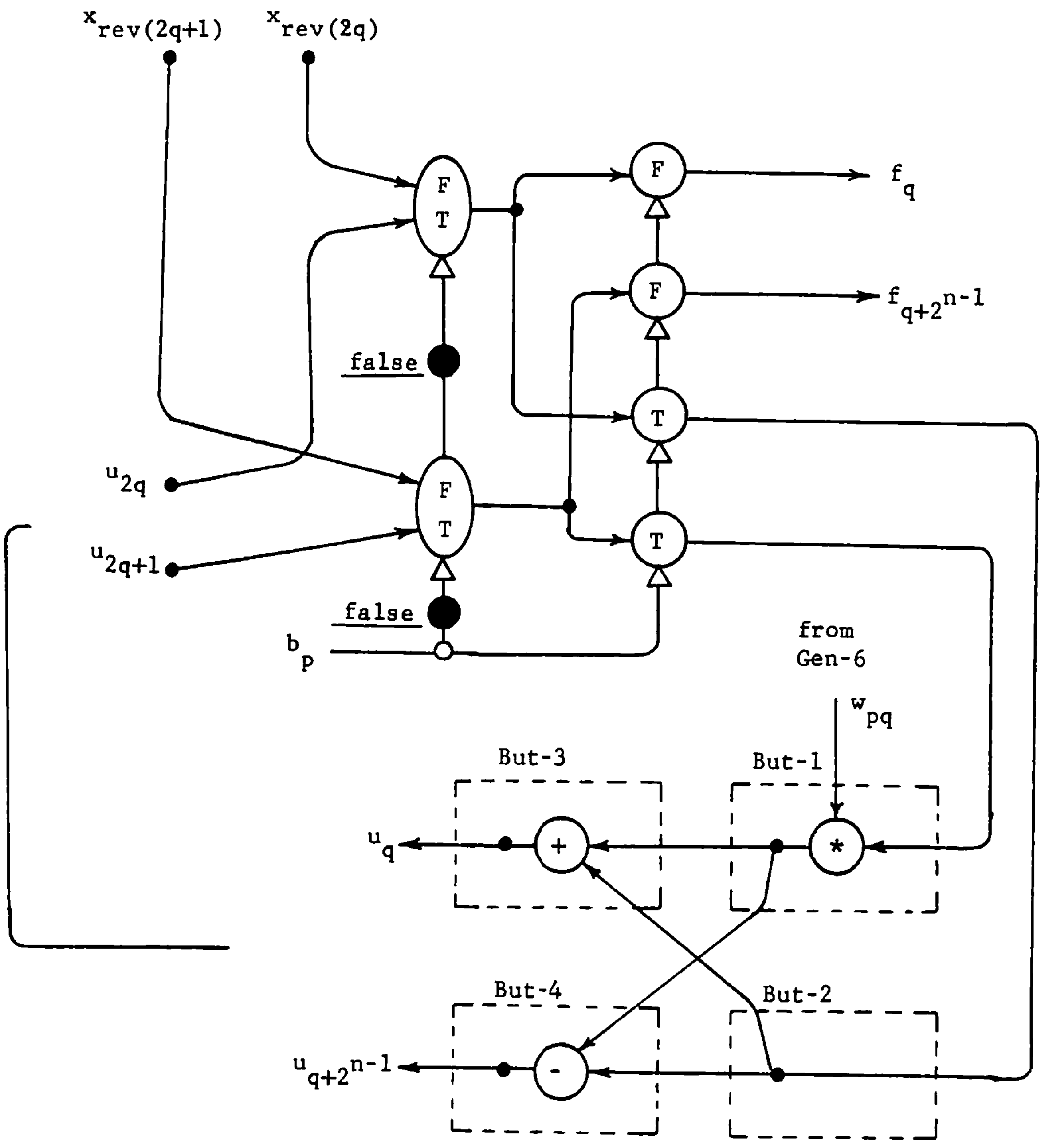


(a) partitioning the program graph for phase factor generation

Figure 26. Machine level representation of phase factor generation.

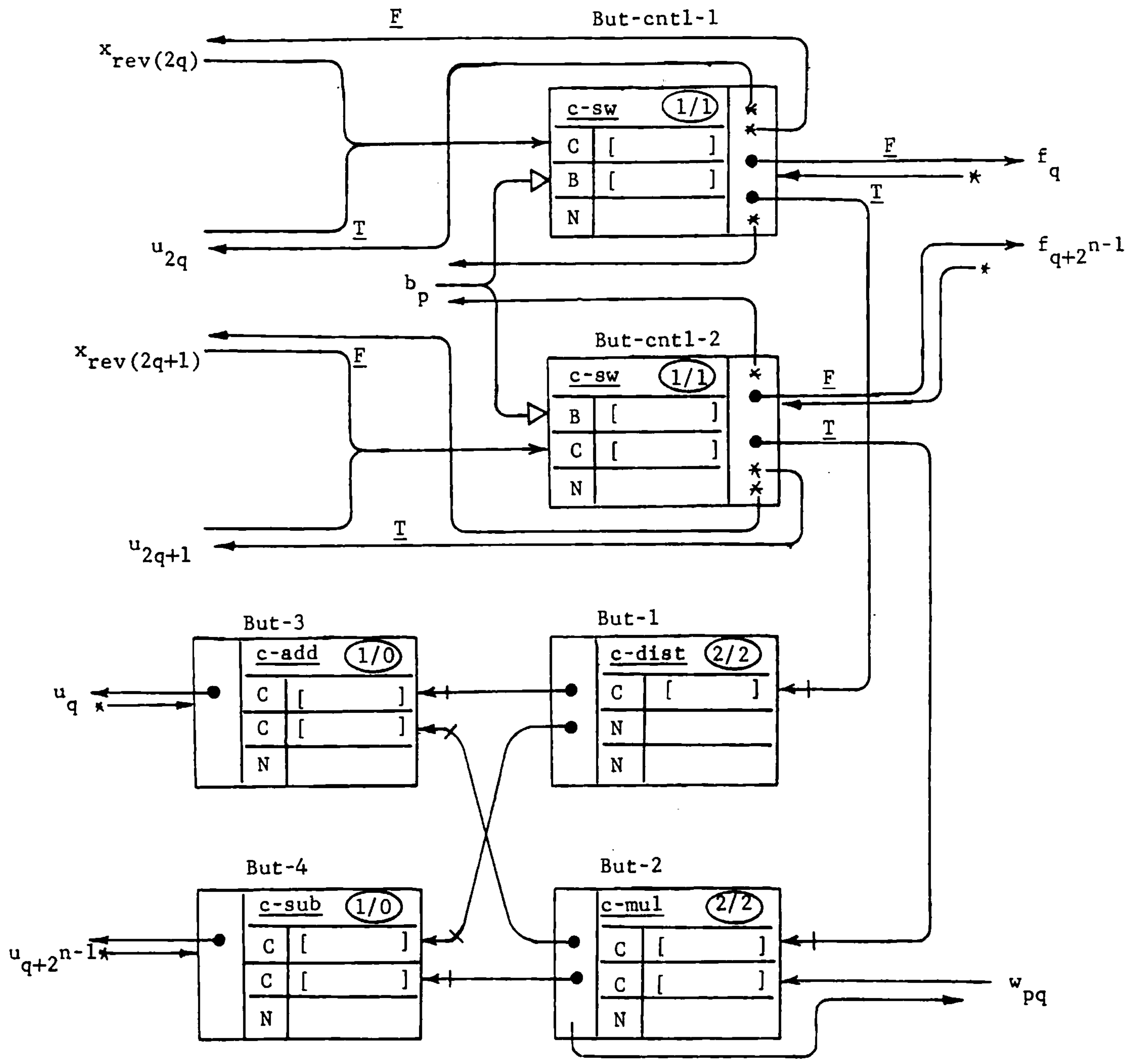


(b) machine level implementation



(a) partitioning the program graph

Figure 27. Machine level representation of the butterfly section.



(b) machine level implementation

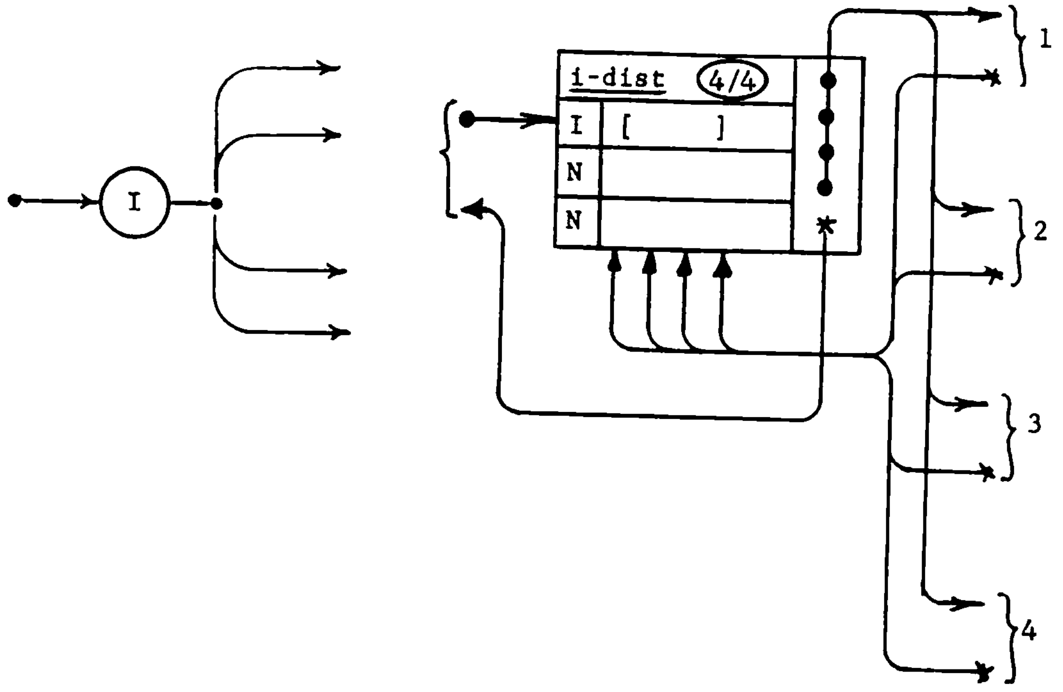


Figure 28. One unit of a distribution tree and its implementation.

illustrated in Fig. 27. Generating the cell configuration for performing the required arithmetic is straightforward. Each butterfly section also contains two control cells to receive either a pair of inputs (through $x_{rev(2q)}$, $x_{rev(2q+1)}$) to initiate a new FFT computation, or a pair of intermediate results (through u_{2q}, u_{2q+1}) from the previous stage of the current FFT computation.

Distribution Trees

For each stage of the FFT computation, the values b_p, n-p and w_p must be distributed to the Phase Factor Generation sections and the Butterfly sections. This may be done by three Distribution Trees made up of units as shown in Fig. 28. Use of these units to construct Distribution Trees allows enough skew in the execution of the Phase Factor Generation and Butterfly sections that the computation for successive stages of the FFT may overlap substantially.

VI. Routing Network Structure and Performance

The Arbitration, Distribution and Control Networks of the data flow processor are examples of *routing networks* that perform the function of directing packets to one of several or many physical units of the processor. If the parallelism represented in the data flow form of algorithms such as the FFT is to be exploited by the kind of machine we have described, these routing networks must be structured so they can handle many packets concurrently. If a high degree of parallelism is supported, then it is not crucial that each unit acts in the fastest possible time on information it receives to achieve balanced utilization of sections of the machine.

A structure for the Arbitration Network is shown in Fig. 29a. It is built of arbitration units, switch units and buffer units: Each *arbitration unit* passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any ambiguity about which packet should be sent next. A *switch unit* assigns a packet at its input to one of its output ports according to some property of the packet, the operation code in the case of the Arbitration

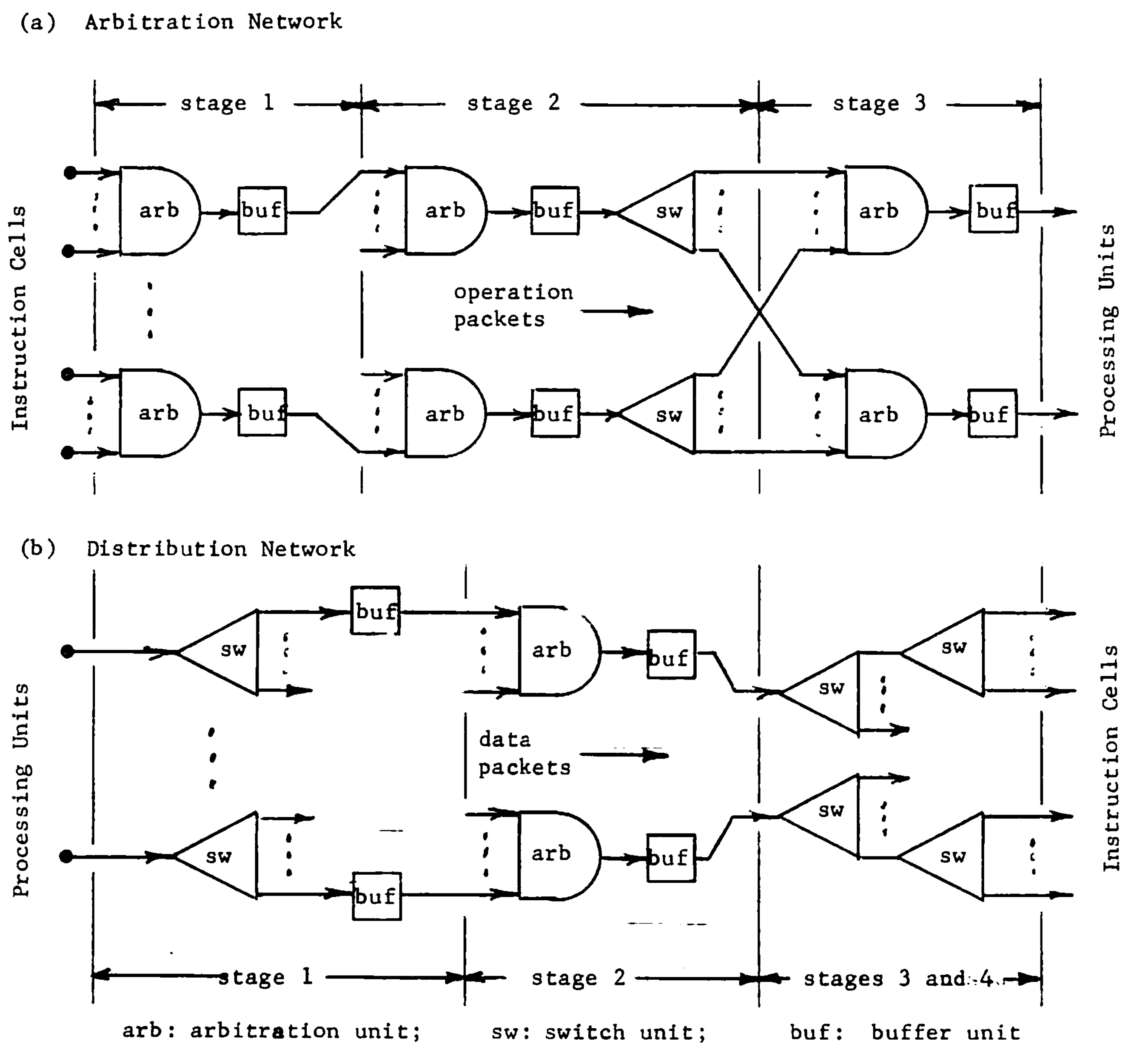


Figure 29. Structure of the Arbitration and Distribution Networks.

Network. A *buffer unit* stores a packet until the succeeding switch or arbitration unit is ready to accept it. The network shown has three stages: stages 1 and 2 have arbitration units that funnel operation packets from many Instruction Cells into a smaller number of more heavily utilized channels; the switch units in stage 2 split the streams of operation packets into separate streams for each Processing Unit; and the output streams of the switch units are merged by stage 3 into a single stream for each Processing Unit.

The Distribution Network (Fig. 29b) is structured in a similar manner and provides a path for data packets from each Processing Unit to each Instruction Cell. Switch units direct each data packet toward the appropriate Cell by examining bits of the cell identifier in the packet's destination address. A few arbitration units are needed in the Distribution Network to provide for merging the flow of data packets from different Processing Units to the same group of Instruction Cells.

Since the Arbitration Network has many inputs, a serial format is appropriate for packet transfer between Instruction Cells and the Arbitration Network to reduce the number of connections needed. However, to achieve a high rate of packet flow at the output ports, a parallel format is required. For this reason, serial-to-parallel is done within the buffer units as a packet travels through the Arbitration Network. Parallel-to-serial conversion is performed in the Distribution Network for similar reasons.

The structure of the Control Network is similar to that of the Distribution Network. However, the packets passing through the Control Network convey either simple boolean values or acknowledge signals, and parallel-to-serial conversion of packets is not required; thus the Control Network is composed of only switch units and arbitration units.

We now turn to an analysis of the performance achievable by the data flow processor in performing the FFT computation using the programs of Figs. 23, 25, 26, 27, and 28.

Computation by the data flow processor will be at the maximum rate permitted by the three routing networks (Arbitration, Distribution and Control) provided the capacities of the Processing Units are not exceeded and provided sufficiently many Instruction Cells are enabled to keep the Arbitration Network supplied with operation packets. Two factors control the number of enabled instructions: delays in the passage of packets through the routing networks and constraints on the enabling of instructions imposed by the structure of the machine level program.

Our plan of analysis is as follows: First we determine the maximum computation rate permitted by the Processing Units for the FFT program. Then we consider suitable structures for routing networks able to support this computation rate, and compute the minimum packet transit time for each network. Finally, we show that the processing rate assumed is consistent with the sequencing constraints embodied in the machine level FFT program.

Table 3 gives the number of operation packets that must be processed during one stage of the 1024-point FFT computation, and the number of data and control packets that must be distributed. For determining computation rate, we shall use as a basic measure the rate at which a Processing Unit can receive bytes at a port. Let this rate be 5 MHz, corresponding to use of a medium speed logic family. Since serial-to-parallel conversion is carried out in the Arbitration Network, this is also the maximum rate at which a Processing Unit receives operation packets. We suppose that operation packets can indeed be received every 200 nanoseconds by the Distributor, Int-Processor and Cntl-Processor units, and at half this rate by the two complex arithmetic Processing Units. As shown in Table 4, the Processing Units are able to support execution of one stage of the 1024-point FFT every 512.2 microseconds where this limit on computation rate is set by the speed of the Cntl-Processor unit.

For the 1024-point FFT, 7349 Instruction Cells are required to hold the machine level data flow program, so let us hypothesize a processor having 8192 = 2^{13} Instruction Cells. We suppose the

	Instruction Cells	Multiplier			Adder			Distributor			Int-Processor			Cntl-Processor		
		O	D	C	O	D	C	O	D	C	O	D	C	O	D	C
Butterfly Units	3072	512	1024	1024	1024	1024	2048	512	1024	512	--	--	--	1024	1024	2048
Phase Factor Generation	3584	256	256	--	--	--	--	512	1024	512	512	0	1536	1536	1280	2048
Distribution Trees	684	--	--	--	--	--	--	684	1368	2052	--	--	--	--	--	--
Loop Control	5	--	--	--	--	--	--	--	--	--	3	3	2	1	2	0
Phase Constant Queue	4	--	--	--	--	--	--	4	5	4	--	--	--	--	--	--
Totals	7349	768	1280	1024	1024	1024	2048	1712	3421	3080	515	3	1538	2561	2306	4096

O - operation packets; D - data packets; C - control packets

Table 3. Packet counts for one stage of the FFT computation.

Table 4.

Processing Unit	Time per Packet	Operation Packets Per Stage	Period for One Stage
Multiplier	400 ns	768	307.2 microsec.
Adder	400 ns	1024	409.6 microsec.
Distributor	200 ns	1712	342.4 microsec.
Int-Processor	200 ns	515	103.0 microsec.
Cntl-Processor	200 ns	2561	512.2 microsec.

Table 5. Design Parameters for an Arbitration Network

	fan-in	fan-out	format	arbitration units	flow rate
Stage	P_i	q_i	$s_i \times t_i$	n_i	$R_i = n_i/(s_i \times T)$
Stage 1	8	1	48 × 3	1024	106.6 MHz
Stage 2	8	1	12 × 12	128	53.3 MHz
Stage 3	8	1	3 × 48	16	26.6 MHz
Stage 4	4	4	1 × 144 4	20 MHz	
Stage 5	4	5/4	1 × 144 4	20 MHz	

Arbitration Network has the structure specified in Table 5. Each stage consists of a rank of arbitration units having fan-in p_i, a rank of buffer units, and rank of switch units having fan-out q_i. The number of arbitration or switch units in stage i is n_i and these numbers satisfy the relation

$$n_i q_i = n_{i+1} p_{i+1}, \quad i = 1, \ldots, 4$$

which expresses the condition that the number of output links from stage i must equal the number of input links to stage i + 1. In stage i, the input packets are represented by s_i bytes of t_i bits each. The formats specified in the table assume that operation packets are 144 bits in length, and that serial-to-parallel conversions are done by the buffers in stages 1, 2 and 3. The packet flow rate R_i for each stage is computed as the number of channels (one per arbitration unit) divided by the time required to transmit the bytes of a packet serially; the time T for transmitting one byte is assumed to be 200 nsec.

The design parameters of this Arbitration Network have been chosen so that the full 20 MHz capacity of the Processing Section can be met. The early stages of the network have generous capacity to accommodate shifts of activity among the Instruction Cells during the running of a computation. For this network, the minimum transmit time is

$$T_A = \sum_{i=1}^{5} s_i\, T = (48+12+3+1+1)T = 13 \text{ microseconds}$$

Let us suppose that the Control Network and the Distribution Network of our hypothetical processsor are similarly designed to accommodate the flows of data packets and control packets indicated in Table 3. It is reasonable to assume approximately equal transit times for the Distribution Network and the Arbitration Network, and about one fifth of this time for the Control Network as it is much simpler than the others:

T_D = 13 microseconds -- Distribution Network delay

T_C = 3 microseconds -- Control Network delay

Now we may ask whether these transit times and the computation rates of the Processing Units are consistent with the sequencing constraints imposed by the FFT program. To answer this question, we must determine how the rate of executing the sequence of n stages of the FFT computation is limited by the structure of the machine level program. We consider the Phase Factor Generation and Butterfly sections of the program since these make up the major portion of the iterative computation. The following analysis is readily extended to include the iterative Loop Control section and the distribution trees without effect on the conclusion. For computing the period of computation we may assume that the boolean values arriving at input b_p are all **true** so the switch instructions in cells Gen-cntl, But-cntl-1, and But-cntl-2 always transmit results and signals to their T-destinations and never to their F-destinations. Thus cell Gen-1 does not participate in the periodic behavior we wish to analyze.

We shall make two simplifying assumptions of a conservative nature: The first is to suppose that each Butterfly Unit sends its results to itself rather than to other Butterfly Units. This assumption is justified by the symmetry of the interconnection pattern of the Butterfly Units. We further assume that the boolean output of cell Gen-2 is always **true** since this choice invokes execution of the multiplication in cell Gen-5, and will yield the worst case period for the computation.

With these assumptions, the cyclic execution of the FFT program may be accurately represented by a special kind of Petri net known as a marked graph [12]. In Fig. 30 each node of the marked graph corresponds to an Instruction Cell participating in the cyclic computation or to a source of input values from the Loop Control Section. Each directed arc represents a data path between cells specified by one destination of an instruction. The arrowheads of the arcs indicate the type of the packets -- data, boolean, or signal -- that flow over the corresponding path. Tokens

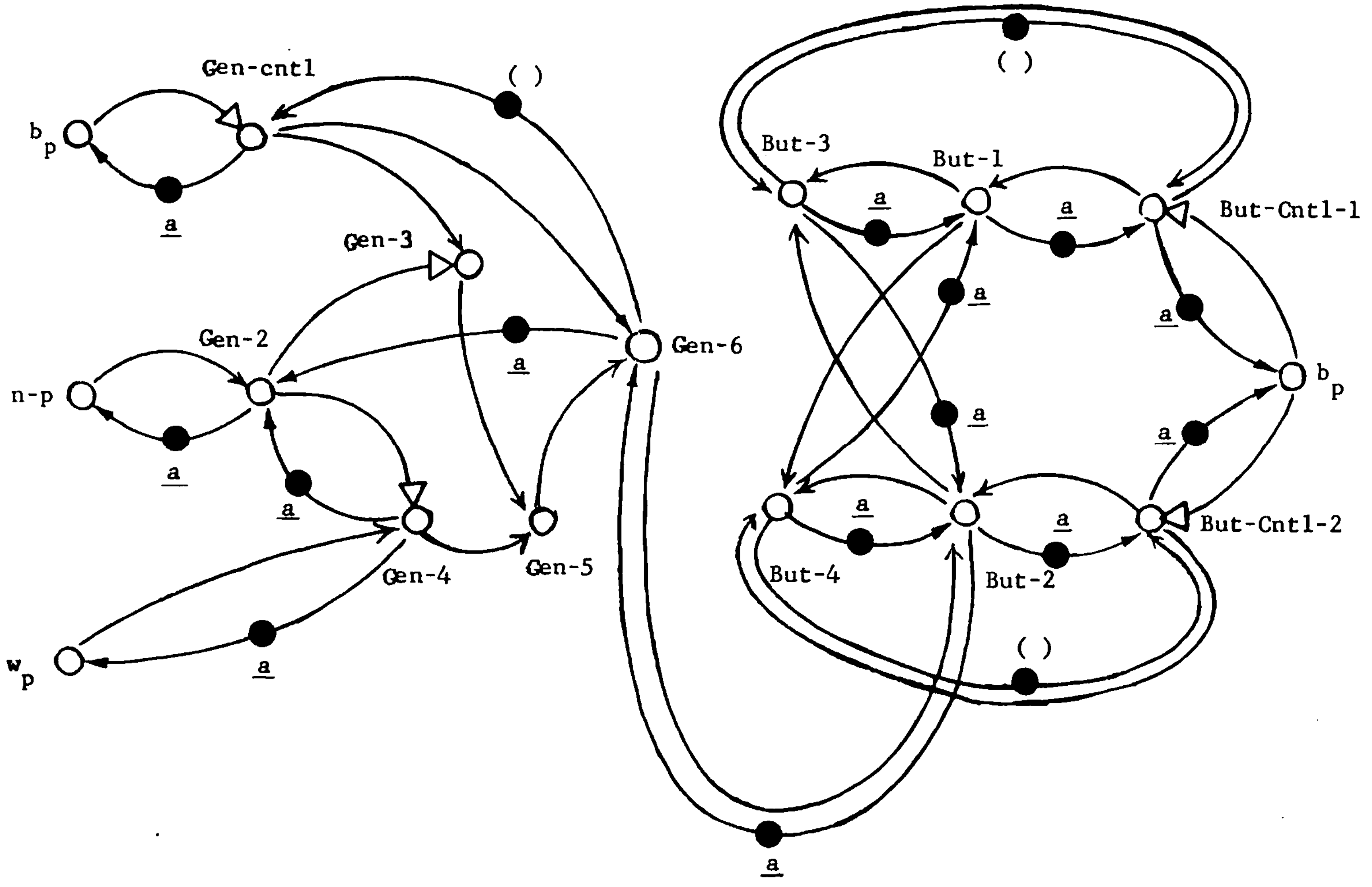


Figure 30. Marked graph description of the FFT computation.

are placed on arcs of the marked graph to represent a live and safe initial configuration of the data flow program. Signal values are denoted by *a* and data tokens carry unknown values indicated by ( ).

We regard each arc of the marked graph as having an associated "propagation delay" which is the time interval from the moment a token is placed on the arc by its origin node to the moment presence of the token is observed by the destination node. For data arcs we take this time to be $T_A + T_D + T_P$, the time for an operation packet to pass through the Arbitration Network plus the time for the data packet resulting from instruction execution to pass through the Distribution Network plus the time T_P for processing an operation packet by a Processing Unit. Similarly, the propagation delay for boolean and signal arcs is $T_A + T_C + T_P$.

Now our question of computation rate for the FFT program becomes a question about the minimum period for the cyclic behavior of a marked graph when each arc has a known propagation time. This problem was solved by Karp and Miller [21]: The minimum period is determined by the directed cycle in the marked graph having the largest value of total delay divided by the number of tokens on the cycle. Assuming

$$T_A = 13, \quad T_D = 13, \quad T_C = 3, \quad T_P = 4$$

we find there is one critical cycle in the program -- one involving cells Gen-6, Gen-cntl, Gen-3 and Gen-5. This cycle has one token and a total delay of $4 \times (T_A + T_p + T_D) = 120$ microseconds. Evidently, the sequencing constraints present in our FFT program are not a significant factor in determining the performance of the data flow processor. Indeed, its performance could be improved substantially by raising the number or performance of the Processing Units, and increasing the capacity of the routing networks.

VIII. Conclusion

The prospective performance of our hypothetical data flow processor is attractive in comparison with conventional stored program computers. However, whether such a machine is practical depends on the feasibility and cost of its construction. In the interest of obtaining a more practical design, it is attractive to divide the Instruction Cells into groups of eight or more cells and to implement each group together with associated portions of the Arbitration, Distribution and Control Networks by a combination of RAM chips and common control logic.

A limitation of the present data flow processor is that one Instruction Cell is required for each instruction, imposing a practical limit on the size of programs that may be run. This limitation may be overcome by including an auxiliary memory system that has space for all instructions of the data flow program and using a smaller number of Instruction Cells arranged to hold the most active instructions during program execution. The Instruction Cells then form a "cache" whose contents changes as activity shifts from one section of the program to another. An outline of the mechanisms required for this extension of the data flow architecture has been given in [17], and some ideas on the structure of memory systems for a data flow computer are given in [1, 2, 15]. These papers are in harmony with the principles of packet communication architecture.

We are also extending the generality of data flow architectural concepts by developing mechanisms to support procedure definition and invocation, and data structures. Data flow program graphs as described by Dennis [14], among others, encompass procedures and a general data structure capability, so the major problems concern development of satisfactory architectural schemes for implementing these capabilities. Issues in the design of multilevel memory systems for data flow computers have been studied by Ackerman [1, 2]; Rumbaugh's machine [30] implements procedures within a more conventional framework. Recently an extension of the processor design of the present paper to handle procedures and data structures with a high level of generality has been presented in the work of Weng [18, 37]. Other approaches are being developed by Arvind, Gostelow and Plouffe [7], and by Keller, Patil and Lindstrom [23].

REFERENCES

[1] Ackerman, W. B. A Structure Memory for Data Flow Computers. Technical Report LCS/TR-186, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., August 1977.

[2] Ackerman, W. B. A structure processing facility for data flow computers. Proceedings of the 1978 International Conference on Parallel Processing, IEEE, August 1978.

[3] Ackerman, W.B., and J. B. Dennis. VAL -- A value-oriented algorithmic language: preliminary reference manual, Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass, June 1979.

[4] Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, December 1968.

[5] Amdahl, G. M., T. C. Chen, and C. J. Conti. IBM system/360, model 92. Proceedings of the AFIPS 1964 Fall Joint Computer Conference, Part 2, 1964.

[6] Arvind, and K.P. Gostelow. A computer capable of exchanging processors for time.Information Processing '77, North Holland, New York 1977, 849-854.

[7] Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Technical Report 114A, Dept. of Information and Computer Science, University of California, Irvine, December 1978.

[8] Bahrs, A. Operation patterns. Lecture Notes in Computer Science, 5, Springer Verlag, New York, 1974, 217-246.

[9] Barnes, G. H., R. M. Brown, M. Kata, D. J. Kuck, D. L. Slotnick and R. A. Stokes. The Illiac IV computer. IEEE Trans. on Computers C-17, 8 (August 1968), 746-757.

[10] Batcher, K. E. STARAN parallel processor system hardware. Proceedings of the AFIPS Conference, 43, 1974, 405-410.

[11] Bruno, J., and S. M. Altman. A theory of asynchronous control networks. IEEE Trans. on Computers, C-20 (June 1971), 629-638.

[12] Commoner, F., A. W. Holt, S. Even, and Pnueli. Marked directed graphs. J. of Computer and System Sciences, 5, 1971. 511-523.

[13] 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, 210-215.

[14] Dennis, J. B.. First version of a data flow procedure language. Lecture Notes in Computer Science, 19, Springer-Verlag, New York, 1974, 362-376.

[15] Dennis, J. B.. Packet communication architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, 1975, 224-229.

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

[17] Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, 1975, 126-132.

[18] Dennis, J. B., and K.-S Weng. An abstract implementation for concurrent computation with streams. Proceedings of the 1979 International Conference on Parallel Processing, IEEE, August 1979.

[19] Gold, B., and C. M. Rader. Digital Processing of Signals. McGraw-Hill, New York, 1969.

[20] Hack, M. Analysis of Production Schemata by Petri Nets. Technical Report MAC-TR-94, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., February 1972.

[21] Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. of Applied Mathematics, 14 (November 1966), 1390-1411.

[22] Karp, R. M., and R. E. Miller. Parallel program schemata. J. of Computer and System Sciences, 3, 2 (May 1969), 147-195.

[23] Keller, R. M., S. S. Patil, and G. Lindstrom. A loosely-coupled applicative multi-processing system. Proceedings of the National Computer Conference, 1979.

[24] Kosinski, P. R. A data flow language for operating systems programming. SIGPLAN Notices, 8 (September 1973), 89-94.

[25] Miller, R. E., and J. Cocke. Configurable Computers: a New Class of General Purpose Machines. Report RC 3897, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., June 1972.

[26] Misunas, D. P. Deadlock avoidance in a data-flow architecture. Proceedings of the Milwaukee Symposium on Automatic Computation and Control, IEEE, April 1975.

[27] Peterson, J. Petri nets. Submitted for publication.

[28] 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, 293-302.

[29] Rodriguez, J. E. A Graph Model for Parallel Computation. Technical Report MAC TR-64, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., September 1969.

[30] Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Technical Report TR-150, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., May 1975.

[31] Rumbaugh, J. E. A data flow multiprocessor. Proceedings of the 1975 Sagamore Computer Conference on Parallel Procéssing, IEEE, August 1975, 220-223.

[32] Seeber, R. R., and A. B. Lindquist. Associative logic for highly parallel systems. Proceedings of the AFIPS Conference, 24, 1963, 489-493.

[33] Shapiro, R. M., H. Saint and D. L. Presberg. Representation of Algorithms as Cyclic Partial Orderings. Report CA-7112-2711, 1, Applied Data Research, Wakfield, Mass., December 1971.

[34] Tesler, L. G., and H. J. Enea. A language design for concurrent processes. Proceedings of the AFIPS Conference, 32, 1968, 403-408.

[35] Watson, W. J. The Texas Instruments advanced scientific computer. Proceedings of the Sixth Annual IEEE Computer Society International Conference, 1972, 291-293.

[36] Weng, K-S. Stream-Oriented Computation in Recursive Data Flow Schemas. Technical Memoranda 68, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., October 1975.

[37] Weng, K.-S. An Abstract Implementation for a Generalized Data Flow Language. Technical Report, MIT Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., forthcoming.

[38] Wirth, N. The programming language Pascal. Acta Informatica, 1, 1971, 35-63.