ESD-TR-77-193

Technical Note

1977-33

J. Gorski-Popiel

A Digital Microprocessor Channel Vocoder

10 August 1977

Prepared for the Department of the Air Force
under Electronic Systems Division Contract F19628-76-C-0002 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the Department of the Air Force under Contract F19628-76-C-0002.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Raymond L. Loiselle

Raymond L. Loiselle, Lt. Col., USAF
Chief, ESD Lincoln Laboratory Project Office

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

A DIGITAL MICROPROCESSOR CHANNEL VOCODER

J. GORSKI-POPIEL
Group 64

TECHNICAL NOTE 1977-33

10 AUGUST 1977

LEXINGTON MASSACHUSETTS

ABSTRACT

A complete, real-time, channel vocoder delivering good speech quality with a 2400-bit/second data transmission rate was implemented using purely digital circuitry in the form of a high-speed programmed microprocessor.

Necessary algorithms are presented and their effect on the machine design is discussed in detail. The end product is a very high-speed computing machine (measured in program throughput terms). It turned out to have a high degree of programming flexibility, which would make it adaptable to other tasks. This was a bonus, not an original goal. The project was conceived and successfully realized as the most practical way to build a vocoder for actual use in the LES-8/9 satellite communications system.

TABLE OF CONTENTS

I. INTRODUCTION

A digital vocoder is a device which extracts from samples of speech those attributes which are most essential for accurate synthetic speech reproduction, subject to the constraint that the data link between the transmitter and receiver utilizes a minimum data rate. The microprocessor described in this report was designed specifically to implement a 2400-bit/second channel vocoder delivering good speech quality in real time. The judge of what does and does not constitute acceptable quality is the human ear. The criteria are thus highly subjective and ill suited to precise mathematical description. As a result, vocoder algorithms are largely empirical in nature. The good ones have taken many years to develop. Not unexpectedly, however, one fundamental fact has emerged: — computational complexity and final speech quality for a given data rate are directly related. The Gold-Rader-Tierney channel vocoder algorithm used in this project was developed over a period of several years by a number of people with the goal of excellent speech quality. It is thus rather complex and takes a great deal of computation to implement. Several successful implementations were built using analog circuitry for the spectrum analysis and synthesis, and digital pitch detectors. However, the algorithm has never before been implemented in an all-digital machine in real time, because of the high required machine speed. The processor described in this report is the first machine with sufficiently high-speed capabilities to realize the Gold, et al. channel vocoder algorithm in full duplex real-time form. This was achieved using commercially available Schottky-clamped TTL logic and an 8-MHz system clock rate. To give a feeling for the speeds involved, an IBM 370 has a program throughput rate roughly two orders of magnitude too slow.

Many vocoder algorithms were developed under the guidance of B. Gold over the last decade or more. A great deal of algorithm simulation on large machines has been done. The most recent work of interest for this project was done by P. Demko and J. Tierney, both of Group 27 (unpublished internal report). They made use of a general purpose computer to simulate a 16-channel vocoder in non-real time. Finite length words were used in order to determine the minimum required computer accuracy for acceptable reproduction of vocoded speech. A given word length once decided upon was used throughout the computation, all of which was done in fixed point arithmetic. Under constraints of this nature, computational optimization is equivalent to finding optimal scaling of the processed numbers such that dynamic range is maximized. The results of their work are therefore of great value to anyone interested in building a fixed point microprocessor to implement a channel vocoder. Specifically, their results indicated that a 12-bit coefficient and sample word together with 16-bit processing was a good choice. 18-bit processing gave only marginal improvement. The machine described in this report follows the Tierney-Demko simulation fairly closely. One major exception is the use of 16-bit operations throughout, including coefficient lengths. Other exceptions are of a more detailed and minor nature and will be pointed out as we come to the appropriate point.

The main thrust of this report is the description of the actual computing machine; however, since the reason for its existence is a realization of a channel vocoder a description of the salient features of the algorithm implemented will be given.

The method by which the machine was designed will also be discussed. This should be helpful to others faced with similar design problems. At the

beginning of the project I knew of no machine design capable of the high program throughput speed needed for the channel vocoder algorithm. Existing microprocessor designs and standard digital design procedures were, with some exceptions, not too helpful. A new approach to the problem had to be worked out. The eventual success of the project was, I believe, almost entirely due to the design approach developed; that is, simultaneous and closely coupled design of both hardware and software. To start with, we knew the algorithm to be realized, and could therefore write down all the mathematical expressions, and thus the required computational forms. Using available integrated circuits the problem then reduced to fitting them together in such a way that the computational forms could be executed in the most efficient way. Each required operation was carefully scrutinized and implemented in software or in hard-wired logic, whichever of the two achieved greater efficiency. As an example, consider a sequence of identical operations performed on varying data. In software, it is usual in such cases to make use of a so-called DO LOOP. An index is set, a test performed and a decision made whether to return to the beginning of the sequence or to continue. Symbolically, such a procedure may be expressed as follows:

```
1.  Set I = 0, K = M
2.  Take in New Data
3.
.   {Perform Required
.   {   Operations    }
n
n + 1.  Set I = I + 1
n + 2.  If (K - I) > 0 go to Step 2, continue otherwise
n + 3.      ----
n + 4.      ----
n + 5.      ----
```

In this sequence, steps (n + 1) and (n + 2) do not contribute to the computation, but only to the control. They could thus be considered overhead, costing both program memory and execution time. If, however, we eliminate the DO LOOP control instructions many more program steps would have to be written. An optimum solution therefore is to build in a DO LOOP mechanism in hardware. For example, one which will give rise to the following software:

```
1.  Do 2, n, M
2.  Take in New Data
3.
.   {Perform Required
.   {   Operations    }
.
n
n + 1
n + 2
n + 3
```

The above means DO step 2 through n, M times and on completion continue with the program. It should be noted that steps n + 1 and n + 2 are now no

longer in the loop. All needed controls are line 1. If each program line takes τ sec to execute, the above solution, besides saving two lines of program, also shortens the executive time by $2M\tau$ sec. For large M this can be quite considerable.

The software operations, instruction set, and machine architecture are designed together as described, modifying all three as necessary while working through the required mathematical expressions. It will, of course, be appreciated that after several steps in the process, one will have to go back to the beginning and re-assess the impact of the latest changes on the previous computational procedures. Thus, this is an iterative design procedure with feedback, which stops when all required computations are implementable and the job can be done within the required time. In terms of engineering esthetics, it is a very satisfying process in that it meets all requirements, and allocates tasks efficiently between hardware and software.

After following the above design philosophy for some time it suddenly dawned on me that the design mode allows computational time problems to be dealt with especially easily. It is quite straightforward to add more paralleling, more pipelining, and more hardware as required without starting over. Another very interesting observation was made when the design was completed. Despite the decidedly dedicated nature of the design, the end product is by any reasonable definition a general purpose computing machine. It does have peripherals which are specifically geared for the vocoding process such as pre-sample filtering, special format data storage, and acquisition systems. However, the machine itself is quite general purpose. It can do addition, subtraction, and multiplication. It has DO LOOPS and conditional and unconditional jumps. It

can perform conditional operations and make decisions as a result of some operations. It also has been programmed to implement a self-diagnostics program, a task totally different from the vocoding algorithm.

When faced with the need for a machine to implement a vocoder or a task of similar complexity, it is perhaps natural to draw on the extensive past experience accumulated by general purpose computer designers. This leads directly to a simple "classical" architecture based on familiar design concepts and is capable of achieving fastest possible machine cycle times, a fact frequently quoted as its justification. Implementations along these lines can easily achieve execution times per instruction a factor of 5 or so shorter than those of the currently proposed machine. Complex signal processing operations are then carried out by very complex and lengthy software.

Perhaps the single most interesting result to emerge from this project is the fact that substantially greater total program throughput rate may be achieved by settling for a slower basic cycle time, but concentrating instead on making each instruction as powerful and efficient as possible. This claim is borne out by the fact that at this writing no full-duplex channel vocoder of similar complexity has been implemented digitally, despite a keen interest in such devices, other than the machine described here. It is believed that for tasks involving signal processing or filtering where maximum throughput of mass real time data is the keynote the proposed approach will yield a more efficient end product. The chief contributing factor is extensive use of "firmware." This term means PROM implemented special functions (like log tables for example) which as a result can be recalled with a single command (not unlike a subroutine in a Fortran program). This greatly shortens the required software and

correspondingly speeds execution. This development could not have been possible without the recent introduction of LSI and large PROMs since they make it possible to realize custom parallel architecture and fast look up tables with comparatively little design effort, at small cost in power, size, parts count and ultimately dollars.

After the final design is completed, debugged and working it is almost inevitable that the question comes up "If it had to be done again would I do it the same way?" Equally inevitably the answer is: "not quite." This case is of course no exception. Following is a number of comments arrived at by hindsight. They may prove useful to anyone faced with a similar design problem.

The difficulty in creating a sufficiently flexible addressing scheme was underestimated. As a result the part of the machine dealing with this problem was underdesigned at first creating a lot of headaches later on. All this would have been avoided if addressing received greater attention at the very outset. If this had been done it is also very likely that a better, more flexible addressing scheme would have evolved.

Difficulty of debugging rises exponentially with the number of ICs used. It is therefore very important to include eventual debugging procedures into the design process. This may raise the IC count slightly, but will repay itself manyfold later on. One very attractive way to do this is to set aside a reasonably large part of the program memory for a self diagnostic routine. This would exercise all possible machine modes one at a time (if feasible) or jointly by operating on some predetermined numbers. During each operation the output of the arithmetic section, for example, is monitored by comparing it with a precomputed value. The whole procedure should be programmed in such a

a way that if a disagreement is detected the error will have originated in only a small part of the machine. In this way the detection of several errors should pinpoint the malfunctioning of individual ICs. In the present machine self diagnostics was added at the end, and was available only by connecting a separate specially designed board. This was due mainly to the non-availability of sufficiently large ROMs. The diagnostics unfortunately created problems on its own mainly due to propagation delays. It is felt that self-diagonstics should be an integral and permanent part of the machine. If the machine is to be a subsection of a much larger system composed of other programmable machines each with its own self-diagnostics all of these diagnostic routines could be tied together making it possible to debug even very large systems with relative ease and in a very small fraction of the time it would take otherwise. Since self-diagnostic is essentially an exercising of the machine itself, it is estimated that its implementation should raise the IC count by not more than 5%.

As already mentioned, the advocated design procedure is only possible due to the introduction of LSI. With the appearance of even more complex LSI modules the process becomes more flexible still. A good example of this would be the use of the AM 2901 module. This is a 40-pin device containing an ALU, shift registers, buffers, ROM and RAM memory arranged to create very primitive arithmetic operations under control of the ROM which is already micro-programmed. A unit like this could have been used to advantage in both the addressing and arithmetic sections. Another example would be the incorporation of array multipliers, possibly custom designed, arranged on a single chip.

II. THE VOCODING ALGORITHM

Essentially the program consists of four distinct groups.

2.1 Spectrum analysis and real-time pitch computation.

The spectrum analysis operation extracts the energy content of 16 bandpass filters fed by the speech samples. The sampling period used was 140 μsec, or a rate of 7.14 KHz. Let us assume x(n) represents the current speech sample digitized to (12 bits in our case), and x(n-1) the previous one. The first set of computations are 49 bandpass poles defined by the following difference equations:

$$y_i(n) = k_{1i}[2\ y_i(n-1) - x(n-1)] - k_2\ y_i(n-2) + x(n) \qquad (1)$$

where

$$i = 1, 2..., 49 \quad .$$

The k_{1i} are 49 distinct constants, while k_2 is a single constant equal for all 49 filters. The $y_i(n)$, $y_i(n-1)$ and $y_i(n-2)$ are current, past and past twice removed filter outputs. Digital filters of this type are referred to as recursive. It is important to note that they contain by virtue of $y_i(n-1)$ and $y_i(n-2)$, computational feedback. A system of this kind may lead to instability especially in fixed point arithmetic machines. This problem, together with possible solutions will be discussed later.

The k_{1i} and k_2 are respectively given by

$$k_{1i} = r \cos(\beta_i T) \qquad (2)$$

and

$$k_2 = r^2 = \exp(-\alpha T)$$

where T = sample period (140 μsec), α = real part of the complex pole pair

($2\pi \times 60$ Hz), β_i = complex part of the pole pair. Table 2.1 gives the values of β_i and a summary of the relevant features of all 49 pole pairs. It will be seen that the poles lie on a line parallel to the imaginary axis. Furthermore, their positions have been chosen such that if combined in the manner to be described below, the bandpass filters formed will approximate a linear phase characteristic. Filters of this type are referred to as Lerner[4] filters.

The 49 bandpass poles are summed into 16 sets according to the pattern shown in Table 2.2 Each pole has a weight attached to it which starts with 0.5 for the first and then continues with alternating sign but of unit magnitude to end up with 0.5 again on the last pole for an odd number of poles and -0.5 for an even number. Thus, the first output is:

$$f_1(n) = |0.5\ y_1(n) - y_2(n) + y_3(n) - 0.5\ y_4(n)| \tag{4}$$

and the eleventh

$$f_{11}(n) = |0.5\ y_{21}(n) - y_{22}(n) + y_{23}(n) - y_{24}(n) + 0.5\ y_{25}(n)| \quad . \tag{5}$$

The envelopes of the rectified bandpass outputs $f_i(n)$ are lowpass filtered using a third-order transitional Gaussian to 12 dB characteristic cutting off at 35 Hz. Here the design differs from the original where a Bessel filter was used. The Gaussian characteristic has better step response characteristics and was therefore chosen here. The filters are realized in two steps. A first order section whose output is:

$$r_i(n) = \frac{\alpha}{2} [2\ r_i(n-1) - 2\ f_i\ (n-1)] + f_i(n) \tag{6}$$

is followed by a second order section:

TABLE 2.1
LERNER POLE POSITIONS

Pole Designation	Real Coordinate (Hz)	Imaginary Coordinate (Hz)	Resonant Frequency (Hz)	Q
1	60	160	170.9	1.42
2	60	200	208.8	1.74
3	60	280	286.4	2.39
4	60	320	325.6	2.71
5	60	400	404.5	3.37
6	60	440	444.1	3.70
7	60	520	523.5	4.36
8	60	560	563.2	4.69
9	60	640	642.8	5.36
10	60	680	682.6	5.69
11	60	760	762.4	6.35
12	60	800	802.3	6.69
13	60	880	882.0	7.35
14	60	920	921.9	7.68
15	60	1000	1001.8	8.35
16	60	1040	1041.7	8.68
17	60	1120	1121.6	9.35
18	60	1160	1161.6	9.68
19	60	1240	1241.4	10.35
20	60	1280	1281.4	10.68
21	60	1360	1361.3	11.34
22	60	1400	1401.3	11.68
23	60	1480	1481.2	12.34
24	60	1560	1561.2	13.01
25	60	1600	1601.2	13.34
26	60	1680	1681.2	14.01
27	60	1760	1761.0	14.68
28	60	1800	1801.0	15.01
29	60	1880	1881.0	15.68
30	60	1960	1961.0	16.34
31	60	2040	2040.9	17.01
32	60	2080	2080.9	17.34
33	60	2160	2160.8	18.01
34	60	2240	2240.8	18.67
35	60	2320	2320.8	19.34
36	60	2400	2400.8	20.01
37	60	2440	2440.7	20.34
38	60	2520	2520.7	21.01
39	60	2600	2600.7	21.67
40	60	2680	2680.7	22.34
41	60	2760	2760.6	23.01
42	60	2800	2800.6	23.34
43	60	2880	2880.6	24.01
44	60	2960	2960.6	24.67
45	60	3040	3040.6	25.34
46	60	3120	3120.6	26.01
47	60	3200	3200.6	26.67
48	60	3280	3280.6	27.34
49	60	3320	3320.5	27.67

TABLE 2.2

LERNER FILTER DATA

Filter	Number of Poles	Poles	3-dB Bandedges (Hz)	3-dB Bandwidth (Hz)
1	4	1, 2, 3, 4	180 - 300	120
2	4	3, 4, 5, 6	300 - 420	120
3	4	5, 6, 7, 8	420 - 540	120
4	4	7, 8, 8, 10	540 - 660	120
5	4	9, 10, 11, 12	660 - 780	120
6	4	11, 12, 13, 14	780 - 900	120
7	4	13, 14, 15, 16	900 - 1020	120
8	4	15, 16, 17, 18	1020 - 1140	120
9	4	17, 18, 19, 20	1140 - 1260	120
10	4	19, 20, 21, 22	1260 - 1380	120
11	5	21, 22, 23, 24, 25	1380 - 1580	200
12	5	24, 25, 26, 27, 28	1580 - 1780	200
13	6	27, 28, 29, 30, 31, 32	1780 - 2060	280
14	7	31, 32, 33, 34, 35, 36, 37	2060 - 2420	360
15	7	36, 37, 38, 39, 40, 41, 42	2420 - 2780	360
16	9	41, 42, 43, 44, 45, 46, 47, 48, 49	2780 - 3300	520

$$C_i(n) = k_3[2C_i(n-1) - 2\, r_i(n-1)] - k_4[C_i(n-2) - r_i(n-1)] + r_i(n) \qquad (7)$$

α, k_3 and k_4 are constants computed to give the required characteristic.

The outputs $C_i(n)$ give the energy in the 16 channels and form the output of the spectrum analysis section. Much of the basic bandwidth compression of the vocoder has occurred at this point. The original sampled speech in a bandwidth of about 3 KHz is now represented by 16 abstracted spectral-energy functions whose total bandwidth is just 16(35) = 560 Hz.

The pitch extraction, both real time and non-real time is described in great detail in Ref. 1. Thus, only those parts necessary to illustrate the computational structure will be discussed here. The purpose of the real time pitch computation is to make preliminary estimates of the pitch period. This is done on samples of only the bottom 900 Hz since it is known that the fundamental pitch period will never be in excess of 900 Hz. The actual current sample is labelled x_{n+1}, the previous sample once removed x_n and the sample twice removed x_{n-1}. For computational purposes however x_n is treated as the current sample thereby making x_{n+1} the immediate future and x_{n-1} the immediate past sample. A parameter Δ is now defined by (See Fig. 3 for flow diagram).

$$\Delta_o = 0$$

$$\Delta_{n+1} = \begin{cases} +1 & \text{if } x_{n+1} > x_n \\ \Delta_n & \text{if } x_{n+1} = x_n \\ -1 & \text{if } x_{n+1} < x_n \end{cases} \qquad (8)$$

$$\Delta_{n+1} - \Delta_n = \begin{cases} +2 & x_n \text{ is a positive peak} \\ 0 & \text{there is no peak} \\ -2 & x_n \text{ is a negative peak} \end{cases} .$$

When peaks are detected, their magnitudes are stored in P_{cp} and P_{cn}, i.e., if $\Delta_{n+1} - \Delta_n = 2$, the content of the current positive peak P_{cp} is shifted into the past positive peak storage location labelled P_{pp} and x_n is written into P_{cp}. Similarly for a negative peak P_{cn} is shifted into the past negative position P_{pn} and x_n is placed into P_{cn}. For the majority of cases when no peak is detected the contents of P_{cp}, P_{cn}, P_{pp} and P_{pn} are not disturbed. At any one sample time magnitudes of current and past positive and negative speech waveform peaks are available. Six parameters, defined in Eq. (9) are next formed.

$$\begin{aligned} m_o &= |P_{cp}| & m_3 &= |P_{cn}| \\ m_1 &= |P_{cp} - P_{pn}| & m_4 &= |P_{cn} - P_{pp}| \\ m_2 &= |P_{cp} - P_{pp}| \text{ if } P_{cp} \geq P_{pp} & m_5 &= |P_{cn} - P_{pn}| \text{ if } P_{cn} \geq P_{pn} \\ &= 0 \text{ for } P_{cp} < P_{pp} & &= 0 \text{ for } P_{cn} < P_{pn} \end{aligned} \qquad (9)$$

The detailed rationale for the above choice of measurements is described in Ref. 1. Basically, if time intervals between them are measured, m_o, m_1, m_3, and m_4 give a good indication of the period for wave shapes with a strong fundamental component present. m_2 and m_5 provide a correct period for strong second harmonics and only some fundamental component waves. This information is extracted by the following procedure.

For each m_i whenever a new m_i is computed for a time τ_i, called the blanking period, no computations are performed, then a parameter α_i is computed given by

$$\alpha_i = m_i \times \exp(-N \ \ell n 2/P_{av_i}) \tag{10}$$

where N is effectively zero during blanking and is then incremented by unity every sample period. α_i therefore represents an exponential run-down which reaches half its original value when $N = P_{av}$. Updating of α_i stops when a new m_i is found which is not less than the current value of α_i. The time in multiples of the sampling period, from the beginning of the blanking interval until the current cessation of the α_i rundown is stored in P_i which in turn defines all the above parameters thus:

$$\begin{aligned} P_{av_{old}} &= \frac{1}{2}\,(P_{av_{new}} + P_i) \\ \tau_i &= 0.4\ P_{av_{new}} \quad . \end{aligned} \tag{11}$$

The flow diagram for the above is shown in Fig. 1. The six P_i are the initial pitch estimates. Blanking and run-down procedures are helpful in reducing spurious very short pitch period estimates and those produced by noise. Three sets of P_i are kept in memory; the current set and the two most recent past sets.

2.2 Non-Real Time Pitch Computation

The remainder of the pitch extraction process depends only on the computed 18 values of P_i. Also since a new pitch estimate has to be made only once every 18 msec (this corresponds to slightly over 71 sample periods), the computation

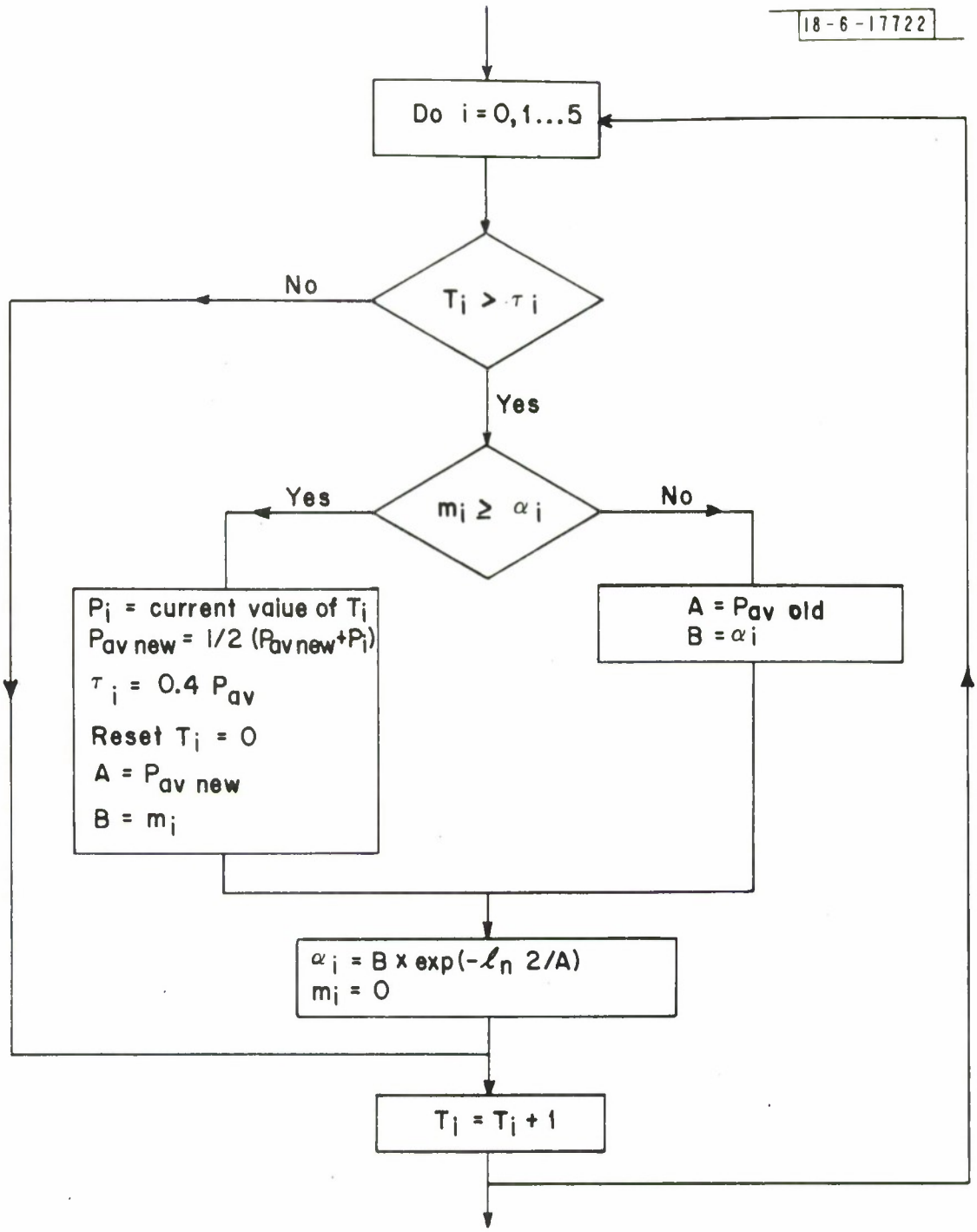


Fig. 1. Real-time pitch computation flow diagram.

from now on can, within broad limits, be done whenever convenient. It does not have to be finished during any one sample period. It may be spread out over several or done all at once every 10 msec. The term "non-real time pitch computation" may be misleading since the operation does result, in conjunction with the rest of the algorithm, in real time speech processing. The name merely designates those parts of the computations which do not have to be performed every input-sample period. The approach adopted here was to do the whole non-real-time pitch computation all at once every 10 msec. This simplifies control, since the operation does not have to be interrupted.

The computation consists of arranging a table of period estimates and then choosing the most likely candidate. At the same time attention is also directed to the energy of the signals involved. If a certain threshold level is not exceeded, the samples are assumed to be caused by background noise and not speech. They are therefore labeled for what they are despite any possible detected periodicity that may be associated with them. If the energy level is exceeded but the dispersion of the pitch estimates is large (e.g., no two estimates are alike), the speech sample may in fact have no defined pitch at all as in an incoherent sound like "s." Both of the above cases are "Hiss." In all other cases a 7-bit word representing the most likely pitch period (referred to as "Buzz") as a multiple of the sampling period is found. The above process is commonly called the Buzz-Hiss or voiced-unvoiced decision. Table 2.3 gives some examples to further clarify the procedure.

The word used to represent Hiss is 0 0 0 0 1 1 1. Thus if the energy

TABLE 2.3

Pitch Word	Decimal Equivalent	Pitch		Comments
		Period = 140 μsec × Pitch Word	Frequency	
0 0 0 0 0 0 0	0			Hiss
0 0 0 0 1 1 1	7	0.98 msec	1020.4 Hz	
0 0 0 1 0 0 0	8	1.12 msec	892.9 Hz	True Pitch Estimate
0 0 0 1 1 1 1	15	2.10 msec	476.2 Hz	
0 0 1 0 0 0 0	16	2.24 msec	446.4 Hz	
0 0 1 1 1 1 1	31	4.34 msec	230.4 Hz	
0 1 0 0 0 0 0	32	4.48 msec	223.2 Hz	
0 1 1 1 1 1 1	63	8.82 msec	113.4 Hz	
1 0 0 0 0 0 0	64	8.96 msec	111.6 Hz	
1 1 1 1 1 1 1	127	17.78 msec	56.2 Hz	

threshold is not exceeded or the dispersion is too large, this word is put out for transmission as the current pitch estimate.

The computations involved here are as follows: Let the current 6 P_i be designated as column 1, the one preceeding this column 2 and the one before that as column 3. Also let P_{ij} denote the entry in the i^{th} column and j^{th} row. A 6 x 6 matrix is now formed which includes besides the above 3 as columns 1, 2 and 3 also the following:

$$P_{4j} = P_{ij} + P_{2j}$$
$$P_{5j} = P_{2j} + P_{3j} \qquad (12)$$
$$P_{6j} = P_{1j} + P_{2j} + P_{3j}$$

The reason for these rows is that for waves rich in harmonics the original estimates P_{1j} to P_{3j} may erroneously detect second or third harmonics. For such cases P_{4j} to P_{6j} are more likely to give the correct pitch estimate. Next a set of window functions for each entry of the first column is defined as

$$W_k(P_{1j}) = k \times 6.25\% \text{ of } P_{ij} = 0.0625\ k\ P_{ij} \qquad (13)$$

where

$k = 1, 2, 3$ and 4.

A score NC_{qk} is then incremented by unity every time if

$$|P_{1q} - P_{ij}| \leq W_k(P_{1j}) \qquad (14)$$

For each of the 6 q values all i and j are used. The score is augmented by a bias term BT_k (where BT_k = 8, 6, 3, and 1 for k = 1, 2, 3 and 4, respectively) giving

$$C_{qk} = NC_{qk} + BT_k \qquad (15)$$

The P_{1q} resulting in the largest C_{qk}, P_{1max} say, is then compared with a threshold term CT. The pitch to be transmitted

$$P_{TR} = P_{1max} \quad \text{if } P_{1max} > CT$$
$$= \text{Hiss word if } P_{1max} \leq CT \qquad (16)$$

The threshold term CT = 13.

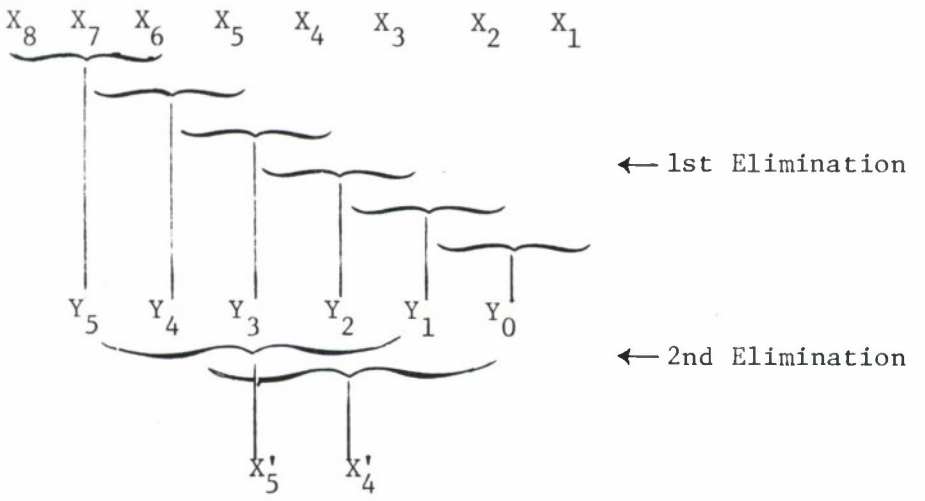
The process just described will, with reasonably high probability, give a correct pitch estimate. Unfortunately, errors are also inevitable. Their effect can, to a noticeable degree, be eliminated if data smoothing is employed on the series of final pitch estimates. This process removes rapid alterations between Buzz and Hiss and results in the smoothing out of implausibly rapid time variations in the pitch estimates. The salient features of this procedure follow.

At the end of the pitch evaluation, after the Buzz- Hiss decision has been made, a shift register is loaded with a 1 if the current pitch word is Buzz and a 0 if it is Hiss. Using an 8-bit register, the decision for the past 8-pitch words is stored. The low-pass filters in the spectrum analysis introduce a 60-msec delay in the spectrum data. The pitch data on the other hand is delayed at most 10 msec. So, in order to time align pitch and spectrum in formation, the current spectrum information should be combined with the pitch word computed 50 msec ago, i.e., 5 final pitch outputs of delay. The 5th-bit in the above pitch register then represents the pitch word of current interest. Let the 8 entries in the register be denoted by X_i. The first smoothing is done over adjacent sets of 3 X_i. The center entry is always altered to the majority. Thus, for example,

X_{l+1},	X_i,	X_{i-1}	Majority	
0	1	1	1	hence no change, $X_i = 1$
0	1	0	0	X_i is changed to 0

The decisions are made on overlapping sets of 3, moving one X_i down at a time. A second set of eliminations is done on the results of the first but now over

a set of five. Again the center X_i is changed according to the majority. The overall pattern is shown below:



It can be shown that this pattern is representable by the following logic expressions:

$$
\begin{aligned}
Y_1 &= X_2 \cdot X_3 + X_2 \cdot X_4 + X_3 \cdot X_4 \\
Y_2 &= X_3 \cdot X_4 + X_3 \cdot X_5 + X_4 \cdot X_5 \\
Y_3 &= X_4 \cdot X_5 + X_4 \cdot X_6 + X_5 \cdot X_6 \\
Y_4 &= X_5 \cdot X_6 + X_5 \cdot X_7 + X_6 \cdot X_7 \\
Y_5 &= X_6 \cdot X_7 + X_6 \cdot X_8 + X_7 \cdot X_8
\end{aligned}
\tag{17}
$$

and

$$
\begin{aligned}
X_5' = {} & Y_1 \cdot Y_2 \cdot Y_3 + Y_1 \cdot Y_2 \cdot Y_4 + Y_1 \cdot Y_2 \cdot Y_5 + Y_1 \cdot Y_3 \cdot Y_4 \\
& + Y_1 \cdot Y_3 \cdot Y_5 + Y_1 \cdot Y_4 \cdot Y_5 + Y_2 \cdot Y_3 \cdot Y_4 \\
& + Y_2 \cdot T_3 \cdot Y_5 + Y_2 \cdot Y_4 \cdot Y_5 + Y_3 \cdot Y_4 \cdot Y_5
\end{aligned}
$$

Using X_5' and the original X_5 in the shift register, the following pitch editing process ensues:

Decimal Equivalent	X_5'	X_5	Editing Procedure
0	0	0	Transmit Hiss word as computed.
1	0	1	Transmit Hiss word despite computed Buzz.
2	1	0	Find Median over $X_2 \rightarrow X_8$ and transmit as new Buzz.
3	1	1	Transmit Buzz word as computed.

The first two and the last procedures are self explanatory, the third one needs more elaboration. If originally $X_5 = 0$, Hiss would have been the decision. However, $X_5' = 1$ indicating that a Buzz word is needed. None is available so a new one has to be derived from neighboring ones. It appears that for editing of this type, medians are an optimal choice. A median is defined as that value of a distribution for which half its members are smaller and half larger. Since the median estimation is done with actual pitch values, 8 past pitch estimates (denoted by P_{IP} in flow diagram) have to be also stored.

In order to estimate what will happen in the future as well as in the past, 72 samples x(n) of the speech input are stored in FIFO memory. Samples coming through the analog pitch channel are clocked into a buffer directly (the px(n + 72) buffer). The current samples to be used for computation are those extracted from FIFO memory, therefore, the px(n + 72) buffer contains the value of pitch 72 sample periods into the future. During every sample period,

$$\begin{aligned} X_{max} &= px(n+72) \quad \text{if } px(n+72) > X_{max} \\ &= X_{max} \quad \text{otherwise} \end{aligned}$$

and (18)

$$\begin{aligned} X_{min} &= px(n+72) \quad \text{if } px(n+72) < X_{min} \\ &= X_{min} \quad \text{otherwise} \end{aligned} \quad .$$

During the non-real pitch computation

$$\Delta_2 = \Delta_1$$

$$\Delta_1 = |X_{max} - X_{max}| \qquad (19)$$

Also, X_{max} is reset to 00...0 and X_{min} to 0 1 1 1 1...1. In this way during a current P_{TR} computation Δ_1 gives maximum deviation of pitch samples 10 msec into the future, whereas Δ_2 provides the same information 10 msec into the past. The greater of the two values Δ_1 and Δ_2 is now compared to a threshold level (best value here is found empirically). If this level is not exceeded P_{TR} is made equal to the Hiss word, otherwise Buzz computation is pursued.

A flow graph of non-real time pitch extraction is shown in Fig. 2.

2.3 Synthesis

The purpose of this operation is to reconstruct synthetic speech from the received pitch and spectrum data.

The received data contains two 7-bit words representing pitch and 35 bits representing spectrum energy approximately every 20 msec. More information about the format and timing of the 2400 bit/sec data stream transmitted from analyzer/pitch detector to synthesizer over a digitial transmission link will be given in Section 2.4. The pitch words are used to control the period of a digital-equivalent impulse train. If Hiss is received, +1, -1 impulse pairs are output with period selected by a random number generator. These impulses are used as inputs into a cascade of (usually) 3 or 4 second-order difference equations whose aim is to approximate the vocal tract impulse response during vowel production. Functions of this kind are referred to as formant filters.

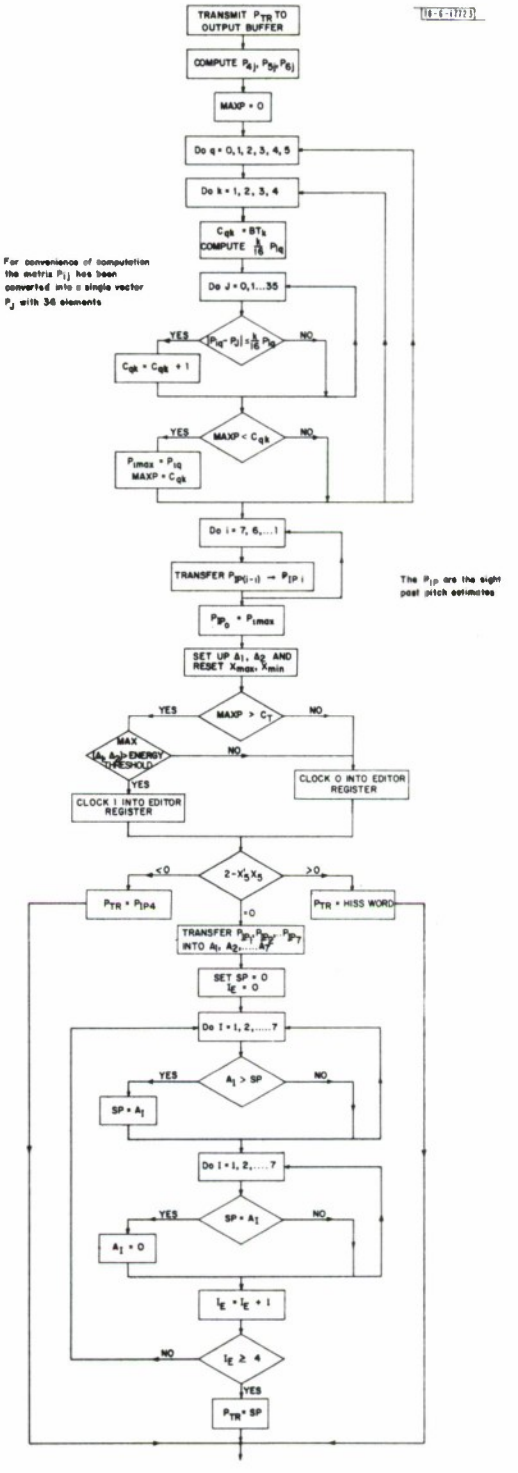


Fig. 2. Non-real pitch computation flow graph.

They also have a "smearing out" effect on the sharp impulses used as inputs. For more information see Ref. 2. The computational form of the formant filters is identical to that given in Eq. (7). The constants, however are different. The original k_3 is now designated k_5^i and varies for each of the formant sections. The original k_4 is changed to k_6 and is the same for all sections.

The formant filter output is the input into 49 band-pass poles identical to the ones used in the analysis section. Their outputs are weighed and summed also in the same way; however, no moduli (envelopes) are taken. Let the 16 results be denoted by $A_i(n)$. The spectrum information extracted from the received data is decoded and converted into a 16-element vector. Each entry is passed through a third order low-pass filter of the type described in the analysis (see Figs. 6 and 7) giving 16 outputs $B_i(n)$. The $A_i(n)$ are used to modulate the $B_i(n)$ to give the 16 $E_i(n)$ according to the following scheme:

$$\begin{aligned} E_i(n) &= \frac{1}{16} B_i(n) \quad \text{if} \quad A_i(n) \geq 0 \\ &= -\frac{1}{16} B_i(n) \quad \text{if} \quad A_i(n) < 0 \end{aligned} \Bigg\} \text{for } i \text{ odd}$$

$$\begin{aligned} &= -\frac{1}{16} B_i(n) \quad \text{if} \quad A_i(n) \geq 0 \\ &= \frac{1}{16} B_i(n) \quad \text{if} \quad A_i(n) < 0 \end{aligned} \Bigg\} \text{for } i \text{ even} \qquad (20)$$

Another set of band-pass poles, again with the same coefficients as in the analysis is used but now in the reverse fashion. Thus,

For Band-Pass Pole	Input
1	$\frac{1}{2} E_1$
2	$- E_1$
3	$E_1 + \frac{1}{2} E_2$
4	$- \frac{1}{2} E_1 - E_2$
5	$E_2 + \frac{1}{2} E_3$

etc.

With these inputs, the first 25 band-pass pole outputs $G_i(n)$ are computed. It will be noted from Table 2.1 that the resonant frequency of the 25th pole pair is 1601 Hz, and that of the 26th 1681 Hz. Also, as will be discussed in more detail later, the pre-sample input and post-sample output analog filters cut off at 3.3 kHz with very sharp rejection. Thus, the second harmonic of the 26th band-pass pole with a center frequency 3362 Hz, lies in the rejection band of the analog output filter. Therefore, the 26th and up to the 49th band-pass poles contribute nothing to the synthesis that is not already provided by the post-sample output analog filter. These poles may therefore be neglected.

The final output is then

$$x'(n) = \sum_{i=1}^{25} 2\, G_i(n) + \sum_{i=1}^{16} E_i(n) \qquad (21)$$

x'(n) is converted into an analog signal via a D/A and fed into the voice out-output.

2.4 Frame Data Encoding and Decoding

The data extracted from the analysis and pitch computation sections is

further condensed and transmitted in 49-bit frames. Since the data rate is 2.4 kb/sec a frame duration is (49/2400 = 20.41667 msec. There are two 7-bit pitch words per frame, representing a pitch computation every half frame, i.e., every 10.20833 msec. The remaining 35 bits are used to represent the energy in the 16-analysis channels. As can be seen from Table 2.3, 7-bit pitch words are adequate to cover the required pitch range. Therefore, no further coding is needed here. However, 35 bits every 20 msec is not adequate to transmit the spectrum information. The analysis section supplies 16 words, 16 bits each, a total of 256 bits which now have to be compressed into 35 bits with a minimum sacrifice in information content. Even if 4 or 5 bits per sample with logarithmic quantization was used (a reasonable approach), we would still have 64 or 84 bits per frame, too many by about a factor or 2. We must take advantage of the high degree of correlation observed to exist between speech spectral samples. After a great deal of work, a technique was found some years ago which removes some redundancy and is easy to implement.[3] The 16 spectrum samples $S_i(n)$ say can be thought of as elements of a 16-dimensional vector S_i. There exists a linear transformation, the Hadamard matrix, with elements limited to +1 and -1, which transforms S_i into S_i', i.e.,

$$S_i' = [H] \cdot S_i \tag{22}$$

such that the elements of S_i' are arranged by decreasing order of information content.

As noted above, logarithmic rather than linear quantization can be used to provide maximum dynamic range for a given number of bits. This follows from the empirical observation that speech perception of the human ear is roughly logarithmic in nature.

Thus, the encoding of spectral samples consists of taking the logarithm of the spectral envelopes, transforming them using the Hadamard matrix, and transmitting 35 bits of information about the result with a maximum number of bits assigned to the first element and progressively fewer to the adjacent ones. The data converted to a 2.4 kb/sec serial stream is transmitted over the communications link. At the receive end, the 35-spectral bits are decoded and a receive spectral vector RS_i' is formed. Then

$$RS_i = [H]^{-1} * RS_i' \quad . \tag{23}$$

It can be shown that $[H]^{-1} = [H]/\ O(H)$, where O(H) denotes the order of the H matrix, in this instance 16. The antilogs of the elements of RS_i then give the spectrum inputs into the synthesis section.

Details of the encoding and decoding process are given next. The encode bit lineup, after log taking and Hadamard transformation is shown below:

			A_4	A_3	A_2	A_1	A_0	7-bit log					
BITS		16	15	14	13	12	11	10	9	8	7	6	5
Case A	A	0	x	x	x	x	x	n	n	n	n	n	n
B	s	s	x	x	x	x	n	n	n	n	n	n	
C	s	s	x	x	x	x	n	n	n	n	n	n	
D	s	s	x	x	x	x	n	n	n	n	n	n	
			A WORDS										

x = information bits
n = unused bits
s = sign bits

A 7-bit logarithm is used. After going through the Hadamard transformation the top line (Case A) may be multiplied by up to 16 (since the top row of the Hadamard matrix consists of all +1's). Hence, the indicated 7-bit log lineup. All other rows of the Hadamard matrix contain an equal number

of +1 and -1's, so the maximum shift cannot exceed three binary places. Bit 15 therefore is the effective sign bit for cases B, C, and D.

Case A: Bit 15 is a value bit, but the whole word is known to be positive. Since this case corresponds to $S'_o(n)$ all 5 bits A_0 through A_4 are used to describe it.

Case B: If the A word is positive and less than 7, the sign bit (0 in this case) plus the 3 bracketed bits are used, giving the representation 0 A_2 A_1 A_0. If A is positive, but greater than 7, 0111, i.e., 7 is used. For negative numbers if larger than -8 the 4-bit representation is used as it stands. For A less than -8, just -8, i.e., 1000 is chosen.

Case C: The same truncation principle as above is used here, except now only on two bits A_4 (the sign bit) and A_0. Thus only numbers between 01 and 10 are generated.

Case D: Only the sign bit A_4 in transmitted for this set.

The 16 elements of S'_i are then assigned to the following groups:

TABLE 2.4

Element of S'_i	Case	No. of Bits in Representation	Total No. of Bits Used	S_{21}	S_{11}
$S'_o(n)$	A	5	5	0	0
$S'_1(n), S'_2(n), S'_3(n), S'_4(n)$	B	4	16	0	1
$S'_5(n), S'_6(n), S'_7(n)$	C	2	6	1	0
$S'_8(n) \ldots\ldots S'_{15}(n)$	D	1	8	1	1
			Total 35 bits		

The values S_{21} and S_{11} are merely used as a digital counter to distinguish the four different cases. Denoting the encoded outputs by O_i, it will be found that Boolean expressions may be derived for them in terms of the A_i and S_{11}, S_{21}. These are given below:

$$
\begin{aligned}
O_o &= A_o \cdot \overline{S}_{11} \cdot \overline{S}_{21} \\
O_1 &= \overline{S}_{21} \cdot \left\{ A_1 \cdot \overline{S}_{11} + S_{11} \cdot \left[\overline{A}_4 \cdot (A_o + A_3) + A_4 \cdot A_o \cdot A_3 \right] \right\} \\
O_2 &= \overline{S}_{21} \cdot \left\{ A_2 \cdot \overline{S}_{11} + S_{11} \cdot \left[\overline{A}_4 \cdot (A_1 + A_3) + A_4 \cdot A_1 \cdot A_3 \right] \right\} \\
O_3 &= \overline{S}_{21} \cdot \left\{ A_3 \cdot \overline{S}_{11} + S_{11} \cdot \left[\overline{A}_4 \cdot (A_2 + A_3) + A_4 \cdot A_2 \cdot A_3 \right] \right\} + \\
&\quad + S_{21} \cdot \overline{S}_{11} \left\{ \overline{A}_4 \cdot (A_3 + A_2 + A_1 + A_o) + A_4 \cdot A_3 \cdot A_2 \cdot A_1 \cdot A_o \right\} \\
O_4 &= A_5 .
\end{aligned}
\tag{24}
$$

The encoded words to be transmitted are shown in Table 2.5 by an X. N denotes "do not use."

TABLE 2.5

	Encode Outputs				
Case	O_4	O_3	O_2	O_1	O_o
A	X	X	X	X	X
B	X	X	X	X	N
C	X	X	N	N	N
D	X	N	N	N	N

At the receive end the serial data in is converted into an 8-bit word OP_i, and lined up as shown in Table 2.6

TABLE 2.6

Decode Inputs

Case	OP_7	OP_6	OP_5	OP_4	OP_3	OP_2	OP_1	OP_0
A	N	N	N	X	X	X	X	X
B	N	N	N	N	X	X	X	X
C	N	N	N	N	N	N	X	X
D	N	N	N	N	N	N	N	X

The function of the decoding is to produce a lineup of numbers such that dynamic ranges are maximized. This implies using the highest possible position, closest to the sign bit. At the same time, care must be taken to insure that the process to follow does not produce an overflow with these numbers. In this case, the following computation is the inverse Hadamard transformation. This implies, in the first row, a summation of all received values. If we now assume that OP_7 in Table 2.6 is equated with the sign bit, then OP_6 to OP_0 represents the first through seventh value bits. For Case A, all N's will be 0 since it is known that this number is positive. For the others no such guarantee exists, therefore the N's in these cases have to be assumed to be sign extensions. Two extreme cases arise, one when all values received are positive, the other when they are all negative. For the first case the largest value word A can have is 31. Word B can be 7; however, there are four of these contributing a total of 28. The three words C can contribute a maximum of three, and D can at most be 0 and it does not contribute. The total is 62. However, one must add eight to the word A. This follows from the fact that in 2's complement arithmetic truncation, whether the number is positive or negative, always adds an error in same direction. Since in the construction of word A only additions are used

(16 of them) the number will, on average, be smaller by half the order of the transformation, 8 in this case. So the total maximum positive sum for all received words, using the lineup in Table 2.6 is 72. Using a similar procedure for an all-negative numbers input (A can be only zero here) and adding 8 to A gives a minimum of -38. Thus 7 bits are sufficient to represent all eventualities and identification of OP_7 with the sign bit is exactly right.

III. COMPUTATIONAL FORMS AND STRATEGY

As indicated at the beginning, the machine we built was designed to implement the computational forms produced by the channel vocoder algorithms. It was of central importance that these forms be executed in a minimum of time and program storage. This section lists the required computational capabilities and discusses the most efficient way to achieve them.

Let us begin then with the 49 band-pass poles. The equations to be executed assume the form given in Eq. (1). There are 49 such poles in the analysis and 74 in the synthesis operation, a total of 123. If each line of program is executed in one basic cycle time T_c, and n lines of code are needed per band-pass pole, and execution time of $123n\ T_c$ will be required. Given the basic algorithm and a minimum value of T_c which the hardware is capable of, minimizing n without much increasing T_c is the only possible approach to minimizing total execution time. This implies parallel hardware. An array multiplier executing a complete multiply during a single T_c is therefore desirable. Current Shottky TTL 16 x 16 array multipliers can be made to give a product in about 100 nsec (typical). This would imply an upper limit of 10 MHz on the machine cycle frequency. Assuming for the moment that T_c = 100 nsec each unit increase in n

adds 12.3 μsec to the execution time. This represents nearly 10 percent of the total 140 μsec sample period available for the basic spectrum computation.

The machine must be able to execute the adds and subtracts in parallel with the multiplies. These operations must be capable of being expressed by the following functional equations.

(Operand 1) (operation) (Operand 2) → (Result Destination) (25)

This implies three simultaneous addresses. Assuming at least 8 bits per address, a minimum of 24 bits of address code will be needed.

The two multiplies already imply two cycle times. Since an addition has to be performed after the multiplies, a minimum of three T_c must be available. The aim therefore will be to build the machine such that no more than three-cycle times are needed for each second-order iteration of the type shown in Eq. (1). If a hard wired DO LOOP mechanism of the type discussed in the introduction is used, no additional increase in n due to looping will be required.

Equation (2) implies the need for a ROM in which the 49 K_{1i} are stored. Equation (3) requires a mapping of X into exp(-X). Again this may be done by a ROM. The next computational forms appear in Eq. (4): scaling by half and the taking of a modulus. The fastest way of achieving a scaling by one half is to use a multiplexer appropriately wired. The taking of the modulus (absolute value) may be done as follows: let us assume that an ALU is available which shifts the input through to the output on one of its channels when, for example, the code word $x_1\ x_2\ x_3$ is supplied and provides the inverse of the input when the code word is complemented, i.e., becomes $\bar{x}_1\ \bar{x}_2\ \bar{x}_3$. The modulus can then be implemented by connecting the x_i through exclusive OR's. The other free

input into the exclusive OR's is the sign bit of the input word. When this is positive, the sign bit is zero and the outputs of the exclusive OR's are $x_1\ x_2\ x_3$ implying a straight shift through. For negative inputs, the sign bit is one and the x_i will become inverted, i.e., $\bar{x}_1\ \bar{x}_2\ \bar{x}_3$, forcing the input to appear inverted on the output. This process takes a single T_c and can be initiated by a single command (MOD for example).

Equations (6) and (7) do not impose any new requirements. The scaling by 2 (incidentally this already occurs in Eq. (1)) can again be solved by the use of a multiplexer. It is hoped that Eq. (6) would take only two program steps since only one multiply and one subsequent add is needed.

Equation (8) requires a 3-way decision. This can be achieved in two conceptually quite different ways. The machine can be made to access one of three different addresses depending on whether $x_{n+1} > x_n$, $x_{n+1} = x_n$ or $x_{n+1} < x_n$ and write 1, Δ_n or -1 into Δ_{n+1}, respectively. This, in addition to the decision step, requires at least three additional program lines and an unconditional GO TO command. If pipe-lining is used the number of lines required may double or even treble. If, however, those lines can be used for other essential operations as well, this may be a very acceptable solution. An alternative method is to use an approach similar to that proposed for the modulus function. Thus, operation A or B is performed, depending on the outcome of a comparison between two numbers. Two-way comparisons are reasonably simple to implement. The three-way case becomes much more difficult. It may, of course, be hardware implemented. The additional complexity this introduces, however, was deemed too high a price, especially since two lines of software using two-way comparisons can be used to make a three-way decision. The following three single-line software capabilities

were developed:

1. COMP(A > B) IF MET (X OP_1 Y); IF NOT MET(X OP_2 Z)
2. COMP(A = B) IF MET (X OP_1 T); IF NOT MET (X OP_2 Z) (26)
3. COMP(A < B) IF MET (X OP_1 T); IF NOT MET (X OP_2 Z) .

Any of the X, Y or Z can be either of the two comparison words A or B. Operations OP_1, OP_2 may, but do not have to involve the two numbers. Thus, they can be an addition or subtraction involving say X and Z, alternatively, OP_1 could specify a shift through whilst OP_2 a negation of X. Such an operation could be used to implement the modulus function discussed above. The instruction for this would be:

COMP(X < 0) IF MET (SHIFT-X); IF NOT MET (SHIFT X) . (27)

Despite the introduction of these very powerful instructions, the modulus instruction was retained as a separate entity. This enhances programming flexibility.

The flow diagrams of Eq. (8) is shown in Fig. 3. It takes two dedicated lines of software to program. A write into Δ_{n+1} in a third line is also needed, however, this can be shared with other operations.

Equation (9) needs no new facilities. The MOD and COMP instructions will give single line software for their implementation. Similarly, computational forms needed for Eqs. (10) to (16) are already available. Equation (17) are Boolean expressions. One can either utilize the logic expression facilities of available ALUs or (if the number of lines involved is not too large) use one or more ROMs programmed to give the required expressions. In the present case 8 input lines and one output is required, making it ideal for a single ROM realization.

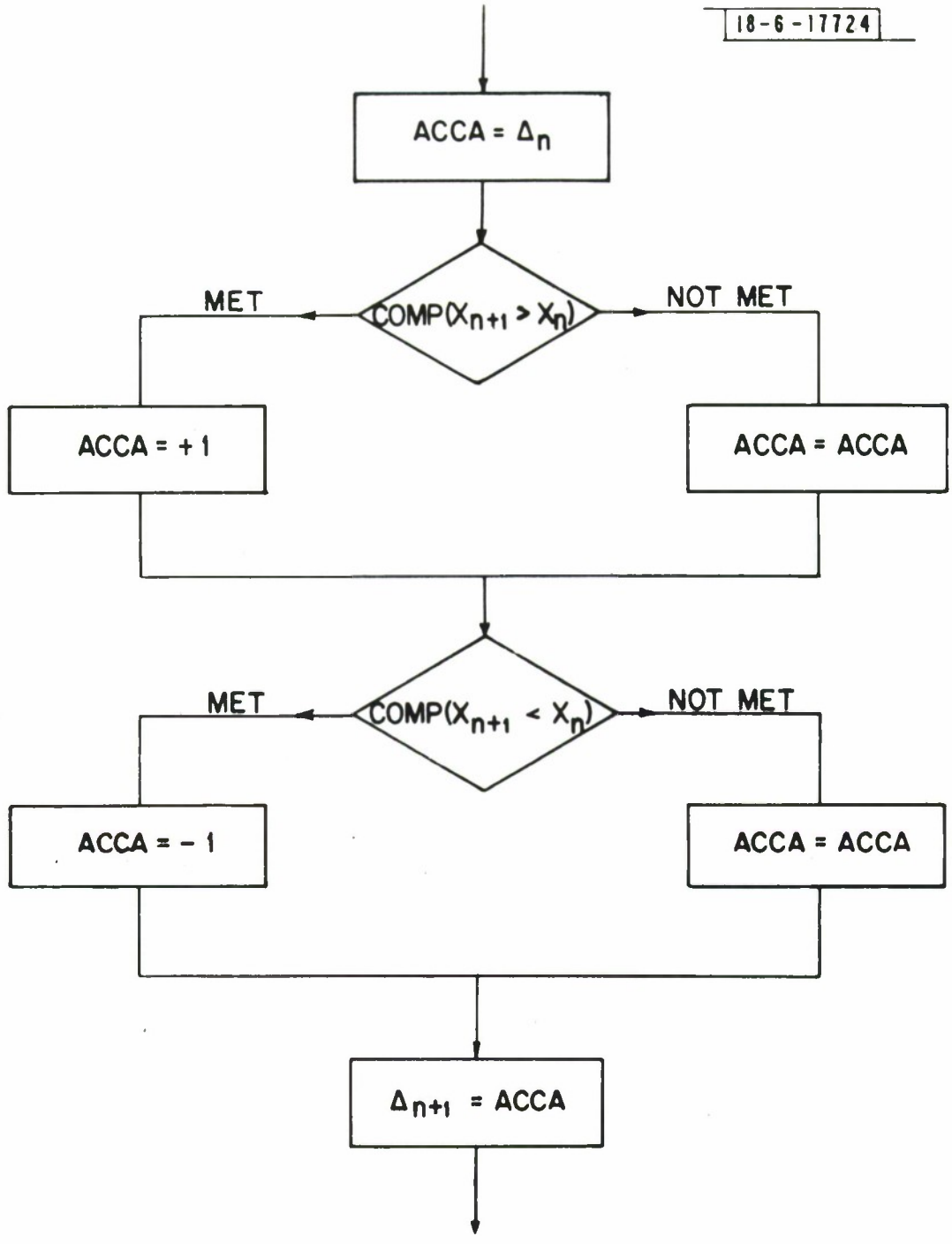


Fig. 3. Flow diagram for computation of $\Delta_{n+1} = +1$, Δ_n or -1 depending on whether $X_{n+1} > X_n$, $X_{n+1} = X_n$ or $X_{n+1} < X_n$.

Nothing new is required again for Eqs. (18) and (19). Equation (20) could be implemented in software. All the even $A_i(n)$ collected in an array $AE_i(n)$ say, used in a DO LOOP containing one COMP instruction and one multiply (by 1/16) would give the even $E_i(n)$. Similarly for the odd $E_i(n)$. This would take some six lines of coding. It appears, however, that with very little extra hardware a single line command to compute the $E_i(n)$ from the $A_i(n)$ can be obtained. This is based again on the principle of complementing commands for opposite functions. Depending on whether the most significant bit out of a storage register (designated ACCA for accumulator A) containing $A_i(n)$ is 0 on 1, a hard wired 1/16 $B_i(n)$ is either shifted through directly or inverted. This is done with or without an additional sign reversal under control of the least significant bit of the DO LOOP counter which is 0 for even and 1 for odd consecutive passes. This command is designated DOIB for Direct or Inverse through accumulator B. The hardware implementation only requires the addition of a few gates.

The next Equation, (21), again does not require any new features. Equation (22), however, is a matrix operation and would therefore imply a series of multiplications. Fortunately H is composed entirely of +1 and -1 and only additions and subtractions are needed. Also any one entry of H, h_{ij} is given by

$$h_{ij} = \Big(\big((i_o \cdot j_3) \oplus (i_1 \cdot j_2)\big) \oplus (i_2 \cdot j_1)\Big) \oplus (i_3 \cdot j_o) \quad . \quad (28)$$

This represents an 8 input (4i and 4j) values to one output (h_{ij}) transformation. So a single ROM can give all h_{ij} values. A command HAD inside DO LOOP takes over addition and subtraction under control of h_{ij} and gives a single line realization. Sixteen runs through this DO LOOP give one line of S. HAD will, of course, be also used for the inverse Hadamard transformation.

Finally, Eq. (24) is also realized using one ROM.

The discussion in this section should give some feel for the way in which the machine was designed. Those parts which may seem a little vague should become much clearer in the more thorough discussion of machine structure presented in the next sections.

IV. OVERALL MICROPROCESSOR STRUCTURE

The discussion from now on is concerned exclusively with the final machine design. This is the end product of many iterations of the type described in the previous sections.

A block schematic of the machine is shown in Fig. 4. Three-level pipelining is used. The program counter, together with its controls, program and address code ROMS make up the first time zone. The second contains program decoding, an arithmetic section devoted entirely to RAM memory addressing, hardware necessary to implement program operations such as the DO LOOP, GO TO etc., and ROMS containing constants like the K_{1i} of (1), exp(x), etc. and the Hadamand transformation ROM designated HAD ROM. The third zone contains arithmetic processing. Each zone is separated from the others by a layer of clocked buffers. Buffer transfers occur at a clock rate CLL equal to $1/T_c$. Thus, the hardware in each zone is on its own to complete all the operations required of it during one period of duration T_c. The final machine clock was chosen to be 8 MHz or less (this remark should become clearer after reading Section 5.14). Thus T_c is not less than 125 nsec. The remaining blocks in Fig. 4 are the input/output sections for voice sample and communications link data and the machine control logic. They are outside the pipelining structure since they have no functions which must be completed during one T_c. As such they are really not a part of

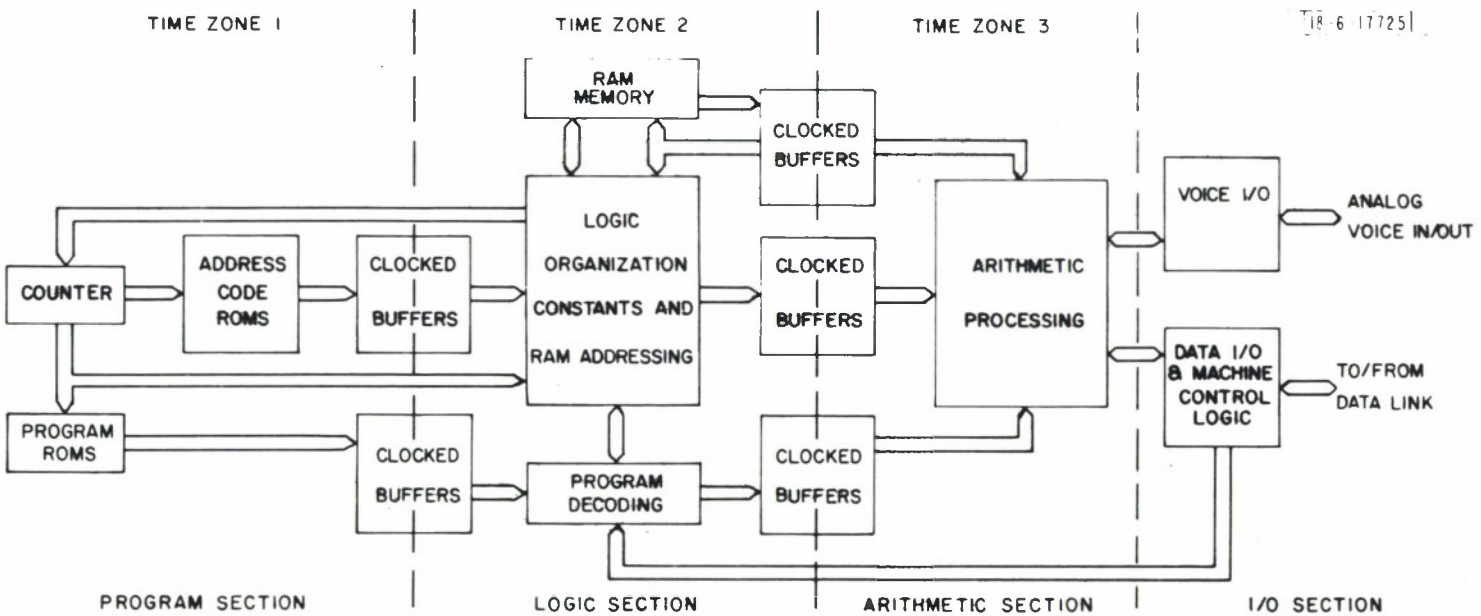


Fig. 4. Microprocessor block schematic.

the basic microprocessor.

Basically, pipelining implies that the arithmetic section executes the commands processed by the logic section in the prior T_c which appeared in the program $2T_c$'s ago. Schematically, this can be represented as follows:

TABLE 4.1

Number of T_c Intervals	Time Zone 1	Time Zone 2	Time Zone 3
1	PR Step N	---	---
2	PR Step N+1	Logic N	---
3	PR Step N+2	Logic N+1	Arithmetic N
4	PR Step N+3	Logic N+2	Arithmetic N+1
5	PR Step N+4	Logic N+3	Arithmetic N+2

"Logic N" denotes that part of the command in program step N which is executable in the logic section, while "arithmetic N" denotes the part which requires the arithmetic section to perform some task. Not every programming step requires action from both logic and arithmetic parts. For example, the DO LOOP command has an effect only on the logic section, while some add instructions will involve the arithmetic section alone. Generally, though, most programming steps will require some action in both sections. It is of course necessary at times to store in temporary memory (RAMs) results of arithmetic operations. Since two time zones are involved in this, care must be taken to give the appropriate commands at the proper times. This type of operation is indicated in Fig. 4 by the lines emanating from the arithmetic processing unit into time zone 2 to RAM memory and logic organization units. Similarly a DO LOOP requires control of the program stepping and an indication of program position. This is indicated

in Fig. 4 by the lines crossing from time zone 2 into 1 and vice versa. These and other similar cases will be discussed in detail in Section 5.

Table 4.1 should not be construed as an example of how the program is written. As a matter of fact each line of program crosses 3 time zones. Thus, a typical line would be:

PR STEP N; LOGIC N; ARITHMETIC N.

If an arithmetic operation result in line N say is to be processed in the logic section, the associated logic command will then appear in a line below, i.e., N+1. In this way the program step and line number becomes the same thing. Section 6 will give more detail on programming procedures.

V. DETAILED MICROPROCESSOR STRUCTURE

Structural details of the machine are given in Figs. 5, 6, 7, and 8. The discussion will proceed in terms of functions performed and resultant hardware realizations. In this way all interrelated blocks appear in the same description. Some units which perform more than a single function will appear in several places; however, their description will be from a different point of view.

5.1 Program Counter and ROMs

The ROMs used to store the program and addressing information are 256 x 4 field-programmable devices with a typical access time of 40 nsec and a guaranteed maximum of 60 nsec. Since T_c = 125 nsec, it was considered prudent to include in the first time zone only the program counter and the program and address code ROMs. The 256 counts needed are applied by two 4-bit counters. Addresses are supplied via line drivers (not shown) to the 8 program ROMs, the 6 address code ROMs, the Page ROM and the Shift ROM. The counter output is

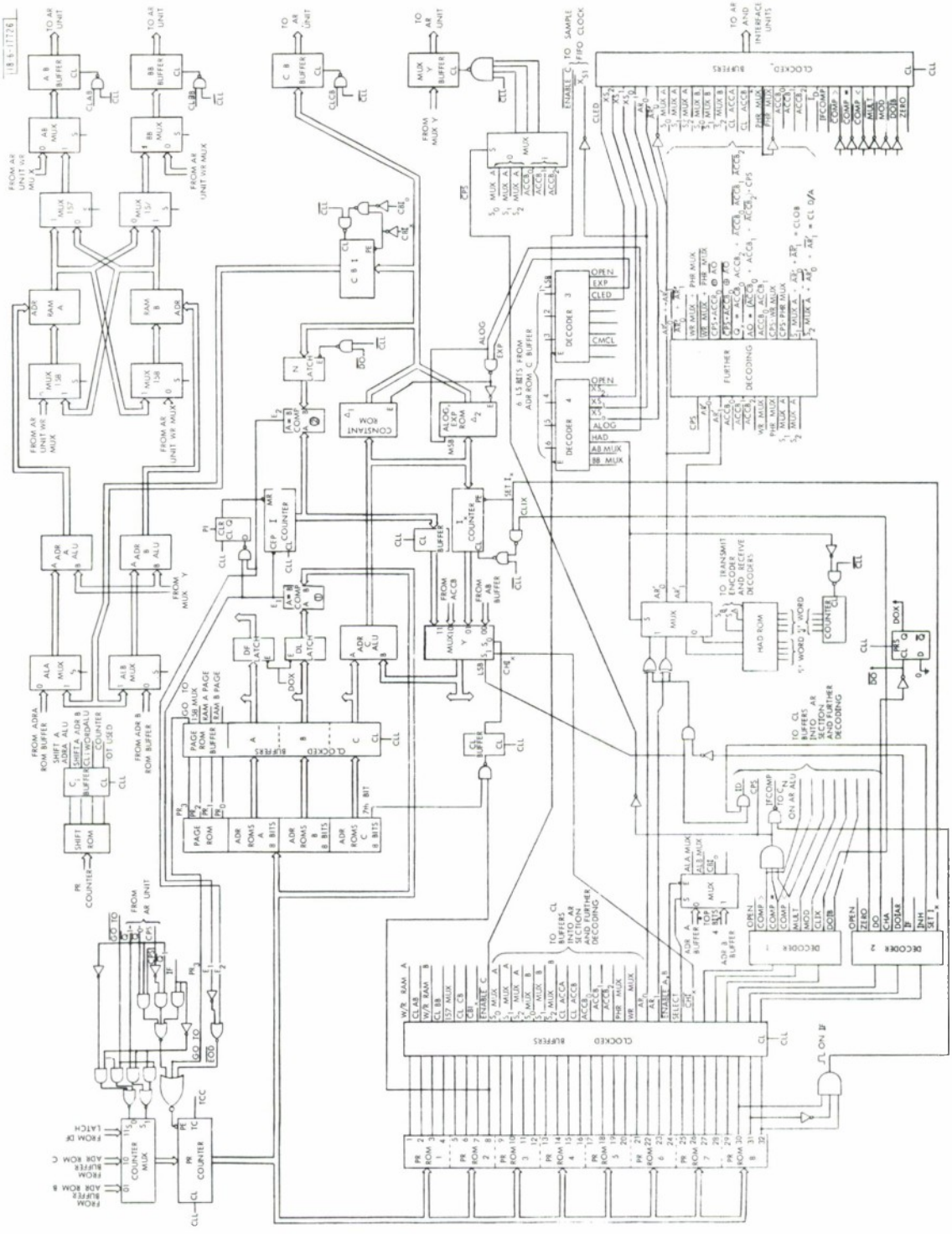


Fig.5. Vocoder microprocessor program and program decode sections.

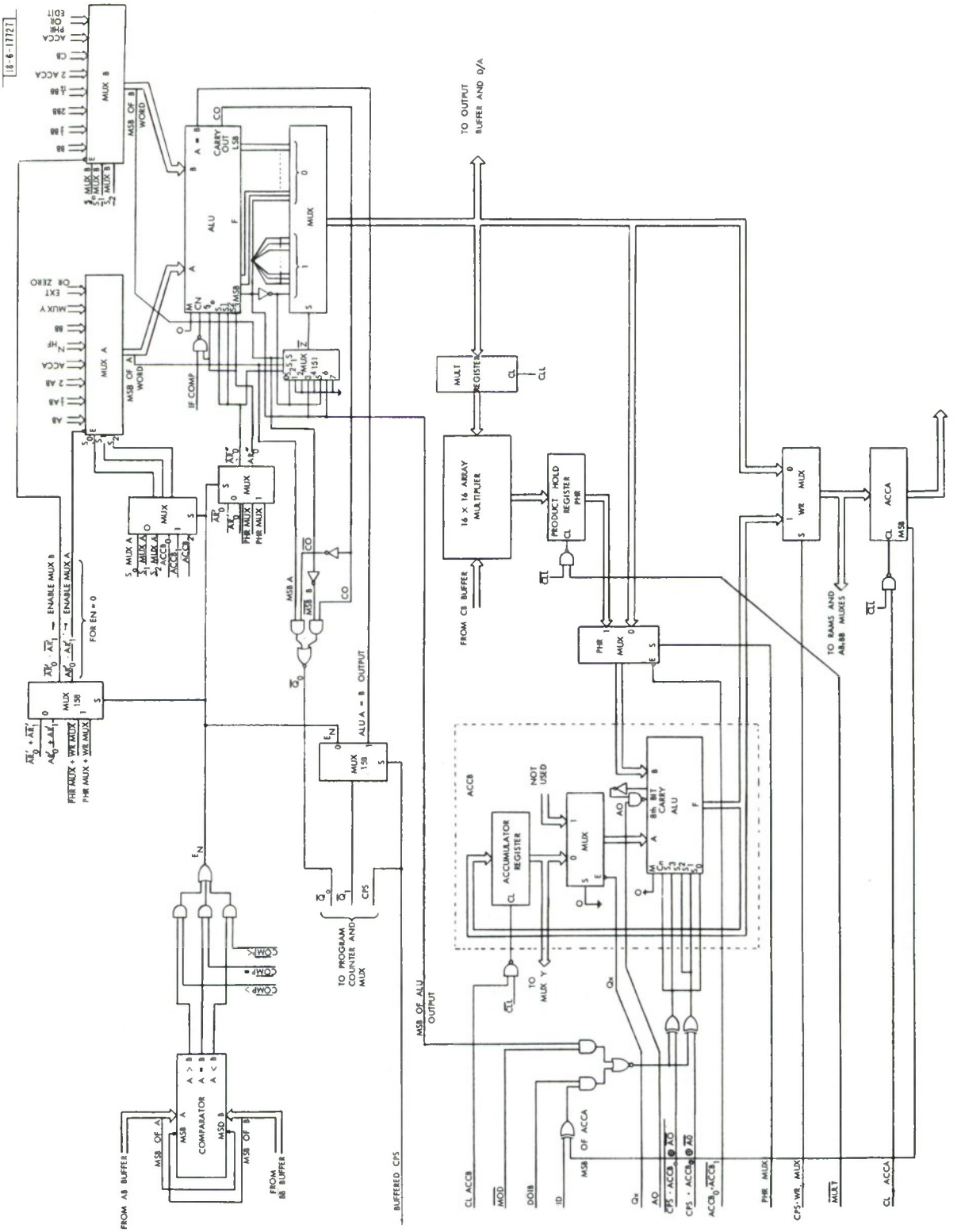


Fig. 6. Vocoder microprocessor arithmetic unit.

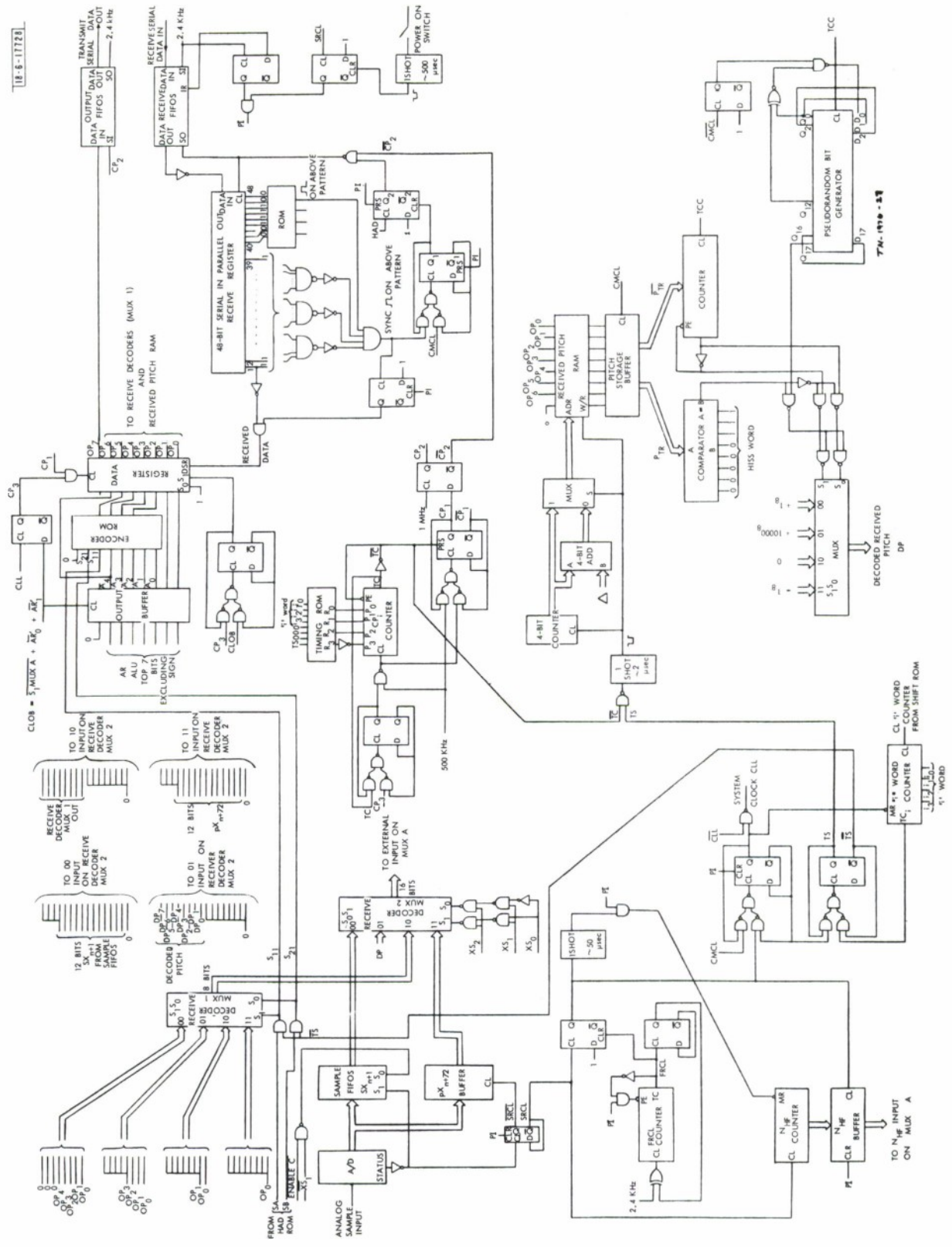


Fig. 7. Vocoder microprocessor interface units.

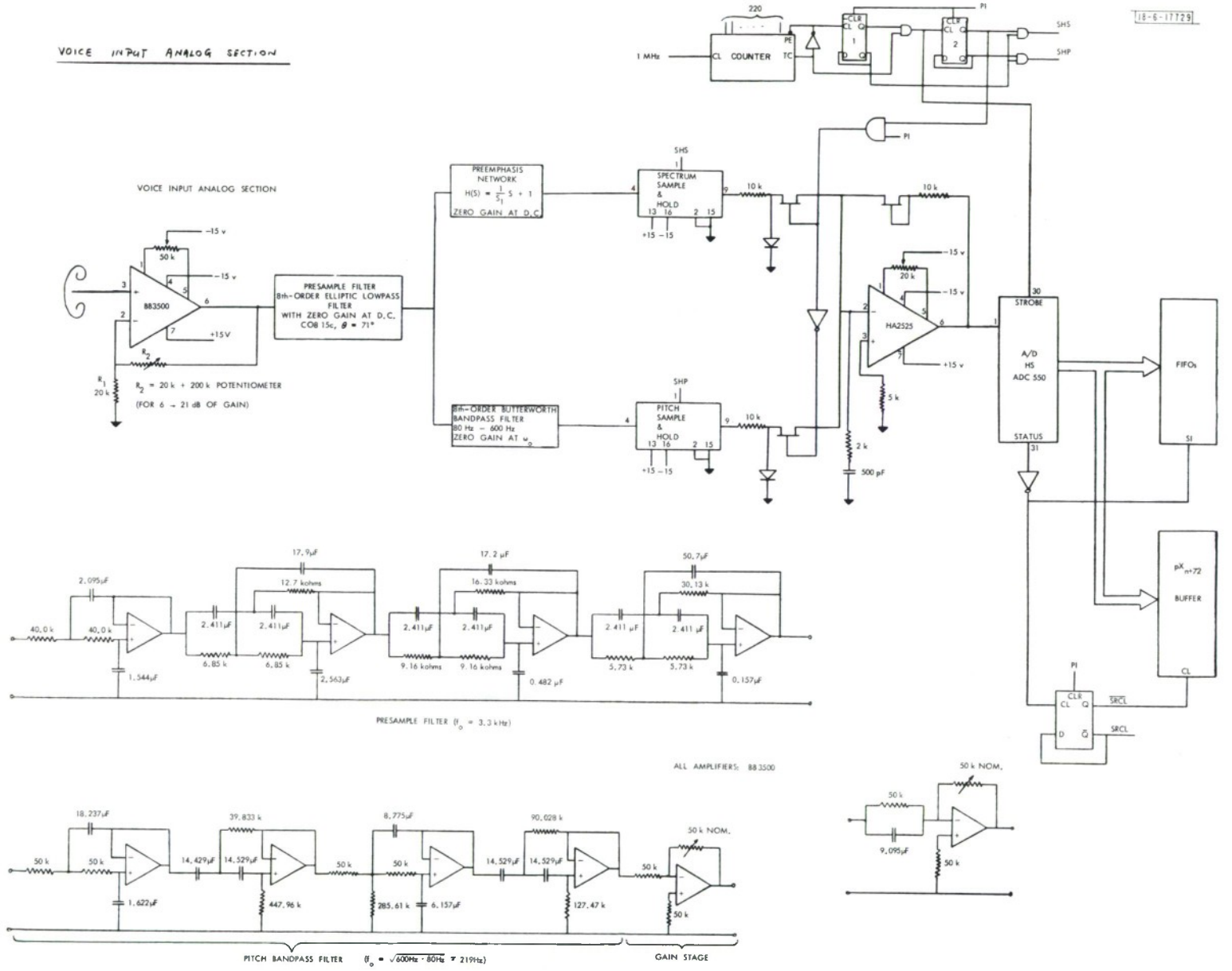


Fig. 8. Preemphasis network and gain stage (f_o = 350 MHz).

also used as the B input in comparator 1 for DO LOOP purposes (see Section 5.6). The counter clock CLL (See Setion 5.14) is the basic system clock, nominally 8 MHz. The all-full indicator on the counter designated TCC is used in pitch decoding (see Section 5.13). Presetting of the counter is under control of the DO LOOP, IF and GO TO instructions (see Sections 5.6 and 5.7). The PE (parallel enable) inputs are forced low at the appropriate point and an external input, supplied by a multiplexer is set into the counter after a low-high transition of the next CLL. The counter multiplexer consists of 4 dual 4 input units. Only 3 inputs are used.

Mux Select		
S_1	S_0	Inputs
0	0	Not used. Unconnected
0	1	8 bits from ADR ROM B BUFFER
1	0	8 bits from ADR ROM C BUFFER
1	1	8 bits from DF LATCH

The select lines are under control of the IF and GO TO commands (see Section 5.7).

The outputs of all ROMs described here go into clocked buffers (clocked by CLL) which separate the first from the second time zones. In addition program ROMs 2 line 4 in conjunction with program bit 7 of Address ROM C provide via a Nand gate and clocked buffer the S_0 input to Mux Y. Details are given in the next three sections. Also program ROM 8 bits 2, 3, and 5 provide an indication in the first time zone of the IF command. The reason for this is that COMP is executable in the arithmetic section while IF is performed a time zone earlier in the logic section. Therefore the COMP - IF operation (see Section 5.6 for

greater details) is coded in two consecutive program lines. On the other hand information whether it is to be just a COMP or a COMP-IF instruction is needed at the same COMP appears hence the jumping at a time zone to provide IF indication.

5.2 RAM Memory and Buffers

In Fig. 5 two sets of RAMs referred to as RAM A and RAM B are shown. Each consists of two 256 x 16 arrays arranged as page 0 and page 1 giving a total of 1024 16-bit words of random-access memory. The RAMs are fed by multiplexers. One of the inputs into these comes from the arithmetic section Write Mux output. This path enables the results of arithmetic processing to be stored in RAM memory. The other path comes from the other RAM array. It is possible therefore to shift data from RAM A into RAM B or vice versa without affecting any other part of the machine. The two-page controls for both RAMs are also on the Page ROM. An output multiplexer is also provided for each set of RAMs enabling either set to write into the AB or BB buffers. These buffers are individually clocked by commands Clock AB and Clock BB, respectively. Thus, they provide the dual functions of individual storage and time zone 2 to 3 isolation. Two further multiplexer arrays in front of the AB and BB buffers are also used. They are labeled AB Mux and BB Mux, respectively. Under program control, the Write multiplexer output in the arithmetic section ca be channeled through. This facility enables the AB and BB buffers to be used as temporary storage for arithmetic operations. This, as will be later shown, enhances the versatility of the COMP and COMP-IF instructions.

5.3 Address Processing

Address ROMs A and B in the first time zone and their clocked buffers in the second, provide conventional addressing for the two RAM arrays. The buffered 8-bit address is channeled through a multiplexer (ALA Mux and ALB Mux) and through an ALU (ADR A and ADR B ALU). The ALUs are conventionally in A plus B mode. The other input comes from MUX Y which in turn has the value of the DO LOOP counter I, on its output (unforced state of $S_0 = S_1 = 1$ on MUX Y). Outside a DO LOOP I = 0, thus the address reaching the RAMs will be the 8-bit word provided by the address ROMs. This arrangement also satisfies all addressing needs inside a conventional DO LOOP. The first address of a consecutively numbered array is provided by the addressing ROMs. The I counter then increments once every run through a LOOP, providing a unit increment for the accessed array address via MUX Y and the ADR ALU. If unconventional incrementing or random addressing is needed in a LOOP, the other MUX Y inputs can be used. As an example, accumulator B (ACCB) may be incremented by some fixed integer k stored in Accumulator A by the command

```
ACCB = ACCB + ACCA    .
```

once during DO LOOP execution. The contents of ACCB can then be channeled via MUX Y to the appropriate address ALU. Another possibility is to use the AB buffer. This provides the most versatile addressing but is restrictive in the sense that one RAM array becomes unavailable. But for example, a set of random numbers computed by the program and stored in say RAM A in consecutive locations, may be used in a subsequent DO LOOP to address RAM B via the AB buffer and MUX Y. Also the address ALU must be in shift A mode and the ALA MUX

in state 1, i.e., it channels the contents of CBI to RAM A. In this way the RAM A address can be incremented consecutively, whilst RAM B is under AB buffer addressing control. CBI is a counter which may be incremented by unity on command CBI_0; alternatively a word appearing on the output of the Δ ROMs can be written into it. Unit incrementing of CBI was already described in conjunction with use of the AB buffer for addressing. The presetability of CBI provides another way of random addressing, this time of both RAMs (if required); however, the random sequence has to be programmed into the Δ ROM. The address C ALU has an unforced state of A plus B. Thus a base address provided by ADR ROM C can be incremented in a DO LOOP via MUX Y. Thse addresses are supplied to the Δ_1 ROM which in turn contains a preprogrammed random sequence of addresses in a consecutive array. These are then clocked into CBI and are available to RAM A and B, via inputs 1 on the ALA and ALB multiplexers and ADR ALUs (usually the latter will be in Shift A mode).

Facilities are also provided for nested looping. Usually in such cases the innermost loop would be a conventional DO LOOP with incremental addressing provided by the I counter as described above. For the outer loops, however I = 0 and unit incrementing has to come from somewhere else. One of the functions of the I_x counter is to provide this facility. I_x may be set to some value specified by ADR ROM C and then incremented by unity at every pass through the loop. In this configuration MUX Y is in the 0 1 state. For more than two loop nestings, incrementing has to come from either ACCB or the AB buffer. In principle, since the AB buffer can be accessed from either RAM array and from the arithmetic sections, unconstrained addressing and incrementing for any degree

of loop nesting is possible. However, software has to be used to compute the addressing and their possible increments. I, I_x and to a less dedicated degree ACCB can provide hardware oriented nested DO LOOPING up to 3 deep. The I_x counter can also be used as intermediate address storage. The flexibility of the system is best illustrated by the fact that up to 4 levels of indirect addressing are possible. Thus, an address out of ADR ROM C can be modified by the ADR C ALU by adding some increment to it. This in turn produces an output from the Δ_1 ROM which can be modified a fourth time by the ADR A or B ALUs.

5.4 Decoders

The need for high-speed program throughput means that indivdual instructions must be very powerful and flexible. As a consequence, most commands have to be independent of all others. Unfortunately, this in turn generates a need for wide program words and an attendent increase in ROMs and allied hardware. Some effort was therefore put into collecting non-interfering commands into groups (of not more than 7). Each group is then accessed by a one of 8 decoder. Only one command of each group can be used at a time; however, a 7 to 3 independent line compression has been achieved. The eighth output of each decoder is not used. In this way if none of the particular group of commands is needed the eighth output is automatically accessed ensuring non-interference with the rest of the program. Of the 32 lines of program code the bottom 6 are used as inputs into two decoders (labeled DEC1 and DEC2). This produces an additional set of 14 commands. An additional two decoders (DEC3 and DEC4) are driven by the bottom 6 lines of ADR ROM C. Many of the envisioned operations required of the machine do not need ADR ROM C. With the exception of a multiply, all arithmetic operations are an example of this. As a consequence the demands imposed

on the ADR ROM C by the operations described to date will not be heavy, leaving it available for other things. Capitalizing on this fact, the two additional decoders ensure fuller use of ADR ROM C. The two groups of commands available from DEC3 and DEC4 are those least frequently used. A full listing is given in Table 5.1.

The 3 COMP instructions, already described in Section 3, certainly are mutually exclusive. MULT used to initialize a multiply, MOD to produce a modulus and DOIB for ACCB control, are all executable in the arithmetic section and do pose a certain constraint on programming flexibility. CLIX is used to update I_x by unity. It is a second time zone command and not a very frequent one. The command ZERO is used to disable the receive decoder multiplexer 2

TABLE 5.1

STATE	DEC 1	DEC 2	DEC 3	DEC 4
0 0 0	OPEN	OPEN	OPEN	OPEN
0 0 1	COMP >	ZERO	EXP	XS_2
0 1 0	COMP =	DO	CLED	XS_1
0 1 1	COMP <	CHA	Not Used	XS_0
1 0 0	MULT	DOIAR	"	ALOG
1 0 1	MOD	IF	"	HAD
1 1 0	CLIX	INH	CMCL	AB MUX
1 1 1	DOIB	SET I_x	Not Used	BB MUX

(see Fig. 7). The output of the multiplexer goes to channel 7 of the arithmetic ALU multiplexer A. Selection of this channel together with the ZERO command makes 0 available for arithmetic processing. DO is the DO LOOP command to be

described in greater detail in Section 5.6. CHA ensures that the ADR ROM C buffer output is shifted directly through the ADR C ALU. DOIAR shifts the arithmetic ALU commands through directly, or inverts them depending on whether the least significant bit of the MUX Y output is 0 or 1. IF (see also Section 3.7) initializes a conditional jump, INH is used in conjunction with DOIB if the effect of the least significant bit of MUX Y on the DIOB mode of operation is to be inhibited. This permits ACCB operations to come through as specified or inverted depending on whether the word contained in ACCA is positive or negative, respectively (for ACCB operations, see Section 5.8). Set I_x writes the 8-bit word appearing on the output of ADR C ALU into the I_x counter. The EXT command disables ROM Δ_1, enables ROM Δ_2 and makes the exponent data available (see also Section 5.5). CLED clocks the edit shift register (described in Section 2.2). CMCL is used once in the program. It therefore constitutes a clock indicating completion of a program run. It is used for pitch computation and timing (see also Section 5.14). XS_2, XS_1, and XS_0 all mutually exclusive, are used to select px(n + 72), px(n + 1) or sx(n + 1) (see Section 5.12) and Decoded Pitch, respectively from Receive Decoder MUX 2. ALOG is the antilog command, it, like EXP, disables ROM Δ_1 and makes available antilog data (see Section 5.5). HAD is the Hadamard matrix transformation command. It channels a multiplexer to the HAD ROM input (see Section 5.11). The 8 inputs into which are divided into two four-bit words, the 'i' word, which is updated once every run through the program and the 'j' word, which is updated every time HAD appears. A DO LOOP containing 16 HAD instructions therefore varies j between 0 and 15. This is done for every value of i (0 to 15). The output of the HAD ROM is the appropriate add

or subtract command specified in the Hadamard transformation (discussed in Sections 2.4 and 3). The multiplexer channels the HAD ROM output through to the arithmetic ALU. AB MUX and BB MUX are the controls on the two multiplexers in front of AB and BB buffers, respectively. The natural unaccessed state of the decoder output is a 1. The RAM outputs therefore are channeled through these multiplexers. A 110 on DEC 4 inputs forces a 0 into line 6, i.e., AB MUX and this selects the Write MUX output through the AB multiplexer. The same happens with the BB multiplexer when 111 is applied on DEC 4.

Decoders 1 and 2 are permanently enabled. This cannot be permitted for the 3rd and 4th decoders since otherwise conventional use of ADR ROM C might be translated into an unwanted command out of these devices. An enable line designated ENABLE C is provided. Every command in decoders 3 and 4 must be accompanied by an ENABLE C. The 8th line out of the program clocked buffers in Fig. 5 is shown as $\overline{\text{ENABLE C}}$. The bar indicates that this line is nominally high and becomes 0 only when accessed, this being the inverse of conventional usage. A zero on the decoders enables them.

5.5 ALOG and Exp Routines

The computation of logarithms is done by a software subroutine. Sixteen-bit input numbers are mapped into all 7-bit words (0 to 127) in a one-to-one transformation such that adjacent input words are offset from each other by a constant on a logarithmic scale. The inverse transformation, referred to as the antilog (hence, the ALOG label) is programmed into ROM Δ_2. Thus for input words m = 0 through 127, 128 16-bit output words are available. A 60-dB dynamic range was assumed to be very adequate. This led to the choice of 0.5 dB steps

and a consequent dynamic range of 64 dB. Thus, generally:

$$20 \log_{10} \left[\frac{\text{ALOG } (m + 1)}{\text{ALOG } (m)}\right] = 0.5 \tag{29}$$

giving

$$\text{ALOG } (m) = \text{ALOG } (m + 1) \times 10^{-\frac{1}{40}} \quad . \tag{30}$$

The largest number representation, assigned to ALOG (128), is $1 - 2^{-15}$. It was assumed that for the case at hand this was sufficiently close to unity to define:

$$\text{ALOG } (127) = 10^{-\frac{1}{40}} = 0.9440608763$$

as a result the general expression for ALOG (m) is given by

$$\text{ALOG } (m) = 10^{-\left(\frac{128 - m}{40}\right)} \quad . \tag{31}$$

This expression was used to evaluate the ALOG transformation table.

Let us assume that X is a number whose logarithm is required. The procedure is based on the following algorithm:

$$\left.\begin{array}{lll} \text{IF } \left[X - \text{ALOG } (m)\right] > 0 & m = m + \frac{m}{2} \\ = 0 & m = m \\ < 0 & m = m - \frac{m}{2} \end{array}\right\} . \tag{32}$$

The evaluation, irrespective of X, is always started with m = 64. It takes exactly 7 iterations at which point the latest value of m is the required LOG (X).

The ALOG command, besides disabling ROM Δ_1 and enabling ROM Δ_2 also puts a zero into the most significant bit on the input of ROM Δ_2. This ensures that only bits 0 to 127 are available as ROM address inputs when the log-taking routine is in use. When ALOG is not activated, the MSB is held at a high with only addresses

128 to 255 available as inputs. This part is reserved for the EXP (m) mapping. EXP out of decoder 3 enables ROM Δ_2 and makes available at the ROM output the following transformation, given an input address m:

$$\text{EXP (m)} = e^{-\frac{\ell n 2}{m - 128}} \quad . \tag{33}$$

Thus for $128 \leq m \leq 255$, $\quad 0 \leq \text{EXP (m)} \leq 0.994557$

This table is used exclusively for the exponential rundown procedure during real time pitch computation, as described in Section 2.1.

5.6 DO LOOP

Dedicated DO LOOP operation has already been suggested in the introduction as a means of speeding up of the program throughput. The command is specified by

$$\text{DO, DF, DL, N} \tag{34}$$

where DF is the first line of the loop, DL the last and N the number of times the loop is to be executed. Usually DF is the line immediately following the DO instruction, however this need not be so. DL is stored in ADR ROM B. The output of the DL latch goes to one of the two inputs on a comparator the other being the current program counter setting. The DO therefore is a first-time zone operation. Comparator 1 gives an output the moment the counter setting reaches the value DL. This output, denoted E_1 in Fig. 5, performs two tasks. It clocks the I counter incrementing I by unity and, if the output of the comparator 2 (i.e., E_2) is low, it parallel enables the program counter ensuring that the next value the counter will assume is that available from the counter multiplexer. The unforced state of the multiplexer is 1 1 which channels the content of DF. So, as long as there is a parallel enable during program step DL, the

next setting is step DF. This is essentially the looping mechanism. Comparator 2 has the current value of I as one of its inputs and the output of the N latch as the other. When equality of I and N is reached, the DO LOOP has been executed the required N times. E_2 goes high inhibiting the parallel enable action of E_1 the counter therefore continues sequentially which has the effect of exiting from the DO LOOP. Two-level indirect addressing permits N to be a variable. For example, an array of N values N_j say is stored in consecutive address locations $A_j(N_j)$, (j = 0, 1,....n). Let us assume the DO is nested in an outer one controlled by I_x. If for I_x = 0 ADR ROM C supplies $A_o(N_o)$, N_o will be used for the inner DO. The next pass through the outer DO increments I_x by unity giving the new address for ROM Δ_2 as $A_o(N_o) + 1 = A_1(N_1)$ as a result this time the inner DO is executed N_1 times. Continuing in this way access is made to all N_i, making it possible to individually control the number of executions of the innermost DO LOOP.

5.7 IF, COMP, COMP-IF and GO TO Instructions

The IF instructions produces a conditional jump. The decision may be up to 3-way depending on whether two numbers A and B say satisfy the A > B, A = B or A < B conditions. The indicators used are the carry-out line from the ALU, C_o, the A = B output and the most significant bits of the A and B words, MA and MB, respectively. Table 5.2 lists all possible states. This table was used to implement a hard-wired IF instruction.

Justification for the COMP instruction and a short description of its operation was already provided in Section III and Eqs. (26) and (27). The two quantities to be compared are the contents of the AB and BB buffers, respectively.

TABLE 5.2

State	MA	MA	ALU C_o	ALU A = B (Q_1)	RESULT	
1	0	0	0	0	A positive; B positive	A > B
	0	1	0	0	A positive; B negative	
	1	1	1	0	A negative; B negative	
2	0	0	1	1	A positive; B positive	A = B
	1	1	1	1	A negative; B negative	
3	0	0	1	0	A positive; A positive	A < B
	1	0	0	0	A negative; B positive	
	1	1	0	0	A negative; B negative	

They will be referred to as AB and BB in the subsequent discussion. The comparator used in this operation is shown in the top left corner of Fig. 6. Since AB and BB may be positive or negative independently, the comparator must be able to handle 2's complement numbers. The outputs from the comparator are combined with the COMP lines to give

$$E_N = \overline{\text{COMP}} > \cdot (A > B) + \overline{\text{COMP}} = \cdot (A = B) + \overline{\text{COMP}} < \cdot (A < B) \quad . \qquad (35)$$

The COMP lines are active low whereas the comparator outputs active high. Therefore, a comparison if met makes $E_N = 1$, if not met $E_N = 0$. E_N controls the select lines of 3 other multiplexers. During a COMP operation the ACCB control lines are not used for Accumulator B control. This accumulator becomes unavailable for other than shift through operations. Similarly, the PHR multiplexer selects lines PHR MUX and write multiplexer control WR MUX are no longer

available for their prime duties. The two multiplexers are connected in the 0 channel mode, i.e., select AR ALU output. The 3 control lines $ACCB_0$, $ACCB_1$, and $ACCB_2$ are one of the two sets of 3 inputs (channel 1) into a multiplexer selected by E_N. The zero channel is S_0 MUXA, S_1 MUXA and S_2 MUXA. The outputs of this multiplexer drive the three select lines of MUXA. Thus, when a comparison is not made or made but not met the MUXA outputs are selected by the control lines originally intended for the job, i.e., S_0, S_1 and S_2 MUXA. If, however, a COMP instruction is met $E_N = 1$ forces $ACCB_0$, $ACCB_1$ and $ACCB_2$ to take over control of MUXA. Thie mechanism permits selection of different MUXA inputs as the A word of the ALU depending on whether a comparison is met or not met. The word to be selected if the comparison is met is specified by $ACCB_0$, $ACCB_1$ and $ACCB_2$ if it is not met S_0 MUXA, S_1 MUXA, and S_2 MUXA does the selection. Similarly, the lines controlling the arithmetic operation are either the originally intended ones AR_0 and AR_1 if no comparison is made or one made and not met or their place is taken by PHR MUX and WR MUX, respectively, when $E_N = 1$. This channeling is effected via two further multiplexers shown in Fig. 6. Since the two sets of controls are totally separated, completely independent arithmetic operations may be performed on two ALU inputs depending on the outcome of a comparison between AB and BB. The following are three examples selected to illustrate the use of the COMP instructions. The notation will be described in greater detail in Section 6.

1. COMP> ($\frac{1}{2}$ AB, 2AB) (SFTA, SFTA)

2. COMP = (ACCA, MUXY) (SFTA, +) CB → ACCA

3. COMP< (BB, AB) (-, SFTB) PHR → ACCB

1. This means: if AB > BB shift $\frac{1}{2}$ AB through to the natural output of the arithmetic section (which is the WR MUX output). If the comparison is not met shift through 2AB.

2. If AB = BB clock the content of ACCA back into itself. This is equivalent to saying that ACCA should not be disturbed. If the comparison is not met the content of MUXY incremented by the content of buffer CB and the results clocked into ACCA.

3. For AB < BB clock BB - PHR into ACCB. If AB $\geq$ BB place only the PHR into ACCB. The second entry in the first bracket, i.e., AB is in this case just a dummy since the ALU selects only channel B anyway. Any one of the MUXA input could be used here, however, AB with a command of 0 0 0 is the most convenient. Although internal ACCB operations are not possible during a COMP, clocking into ACCB does not require $ACCB_0$, $ACCB_1$, or $ACCB_2$ and is therefore permissible.

The COMP instruction may also be used as the decision stage for a conditional jump. If in the line following a COMP instruction an IF appears, the program counter will go to a location specified in ADR ROMC or continue in its natural sequence depending on whether the COMP was or was not met, respectively. The operations specified by the COMP itself have no bearing on the conditional jump. Table 5.3 combines all the relevant states and their consequences for both kinds of conditional jumps.

TABLE 5.3

No.	IF	CPS *	(A = B or E_N) Q_1	Q_o	Meaning		Action
1	0	1	0	0	AB > BB	conventional IF	No P.E.
2	0	1	0	1	AB < BB		P.E. to ADR ROMB
3	0	1	1	X	AB = BB		P.E. to ADR ROMC
4	0	0	1	X	Met	COMP IF	P.E. to ADR ROMC
5	0	0	0	X	Not met		No P.E.
6	1	X	X	X	No IF		No P.E.

P.E. stands for parallel enable. X means a don't care state. The action to be taken in line 2 for example means: set the program counter to go to the steps specified in the ADR ROM B on the next rising clock edge. The parallel enable on the program counter is active low, therefore in conjunction with Table 5.3.

$$\text{P.E.} = \text{IF} + \overline{\text{CPS}} \cdot \overline{Q}_1 + \text{CPS} \cdot \overline{Q}_o \cdot \overline{Q}_1 \quad . \tag{36}$$

The GO TO command is a straightforward unconditional jump. It comes from the top line of the page ROM and goes to the P.E. input on the program counter. The address to which the program is to go is in ADR ROMC. The complete P.E. on the program counter taking Eq. (36), GO TO and EOD** into account is

$$\text{P.E.} = (\text{IF} + \overline{\text{CPS}} \cdot \overline{Q}_1 + \text{CPS} \cdot \overline{Q}_o \cdot \overline{Q}_1) \cdot \text{GO TO} \cdot \text{EOD} \quad . \tag{37}$$

---

* CPS = COMP > · COMP = · COMP <.

** EOD = END of DO output. This goes high at the end of a DO LOOP.

The relevant data controlling the select lines on the program counter multiplexers is collected in Table 5.4.

TABLE 5.4

IF	GOTO	Q_1	Q_o	S_1	S_o	Select	Meaning
0	0	X	X	X	X	X	Does not occur
0	1	0	1	0	1	ADR ROM B	AB < BB in conventional IF
0	1	1	X	1	0	ADR ROM C	AB > BB in conventional IF
1	0	X	X	1	0	ADR ROM C	Unconditional jump
1	1	X	X	1	1	DF LATCH	Static condition; used in DO

From the above

$$S_o = (IF + \overline{Q}_1) \cdot GOTO$$
$$S_1 = IF + Q_1 \cdot GOTO \quad . \tag{38}$$

5.8 Arithmetic ALU and Accumulator B

The inputs into the arithmetic ALU are fed by two 8-input multiplexers referred to as MUXA and MUXB (see Fig. 6). Selection of outputs is conventinally handled by six dedicated lines; S_0 MUXA, S_1 MUXA and S_2 MUXA for multiplexer A and S_0 MUXB, S_1 MUXB and S_2 MUXB for the other. An exception to this rule for multiplexer A alone occurs only during the COMP instruction described in the previous section. Table 5.5 lists all available inputs and their locations.

TABLE 5.5

S_o	S_1	S_2	MUX A	MUX B
0	0	0	AB	BB
0	0	1	$\frac{1}{2}$ AB	$\frac{1}{2}$ BB
0	1	0	2 AB	2 BB
0	1	1	ACCA	$\frac{1}{16}$ BB
1	0	0	N_{HF}	CB
1	1	0	MUX Y	ACCA
1	1	1	EXT or ZERO	PHR or EDIT

The first 3 lines in each multiplexer are the AB and BB buffer outputs direct, scaled by $\frac{1}{2}$ and 2. This is the fastest way of achieving scaling. MUX B also provides a $\frac{1}{16}$ scaling. This is used for quick implementations of Eq. (20) and also in non-real pitch evaluation of the window function Eq. (13). ACCA is available in both MUXES. The reason for this is that the ALU can only provide A minus B. In this way ACCA minus X or X minus ACCA are both possible. For the same reason BB appears in MUX A as well. 2 ACCA was found very useful for a number of operations. MUX Y is multiplexer Y output. This permits constants, chiefly used for addressing to be brought into the arithmetic section. Since MUX Y is in the second time zone, its output has to be buffered as shown in Fig. 5. CB permits constants from the ΔROM's to be available in the arithmetic section. The last setting in the MUX A column contains both EXT and ZERO since this address makes the output of receive decoder MUX 2 available. Whether what comes in on lines A is one of the 4 inputs into the multiplexer labeled EXT or ZERO depends on further commands out of decoder 4 for EXT and decoder 2 for zero. Position 7 in MUXB is taken up by the output of the product hold register labeled

PHR, also the two least significant lines are connected through a two input multiplexer to X'5 X5 (described in Section 2.2 when the editing procedure was explained) and the two least significant bits of the PHR output. Edit, which happens only once during the program, is controlled by an unused setting of the ACCB controls, i.e., $ACCB_0 = 0$, $ACCB_1 = 1$, $ACCB_2 = 1$. Under this control (with 1 1 1 on MUXB select) the product hold register is cleared to zero and the two lines X'_5 X_5 are channeled through.

The arithmetic ALU is required to perform only 4 different operations: shift A, A minus B, A plus B and shift B. Two dedicated control lines AR_0 and AR_1 are used. The control on the ALU packages themselves (the 74S181's) requires 6 lines. This and other relevant data is summarized in Table 5.6.

TABLE 5.6

AR_0	AR_1	Meaning	ALU Controls						Enable	
			M	C_n	S_3	S_2	S_1	S_0	MUXA	MUXB
0	0	SHIFT A	0	0	0	1	1	0	0	1
0	1	A - B	0	0	0	1	1	0	0	0
1	0	A + B	0	1	1	0	0	1	0	0
1	1	SHIFT B	0	1	1	0	0	1	1	1

ALU controls for only A - B and A + B are used. For SHIFT A the B is set to zero. Similarly, when just B is required, the A + B command is used and A is forced to zero by disabling MUXA. It will be seen that the two sets of controls are duals of each other for 5 of the 6 controls and the first one, M is zero throughout. A very straightforward and consequently low delay realization is

possible. The system is given in Eq. (39).

$$\begin{aligned} M &= 0 \\ C_n &= S_3 = S_o = AR_o \\ S_2 &= S_1 = \overline{AR}_o \end{aligned} \tag{39}$$

and the MUX enable lines

$$\begin{aligned} \text{ENABLE MUXA} &= AR_o \cdot AR_1 \\ \text{ENABLE MUXB} &= \overline{AR}_o \cdot \overline{AR}_1 \quad . \end{aligned} \tag{40}$$

As already mentioned during the description of the COMP instruction, PHR MUX and WR MUX commands replace AR_o and AR_1, respectively, when a comparison is met. The enable lines on MUXA and B and also the ALU controls are therefore available through multiplexers controlled by E_N. Besides the above and the operations performed in conjunction with the IF, one further is associated with the ALU. This is DOIAR which stands for direct or inverse operation in arithmetic ALU. It will be noticed from Table 5.6 that the commands of SHIFT A and SHIFT B are logic duals of each other. Similarly so are those of A - B and A + B. This choice is deliberate and is utilized to provide a single line conditional command. The condition used is the value of the least significant bit of multiplexer Y (designated LSBY). The function $\overline{\text{DOIAR}} \cdot \text{LSBY}$ controls a pair of exclusive OR's, the other inputs into which are AR_o and AR_1. When DOIAR is not used its value is high and consequently, irrespective of LSBY, the exclusive OR's transmit AR_o and AR_1 unchanged. During a DOIAR command the outputs of the exclusive OR's are AR_o and AR_1 if LSBY = 0, $\overline{AR}_o$ and $\overline{AR}_1$ if LSBY = 1. The examples below illustrate the usage of the command:

1. DOIAR (ACCA, CB) → ACCB
2. DOIAR (AB - PHR) → ACCB

1. Leaves value of ACCA unchanged if LSBY = 0 or replaces it by CB if LSBY = 1.

2. Clock AB - PHR into ACCB if LSBY = 0, and AB + PHR if LSBY = 1.

It should also be pointed out that the LSBY may assume difference values. Thus inside a DO LOOP LSBY will alternate between 0 and 1 on consecutive passes. If the ACCB or AB inputs on MUXY are used the evenness or oddness of the numbers contained there will be the controlling factor.

The arithmetic ALU output is channeled through a specially arranged set of two input multiplexers. Their purpose is to provide saturation. Operation of this kind implies non-modulo arithmetic. When a positive number is to appear on the ALU output exceeding the largest representation permissible, the output is forced to this largest permissible value rather than "wrap around" as would be the case in ordinary modulo arithmetic. Under similar conditions for negative numbers the output is clamped to the most negative permissible value. In the representation used here, this implies $1 - 2^{-15}$ and -1 as the two limiting values. The choice of this kind of computational strategy is dictated by stability considerations. It may be shown that in any kind of computation which contains feed-back (examples of this are all the digital filters used here), a self-sustaining instability is likely if overflow and conventional "wrap around" is to occur. Furthermore, this will be so even if all else is totally ideal. Saturation arithmetic on the other hand guarantees complete stability even if accidential overflow was to occur. Table 5.7 summarizes and comments on all possible states. SA, SB and SF are the sign bits of the A, B, inputs and the

ALU output. OP is the operation with OP = 0 for Add and OP = 1 for Subtract. Actually OP is equivalent to AR''_0. The last column, labeled OUTPUT, indicates whether the result of the operation is correct (in which case an O.K. appears) or in the event of an overflow, what kind of clamping is needed. The State column is the decimal equivalent of OP SA SB. The clamping multiplexer following the ALU has one of its inputs (channel 0) connected to the ALU output. The other has the inverted sign bit, i.e., $\overline{SF}$ connected to the most significant bit and SF itself to all others.

TABLE 5.7

State	OP	SA	SB	SF	Comments	Output
0	0	0	0	0	A + B, both positive, answer positive	O.K.
0	0	0	0	1	A + B, both positive, answer negative	positive clamp
1	0	0	1	X	Effective subtract of 2 positive numbers	O.K.
2	0	1	0	X	Effective subtract of 2 positive numbers	O.K.
3	0	1	1	1	A + B, both negative, answer negative	O.K.
3	0	1	1	0	A + B, both negative, answer positive	negative clamp
4	1	0	0	X	A - B, both positive	O.K.
5	1	0	1	0	Effective add of 2 positive numbers answer positive	O.K.
5	1	0	1	1	Effective add of 2 positive numbers answer negative	positive clamp
6	1	1	0	0	Effective add of 2 negative numbers answer positive	negative clamp
6	1	1	0	1	Effective add of 2 negative numbers answer negative	O.K.
7	1	1	1	X	A - B, both negative	O.K.

ACCB consists of an ALU also with saturation clamping. Although it is describes as the ACCB, the content of the register is usually implied when

a number is referred to as ACCB. The unit is intended primarily for arithmetic operations and is therefore designed for 16-bit words. In addressing, only half the number is needed and so only the top 8 bits of ACCB are channeled over to MUXY. Three dedicated lines designated $ACCB_0$, $ACCB_1$ and $ACCB_2$ control operations. These are listed in Table 5.8.

TABLE 5.8

No.	$ACCB_o$	$ACCB_1$	$ACCB_2$	Internal MUX E_N	PHR MUX E_N	Meaning	Comments
0	0	0	0	1	0	SHIFT EX	ALU in A+B with A=0
1	0	0	1	0	0	ACCB + EX	Straight A+B
2	0	1	0	X	X	Not used	
3	0	1	1	X	X	Edit	
4	1	0	0	0	1	ACCB	ALU in A-B with B=0
5	1	0	1	0	1	ACCB + 1	ALU in A+B, 8th Bit=1, B=0
6	1	1	0	0	0	ACCB - EX	Straight A-B
7	1	1	1	1	0	SHIFT - EX	ALU in A-B with A=0

EX = external input; E_N = MUX enable line.

Operation 5, above, i.e., ACCB + 1 has a 1 in the 8th bit down. In this way the ACCB can be incremented for addressing purposes. This is achieved by disabling the PHR MUX and forcing a 1 into the 8th carry bit. A0 detects this state since

$$A0 = (\overline{ACCB}_0 + ACCB_1 + \overline{ACCB}_2) \cdot CPS \quad . \tag{41}$$

It is fed through a Nand gate to give the required carry.

The MOD and DOIB commands are also associated with ACCB.

5.9 Array Multiplier

Figure 9 gives a very rough schematic of the system used. The boxes with numbers in them are AM25S05 4 by 2,2's complement multipliers. A block labeled i - j refers to multiplier bits i and i + 1 and multiplicant bits j through j + 3. Neither the multiplier nor multiplicand lines are shown, but their presence and position can be inferred from the labeling. Since only the 16 most significant bits are used in the product not all multiplier blocks are needed, thereby saving on hardware. More detailed information on such arrays is given in the manufacturers data sheets. The particular arrangement used here produces a 16-bit product in typically 80 nsec.

The multiplier in the arithmetic section is supplied by the MULT Register. This in turn is clocked in every cycle by CLL and therefore always contains the output of the arithmetic ALU. The multiplicand is the CB buffer output. This buffer is clocked only on the dedicated command CLCB. Since the multiplier is memoryless a new product of the current ALU output word and CB is constantly being produced. The multiply command MULT is used only as a clock on the product hold register PHR. Thus MULT results in the current product being retained for further use. This value is available until the next MULT command appears.

5.10 Intermediate Memory

Besides the RAMs, which could also be calssified as intermediate memory, a number of single word intermediate memories are available. The division into time zones, necessitated by the need for pipelining, provides storage in the

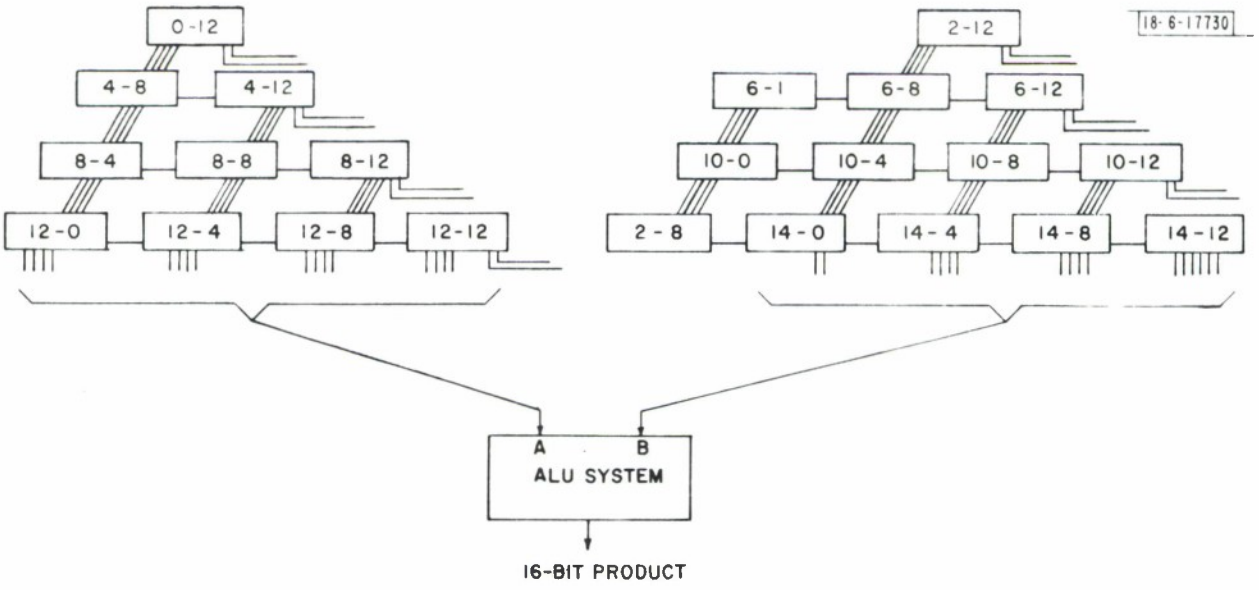


Fig. 9. 16 × 16 multiplier array with 16-bit product.

dividing buffers. Not everywhere, however, is use made of this facility. Thus, all buffers clocked by CLL directly will store their contents for one clock period only. Others like AB, BB and CB buffers perform the dual role of (1.) isolation of 2nd and 3rd time zones and, (2.) intermediate storage. This second effect is achieved by providing dedicated clocking lines for each of them. In this way only when the clocking command is given will the content on the inputs of these buffers be stored in them. Storage of given words for an arbitrary number of computational periods is therefore possible. CBI in the second time zone (Fig. 5) is also a storage register. Also, an existing word may be incremented by unity. Exactly the same is true of the I_x counter. Both of these registers are 8-bits wide. The multiply register, clocked by CLL does not provide intermediate storage, the PHR however does. This was already described in the previous section. ACCA can provide storage since a clock ACCA (CL ACCA) is available, and finally ACCB with CL ACCB provided is also an intermediate store.

5.11 ROM Coding

At a number of points of this report mention has been made of the ROMs used to implement memoryless Boolean functions. This section lists and describes them all.

The ENCODER ROM implements Eqs. (24). The inputs are A_0, A_1, A_2, A_3, A_4, S_{11} and S_{21} starting with the least significant and up to the seventh address bit. The 8th and most significant bit is grounded. The four outputs are labeled 0_1, 0_2, 0_3 and 0_4. The encoder ROM translates the computed output words (stored in the output buffer) into the required format used for serial data transmission. The ROM output feeds the Data Register (Fig. 7.).

HAD ROM produces the expression given in Eq. (28) as one of its outputs, this is denoted by H_1. Another line, denoted H_2, is used to give the complement, i.e., $\overline{H}_1$. The eight inputs are the i and j words. The two remaining outputs are used to produce

$$S_A = i_3 + i_2 \cdot \overline{i}_1 \cdot \overline{i}_0 + \overline{i}_2 \cdot (i_0 + i_1) \qquad (42)$$

$$S_B = i_3 + \overline{i}_2 \cdot \overline{i}_1 \cdot \overline{i}_0 + i_2 \cdot (i_0 + i_1)$$

S_A will be seen to be zero for lines 1 and 3 in Table 2.4. S_B is zero for lines 1 and 2. They are therefore numerically equal to S_{11} and S_{21}, respectively. Actually

$$S_{11} = S_A \cdot \overline{TS} \qquad (43)$$

$$S_{21} = S_B \cdot \overline{TS}$$

where TS = 0 during a spectrum computation and 1 during pitch. More details on this parameter are given in Section 5.14.

The TIMING ROM is driven by the 'i' word and TS. It is used to produce, in conjunction with other circuitry described in Section 5.14, timing pulses for data in an output. Table 5.9 lists the requirements. The R_i are the ROM outputs and the last column represents clocking pulses. More about these will be found in Section 5.14.

TABLE 5.9

TS	i	R_3 R_2 R_1 R_0	No. of CP_1
0	0	0 0 1 0	5
0	1, 2, 3, 4	0 0 1 1	4
0	5, 6, 7	0 1 0 1	2
0	8,...., 15	0 1 1 0	1
1	X	0 0 0 0	7

The EDIT ROM has already been mentioned in conjunction with editing procedure in Section 2.2. Specifically, the EDIT ROM accepts the 8-decision bits generated during the editing process as a parallel address and provides one line only labeled X_5'. The logic implemented is given in Eq. (17).

Finally, there are the two sets of Δ ROMS designated ROM Δ_1 storing constants and ROM Δ_2 used for ALOG and EXP commands described in Section 5.5.

5.12 Data I/O, Acquisition and Synchronization

The data input/output facility at the machine end is provided by the Data Register (Fig. 7). The encoded word for transmission is clocked into it in parallel. The MSB of this word appears always on line OP_6 of the register output. Clocking pulse CP_3 provides the required clocking edge for this register, when clock output buffer (CLOB) is activated.

$$CLOB = \overline{S_1\ MUXA} + \overline{AR}_o' + \overline{AR}_1' \quad . \tag{44}$$

An edge triggered F/F controlled by CP_3 and CLOB in turn controls the select line S_1 on the data register. With $S_0 = S_1 = 1$ the register (a set of 74S194s) is in parallel load mode. CLOB will put $S_1 = 1$ and CP_3, which comes always in

the following line, puts S_1 back to zero. Thus only immediately after CLOB will a parallel load result. Elsewhere the select lines will be in state $S_0 = 1$, $S_1 = 0$ which implies a shift right every time a clocking edge is applied. The clock on the register may be either CP_3 or CP_1. The latter is used in bursts of appropriate length (see Section 5.14) to shift right the available data. Thus the first edge of CP_1 puts the most significant bit available on CP_6 into CP_7. It also pulls into CP_0 the bit applied to the DSR input on the Data Register. For the first spectrum word CP_1 will have five edges. After such a clocking sequence, the five bits originally present will have been clocked out via OP_7 and five new bits now occupy postions OP_0 to OP_4. Another clocking sequence CP_2 whose edges occur half a period behind CP_1 is used to clock the content of OP_7 serially into a first-in first-out memory. Thus, on completion of this procedure the original content of the data register has been transferred into the FIFO, designated output FIFO. The number of clocking edges in the sequence CP_1 (and as a consequence in CP_2 as well) is given in the last column of Table 5.9. These sequences correspond to the number of bits in the encoded final output words as described in Section 2.4. The received data, which appears on the bottom lines of the Data Register after completion of a CP_1 - CP_2 clocking sequence is treated as a word equivalent to the one just clocked into the output FIFO. Thus if CP_1 has five edges the receive word is assumed to represent the first spectrum word and so on. These received words are then decoded as described in Section 2.4 by a hard-wired multiplexer designated Receive Decoder MUX1 (Fig. 7). The output of this multiplexer is the 10 input on Receive Decoder MUX 2.

In order to ensure that the received bits do indeed have the interpreted

meaning, it is necessary to synchronize the received data stream with the internal CP clocks. Since the input data is at 2.4 KHz whereas the CP pulses are at a vastly different rate, again a FIFO is used as a buffer. This is designated Receive FIFO in Fig. 7. The input into this device is clocked at the 2.4 KHz rate, externally supplied as the input data clock. The system first requires Frame synchronization, i.e., it recognizes spectrum and pitch words and lines them up in a specially provided shift register. It then waits until the machine itself is ready to accept data. When this happens (referred to as data sync) data is taken out of this shift register and processed while new data is entered into the input FIFO ready for transfer into the shift register.

5.13 Pitch Decoding

A one-shot triggered at the appropriate time clocks the received pitch word into a RAM, capable of storing 16 8-bit words. The address is generated by a 4-bit counter and channeled via a MUX. Only during the write cycle is the address supplied exclusively by the counter. When in read, an offset is added to the address. Since Δ is manually adjustable this permits an effective delay to be introduced between the just-written and just-read pitch words. This delay adjustment is used to equalize any time offsets that may exist between the received pitch and the spectrum. A buffer stores the currently read pitch word. The buffer output is compared against the Hiss word. Its complement is also used to preset a counter, whenever that becomes full, and is clocked by TCC, the program counter overflow. A gating arrangement ensures that for non-hiss words, the pitch multiplexer (Fig. 7) selects the impulse $1\ 0\ 0\ 0_8$ and is otherwise connected to zero. In this way an impulse separated in time by the multiple

of computational periods specified by the received pitch word is generated. The output of the pitch multiplexer is available to the machine via the EXT input on MUXA and the 01 input on the Receive Decoder MUX2. If a Hiss word is received the pitch multiplexer is connected to the 0 0 or 1 1 states under pseudorandom number control. The 0 0 position supplies a positive unit pulse, whilst the 1 1 position gives a negative unit pulse. As a consequence, pitch excitation during Hiss is a noise signal of unit amplitude but random sign. The pseudorandom bit generator is capable of providing a random sequence with 2^{18} members. Since the sampling clock averages 140 μsec (see next Section) the sequence repetition rate is 2^{18} x 140 μsec = 36.7 sec. This is sufficiently long not to generate a noticeable repetitive pattern.

5.14 Timing

In order to maximize program throughput speed a half frame's work of computing is done without any pauses. Since program execution times will vary depending on input data, the intervals between TCC pulses will also vary, making the internal machine clock non-uniform. On the other hand, the input speech samples are received at a uniform 140 μsec sample rate. Therefore, the effective internal sample rate has to average 140 μsec or less over each half frame. A variation of 130 μsec to 150 μsec can be expected. The internal clocking rate CLL is adjusted until over a half frame the average execution time is just under 140 μsec. Generally, there will be an irrational number of speech samples in a half frame. Since only integer values are acceptable, an interger N_{HF} is generated each 1/2 frame such that the various N_{HF} values averaged over many frames approximate the above irrational number. In this way the frame rate and sample

rate may be synchronized to each other, preventing relative slippage. In the machine itself, using software, I_{NR} is generated by unit incrementing once every program run. At the end of the program a comparison is made. As long as I_{NR} is less than N_{HF} no action is taken. When $I_{NR} = N_{HF}$, the command CMCL is generated which stops the system clock CLL. The system at that time has used up N_{HF} samples and so it has to wait until a new set of N_{HF} samples has been stored in the sample FIFOs.

5.15 Voice Analog Section

A schematic of this section is shown in Fig. 8. The voice output from a microphone is fed into an amplifier with 6 to 21 dB of variable gain. The output feeds a presample low-pass cutting off at 3.3 KHz. This is an 8th order elliptic filter (CO815c, $\theta = 71^{o}$) with zero gain at DC. The output is split into two paths. One goes via a preemphasis network again with zero DC gain into a sample and hold. The other is band-pass filtered by an 8th order Butterworth 80 to 600 Hz filter. Its output is also sampled by a sample and hold module. The first of the two is the spectrum, the second the pitch path. Preemphasis of spectrum samples has been found empirically to improve final speech quality. Fundamental pitch periods lie in the range of 80 to 200 Hz; the band-pass filter therefore permits up to the third harmonic of the highest fundamental to come through. Spectrum and pitch samples are transferred into an A/D converter during alternating periods. Two clocks, denoted by SHS and SHP, respectively for spectrum and pitch sampling, are generated as follows: A presetable 8-bit wide counter has its parallel input hard wired to 220. A 1-MHz clock is used giving a 35 μsec period from preset to all full (255). The 255 state characterized by the overflow TC = 1 is used as a parallel enable for the counter and is a 1 μsec wide

pulse with a 35 μsec period.

The acquisition time of the sample and hold modules, defined as the time an unchanging input must be maintained to get the specified accuracy, is 35 μsec. Both SHS and SHP are maintained high for exactly that length of time. The 140 μsec sample period is divided into four 35 μsec zones. Starting with both SHS and SHP low for 35 μsec, SHS goes high. This lasts for 35 μsec whilst SHP is still low, in the third zone both are low again and in the fourth SHP alone is high. A high is the sample state and during a low the sample is held. The A/D conversion time (to 12-bit accuracy) is 30 μsec. The strobe input into the A/D on transition from 0 to 1 resets the converter to zero and sets the "busy bit" (also referred to as the status) to 1. When the strobe goes low, conversion begins. In our system, starting with the spectrum channel as the strobe goes high the A/D is reset and the status goes high disabling both the FIFOs and the px(n + 72) buffer. One microsecond later, the strobe goes low, and SHS goes high initiating a spectrum sample. The FET switch (active low) channels the pitch S&H to the D/A. This will have already been converted in the pitch S&H into a steady level. Thus the A/D starts its conversion cycle on the pitch sample. Some 30 μsec later the conversion is complete at which time the status goes low. Since SRCL goes high, this transition clocks the A/D output into the px(n + 72) buffer as well as into the FIFOs. In the meantime the spectrum sample period (35 μsec) is being completed. By the time the next strobe comes along the spectrum sample is already held for some 35 μsec; also, the FET switch channels the spectrum sample into the A/D and the spectrum conversion in the A/D begins. At the same time the next pitch sample is already being taken into the pitch S&H. On completion of the spectrum conversion the status output on the A/D

clocks this into the FIFOs but not into the px(n + 72) buffer since SRCL is now toggled into the low state and is not producing a clocking edge. The cycle then repeats. The FIFOs contain both spectrum and pitch samples (alternating) whilst the px(n + 72) buffer holds pitch samples only.

At the output side, the reconstructed samples are fed first into a FIFO. This is done in order to bring their rate back to the constant 140 μsec. The clock used to store the samples in this FIFO is the internally generated program execution rate which varies (as discussed in the previous section) from 120 μsec approximately to 150 μsec. The average rate over one-half a frame is 140 μsec. Thus, extracting the samples from the FIFO at exactly 140 μsec eliminates effectively the internal varying rate. These samples are then put through a low-pass filter identical to the 3.3 KHz presample filter. De-emphasis may be added at this point to compensate for the input pre-emphasis. However, the quality of speech was found to be more natural without it and so it was left out in our system.

A power amplifier feeding earphones or a loudspeaker completes the audio system.

VI. PROGRAMMING

At the end of Section 4 the format in which a program line would be written was already indicated. The program step is just a consecutive number and conveys no information other than that of position. The real information content is in the logic and the arithmetic parts. The two are separated by a semicolon to denote that execution of the former is one time zone ahead of the latter. The following is a list of mnemonics used to denote various operations together with comments and explanations. The simpler operations, such as the

read or write are strictly one-zone operations (logic in this case) however, more complex commands like the COMP - IF for example require both logic and arithmetic sections. The listing therefore is in three parts: logic, arithmetic and combined operations.

6.1 Logic Commands

1.	X → Y	Read constant at address X in ΔROM and write into Y where Y can be either the CB or the CBI buffer.
2.	$_{J}X \rightarrow Y$	For J = A; read X in RAM A and store in Y. Y may be the AB, BB, buffer or a location in RAM B. For J = B it is a read from RAM B and into AB, BB or a location in RAM A.
3.	WR → RA at X	Write content of WR MUX output into RAM A at location X.
4.	WR → RB at X	Write content of WR MUX output into RAM B at location X.
5.	WR $_{A}X \rightarrow$ RB at $_{B}Y$	Write content of RAM A at X into RAM B at Y. The dual with A and B reversed is also permissible.
6.	DO DF, DL, N	Execute program lines DF up to and including DL, N times.
7.	CHA	Channel the A input through the Address C ALU.
8.	CHI_{x}	Channel I_{x} to the output of MUX Y.
9.	GO TO N	Unconditional jump to program step N.
10.	EXP, X → CB	Write content of the Δ_2 ROM, exponent section address X into CB.

11.	ALOG (X) → CB	Write content of Δ_2 ROM, ALOG section, address X into CB.
12.	CLIX	Unit increment I_x.
13.	CBI = CBI + 1	Unit increment content of CBI.
14.	Set I_x = N	Set I_x to be equal to N. Where N can be an arbitrary integer (including zero).
15.	CHACCB	Channel ACCB (8 top bits only) through MUX Y.
16.	CHACCB, CHI_x	Channel content (8 top bits only) of AB through MUX Y.
17.	INH	Inhibit the effect of the least significant bit out of MUX Y on the DOIB command.
18.	CMCL	Half frame clocking computed in software.
19.	HAD	Arithmetic operations under Hadamard matrix control.
20.	BL	Blank; no operation.

6.2 Arithmetic Commands

1.	ZERO	Shift zero through MUX A.
2.	X → ACCA	Clock X into ACCA.
3.	X → ACCB	Clock X into ACCB.
4.	X + Y	Add X to Y.
5.	X - Y	Subtract Y from X.
6.	MULT	Multiply; the content of CB will be multiplied by the content of the multiply register which is the past ALU output and the product will be clocked into the PHR at the end of the CLL clock period.

7. MOD (X) — The modulus of X appears on theoutput of ACCB. X is the number coming through the arithmetic ALU.

8. DOIAR (X OP Y) — Direct or inverse operation in arithmetic ALU. Execute Y OP Y if the least significant bit out of MUX Y (LSBY) is zero or X $\overline{OP}$ Y if it is one. Two pairs of OP and $\overline{OP}$ are plus, minus and Shift A, Shift B.

9. DOIB (X OP Y) — Direct or inverse operation in ACCB. Same as above but the operations are with respect to ACCB. These are given in Table 5.8. In addition the control is LSBY $\oplus$ MSB ACCA. If this is 1 direct operation results if 0 inverse.

10. COMP> (A,B), (OP_1, OP_2) X → Y

If AB > BB do A OP_1 X and write into Y
if AB $\leq$ BB do B OP_2 X and wriete into Y.
COMP< and COMP = are analogous. For greater detail see Section 5.7.

6.3 Joint Commands

There are two such commands. The IF jump and the COMP - IF. Both require an arithmetic operation followed by a logic command in the next line. The IF appears as follows:

```
------;          A - B
If > 0 → a, = 0 → b, < 0 → c; ------     .
```

This means that depending on the outcome of the comparison between A and B, where A is the word in the A channel of the arithmetic ALU and B is the B channel word, go to address 'a' if A > B, go to 'b' if A = B and to address 'c' if

A < B. This is a three-way decision.

For the COMP IF, using the COMP= as an example, a typical program line might look as follows:

```
------  ;  COMP= (AB, ACCA) (+, SFTA) CB → ACCA
If YES N;
```

This means that if the operation is not met (irrespective of what the operations are) go to program step N, otherwise continue sequentially.

VII. CONCLUSIONS

One of the consequences of the design approach described in this report is the constant need to re-assess the effect of the most recent modification on the rest of the machine and then take appropriate action. However, when the final modification is reached and the system just works successfully, the incentive to go back and rework the system for a more elegant solution is lacking. If, therefore, the machine described here seems in places capable of obvious improvements and none are made, this is so mainly due to the lack of time for a more elegant solution.

The final machine built has the following statistics:

Power consumtion: ~ 22 amps at 5 volts
~ 0.3 amps at 15 volts.

Size: 451 DIPs 85% Shottky TTL rest regular TTL
3-1/2 Augat boards of digital hardware
1/2 Augat board of analog hardware
Fits into one standard drawer.

ACKNOWLEDGMENTS

The author would like to acknowledge the contributions of a number of people whose help made the construction of the vocoder possible: Gloria Liias for her extensive programming support, Joe Tierney and Ben Gold for numerous discussions on principles of vocoding and computer design, Bob Meade for help in building the units and last but certainly not least Warren Hutchinson who cooperated with the author in all phases of design and whose contributions were of decisive importance to the success of the project.

REFERENCES

1. B. Gold and L. Rabiner, "Parallel Processing Tecniques for Estimating Pitch Periods of Speech in the Time Domain," J. Acoust. Soc. 46, 442-448 (1969).

2. B. Gold and L. Rabiner, "Analysis of Digital and Analog Formant Synthesizers," IEEE Trans. Audio Electroacoust. AU-16, 81 (1968), DDC 673594.

3. B. Gold and C. M. Rader, "The Channel Vocoder," IEEE Trans. Audio Electroacoust. AU-15, 148 (1967), AD-679147.

4. R. M. Lerner, "Band-Pass Filters with Linear Phase," Proc. IEEE 52, 249-268 (1964).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-193	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Digital Microprocessor Channel Vocoder		5. TYPE OF REPORT & PERIOD COVERED Technical Note 6. PERFORMING ORG. REPORT NUMBER Technical Note 1977-33
7. AUTHOR*(s)* Jerzy Gorski-Popiel		8. CONTRACT OR GRANT NUMBER*(s)* F19628-76-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element No. 63431F Project No. 2029
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Systems Command, USAF Andrews AFB Washington, DC 20331		12. REPORT DATE 10 August 1977 13. NUMBER OF PAGES 92
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Electronic Systems Division Hanscom AFB Bedford, MA 01731		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

channel vocoder
digital microprocessor
LES-8/9
synthetic speech

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A complete, real-time, channel vocoder delivering good speech quality with a 2400-bit/second data transmission rate was implemented using purely digital circuitry in the form of a high-speed programmed microprocessor.

Necessary algorithms are presented and their effect on the machine design is discussed in detail. The end product is a very high-speed computing machine (measured in program throughput terms). It turned out to have a high degree of programming flexibility, which would make it adaptable to other tasks. This was a bonus, not an original goal. The project was conceived and successfully realized as the most practical way to build a vocoder for actual use in the LES-8/9 satellite communications system.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*