LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Structuring the Fast Fourier Transform for Data Flow Computation

Computation Structures Group Memo 193
June 1980

Arnold Chien

S.B. thesis submitted in partial fulfilment of the requirements of the Dept of Electrical Engineering and Computer Science, M.I.T.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

STRUCTURING THE FAST FOURIER TRANSFORM FOR DATA FLOW COMPUTATION

by

Arnold Chien

Abstract: This paper explores ways of using VAL, a high-level data flow programming language. A program for the Fast Fourier Transform is presented and developed in stages according to different space/time goals. Data flow graphs are derived from versions of the program, and the low-level implications discussed.

Supervisor: Jack B. Dennis

Position: Professor of Computer Science and Engineering

CONTENTS

FIGURES

1. Introduction

Data flow computation is a recently revitalized concept geared toward achieving high concurrency. Unlike the traditional Von Neumann machine, the data flow computer has no sequential control constructs such as the program counter. Instead, execution is "data-driven"; that is, program instructions are executed whenever their operands are ready. Since it will often happen that many instructions at once will be ready for execution, concurrency may be easily achieved if multiple processors are present in the machine. The higher speed thus attained has become an attractive goal in an increasingly greater number of applications. This, coupled with the ever-present desire to improve performance/cost ratios, has spurred data flow research on a number of levels.

At MIT, the VAL language was developed in response to the need for high-level software amenable to a data flow architecture. As one might expect, VAL - a "Value-oriented Algorithmic Language" - is radically different from most other programming languages. It is completely devoid of side-effects; every VAL module contains only local computation which produces an explicit value. This quality of "functionality" means that modules may be combined without unforeseen results. A programmer is thus allowed to express algorithms with implicit concurrencies in a way which does not involve arbitrary sequencing constraints.

Consider the following VAL code:

```
function factorial(N:integer returns integer)
   for
      result:integer:=1;
      count:integer:=N;
   do
      if not(count=1)
         then
            iter
               result:=result*count;
               count:=count-1;
            enditer
         else
            result
      endif
   endfor
endfun
```

This function computes the factorial of N by manipulating the local variables "result" and "count". The outcome of the "if" test determines whether the iteration body is to be iterated with new values for the two variables, or whether it is to terminate, returning the value of "result" as the value of the function. Functionality is present here at all levels. The factorial of N is returned as the value of the simple expression "result", which is returned as the value of the "if" block, which is returned as the value of the "for" block, which is finally returned as the value of "factorial".

A reader with no previous knowledge of VAL should be sure that he understands this program, particularly the semantics of the iteration construct, before proceeding. Though much of the software to follow is self-explanatory, the custom of most programmers of thinking in terms of statements and subroutines is deep-rooted enough to warrant a little precaution. (A complete, though preliminary, description of VAL may be found in Ackerman and Dennis(1)).

Very little has been done in the way of developing general VAL programming techniques. We will here explore different ways of using VAL to express one very prominent algorithm, the Fast Fourier Transform, with the hope of gaining some measure of insight into this problem.

2. The Fast Fourier Transform

The discrete Fourier transform of $2^n=N$ sample data points $x_0,x_1,\ldots,x_{N-1}$ is given by the sequence $f_0,f_1,\ldots,f_{N-1}$, where

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

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

(We use j to denote the square root of negative one.) Straightforward calculation of these values involves N^2 product terms. The Fast Fourier Transform, or FFT, improves upon this. As it turns out, the transform on 2^n samples can be expressed in terms of two transforms, each on 2^{n-1} samples. Each of these sub-transforms may be expressed in terms of two transforms, and so forth. We continue in this fashion until we have expressed the original transform of 2^n points in terms of $n\cdot 2^{n-1}$ transformations of two points each. A fortuitous arrangement of the computation for the case n=3 is shown in figure 1. Each column of the figure is a "stage" of the computation; there are n of these. Each stage consists of 2^{n-1} "butterfly" units, which compute two-point transforms. This arrangement is known as the time-decimated, constant-geometry FFT.

Let $u_{p,k}$ be the kth component of the vector of values computed by the pth stage of the FFT. Then $B_{p,q}$, the qth butterfly of stage p, computes

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

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

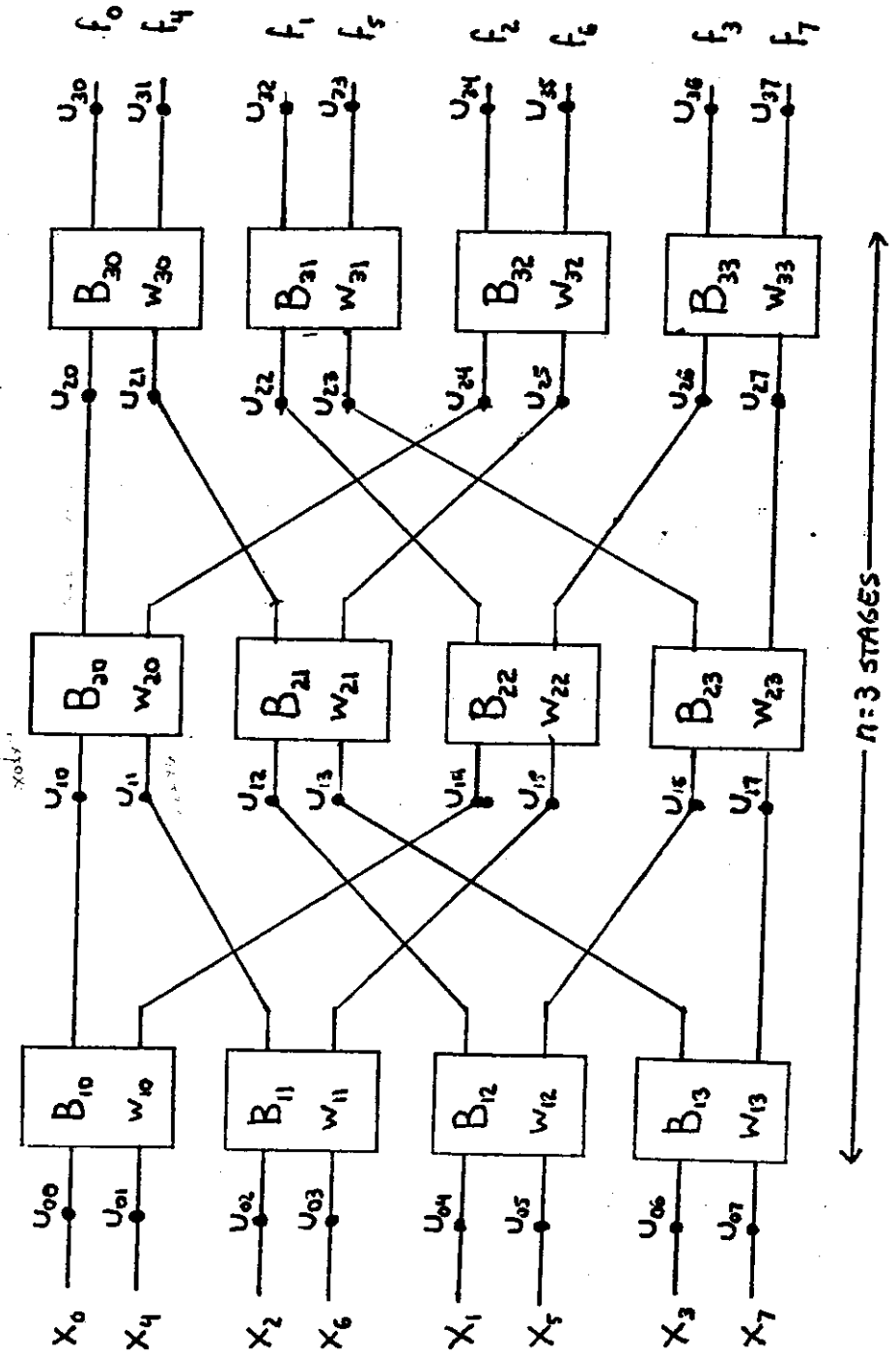


Figure 1. Time-Decimated, Constant-Geometry, Eight-Point FFT

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

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

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

$$0<p\leq n.$$

The function $\underline{quo}(m,n)$ yields the integer quotient of m divided by n. The input values for the first stage are related to the data samples by

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

and where $\underline{rev}(k,n)$ gives the number denoted by the reverse of the n-bit binary representation of k. The output values are

$$f_k=u_{n,k},$$

$$0\leq k<2^n.$$

The reader would be wise to take this all on faith.

3. The Algorithm-Clear Solution

Our first task is to express the FFT in a VAL program which is as straightforward as possible. We would like to be able to look at the program and discern with a minimum of difficulty the algorithm just described. We adopt a modular approach, partitioning the program into functionally coherent modules. The result is shown in figure 2. The program is presented as a collection of functions, with the understanding that they are actually imbedded within one another; in order to make the function calls legal, they must be made to conform with the conventional scope rules. This may be done quite easily, so easily that it's not worth doing.

The butterfly unit described in section 2 is implemented here as two functions, with butterfly1 corresponding to the unit's first output, and butterfly2 to the second. Thus, when these two functions are called with the same arguments, as they are at each pass through the program loop, a complete butterfly is being executed.

The main program module consists of an iteration loop inside an iteration loop. The outer loop corresponds to the stages of the computation, iterating n times, while the inner loop corresponds to the butterflies within each stage, iterating 2^{n-1} times per stage. The running time of the program is therefore roughly proportional to (no. of stages)X(butterflies/stage) = total no. of butterflies = $n \cdot 2^{n-1}$. This accords with what

was said in the previous section.

Some mention should be made about the use of arrays. The values $u_{p,k}$ computed at each stage are treated as values in the array "data". The function array join(A,B) merges the elements of array A with those of array B, retaining in the resulting array the original indices of all elements. The operation "[x:a;y:b]" creates an array with value a at index x and value b at index y ("record[...]" does a similar thing with records). The permutations required at each stage are thus easily handled by manipulating array indices.

We have made no attempt here at achieving concurrency or at otherwise being clever. This, of course, was due to our concern with clarity, not to any lack of cleverness.

Figure 2. Algorithm-Clear FFT Program

```
function FFT (sample_points:array[real], n:integer returns array[complex])
   %define new data type
   type complex = record[re,im:real];
   for
      p:integer:=1;
      data:array complex :=convert_sample(sample_points);
      N:integer:=exp(2,n);
      half:integer:=quo(N,2);
   do if p<n+1
      %last stage?
         then
         %iter, enter new stage
            %if not, enter new stage
               data:=
                  for
                  %This "for" body produces a new array of values from
                  %the previous one, two values at a time.  Each time the
                  %inner body is executed, two consecutive values from
                  %the old array and used to compute two result values;
                  %they are placed in the new array, separated in index
                  %value by "half", or 2^(n-1), thus accomplishing the de-
                  %sired permutation.
                     newdata:array[complex]:=array_empty[complex];
                     q:integer:=0;
                  do
                     if q<half
                     %last butterfly?
                        then
                           iter
                           %if not, enter new butterfly
                              newdata:=
                                 array_join
                                 ([q:butterfly1
                                     (data[2*q],data[2*q+1],phase_factor(p,q)),
                                  q+half:butterfly2
                                    (data[2*q],data[2*q+1],phase_factor(p,q))],
                                  newdata);
                              q:=q+1;
                           enditer
                        else
                           %if so, return final stage results
                           newdata
                     endif
                  endfor
               p:=p+1;
            enditer
         else
            %if so, computation complete
            data
      endif
   endfor
endfun
```

```
function phase_factor(p,q:integer returns complex)
   %calculates the phase factor of the qth butterfly of the pth stage
   let
      two_power:integer:=quo(N,exp(2,p));
      %calculate phase factor exponent
      e_pq:integer:=two_power*quo(q,two_power);
   in
      compexp(e_pq)
   endlet
endfun

function compexp(exponent:integer returns complex)
   %use Euler's identity to compute W^exponent, where W=e-j(2π/N)
   let
      pi:real:=3.14159;
      theta:real:=exponent*(2*pi/N);
   in
      record[re:cos(theta);im:-sin(theta)]
   endlet
endfun

function butterfly1(data1,data2,phasefactor:complex returns complex)
   %compute first butterfly output
   let
      phaseterm:complex:=compmult(data2,phasefactor);
   in
      compadd(data1,phaseterm)
   endlet
endfun

function butterfly2(data1,data2,phasefactor:complex returns complex)
   %compute second butterfly output
   let
      phaseterm:complex:=compmult(data2,phasefactor);
   in
      compsub(data1,phaseterm)
   endlet
endfun

function compmult(op1,op2:complex returns complex)
   record[re:op1.re*op2.re - op1.im*op2.im;
          im:op1.im*op2.re + op1.re*op2.im]
   %(a+bi)(c+di)=(ac-bd)+(ad+cb)i

function compadd(op1,op2:complex returns complex)
   record[re:op1.re + op2.re;im:op1.im + op2.im]
endfun

function compsub(op1,op2:complex returns complex)
   record[re:op1.re - op2.re;im:op1.im - op2.im]
endfun
```

Figure 2. Continued

```
function convert_sample(inarray:array[real]returns array[complex])
   %The sample data points enter the program as an array of real values.
   %Since these values must be used in complex operations, they must
   %be converted to type complex.  This function also does the ini-
   %tial permutation on the input points.
   for
      i:integer:=0;
      outarray:array[complex]:=array_empty[complex];
   do
      if i<N  %end of input array?
         then
            iter  %if not, process input value
               outarray:=array_join
                         ([rev(i,n):record[re:inarray[i];im:0]],
                          outarray);
               i:=i+1;
            enditer
         else
            outarray
      endif
   endfor
endfun

function rev(i,n:integer returns integer)
   %By calling the function"binary_rep", this function gains access
   %to an array of bits which is the binary representation of i.  Each
   %bit is multiplied by the appropriate power of two, and the sum
   %of these products is accumulated in "result".
   for
      result:integer:=0;
      j:integer:=0;
      binary_result: array[integer]:=binary_rep(i,n);
   do
      if j<n
        then
           iter
              result:=result+(exp(2,j)*binary_result[n-1-j]);
              j:=j+1;
           enditer
        else
           result
      endif
  endfor
endfun
```

Figure 2. Continued

```
function binary_rep(i,n:integer returns array[integer])
   %This function constructs the n-bit binary representation of i.  The
   %result array is built in descending order, i.e. from high index to
   %low index, where the index corresponds to the power of two associ-
   %ated with the indexed bit.  A running total is kept of the "switched-
   %on" powers of two, so that, at each pass through the loop, a '1' is
   %added to the array if the current power of two added to the running
   %total does not exceed i.  Otherwise a '0' is added.
   for
      total:integer:=0;
      k:integer:=n-1;
      result:array[integer]:=array_empty[integer];
   do
      if k<=0
         then
            iter
               result:=array_join
                       ([k:if total+exp(2,k)<=i then 1 else 0],
                        result)
               total:=total+(result[k]*exp(2,k));
               k:=k-1;
            enditer
         else
            result
      endif
   endfor
endfun
```

Figure 2. Continued

4. The Serial Solution

4.1. Introduction

We have said that concurrency is the natural goal of any program written for a data flow computer. It is really a question of degree. High concurrency is inseparable from redundant hardware; that is, gains in speed are possible only if we are willing to use up more space and spend more money on the machine logic level. Exactly where we want to be on the continuous spectrum between good performance and good economics depends, of course, on the application. However, mere cognizance of the tradeoff makes the idea of a "serial" solution interesting. If we assume that software iteration modules translate into single hardware modules processing data serially, then the program of figure 2 is already optimal in many ways in terms of saving space and money. We will imagine now that we are cheapskates and see how we can improve upon these savings. That data flow concepts are not necessarily antithetical to such goals will perhaps become clearer in later sections.

4.2. Phase Factor Computation

There are glaring inefficiencies in the phase factor computations of the algorithm-clear program. First of all, even though butterfly1 and butterfly2 use the same phase factor in any given loop pass, phase factor is called in the invocation for each, thus doubling the amount of necessary computa-

tion. This suggests implementing the phase factors for a given stage as an array instead of as function calls, since passing duplicate array values (as butterfly arguments) involves no redundant computation. The question, of course, is how to update the phase factor array at each stage. As it turns out, the phase factors of a new stage may be computed from those of the previous stage by means of the following VAL-like formula (we continue the notation of section 2):

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

where the function bit(r,s,n) returns the rth bit of the n-bit binary representation of s. The initial values of the phase factors are given by

$$w_{0,q}=W^{e_{0,q}}, \text{ where}$$

$$e_{0,q}=2^{n-1}\text{quo}(q,2^{n-1})= 0 \text{ (since } q<2^{n-1}).$$

The derivations may be found in Dennis et. al. (3).

The use of these formulae involves a savings in complex multiplys since a phase factor may not need to be updated for a given stage. It also involves a savings in complex exponentiations, which are particularly costly because of their use of the sine and cosine functions.

The update factors, one per stage, are implemented in the array "queue" of the program in figure 3.

4.3. The "Bit" Function

Using the binary rep function defined in figure 2, we may implement the bit function quite easily, as shown in figure 4a. However, the program in figure 4b, which combines the two

functions, is an improvement. Instead of building the complete binary representation and then picking out the appropriate bit, we may save time by looking for the bit as we are building the binary representation.

Except for the addition of bit and the deletion of phase-factor, the program of figure 3 is supplemented by the same functions as the previous program.

```
function FFT(sample_points:array[real],n:integer returns array[complex])
   type complex = record[re,im:real];
   for
      p:integer:=1;
      data:array[complex]:=convert_sample(sample_points);
      N:integer:=exp(2,n);
      half:integer:=quo(N,2);
      queue:array[complex]:=
      %This loop constructs the array of update factors W^(2^(n-p)).  The
      %index of each array element is equal to the value of p with
      %which it is associated, so that, in any given stage p, the ap-
      %propriate update factor is simply "queue[p]".
         for
            index:integer:=1;
            assign_queue:array[complex]:=array_empty[cmplex];
         do
            if index<n+1
               then
                  iter
                     assign_queue[index]:=compexp(exp(2,n-index));
                     index:=index+1;
                  enditer
               else
                  assign_queue
            endif
         endfor
      %initialize all 2^(n-1) phase factors to the value W^0=1=1+0j
      phasefactor:array[complex]:=array_fill(0,half-1,record[re:1;im:0]);
   do
      if p<n+1
         then
            iter
               phasefactor:=
               %update the phase factors using the "bit algorithm"
                  for
                     q:integer:=0;
                     newphasefactor:array[complex]:=array_empty[complex];
                  do
                     if q<half
                        then
                           iter
                              newphasefactor[q]:=
                                       if bit(n-p,q,n)
                                          then
                                             compmult(phasefactor[q],queue[p]
                                          else
                                             phasefactor[q]
                                       endif
                              q:=q+1;
                           enditer
```

Figure 3. Serial FFT Program

```
                        else
                           newphasefactor
                     endif
                  endfor
               data:=
                  for
                     newdata:array[complex]:=array_empty[complex];
                     q:integer:=0;
                  do
                     if q<half
                        then
                           iter
                              newdata:=
                                 array_join
                                 %Note the change in the phase factor
                                 %argument of the butterfly calls.
                                 ([q:butterfly1
                                     (data[2*q],data[2*q+1],phasefactor[q]);
                                  q+half:butterfly2
                                     (data[2*q],data[2*q+1],phasefactor[q])],
                                  newdata);
                              q:=q+1;
                           enditer
                        else
                           newdata
                     endif
                  endfor
               p:=p+1;
            enditer
         else
            data
      endif
   endfor
endfun
```

Figure 3. Continued

```
function bit(r,s,n:integer returns boolean)
   %Since the array returned by binary rep is bit-ordered, all we
   %have to do to get the rth bit of the n-bit binary representa-
   %tion of s is index.
   let
      result:array[integer]:=binary_rep(s,n);
   in
      if result[r]=1
         then true
         else false
      endif
   endlet
endfun
```

Figure 4a. Simple Implementation of "Bit"

```
function bit(r,s,n:integer returns boolean)
   %This function is similar to the binary rep function of figure
   %2.  The big difference is that there is no concern to build
   %an array of bits.  If the current bit position does not match
   %the desired index, the running total is updated, but no bit
   %is saved.  If there is a match, the bit is used to generate
   %a return value.
   for
      total:integer:=0;
      k:integer:=n-1;
   do
      if k=r
         then
            if total+exp(2,k)<=s
               then true
               else false
            endif
         else
            iter
               total:=total+(if total+exp(2,k)<=s
                                then exp(2,k)
                                else 0
                             endif);
               k:=k-1;
            enditer
      endif
   endfor
endfun
```

Figure 4b. Improved Version of "Bit"

5. The Parallel Solution

5.1. The "Forall"

In VAL, the construct which is most exemplary in expressing concurrency is the forall. Forall blocks are of two kinds. In the construct block, an array is generated whose elements are the values of the expression within the block evaluated at index values specified in the block header. The following generates an array with n elements.

```
forall p in[1,n]
construct
   compexp(exp(2,n-p))
endall
```

The first element is the value compexp(exp(2,n-1)), the second the value compexp(exp(2,n-2)), and so on until the last value compexp(exp(2,n-n)). The eval block generates values just as a construct does, but instead of combining them in an array, they are combined into a single value by some operation, e.g. addition.

The forall represents concurrency in that it imposes no sequencing on its generation of values. It is suitable for the expression of any computation of values which come independently from the same expression. Iteration constructs which sequentially execute the same body of code while varying only the value of some parameter are thus good prospects for replacement. Our strategy for developing a "parallel" FFT program, i.e. one in which we express as much concurrency as possible, is to hunt for iterations in the serial program and kill them

if we can. Clearly, we must retain the outer stage loop since each stage uses results computed by the previous one. No such constraint exists for the inner loop; we may have all our butterflies going at once without affecting the correctness of the computation. Thus we use foralls for both the phasefactor update and the butterfly calculations, as well as for the initialization of "queue". Figure 6 shows two supplementary functions which we also modify in this fashion. Why the bit function of figure 4b is immune to our advances is left as a rather simple exercise for the reader.

5.2 Final Solution for a Non-Optimizing Compiler

The butterfly functions both compute a "phaseterm" by multiplying together the phasefactor and the second datum. This is wasteful since the two functions are called with the same arguments and hence produce the same phaseterm. An inefficiency of this sort might easily be corrected on the machine level by an optimizing compiler. But it is instructive to correct it on the software level by moving the phaseterm computation out and simply passing the result to the butterfly functions. As with the phase factors, we do this with an array updated at each stage. The "phaseterm" array is computed from the "phasefactor" array by multiplying each element of the latter by the corresponding second datum, i.e. the odd elements of the "data" array computed by the previous stage. Note that removing the phaseterm computation leaves only a single expression in each butterfly function, so that there's not much reason for leaving them as functions, modularity already being damaged.

```
function FFT(sample_points:array[real],n:integer returns array[complex])
   type complex = record[re,im:real];
   for
      p:integer:=1;
      data:array[complex]:=convert_sample(sample_points);
      N:integer:=exp(2,n);
      half:integer:=quo(N,2);
      queue:array[complex]:=
      %The following block was discussed in 5.1.
         forall p in[1,n]
         construct
            compexp(exp(2,n-p))
         endall
      phasefactor:array[complex]:=array_fill(0,half-1,record[re:1;im:0]);
   do
      if p<n+1
         then
            iter
               phasefactor:=
               %update phase factors concurrently
                  forall q in [0,half-1]
                  construct
                     if bit(n-p,q,n)
                        then compmult(phasefactor[q],queue[p])
                        else phasefactor[q]
                     endif
               endall
               data:=
                  %compute stage results concurrently
                  forall k in [0,N-1]
                  construct
                  %The first half of the result array is comprised of
                  %values computed by butterfly1; the second half, by
                  %butterfly2.  In the following, note that results
                  %separated in index value by "half" are computed by
                  %the same butterfly "unit".
                     if k<half
                        then butterfly1
                             (data[2*k],data[2*k+1],phasefactor[k])
                        else butterfly2
                             (data[2*(k-half)],data[2*(k-half)+1],
                                                phasefactor[k-half])
                     endif
                  endall
               p:=p+1;
            enditer
         else
            data
      endif
   endfor
endfun
```

Figure 5. Parallel FFT Program

```
function convert_sample(inarray:array[real]returns array[complex])
   forall i in [0,N-1]
   construct
      record[re:inarray[rev(i,n)];im:0]
   endall
endfun

function rev(i,n:integer returns integer)
   forall j in [0,n-1]
      result:array[integer]:=binary_rep(i,n);
   eval plus exp(2,j)*result[n-1-j]
   endall
endfun
```

Figure 6. Modified Supplements to Parallel FFT Program

```
function FFT(sample_points:array[real],n:integer returns array[complex])
   type complex = record[re,im:real];
   for
      p:integer:=1;
      data:array[complex]:=convert_sample(sample_points);
      N:integer:=exp(2,n);
      half:integer:=quo(N,2);
      queue:array[complex]:=
         forall p in [1,n]
         construct
            compexp(exp(2,n-p))
         endall
      phasefactor:array[complex]:=array_fill(0,half-1,record[re:1;im:0]);
   do
      if p<n+1
         then
            iter
               phasefactor:=
                  forall q in [0,half-1]
                  construct
                     if bit(n-p,q,n)
                        then compmult(phasefactor[q],queue[p])
                        else phasefactor[q]
                     endif
                  endall
               data:=
                  forall k in [0,N-1]
                  %compute phaseterm array from updated phase factors
                     phaseterm:array[complex]:=
                        forall q in [0,half-1]
                        construct
                           compmult(phasefactor[q],data[2*q+1])
                        endall
                  construct
                     if k<half
                        then compadd(data[2*k],phaseterm[k])
                        else compsub(data[2*(k-half)],phaseterm[k-half])
                     endif
                  endall
               p:=p+1;
            enditer
         else
            data
      endif
   endfor
endfun
```

Figure 7. Final Parallel FFT Program

6. Data Flow Graph Representations

6.1. Introduction

In order to study possible lower-level implications of our VAL programs and gain a better understanding of how they might work, it would be useful to compile them. Dennis and others have developed a graphical data flow language, in which we imagine VAL compiler output to be expressed.

A data flow program graph is an interconnected set of nodes called "actors" and "links". Data values of any type are represented by "tokens" which travel around the graph. The way they travel is determined by "firing rules", as follows: 1) a node is enabled if a token is present on each of its input arcs and if its output arcs are free of tokens, and 2)an enabled node may fire by removing tokens from its input arcs and placing a token on each output arc. Some common nodes are shown in figure 8. A data flow graph corresponding to the factorial program is shown in figure 9. Note the initial configuration of "false" tokens. Note also the allowance for pipelined operation, given the firing rules.

Quite a bit needs to be done in the area of VAL compilers. Brock and Montz(2) have developed some guidelines for translating certain VAL segments into data flow graphs, but for the most part this is an open question. We will give data flow graphs corresponding to the VAL programs developed earlier, but as to how they might be produced, we have no clear idea. Array con-

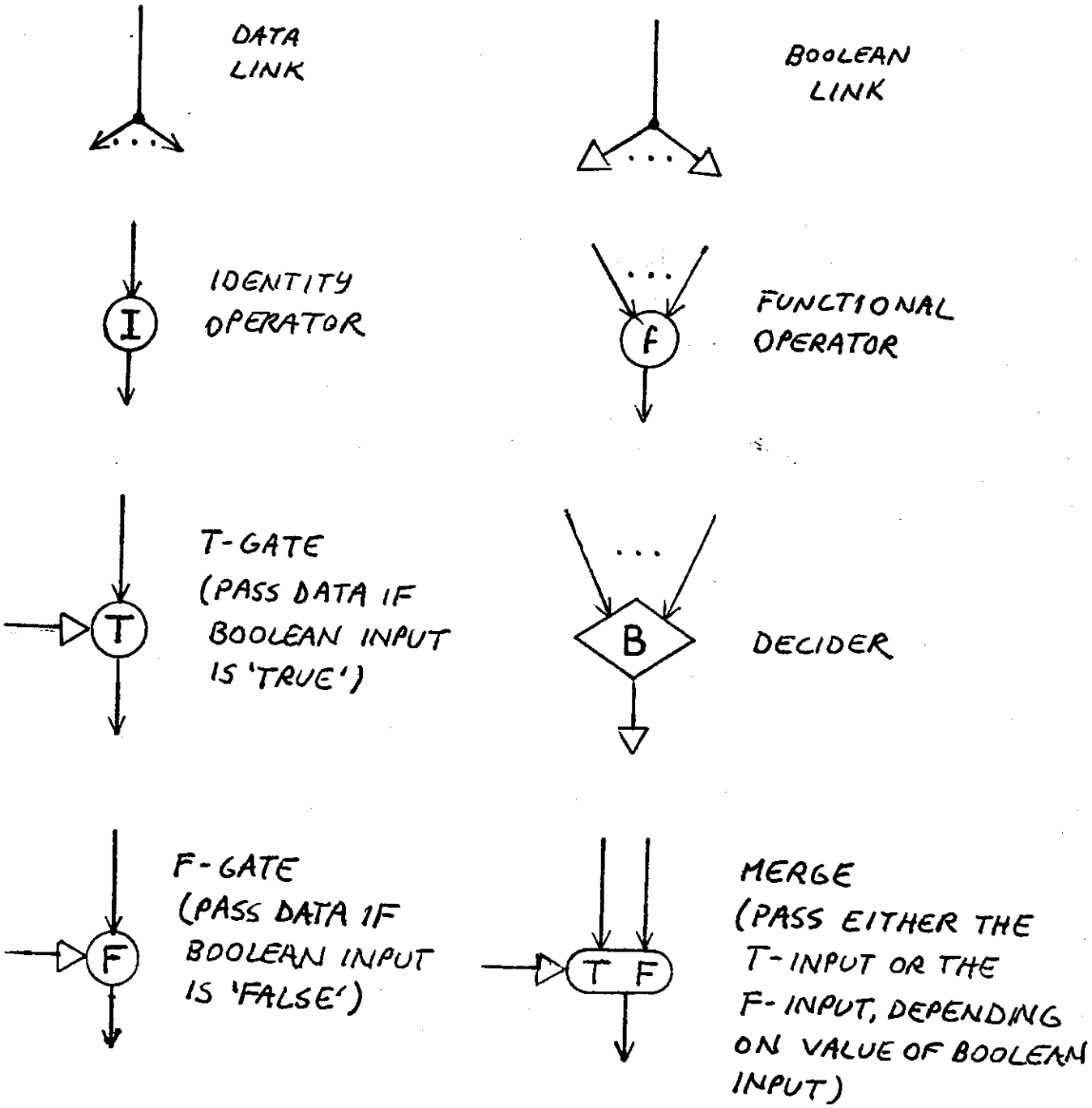


Figure 8. Data Flow Graph Nodes

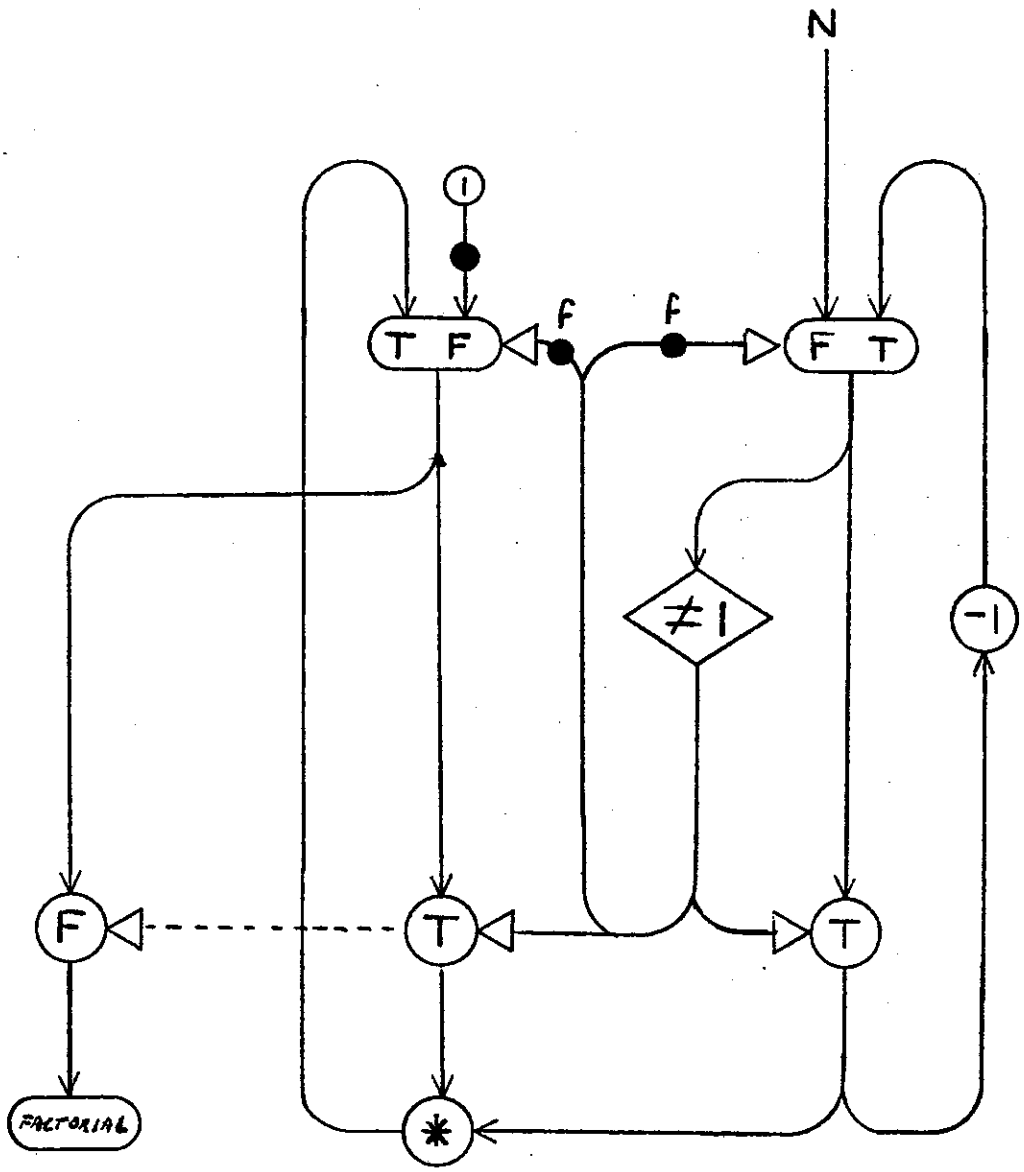


Figure 9. Data Flow Graph For "Factorial"

structs, for example, are not supported in any existing data flow machine designs; getting from VAL arrays to data flow graph constructs which accomplish the same tasks is not trivial. It does, however, make some sense to specify desirable target code before actually attempting to write a compiler. With this rationalization in hand, we will proceed.

6.2. The Serial Solution

Figures 10 through 15 describe a data flow graph for the program of figure 3.

Figure 10 shows the overall program graph, which includes some black boxes. The double lines are an abbreviation for N/2 data paths, while the triple lines denote a data path of width N. Elements of the "phasefactor" and "data" arrays, respectively, travel along these paths in parallel. Since both the phase factor update and butterfly units operate in series, parallel to series and series to parallel conversions are needed. These are provided by fan-in and fan-out trees of "alternators". The parallel interface to these trees requires a particular permutation to assure the correct sequence of values at the serial interface. All this is shown in figures 13 through 15.

For simplicity, we have left the less involved supporting functions as single operator nodes. The encapsulation of bit in figure 12, however, takes a lot of gall. We might be able to do this if we postulated the implementation of the bit function in the ALU, where some simple digital circuitry could do the job very neatly, but, if this were the case, it would have

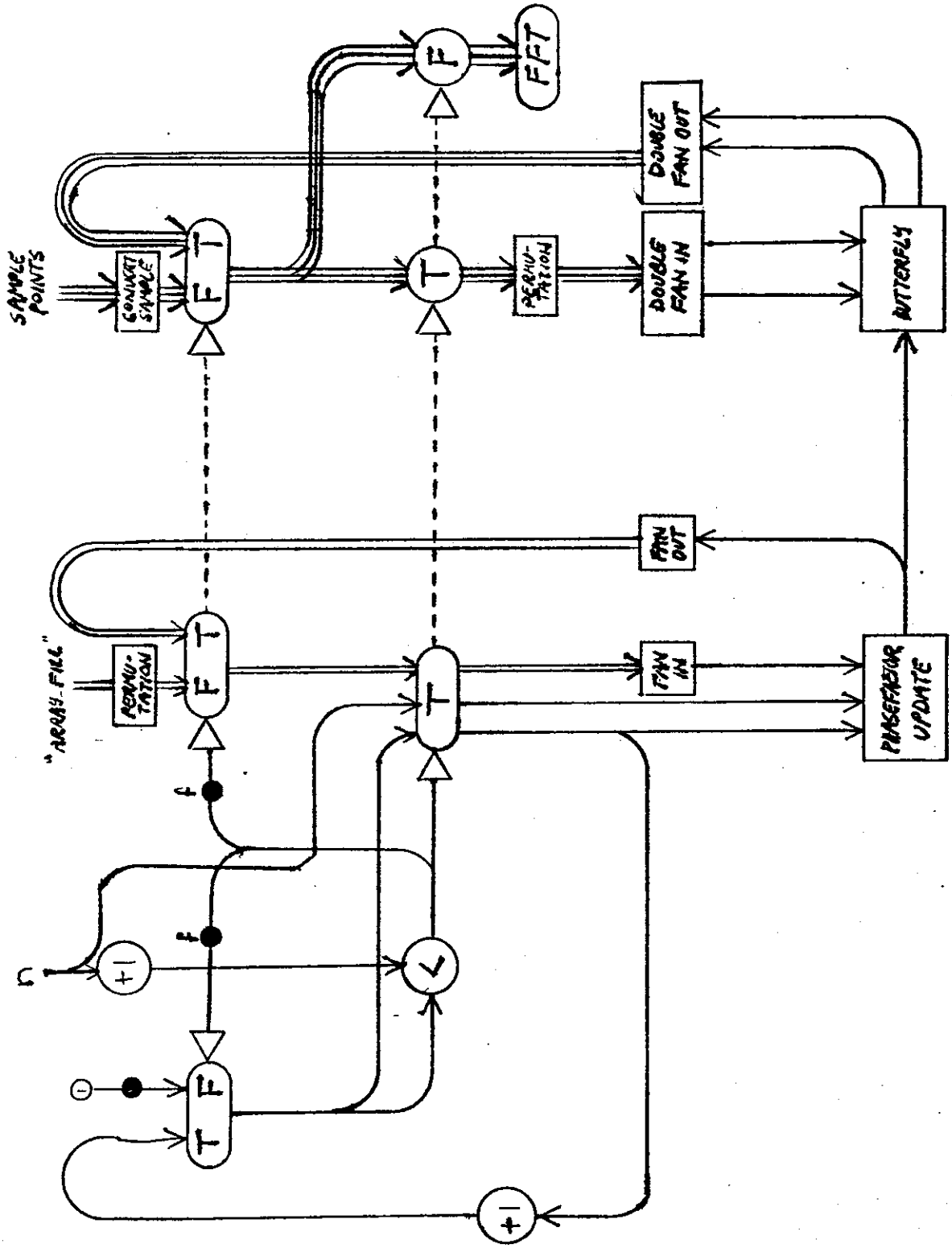


Figure 10. Data Flow Graph for the Serial FFT

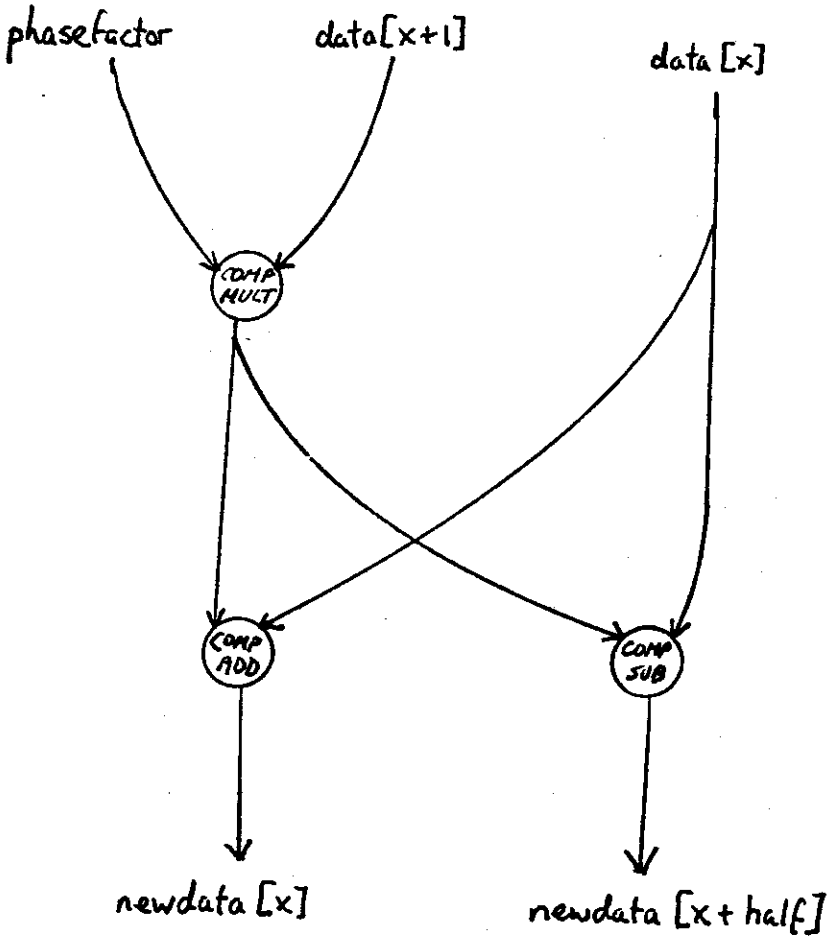


Figure 11. Serial Butterfly Unit

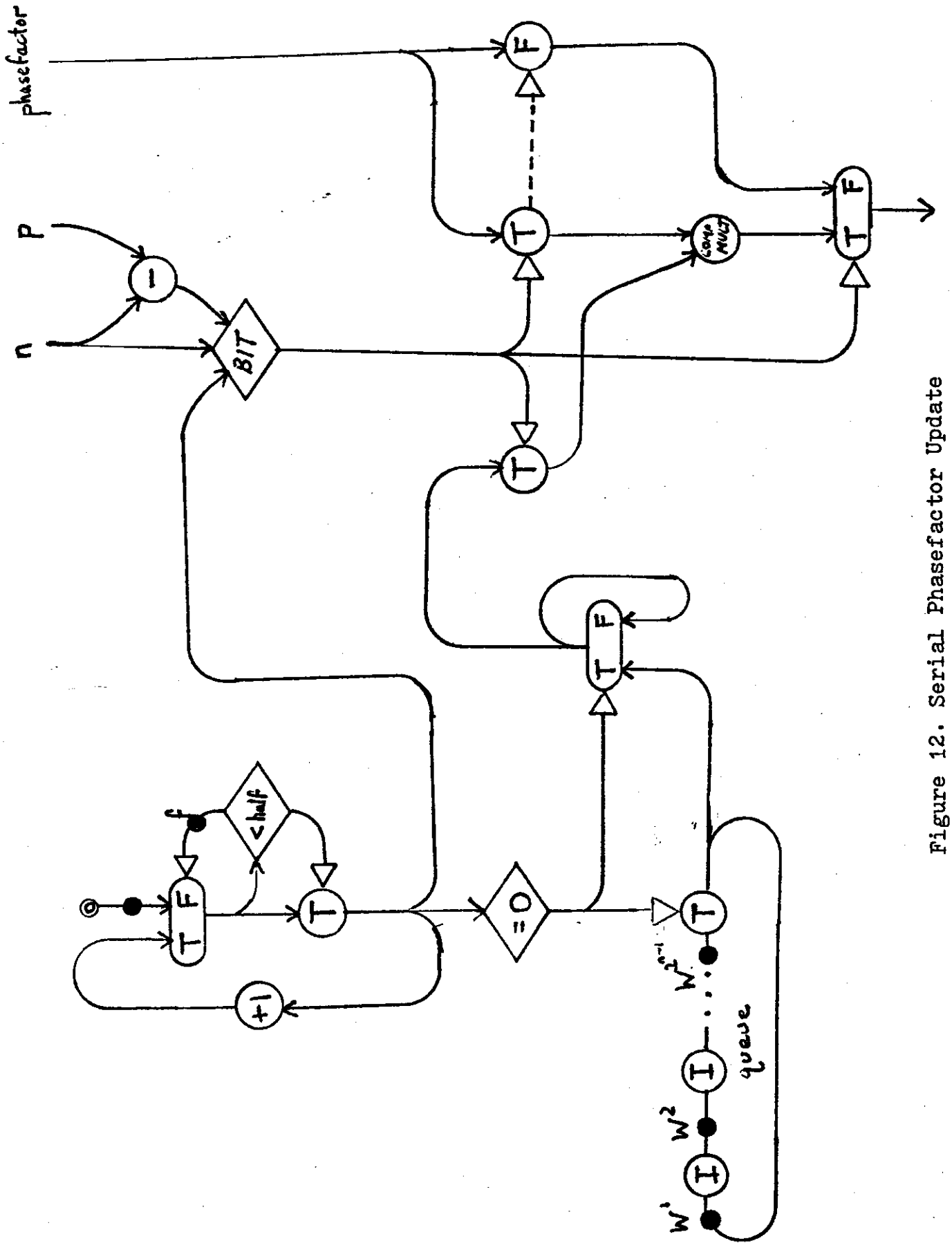
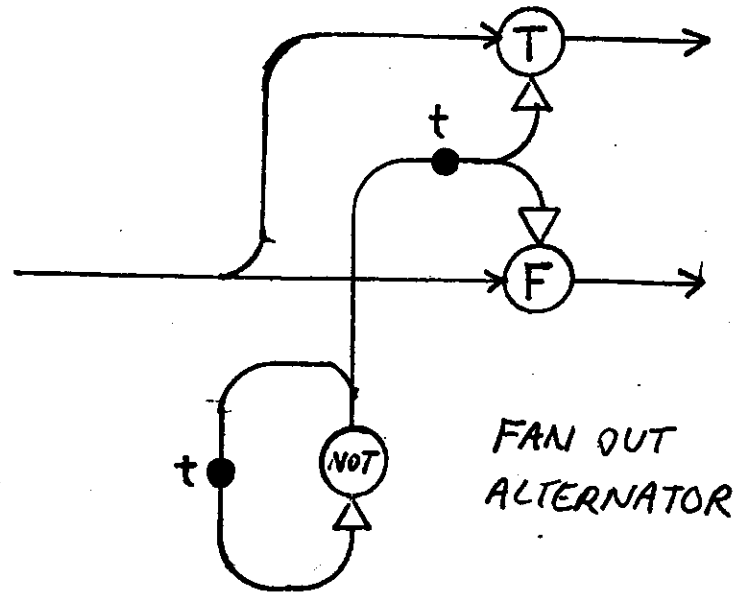


Figure 12. Serial Phasefactor Update



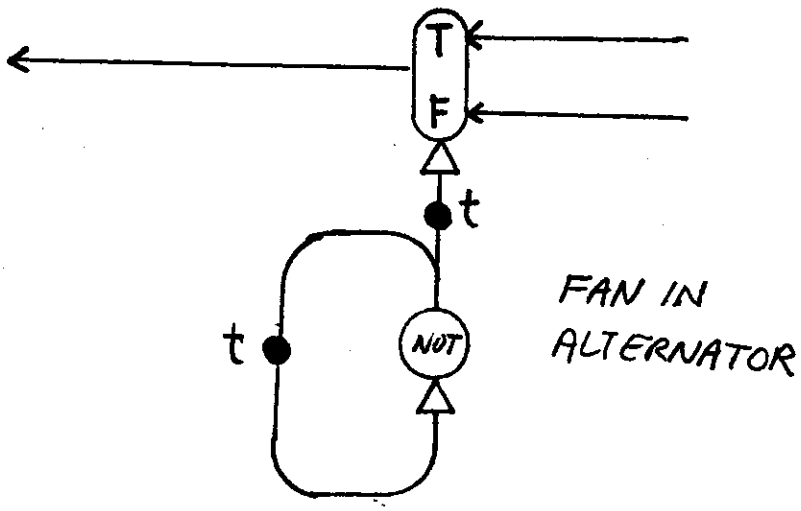


Figure 13. Alternators

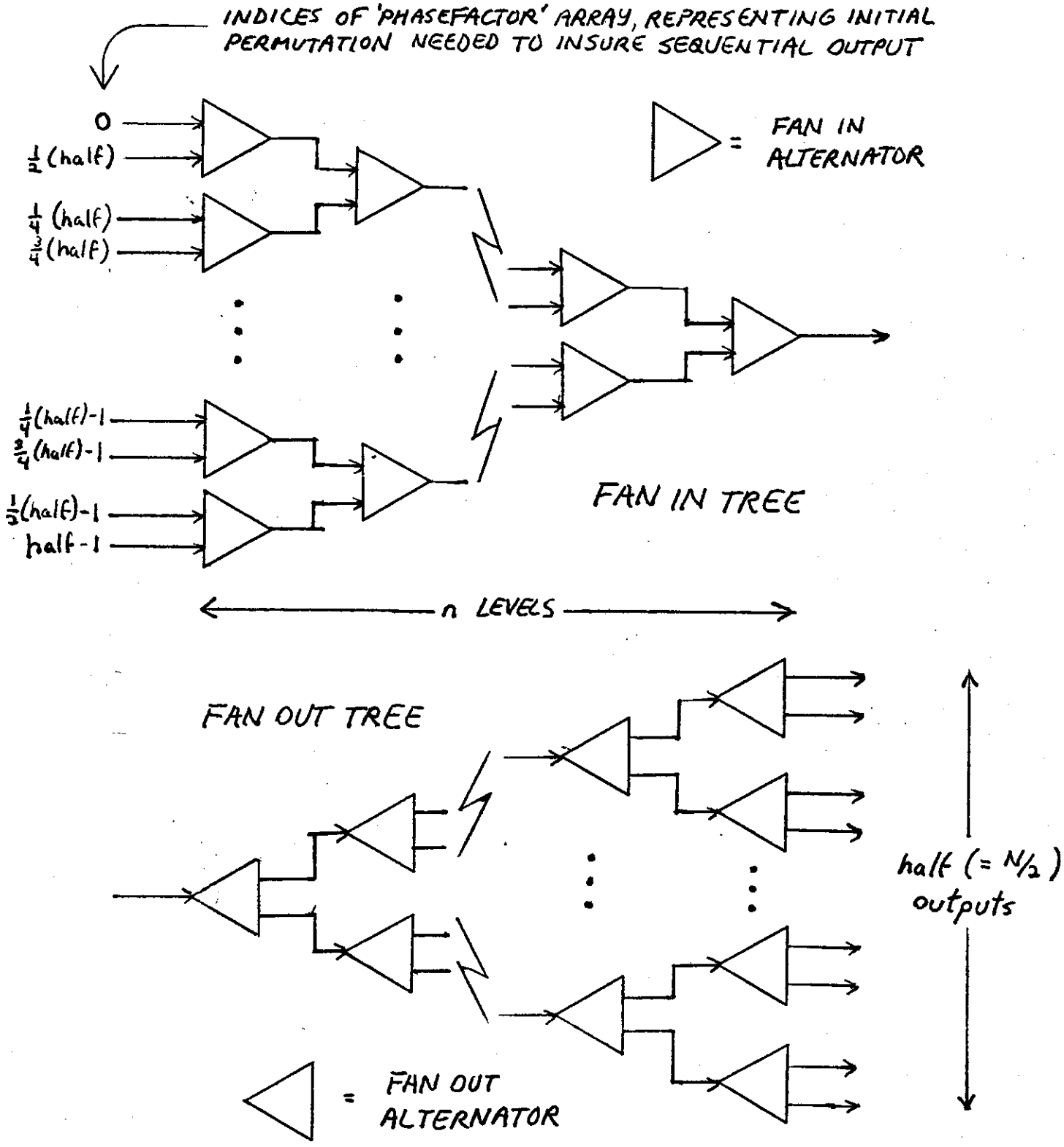


Figure 14. Alternator Trees

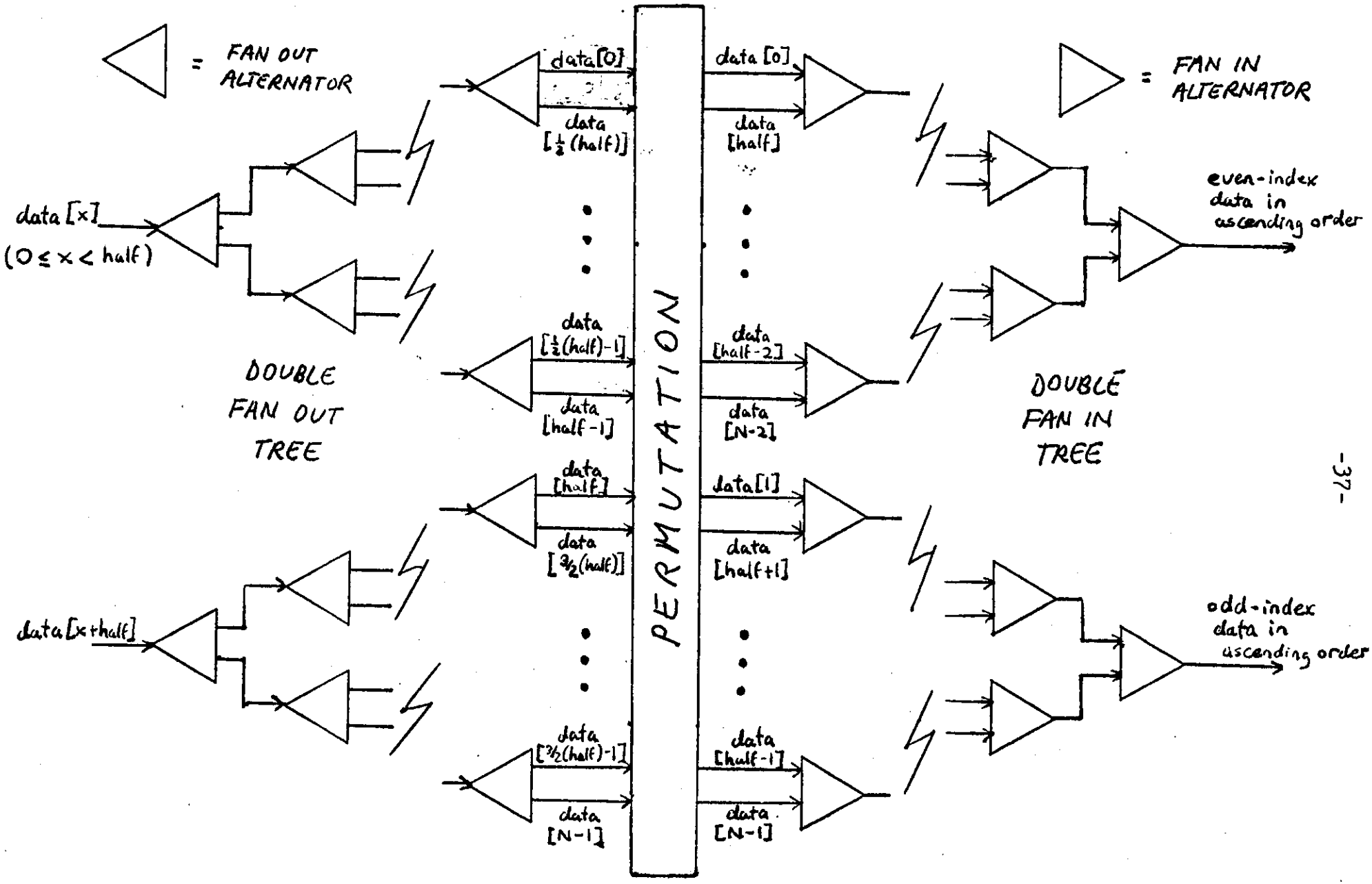


Figure 15. Double Alternator Trees

made more sense to treat bit as a built-in function on the VAL level, instead of as a user-defined function. So for the sake of consistency, figure 16 is provided to show the data flow graph expansion of bit.

6.3. The Parallel Solution

The essential difference between the data flow graph of figure 17, which corresponds to the parallel VAL program, and that of figure 11 is that the phasefactor update and butterfly units operate on all their data at once. This eliminates the need for alternator trees.

As a final note on compiler difficulties: it's hard to see how the subgraphs corresponding to forall blocks (e.g. figure 18), which consist of copies of a computational unit, could be generated if N, the size of the input sample, is not known at compile time. That is, the compiler must know how many copies to make. How it might be passed the necessary information is another issue which we will ignore.

6.4. Comparisons

The obvious advantage of the parallel solution is that of speed. If we assume that the data flow graph translates in a more or less straightforward way into bottom level machine instructions, so that the number of graph nodes corresponds roughly to the number of such instructions, then its obvious disadvantage is that of space and cost. Naturally, the serial solution is slower but more economical. (As a kind of first ver-

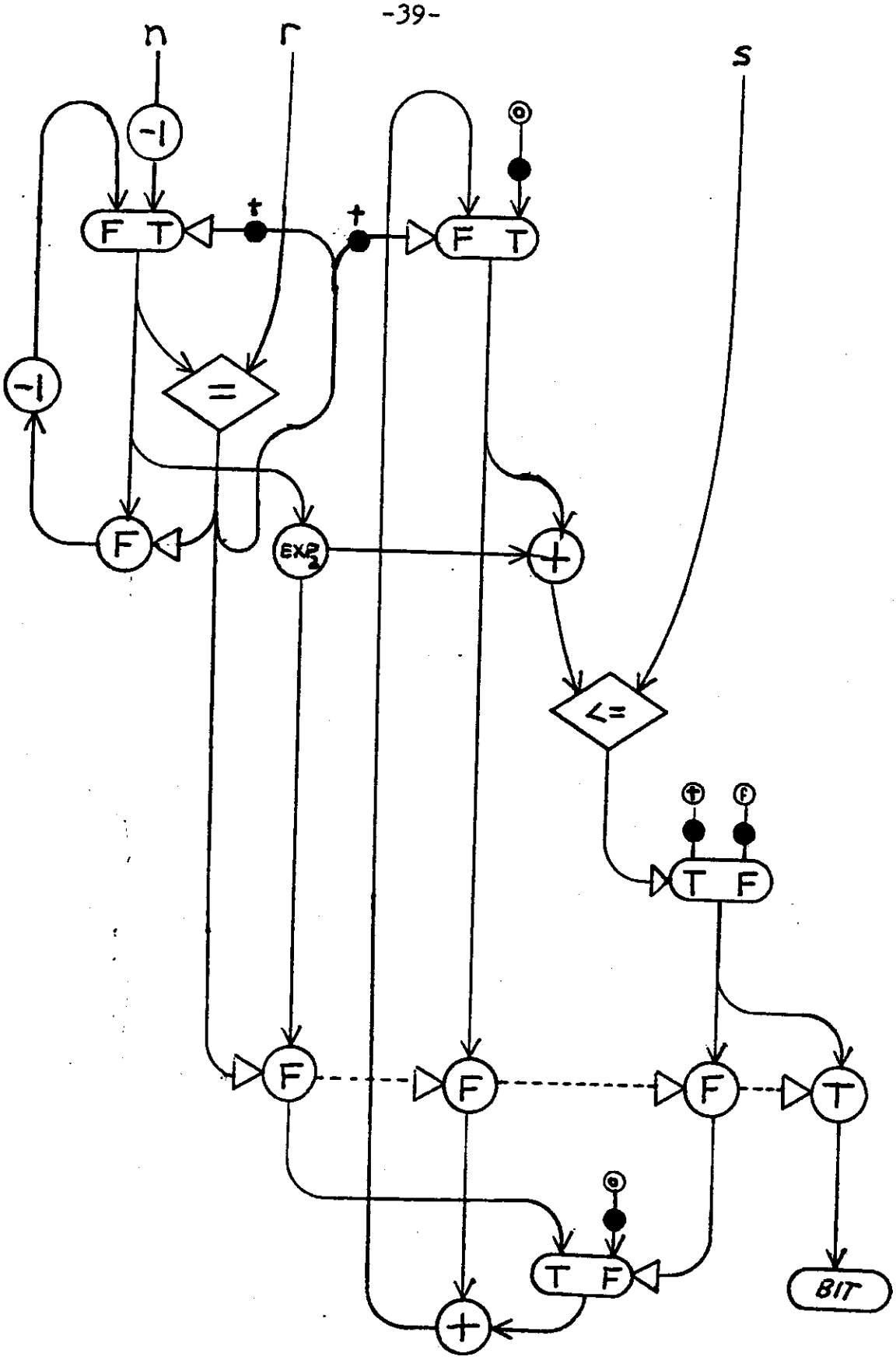


Figure 16. Expansion of "Bit"

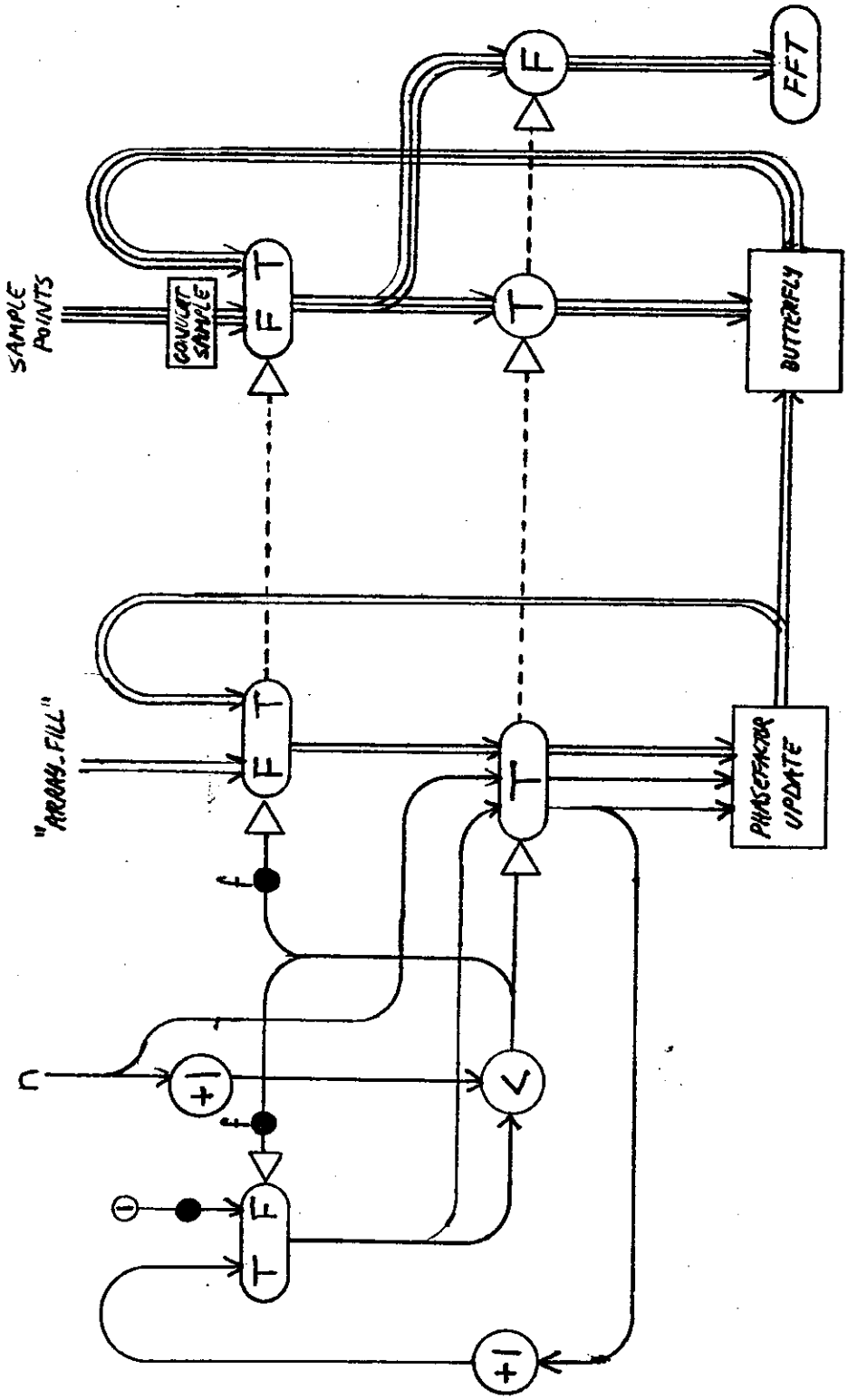


Figure 17. Data Flow Graph for the Parallel FFT

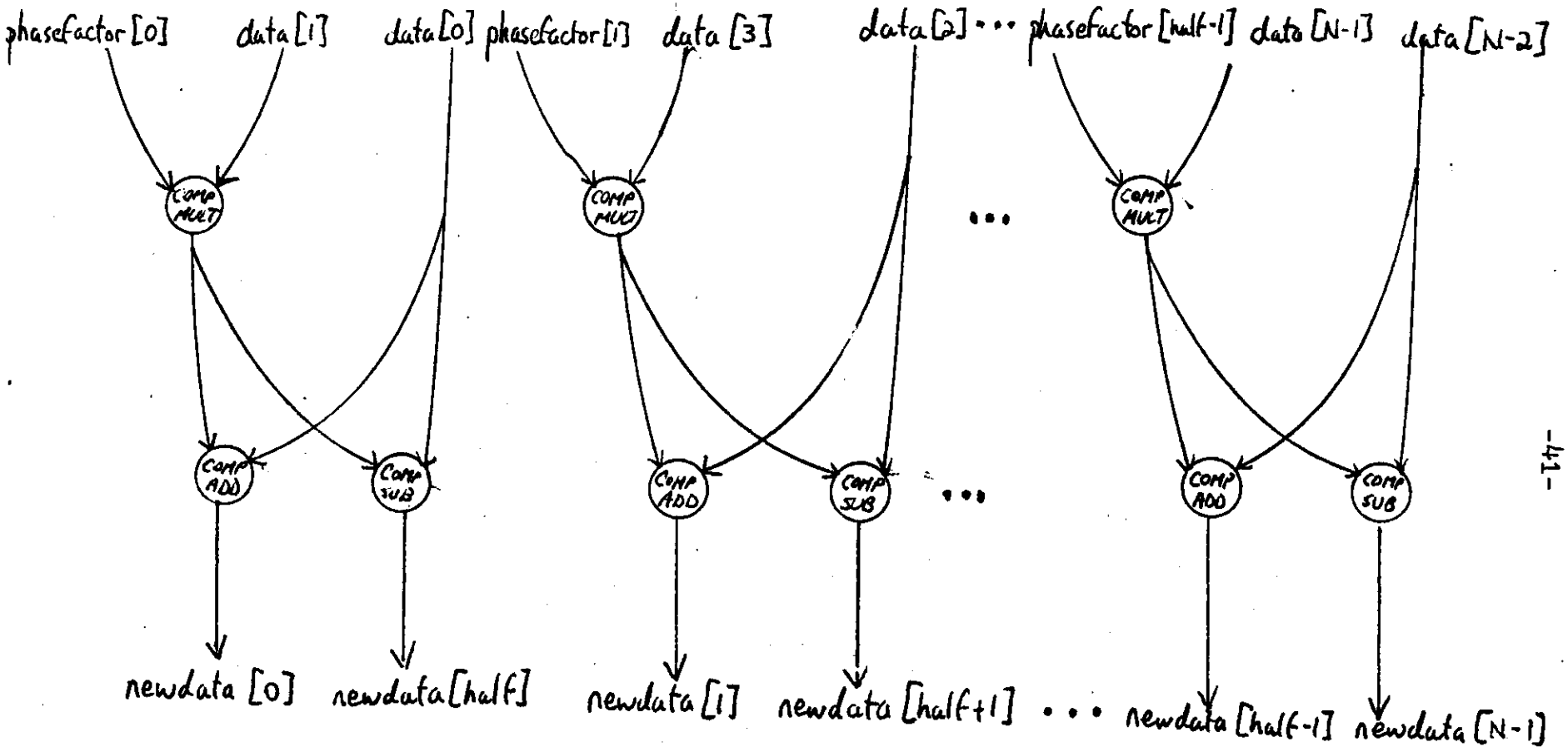


Figure 18. Parallel Butterfly Unit

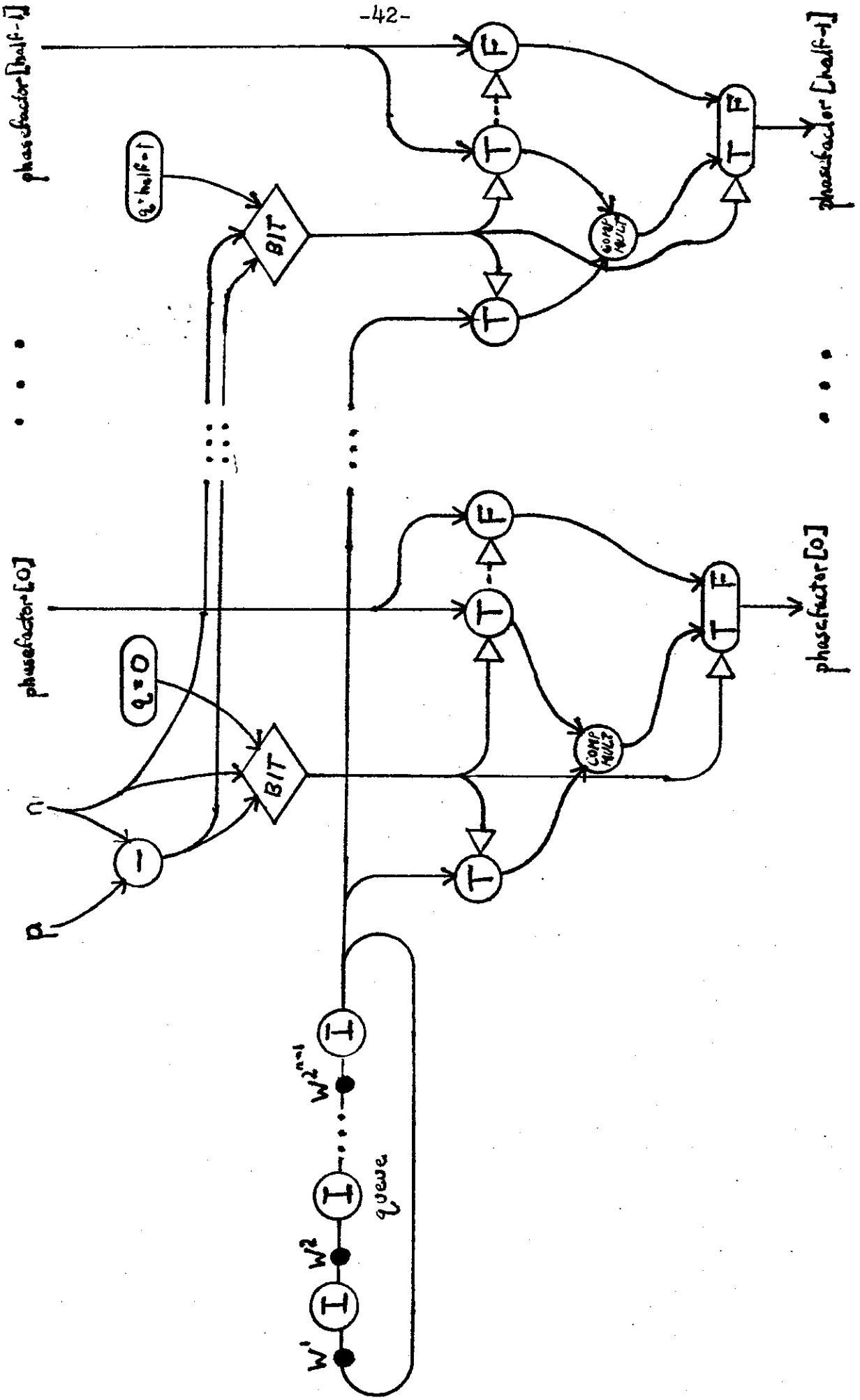


Figure 19. Parallel Phasefactor Update

sion of the serial program, the algorithm-clear solution is both slower and more costly than the serial program and hence is uninteresting at this level; its sole advantage is clarity.) Without going into the details of machine-level implementations, we cannot speak any more precisely about the space/time tradeoffs involved here. We will say, though, that just as potentially attractive solutions might be generated by mixing some of the attributes of the two "polarized" programs. For instance, the serial butterfly combined with the serial phasefactor update could prove too slow even for an application in which saving space is the primary concern. There is no reason why one couldn't, say, substitute parallel phasefactor updating into the program of figure 10 to yield a less compact but faster program.

The use of a data flow design helped us, obviously, in the parallel solution in which we were concerned with exploiting concurrency. What is more interesting is that, even when we hardly worried about concurrency at all, we still got quite a bit of it, as well as the advantages of pipelining. A measure of efficiency was automatically added in to the serial FFT. It would seem that VAL and data flow computation have a wide appeal; they are of no interest only where frugality is the sole concern.

7. Conclusions

Our approach here has been largely empirical. With no preconceived notions of what good VAL programming techniques might be like, we have developed a number of programs and taken note of how we did it. We have found that, for this application at least, good VAL programming overlaps with conventional programming methodology. Modularity remains important for organizing the program and making it readable. And one must still keep an eye out for obvious and not-so-obvious algorithmic improvements, such as those we implemented in the bit function and in phase factor computation. What remains unique to VAL programming is the detection and expression of concurrency. Deciding where forall blocks may be used is by no means the only task a programmer has in this regard, but it is a good example of how he must think in terms of data dependencies if he is to make any progress. Developing a program in stages seems to be a useful heuristic; the parallel program would have been much more difficult to write from scratch. We might say further as a result of our work that the VAL programmer, perhaps more than the conventional programmer, should have some knowledge of underlying machine architecture and of the translation process by which his program is to be mapped onto this architecture. In data flow, the quintessential expression of an algorithm is in the data flow graph. The programmer must have some idea of what kind of graph his program will generate if he is to know how to best use VAL. The space-

time implications are thus also part of his responsibility.

In terms of devloping data flow software techniques, the gains made here are admittedly slight. Much work (for other people) remains to be done in this area, as well as in data flow computation in general.

Acknowledgements

I would like to thank Professor Dennis and Clement Leung for their assistance. I would also like to thank Bertrand Russell, who sustained me throughout.

REFERENCES

(1) Ackerman, W., and Dennis, J. VAL-A Value-Oriented Algorithmic Language: Preliminary Reference Manual. Computation Structures Group, Laboratory for Computer Science, MIT. June 1979.

(2) Brock, D., and Montz, L. Translation and Optimization of Data Flow Programs. Computation Structures Group, Laboratory for Computer Science, MIT. July 1979.

(3) Dennis, J., Leung, C., and Misunas, D. A Highly Parallel Processor Using a Data Flow Machine Language. Computation Structures Group, Laboratory for Computer Science, MIT. January 1977.