TABLE IV
EXAMPLE 2—LOW-PASS: FINAL DESIGN REPORTED BY DUBOIS AND LEICH

a_2	a_1	b_2	b_1
1.0	-1.102534	0.64162667	-1.4031794
1.0	-0.194671	0.80639809	-1.4494845
1.0	-1.407262	0.97502064	-1.5030398
1.0	-1.338983	0.90668656	-1.4782185
0.0	1.0	0.0	-0.4201718

Note: Gain constant $k = 0.0078599138$.

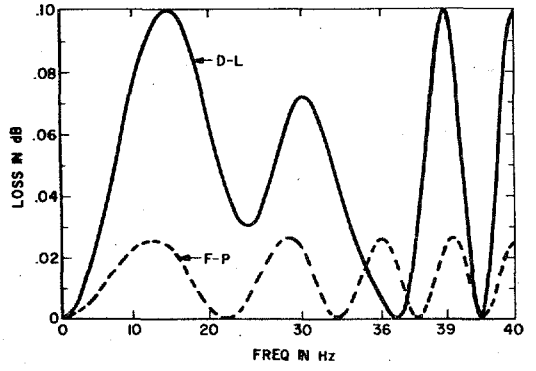

Fig. 3. Example 2: pass-bands from Dubois and Leich (D-L) result and from Fletcher-Powell (F-P) procedure; all specifications satisfied stop-bands different.

TABLE V
EXAMPLE 2—LOW-PASS: THE OPTIMIZED DESIGN FROM FLETCHER-POWELL PROCEDURE WITH MATCHING STOP-BAND

a_2	a_1	b_2	b_1
1.0	-1.102534	0.81285472	-1.4649795
1.0	-0.194671	0.91474971	-1.4829583
1.0	-1.407262	0.97565304	-1.5015193
1.0	-1.338983	0.62768999	-1.3972851
0.0	1.0	0.0	-0.3643101

Note: Gain constant $k = 0.0082154925$.

stop-band results and then compare pass-bands. This time the final design (Table V) had a stop-band matching that of the Dubois and Leich result, and the pass-band response was twice as good. See Fig. 4. Again, the nonlinear scale was used for clarity.

Each solution for the examples cited converged without apparent difficulty in approximately ten major iterations as described earlier.

In view of this evidence and that from many other examples we have run, it is difficult to support the negative statements of Dubois and Leich with respect to the Fletcher-Powell algorithm.

ACKNOWLEDGMENT

The author thanks Dr. James F. Kaiser of Bell Laboratories for helpful discussions.

REFERENCES

[1] L. S. Lasdon and A. D. Waren, "Optimal design of filters with bounded, lossy elements," *IEEE Trans. Circuit Theory*, vol. CT-13, pp. 175-187, June 1966.

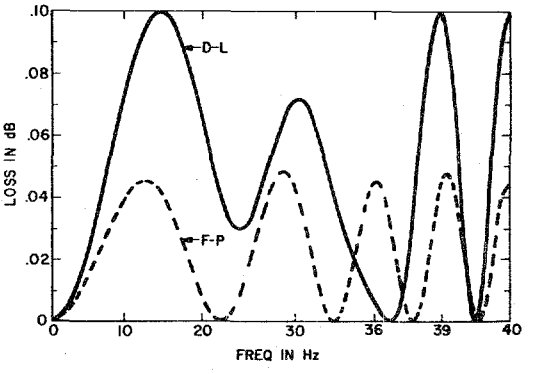

Fig. 4. Example 2: pass-bands from Dubois and Leich (D-L) result and from Fletcher-Powell (F-P) procedure; all specifications satisfied, stop-bands matching.

[2] A. V. Fiacco and G. McCormick, "Nonlinear programming: Unconstrained minimization techniques," New York: Wiley, 1968, chap. 3.
[3] R. Fletcher and M. J. D. Powell, "A rapidly convergent descent method for minimization," *Comput. J.*, vol. 6, pp. 163-168, July 1963.
[4] H. Dubois and H. Leich, Example 2 suggested in review dated Feb. 1976.

Simplified Control of FFT Hardware

DANNY COHEN

Abstract—**A particularly simple way to control fast Fourier transform (FFT) hardware is described. The method produces the indices both for inputs of each butterfly operation and for the appropriate W. In addition, this method allows parallel organization of the memory such that at any stage the two inputs and outputs of each butterfly belong to different memory units, hence can always be accessed in parallel.**

The highly symmetric structure of the discrete Fourier transform (DFT) lends itself nicely to implementation in a variety of parallel processing schemes. Different aspects of the symmetry are exhibited by the approaches described in [1]-[5]. In this correspondence we show a particularly simple way to control a fast Fourier transform (FFT) processor. In contrast to the more familiar control methods which control the FFT using three separate iteration loops, the method described here uses only a single counter, obtaining from it by simple-to-implement logical operations not only the required indices for the memory references required for each butterfly, but also the indices for the values of the complex multiplication constants needed.

Fig. 1 shows the data flow for 16-point FFT. Let $\langle s, t \rangle$ represent the butterfly operation:

$$X'(s) = X(s) + W * X(t)$$

$$X'(t) = X(s) - W * X(t)$$

The sequence of operation suggested by Fig. 1 is:

Manuscript received December 5, 1975; revised June 3, 1976.
The author is with the Information Sciences Institute, Marina del Rey, CA 90291 and the Department of Computer Science, California Institute of Technology, Pasadena, CA 91125.

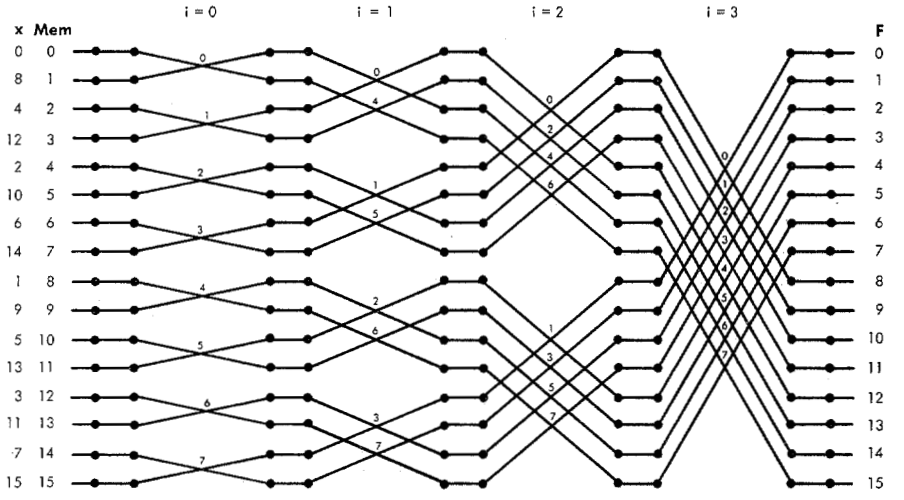

Fig. 1. 16 point FFT. Horizontal lines and W's omitted. Input is bit-reversed. Output is in normal order. The small numbers are the js.

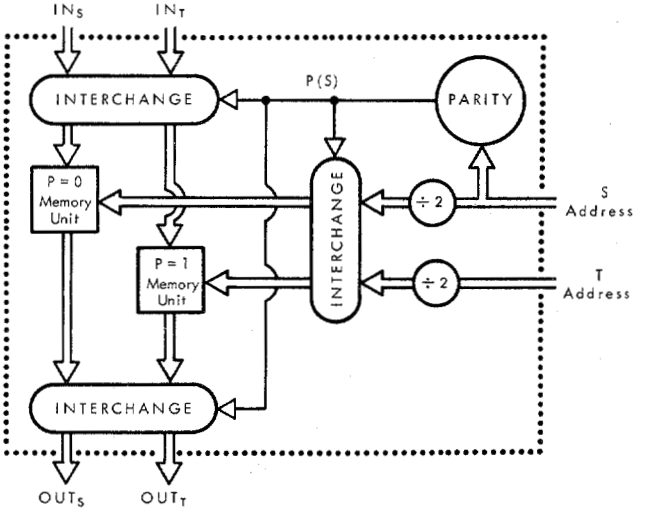

Fig. 2. Memory organization to ensure parallel access according to parity.

⟨0, 1⟩ ⟨2, 3⟩ ⟨4, 5⟩ ⟨6, 7⟩ ⟨8, 9⟩ ⟨10, 11⟩ ⟨12, 13⟩ ⟨14, 15⟩
⟨0, 2⟩ ⟨1, 3⟩ ⟨4, 6⟩ ⟨5, 7⟩ ⟨8, 10⟩ ⟨9, 11⟩ ⟨12, 14⟩ ⟨13, 15⟩
⟨0, 4⟩ ⟨1, 5⟩ ⟨2, 6⟩ ⟨3, 7⟩ ⟨8, 12⟩ ⟨9, 13⟩ ⟨10, 14⟩ ⟨11, 15⟩
⟨0, 8⟩ ⟨1, 9⟩ ⟨2, 10⟩ ⟨3, 11⟩ ⟨4, 12⟩ ⟨5, 13⟩ ⟨6, 14⟩ ⟨7, 15⟩

However, we prefer (for reasons shown later) to recognize it into the following order:

⟨0, 1⟩ ⟨2, 3⟩ ⟨4, 5⟩ ⟨6, 7⟩ ⟨8, 9⟩ ⟨10, 11⟩ ⟨12, 13⟩ ⟨14, 15⟩
⟨0, 2⟩ ⟨4, 6⟩ ⟨8, 10⟩ ⟨12, 14⟩ ⟨1, 3⟩ ⟨5, 7⟩ ⟨9, 11⟩ ⟨13, 15⟩
⟨0, 4⟩ ⟨8, 12⟩ ⟨1, 5⟩ ⟨9, 13⟩ ⟨2, 6⟩ ⟨10, 14⟩ ⟨3, 7⟩ ⟨11, 15⟩
⟨0, 8⟩ ⟨1, 9⟩ ⟨2, 10⟩ ⟨3, 11⟩ ⟨4, 12⟩ ⟨5, 13⟩ ⟨6, 14⟩ ⟨7, 15⟩

which, of course, yields the same results. The reason for this order is that the jth butterfly in the ith iteration is $\langle s, t \rangle$ where:

$$s = ROTATE_n(2j, i) \qquad n = \log_2 N$$

$$t = ROTATE_n(2j + 1, i) \qquad i = 0, 1, \cdots (n - 1),$$

$$j = 0, 1, 2, \cdots (N/2 - 1)$$

where $ROTATE_n(X, m)$ is the value of X rotated left, by m bits, within n bits, e.g., $ROTATE_4(13, 3) = 14$ and $ROTATE_4(5, 2) = 5$.

Pease [1] showed that for every butterfly the two indices involved differ in their parity (the parity of X is defined as zero if the number of *ONE*s in the binary representation of X is even, and as one otherwise). Hence it is possible to organize the N-point memory needed in two banks, according to the parity of the addresses, such that in any cycle two points

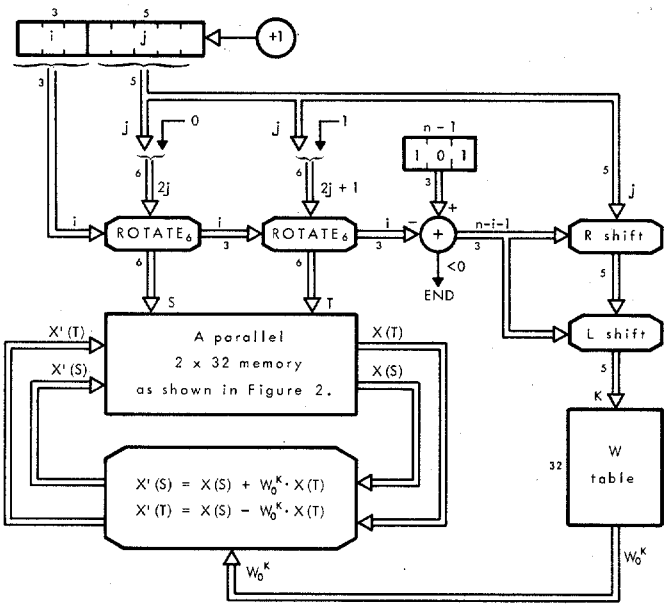

Fig. 3. Control structure for the FFT processor, for $N = 64$ ($n = 6$).

$X(s)$ and $X(t)$ are accessed in parallel, since s and t differ in parity, and $X(s)$ and $X(t)$ are stored in different banks. This memory organization can be implemented simply, as shown in Fig. 2.

One can also show that if the jth butterfly in the ith iteration requires a unit-root W, which is the kth power of $W_0 = \exp(-2\pi i/N)$, then k may be obtained from j by masking out ("zeroing") its $(n - 1 - i)$ least significant bits. Of course, it is easier to store W_0^k, for values of k between 0 and $(N/2 - 1)$, in a ROM rather than compute them. However, if one prefers multiplication to storage, the required W's are in the proper order for incremental multiplications.

By using the above observations, one can design a very simple control structure for an FFT processor, as shown in Fig. 3. This system should be initially loaded with the N sequential data values (applying bit-reversal which does not change parity) two at a time. The "i-j" counter is then cleared, and the FFT operation performed, incrementing this counter by one unit for each butterfly operation until its "i" portion reaches the value "n". The resulting data can be unloaded, two values at a time, using sequential (i.e., not bit-reversed) order.

It is simple to extend all of the above ideas to higher radices, both for computation and for parallel memory accesses, and for memory/compute overlapping.

ACKNOWLEDGMENT

The author wishes to express his deep appreciation to Ivan E. Sutherland of the California Institute of Technology for his help in preparing this document.

REFERENCES

[1] M. C. Pease, "Organization of large scale Fourier processors," *JACM*, vol. 16, pp. 474-482, July 1969.
[2] B. Gold and T. Bially, "Parallelism in fast Fourier transform hardware," *IEEE Trans. Audio and Electroacoustics*, vol. AU-21, pp. 5-16, Feb. 1973.
[3] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. New York: Prentice-Hall, 1975.
[4] B. Gold and C. M. Rader, *Digital Processing of Signals*. New York: McGraw-Hill, 1969.
[5] L. R. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing*. New York: Prentice-Hall, 1975.

A Computer Programming System Using Continuous Speech Input

A. D. C. HOLDEN AND EDWARD STRASBOURGER

Abstract—**A functioning system is described in which spoken Fortran-like programs can be reliably interpreted by a computer. Speech is encoded into cepstral patterns using a shifting overlapping 32-ms Hamming window.**

A vocabulary of 26 words was selected so that the distance between different words, in their cepstral representation, was large. The syntax of individual phoneme strings in a word, and the syntax chosen for the programming language, were used to resolve ambiguous decisions. Feedback to the programmer greatly increased the reliability of the system, since, 1) mistaken decisions could be corrected, and 2) the programmer gradually learned to speak in such a way that system mistakes were greatly reduced.

With two trained speakers, the recognition rate was 50-75 percent on sentences, 75-90 percent on words, and with user feedback and correction, the recognition rate was 100 percent with no more than two repetitions for long statements. The good performance achieved was due to the well-chosen vocabulary, strong syntactic support, and speaker training. The basic drawback for the system at present is that the initial training of the user is a time-consuming process. However, further improvement has since been achieved by using the initial isolated cepstral prototypes to locate new prototypes in samples of continuous speech, and then using the "continuous prototypes" for recognition.

Currently, formant trajectories, derived from a pitch synchronous, linear prediction analysis, are being used, and an automatic stress analysis program provides segmentation, and guides the selection of key allophones.

INTRODUCTION

The well-known unreliability of speech recognition systems so far developed would give the impression that an attempt to design a spoken computer language is premature. Several factors, however, can be exploited to ensure 100-percent reliability in this case.

RELIABILITY FACTORS

The difficulty associated with reliable speech recognition in general [1], [2]-[5] is reduced when the following factors are utilized in the design of a spoken programming language.

1) The designer is free to select a set of allophones, as the basic units of the language, which are distinguishable in the particular recognizer adopted.

2) Ambiguities in the recognition process can be resolved using the syntax of the programming language. Thus the syntax specifies which phonemes can occur in sequence in a legal statement.

3) Visual feedback to the human programmer allows for corrections by the repetition of statements.

4) The programmer, with experience, learns to adapt to the system and this greatly reduces the error rate.

THE SPOKEN COMPUTER LANGUAGE

The *SPO*ken *CO*mputer *L*anguage (SPOCOL) described here resembles a simplified version of Fortran. Fig. 1 gives examples of typical SPOCOL statements and their corresponding Fortran codings. The double use of the first constant in the LOOP and SHOULD statements should be noted. The constant is both part of a variable (the indexing variable in the LOOP

Manuscript received January 15, 1975; revised May 1, 1976.

A. D. C. Holden is with the Department of Computer Science and Electrical Engineering, University of Washington, Seattle, WA 98195.
E. Strasbourger is with the Technion, Haifa, Israel.