THE DESIGN AND CONSTRUCTION OF A SPECIAL PURPOSE COMPUTER FOR SPEECH SYNTHESIS-BY-RULE

by

Robert Jay Steingart

SB, Massachusetts Institute of Technology

(1974)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1976

Signature of Author.....
Department of Electrical Engineering
and Computer Science, March 31, 1976

Certified by.....
Thesis Supervisor

Accepted by.....
Chairman, Departmental Committee on
Graduate Students

THE DESIGN AND CONSTRUCTION OF A SPECIAL PURPOSE COMPUTER FOR SPEECH SYNTHESIS-BY-RULE

by

Robert Jay Steingart

Submitted to the Department of Electrical Engineering and Computer Science on March 31, 1976 in partial fullfillment of the requirements for the Degree of Master of Science.

Abstract

The special purpose computer described in this thesis is designed to convert an input string of phonemes, together with prosodic information, into a set of vocal tract model parameters. A post-processor uses these parameters to produce natural sounding synthetic speech. Three major requirements necessitated the development of a processor with a customized architecture and instruction set: real-time processing capability, low cost, and compact size. Efficient, high-speed operation has been ensured by matching the processor hardware and software with the algorithmic needs of the specific task. The processor has been designed, constructed, and tested.

THESIS SUPERVISOR: Jonathan Allen

TITLE: Professor of Electrical Engineering

ACKNOWLEDGEMENT

I would like to thank my thesis advisor, Prof. Jonathan Allen for suggesting the topic of research. He helped oversee and direct the plan of research, and was instrumental in obtaining the requisite funds.

Discussions in the early stages of design with former staff member Eric Jensen proved to be extremely informative. He was most helpful for his advice concerning the instruction set design.

Jack Allweiss modified and helped in the use of the wirewrap programs to convert the logic diagrams into hardwired boards.

The final draft was edited and typed with the kind assistance of Mark Johnston.

I would also like to thank other members of the group for their ideas and comments at various times over the course of this research.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

List of Figures

List of Figures (cont.)

CHAPTER 1

1.1 Introduction

The development of machines capable of producing synthetic speech has been an area of active research for decades. Early devices, dating back to the nineteenth century, employed mechanical bellows, reeds, switches, and acoustic resonant chambers, controlled by a person to mimic human speech. With practice, a trained operator could manually vary the system parameters to produce a few spoken words.

Recent advances in linguistic theory, digital computers, and digital signal processing have led to electronic analogs of the vocal tract with parameters that can be updated easily and accurately under computer control. Such systems require a large number of computations, and real-time speech production requires large and expensive computers. Rapid technological development in large-scale integrated (LSI) circuitry, however, is having a tremendous impact on the size, price, and speed of digital computing machines. These advances are instrumental in changing speech synthesis from merely a laboratory curiosity into a practical device for institutional or even individual use.

The subject of this thesis is the design and construc-

tion of a processor, using state-of-the-art technology, to implement a compact, inexpensive, natural sounding speech synthesizer capable of real-time speech production. These constraints necessitated the development of a special purpose digital computer, tailored for efficient execution of speech synthesis programs. The machine has been successfully built and tested. The prototype includes approximately 150 integrated circuits and costs under $3000. Current trends in LSI technology, coupled with large volume production, will certainly reduce the size and cost substantially. There is little doubt that the development of a practical, cost-effective speech synthesizer lies in the forseeable future.

1.2 Applications

The applications of speech synthesis hardware are plentiful. Throughout the years we have witnessed man's greater reliance on both mechanical and electrical machines. He has been forced, however, to communicate with machines on their level by using switches and keyboards for input, and printers and cathode ray tube displays for output. Since speech is perhaps man's most convenient form of communication, there is little question that the ability to speak to a machine and to receive a spoken response would greatly facilitate man-

to-machine communication.

Speech synthesis could provide for automatic voice readout of computer-stored information in a form easily assimilated by the human user. A user could request information from a central computer, perhaps over a telephone line, and the computer would respond verbally. Data could be entered either manually with a push button telephone as a keyboard or perhaps verbally once the accuracy of speech recognition devices is improved. Functions performed efficiently on large digital computers, such as automatic information retrieval and numerical computation, would be as far away as the nearest telephone. At present it seems most efficient to have both the phonemes and speech waveforms generated by the central computer, so that no special devices are employed at the terminals. However, once speech synthesizers are cheap enough, the waveform synthesis could be built into the terminals, thus greatly reducing the data channel bandwidth.

Channel bandwidth reduction may be another application of speech synthesis on crowded voice transmission lines. Reasonably accurate reproduction of speech from unencoded digitized samples requires approximately 50,000 bits/second. Delta modulation and pulse code modulation can reduce this figure by perhaps a factor of five, but

a system in which only phonemes and prosodic information is sent and reconstructed into speech by a synthesizer could conceivably operate at less than 100 bits/second. The bandwidth of existing speech channels could be increased by a factor of 500.

Although the savings in bandwidth is impressive, the conservation of channel capacity is not the overriding issue it once was (due to new broadband transmission techniques using lasers, fiber optics, and guided millimeter waves). Furthermore, to conserve bits in a speech synthesis transmission system, certain qualities of an individual's voice are lost, so all voices are more or less indistinguishable. So for the time being, speech synthesis will be attractive only for long-haul channels where bandwidth is still at a premium.

One of the most exciting applications of speech synthesis, and of particular interest to Professor Allen's group at MIT, is the development of reading machines for the blind. Ordinary books could be read with an optical scanner and the characters grouped together as phonemes. Phonemic and prosodic information would then be fed into the synthesizer which would produce speech. Such a machine would overcome the shortcomings of braille, which include the limited availability of braille documents and the difficulty in acquiring proficiency in braille, especially for the aged

and those with a loss of tactile sensitivity.

Deaf people and others with vocal impairment could also benefit from speech synthesis. A typewriter operated voice for those who cannot speak is certainly feasible.

In the more distant future, a translator from one foreign language to another could be built by coupling a speaking typewriter to an automatic voice recognition system.

Finally, a speech synthesizer with controls that closely model the human vocal tract could be used as a research tool. By varying the model's parameters one can perform detailed and controlled experiments on various aspects of human speech production and recognition. This will have application to the further understanding of the psychology and physiology of speech formation and perception.

1.3 The MIT Text-to-Speech System

In the applications mentioned above, the feasibility is dependent upon the availability of a small and inexpensive speech synthesizer capable of producing intelligible speech from unrestricted text. One possible approach to the problem would involve recording all the words in the vocabulary, digitizing the signals, and storing the

bits in a computer memory. Words to be spoken would simply be looked up in the memory and the bits would be reconstructed by a digital-to-analog convertor to produce speech. This system may be feasible for small vocabularies, but the English language contains hundreds of thousands of words and would require an unwieldy memory. In addition, there would be no flexibility to accommodate variations in intonation and stress based upon context. Producing speech by piecing together one word pre-recorded segments thus sounds very unnatural.

Speech synthesis without recourse to any vestige of human speech seems to be the most attractive alternative, since it allows a large and sophisticated vocabulary in a form flexible enough to generate arbitrary messages while minimizing storage requirements. An overview of the complete text-to-speech system being developed by the Natural Language Processing Group at MIT is shown in Fig. 1. This illustrates how the processor developed in this thesis fits into the larger picture. In the first block an optical character reader converts printed text into an alphanumeric list of characters. Ultimately it must be able to read a complete range of type fonts from bound books. In the next section this list is scanned to generate phonemes. An attempt is first made to decompose each word into its morphs, the basic build-

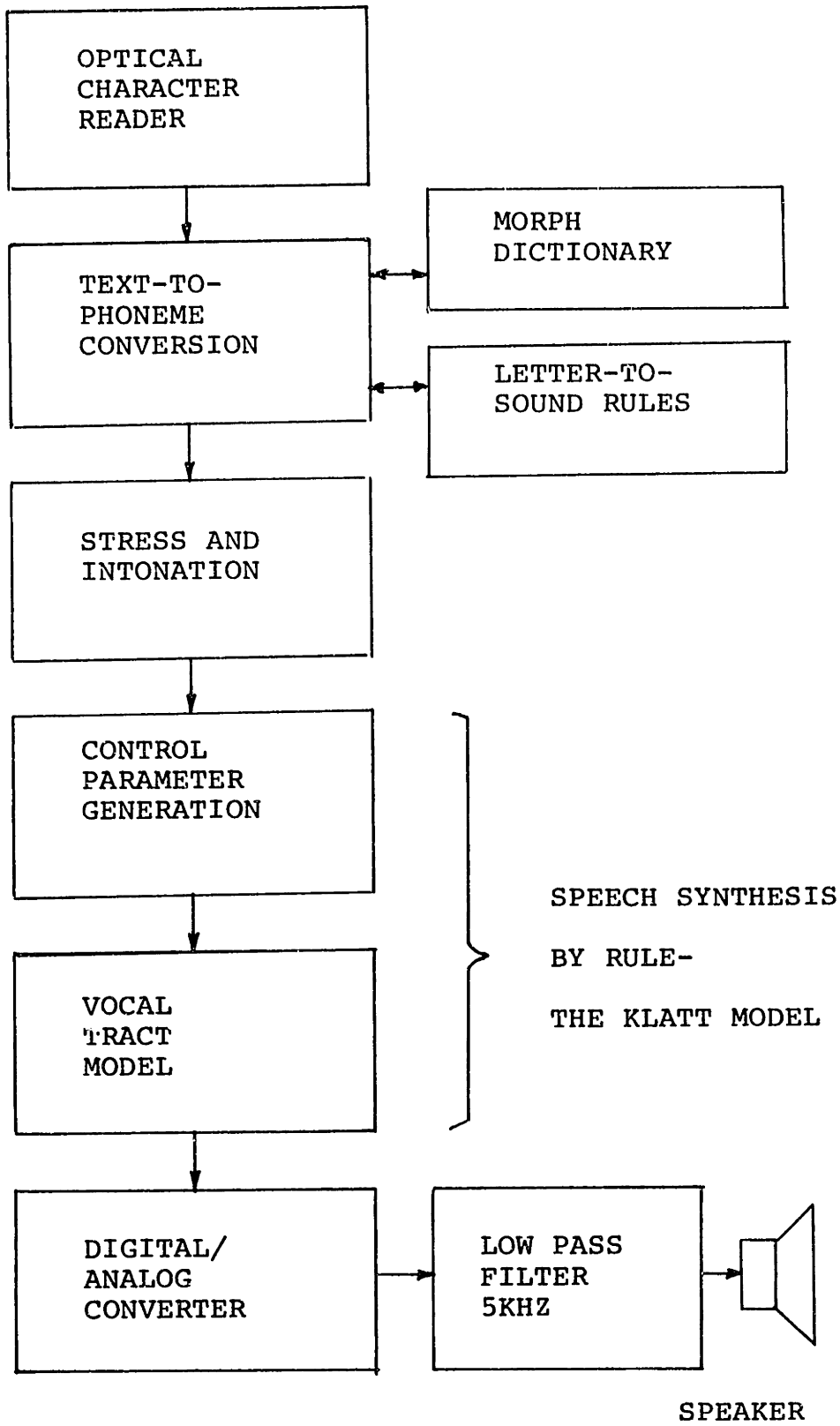


Fig. 1 Block diagram of the text-to-speech system

ing blocks of English words. The phonemic transcription is then looked up in the morph dictionary. If the word cannot be decomposed into morph entries of the lexicon, another program containing letter-to-sound rules is invoked to produce phonemes. Speech reconstituted from these phonemes sounds monotone as it lacks stress and intonation information: this is extracted from context in the next block using linguistic and syntactic rules. By parsing each sentence, phrase and sentence level stress marks can be generated. These prosodic features are necessary to effect natural sounding speech.

The phonemic and prosodic string is then submitted to a processor to calculate control parameters for a dynamically controlled vocal tract model. The hardware implementation of this processor is the subject of this thesis. The last blocks in the figure include the vocal tract model, a digital-to-analog converter, a low pass filter, and finally an electro-mechanical speaker.

CHAPTER 2

Speech Synthesis by Rule -- The Klatt Model

The basic program for speech synthesis employed by the Natural Language Processing Group was developed by Dennis Klatt. The first step in the design of a hardware processor to implement his algorithms was careful examination of his code. Flowcharting two different versions of Klatt's code (the older one written in assembly language and FORTRAN for execution on a Digital Equipment Corp. PDP-9, the newer one written in PDP-10 FORTRAN) made clear certain frequently used procedures.

With the objective of an inexpensive, compact machine capable of performing the necessary computations in real-time, it is obvious that the architecture and instruction set of the processor must match closely the algorithmic tasks specified by the program. By isolating functions that occur frequently and providing for special hardware and software to execute these functions more efficiently, the processor speed has been maximized while minimizing memory requirements. The emphasis has been on maximizing the performance/cost ratio.

The next two sections briefly describe the vocal

tract model and its control strategy. The last section discusses the software and hardware design guidelines derived from Klatt's model.

2.1 The Vocal Tract Model

The Klatt model is based on the theory originated by Fant (Fant, 1960) that the transmission characteristics of the vocal tract are well approximated by a cascade of resonators (poles) and antiresonators (zeros) whose band-widths and center frequencies may be independently controlled. Speech can then be produced by dynamically altering the vocal tract resonances (formants) and supplying the necessary excitation source. The primary building block in Klatt's approach is a digital time-invariant linear filter, a device with transfer function comparable to an analog resonator. Two parameters, the frequency and the bandwidth, specify the input/output characteristics according to the second-order difference equation illustrated in Fig. 2.

$$y(nT) = A \cdot x(nT) + B \cdot y(nT-T) + C \cdot y(nT-2T)$$

where:

$$A = 1 - B - C$$

$$B = 2 \cdot \exp[-\pi(BW)(T)] \cdot \cos[2\pi(F)(T)]$$

$$C = -\exp[-2\pi(BW)(T)]$$

and n = sample number

BW = resonator bandwidth

T = period between samples

F = resonator frequency

x( ) = input sample

y( ) = output sample

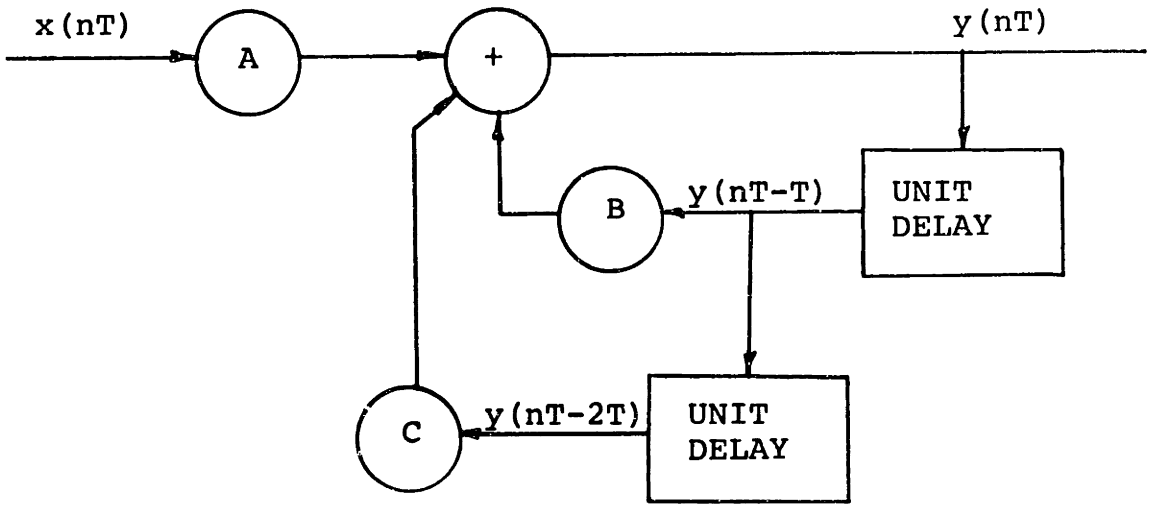


Fig. 2 Difference equation to realize a digital resonator

A special piece of digital hardware is currently being constructed at MIT to implement this filter function. This device consists of a single high speed filter section that can be time multiplexed to effectively model all the resonators in Klatt's vocal tract model. Time multiplexing reduces the amount of hardware significantly: 1/n of the time the single high speed filter

simulates one of the n filters required by the Klatt model.

The model of the human vocal tract built from these digital filters is shown in Fig. 3. (Klatt, 1975) The input to the synthesizer is a set of 38 control parameters that are updated every five milliseconds, fast enough for the most rapid formant transitions. These parameters specify the resonant frequencies and bandwidths of the vocal cavity and the excitation applied to it.

The Klatt model provides for three separate excitation sources: voicing, aspiration, and frication. The voicing source is specified by the fundamental frequency (FØ), voicing amplitude (AV), break frequencies of a glottal shaping network (F_{G1}, F_{G2}), and bandwidths of the glottal network (BW_{G1}, BW_{G2}). FØ controls the frequency of an impulse generator which simulates the vocal cord vibrations. Aspirated sounds like "huh" and fricatives such as "sh" or "th" are caused by a turbulent air stream, and are modelled as white noise. Sources of aspiration and frication are derived from samples of a pseudo-random number generator which are passed through a low pass filter with cut-off frequency F_{NOISE}. There are independent controls for the amplitude of each (AH, AF).

The vocal tract transfer function for nasal and

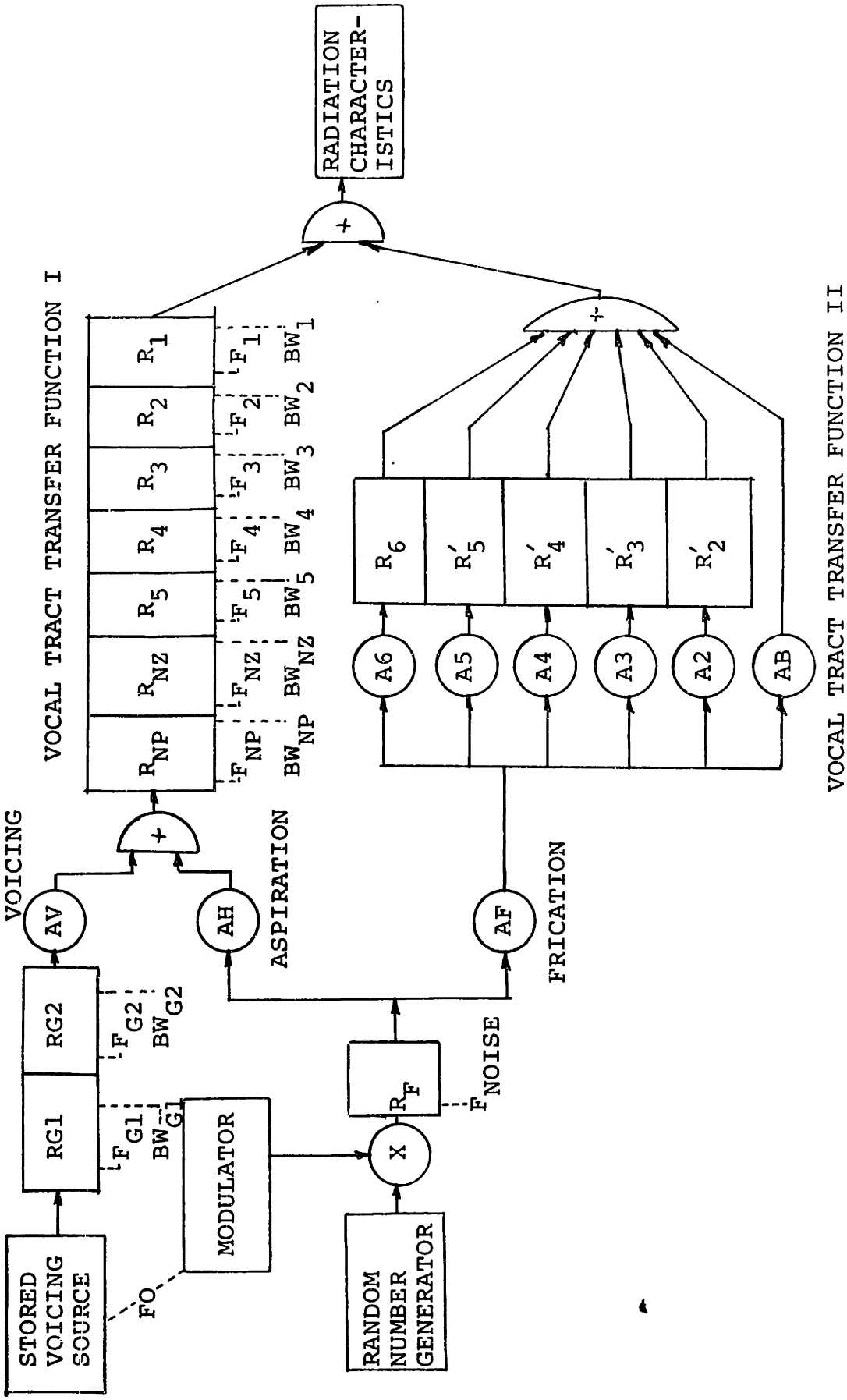


Fig. 3 Vocal Tract Model (Klatt)

laryngeal sources is modelled by seven digital resonators connected in series with independent control parameters for the frequency and bandwidth of each filter. The first two resonators model nasal poles and zeros respectively. A separate transfer function for frication sources uses five digital formant resonators, each with its own amplitude control (A2-A6), in a parallel configuration. AB specifies the amplitude of the by-pass path. Bandwidths for this section are calculated by multiplying the bandwidths, BW_1-BW_6 by a bandwidth scale factor (BW_{SCALE}). This parallel (as opposed to serial) arrangement has been shown to better model fricatives.

To summarize, the control parameters are as follows:

FØ	fundamental frequency	1
F_{G1},F_{G2}	glottal shaping network frequencies	2
BW_{G1},BW_{G2}	" " " bandwidths	2
F_{NOISE}	noise filter cut-off frequency	1
AV	amplitude of voicing	1
AH	" " aspiration	1
AF	" " frication	1
AV_{MAX}	maximum amplitude of voicing	1
AH_{MAX}	" " " aspiration	1
AF_{MAX}	" " " frication	1

F_{NP}, F_{NZ}	Nasal frequencies	2
BW_{NP}, BW_{NZ}	" bandwidths	2
$F_1, \ldots, F_6$	Vocal tract frequencies	6
$BW_1, \ldots, BW_6$	" " bandwidths	6
BW_{SCALE}	bandwidth scale factor	1
AB	Amplitude of bypass path	1
A2,...,A6	Resonator amplitude controls	5
ΔT	period between input parameter updates	1
SAMP	output sampling rate	1
NFRMT	number of formants to be simulated	1
	Total	38

These 38 variables allow for extreme flexibility, however it is possible to set some to constant values. For example, all the bandwidths and the fourth, fifth, and sixth formant frequencies can be set to constants with only a small loss in speech quality. In doing so the number of control variables is reduced and the data rate necessary to drive the vocal tract model in real time is correspondingly reduced.

2.2 The Control Strategy

In the discussion thus far, the function of the control parameters has been described but no mention has

been made of the source of these parameters. The design and construction of a special purpose processor to produce these control parameters from an input string of phonemic and prosodic information demanded a careful study of Klatt's control algorithm.

Klatt's voice synthesis program takes a string of phonemes, stress markers, word boundaries, and syntactic brackets as input and computes parameter values for the vocal tract model. For each phoneme in the input string, an acoustic description is retrieved from a stored table. This phoneme dictionary contains constants such as typical duration, formant frequencies, and formant bandwidths for isolated phonemes. Information classifying the phonemes by feature (e.g. voiced, vowel, nasal, diphthong, ...) is also included in the dictionary. The program then calculates duration and pitch inflection based upon stress markers and syntactic brackets. It is interesting that duration rather than intensity of a vowel segment often determines which syllable is perceived as stressed. Phrase and sentence level stress markers determined from context are used in the generation of the pitch contours.

The next section of code uses a set of rules to compare features of adjacent phonemes and, based on their mutual effects, calculate formant transitions to

produce smooth, free flowing speech. The stored constants in the phoneme dictionary are treated as boundary values from which continuous parameter values are computed by interpolation.

2.3 Hardware and Software Implications

It cannot be overemphasized that the real-time computing capability and compact size necessitated a special purpose processor with a customized instruction set. Currently available low-cost minicomputers with general purpose instruction sets are simply too slow. Klatt's programs run ten to twenty times slower than real time on a DEC PDP-9. The speed and space constraints imply that in addition to a special instruction set, the computer must make use of parallel processing wherever cost-effective in order to improve the machine's cycle time. A strictly serial machine could be built but would require higher performance components at a much greater cost.

In all of Klatt's programs there is a large amount of array processing. Typically, the stored constants in the phoneme dictionary are accessed by a variable offset within a given array. The computed parameter values are also stored in an array with a similar data structure. A convenient method of implementing such a data structure

utilizes a base and index register: the address of the N^{th} element of the array can be determined simply by adding the contents of the Base Register, START, (which is a pointer to the beginning of the array,) to the index register:

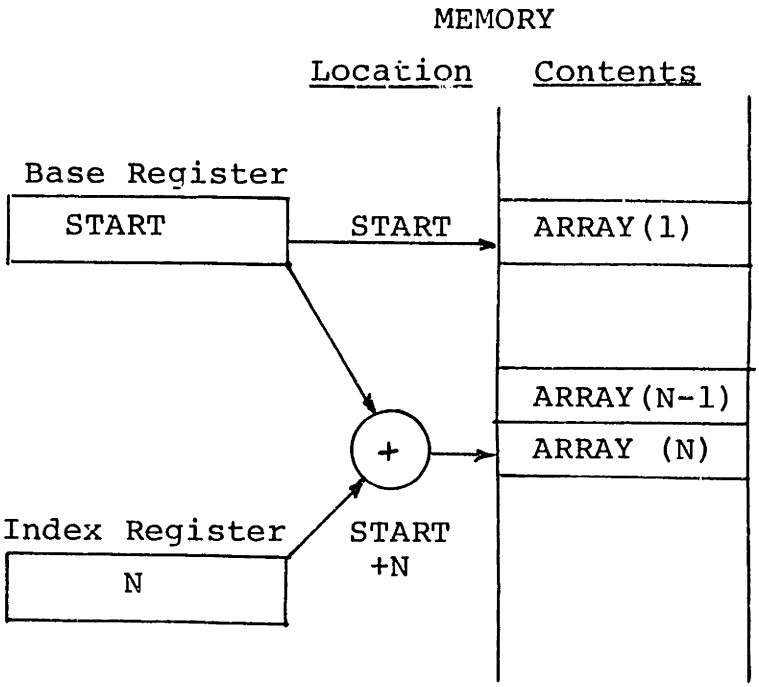


Fig. 4 Indexed Addressing of Arrays

Due to the high frequency of occurrence of this kind of data manipulation, it is desirable to provide for rapid indexed address calculation. Furthermore, the segment of code to implement this type of addressing should be concise to conserve instruction memory. This is discussed in detail in Chapter 3.

Phonemic feature testing is another type of process-

ing employed throughout Klatt's programs. Different blocks of code are executed depending upon the compared features of adjacent phonemes. A typical section of flowchart is shown below in Fig. 5:

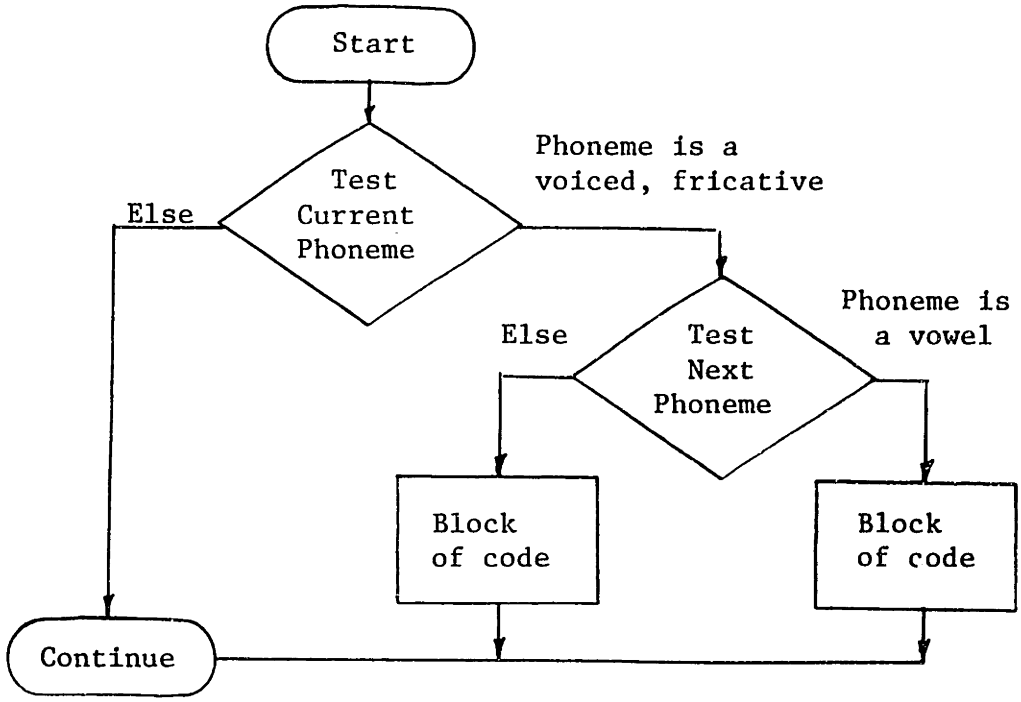


Fig. 5 Section of flowchart exhibiting feature testing

This kind of testing can be conveniently implemented in the following way: a 40 bit data word is generated for each phoneme, each bit corresponding to a specific feature. Setting the bit to ONE means that the feature is present; ZERO means it is absent. Processing an input phoneme requires retrieving this feature word from the phoneme dictionary. Then, to perform a test for a

given feature, the feature word is logically ANDed bit-by-bit with a mask containing all zeros except for the feature bits being tested. With this scheme it is easy to check for several features concurrently simply by setting the appropriate mask bits.

The frequent occurrence of this program structure motivated the development of two special instructions: Skip on Mask and Jump on Mask, which are described in detail in Chapter 3.

A significant amount of computation is involved in calculating the coefficients A, B, and C for the difference equation of Fig. 2. The derivation of the coefficients involves addition, subtraction, multiplication and the transcendental functions cosine and exponential. One of the countless number of hardware/software trade-offs that had to be made centered around which functions to include in the instruction repertoire and which ones to leave in the form of software subroutines. It was decided to include hardware and appropriate microcode for the arithmetic functions and leave the transcendental functions as subroutines. Among the factors influencing this decision were:

1. Addition, subtraction, and multiplication occur often throughout the program. A hardware/firmware multiply instruction has a small incremen-

tal cost and executes approximately ten times faster than a corresponding subroutine.

2. The cost of providing for transcendental functions in hardware is significant. If implemented in firmware, the microcode and its memory would double in size and price.
3. The important criterion in determining which functions should be sped up at the expense of increased complexity and cost is:

$$\begin{pmatrix}\text{relative frequency of} \\ \text{occurrence of a} \\ \text{given instruction}\end{pmatrix} \times \begin{pmatrix}\text{the instruction's} \\ \text{execution time}\end{pmatrix}$$

The execution time of e^x and cosine is relatively unimportant since they account for less than 0.2% of the total number of instructions. A tenfold (1000%) increase in their execution speed would improve the overall performance of the machine by less than 2%.

4. Transcendental functions can be calculated with sufficient accuracy from the first three of four terms of a Taylor series or by interpolation in a table of stored values. Either method involves addition, subtraction, and multiplication. Thus the inclusion of the multiply in-

struction greatly enhances the execution speed for the transcendental functions.

Although not used as extensively as multiplication, a hardware/firmware divide has been included in the instruction set. Both multiplication and division are fixed point (vs. floating point) to minimize circuit and firmware complexity as well as execution time. Keeping track of the location of the binary point is the programmer's responsibility. Since it may be necessary to scale numbers for multiplication and division, a single and double word shift instruction has been included.

For flexibility, a processor to handle speech synthesis programs should include some method for dealing with subroutines. Absence of recursion in the exising programs was partial justification for not using a stack architecture. An extensive discussion of the machine architecture will be found in Chapter 4.

The use of pointers and indirection, although not found in Klatt's code, will facilitate the handling of certain arrays. For example, to insert or delete an item from a list normally involves recopying part of the list. If one wanted to add the letter B to the following list, C, D, E, and F would have to be recopied into locations d, e, f, and g:

original list

location	contents
a	A
c	C
d	D
e	E
f	F
g	-

new list

location	contents
a	A
c	B
d	C
e	D
f	E
g	F

Using pointers, only the pointer from location a would have to be changed and an entry created for B. The example given assumes the list is scanned from top to bottom:

original list

location	data	pointer
a	A	c
c	C	d
d	D	e
e	E	f
f	F	-
g	-	-

new list

location	data	pointer
a	A	b
c	C	d
d	D	e
e	E	f
f	F	-
b	B	c

The input/output requirements of Klatt's control program have certain implications for the design of a special purpose processor. The only input is a phonemic transcription of printed text from an external computer; the only outputs are the coefficients and parameters transmitted to a third device, the vocal tract model. Although a bidirectional I/O bus could have been used to communicate with the other machines, separate input and output buffers simplify the design and are better suited to the task.

Furthermore, an interrupt system is not necessary. If the machine is ready for the next phoneme but the input buffer is empty there is no processing required until the next phoneme is delivered. The machine simply enters a wait cycle until new data is entered into the buffer.

For output the situation is similar: once the 38 parameters are ready the processor dumps them into the output buffer as fast as possible. A small, high speed memory reads the parameters from the output buffer and stores them for 5 msec. for the vocal tract model. A special instruction, OUTPUT, has been devised to transmit all the coefficients at a rate exceeding two million/second.

At a later date, when the entire processor can be reduced to several chips, the program will be stored in a nonvolatile memory (i.e. Read Only Memory -- ROM), so that the program remains intact even when the power is turned off. A separate read/write memory will still be needed, however, to store all the variables. Keeping this in mind now, the processor has separate program and data memories. Separate memories are actually an asset, since they allow for concurrent referencing and therefore faster processing.

Further details of the actual instruction set and computer architecture design that were dictated by Klatt's algorithm follow in Chapters 3 and 4.

CHAPTER 3

Instruction Set Design

After studying Klatt's algorithms, the next step in the design of the processor was the development of the instruction repertoire. The issues of instruction encoding, the number of different instructions, the precision of the data representation, and the size of the memory all influenced the detailed design of the instruction set. An instruction word that is too short in length limits the bits available for the operational code (OP code) and/or the operand address, and can severely restrict the machine's computational power. On the other hand, an instruction that is too long is more powerful but wastes valuable memory space. A well chosen instruction length allows several shorter instructions to be combined into one and can increase processor speed by reducing the number of instruction fetches from memory. By tailoring the instruction set to the particular task, a compromise can be reached that maximizes the computing power and flexibility of each instruction, while minimizing memory requirements. In addition, if the instructions reflect the structure of the underlying algorithms, programs will be easier to write, debug,

modify, and comprehend.

In general, most instructions need five pieces of information:

1. OP code, i.e. the function to be performed
2. The location of the first operand
3. The location of the second operand
4. The location for deposit of the resultant
5. The location of the next instruction

Different machines can be designed depending on whether these data are explicitly or implicitly defined. In a 4-address machine all four addresses, in addition to the OP code, are specified by the programmer. The inherent wide instruction word provides a flexibility rarely needed at a high cost in memory. A 3-address machine assumes the program will be executed sequentially except for jump and skip, and thereby eliminates the next instruction field. The next logical step in shortening the instruction is the 2-address machine which performs the function on the two specified operands and deposits the result in the location of one of the operands (i.e. A + B → B).

The most common addressing scheme, especially among minicomputers, is single addressing. Operations are performed on the specified operand and an implicit register called the accumulator (AC). The resultant is

deposited in the accumulator (i.e. AC + B → AC). A zero-address machine, sometimes called a stack machine, has all addresses defined implicitly. The next instruction is maintained in a program counter, the two operands are the top two cells of a pushdown stack, and the result is returned to the top cell of the stack. Stack machines are particularly good for arithmetic processing where equations are expressed in reverse Polish notation, and for machines with a heavy reliance on recursion.

By writing parts of Klatt's code for each of these machines and by comparing their hardware implementation costs, the 2-address format was chosen as a reasonable compromise with relatively good coding efficiency. Within the specific context of Klatt's algorithm: the 4-address machine is too general and therefore inefficient; there are not enough skips and jumps to justify a 3-address machine; as shown below the single address machine requires more instructions; finally a stack machine is unjustified due to the inefficiency in addressing registers low in the stack, and since the program makes little use of recursion.

Multiple high-speed registers within the CPU reduce the amount of data movement, save macroinstructions, and reduce the number of references to memory. A program for a single address machine which adds the contents of

location A and location B and deposits the result in location B is shown below:

	memory reference
Accumulator (AC) ← loc A	yes
AC ← AC + loc B	yes
AC ← loc B	yes

By having a large number of registers, the program can be written to increase the likelihood that the contents of location A and location B already have been fetched from memory and reside in one of the high speed internal registers. If this is the case, the above code can be reduced to a single line without any memory references:

Register B ← Register A + Register B

In the speech processor, register-to-register instructions utilize the high-speed registers and can operate almost twice as fast as those that reference memory.

The computer's CPU includes 17 internal registers:

a. Register 0 (microprocessor buffer register -- Reg. MB) is invisible to the user. It is used for temporary storage by the microcode.

b. Register 1 is a dedicated program counter (PC).

c. Registers 2-7 are reserved as index or base registers. They are especially useful in array addressing and table look-up.

d. Registers 8-15 are general purpose working registers.

e. Register "Q" is invisible to the user and is used by the microcode in multiplication, division, and shifting.

3.1 Memory Reference Instructions-Address Calculation

Throughout this chapter the following symbols are used:

( ) - refers to the contents of the memory location or register enclosed in parentheses

→ - "is transferred to"

∩ - logical AND

∪ - logical OR

‾ - one's complement

Reg.- register

Reg. D - destination register

E - effective address

I - indirection

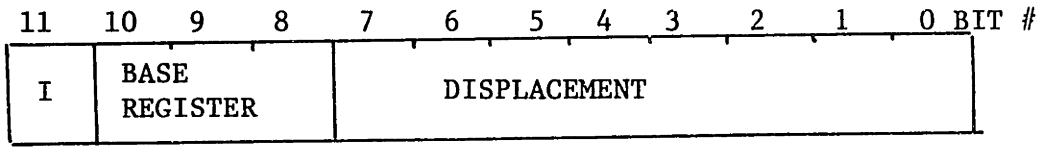
MSB - most significant bit

LSB - least significant bit

LABEL: - the colon indicates the variable to the left is a label corresponding to the address of that location

S - sign bit

Each memory reference instruction includes information necessary to calculate the absolute address from which data is stored or fetched. Addressing information is contained in three distinct fields occupying bits 0-11. Bits 0-7 are the displacement, 8-10 the base register bits, and bit 11 the indirect bit.



The displacement is an 8 bit signed number in two's complement notation with an octal range of -200 to +177. When the base register specified is non-zero, base addressing occurs; the displacement is added to the contents of the base register to generate a memory address. If the base register is 001_2, the program counter is chosen. This provides for relative addressing, a feature especially useful in altering the normal program sequence by jumping relative to the location of the

current instruction. The programmer should note that due to instruction overlap fetching, the PC points two locations beyond the current instruction. When the base register bits are 000_2, rather than adding the displacement to Reg. 0, the displacement is treated as an absolute address with a range of $000\text{-}377_8$.

If the indirect bit is 0, the effective address has already been calculated from the base register and displacement bits. However, when the indirect bit is 1, addressing is indirect and the machine retrieves another address from the location of the address already specified. There is only one level of indirection.

Indirection is desirable especially in programs using pointers. After reviewing Klatt's program, it became clear that pointers and indirection could simplify the handling of data. In a sequence requiring indirect addressing, an instruction capable of indirection can replace two normal instructions, thereby saving memory and increasing execution speed.

Another type of memory reference instructions was included to handle the accessing of variable entries within a list. These instruction, LDX and STX are described in section 3.3.

3.2 Single OP Code (Class I) Instruction

Class I instructions have a 4 bit OP code, usually reference memory, and follow the general pattern shown below:

BIT #	19 18 17 16	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
	OP Code	Reg. D	I	Base Reg.	Displacement

A summary of all instruction mnemonics and formats, including the various modifier fields is given in Appendix C.

OP Code	Function Mnemonic	Explanation
00_8	ADD	To summarize the calculation of the effective address E from bits 0-11: if the Base Reg ≠ 0 and I=0 then E=[(Base Reg)+Signed Displacement or LABEL] I=1 then E=[((Base Reg)+Signed Displacement) or (LABEL)] if the Base Reg = 0 and I=0 then E=[Positive displacement or LABEL]

I=1 then E= { (Positive displacement) or (LABEL) }

The contents of this address are added to the D register (one of the 15 internal registers). The D register also serves as the destination register. All data are in 2's complement notation. Using the above notation:

(E)+(Reg.D) → (Reg. D)

01_8	SUB	(E)-(Reg. D) → (Reg. D)
02_8	OR	(E)∪(Reg. D) → (Reg. D)
03_8	AND	(E)∩(Reg. D) → (Reg. D)
04_8	LOAD	(E) → (Reg. D)
05_8	STORE	(Reg. D) → (E)
06_8	ISZ	Increment Skip if Zero. The first step is (E)+1 → (E). If the final result equals zero, the program counter (PC) is incremented to cause the subsequent instruction to be skipped. The D register field is not used. Additional hardware could use these 4 bits as a modifier field to create more instructions. Presently, the increased complexity was considered unnecessary.
07_8	SDC	Skip if Different Contents. If (E)≠(Reg. D), the next instruction is skipped. The next instruction is executed normally if the two operands are equal. In both cases the contents

of E and Reg. D remain unchanged.

10_8 SM Skip on Mask. The complement of (E), $\overline{(E)}$ is bit-by-bit logically ANDed with (Reg. D). The result is not deposited or stored, however the next instruction is skipped if all the resultant bits equal zero. In other words a skip occurs when:

$$[\overline{(E_0)} \cap (\text{Reg. } D_0)] \cup [\overline{(E_1)} \cap (\text{Reg. } D_1)] \cup \ldots$$
$$[\overline{(E_{15})} \cap (\text{Reg. } D_{15})] = 0$$

where the subscripts refer to the bit being tested.

This function has applications in feature testing, a technique used extensively in speech synthesis programs. Suppose in Reg. D each bit represents a feature such as voiced, nasal, fricative, etc. Setting a bit to 1 means the feature is present. For example, part of the code for a voiced, fricative phoneme may look as follows:

	VOICED	NASAL	FRIC.	SONORANT
Reg. D	1	0	1	0

If one wanted to test the phoneme to determine whether it is voiced, the following mask would be used.

$\overline{E}$	1	0	0	0

to produce $\overline{(E)} \cap$ (Reg. D)

1	0	0	0

Since all bits are not equal to zero, the next instruction is executed. Now if the

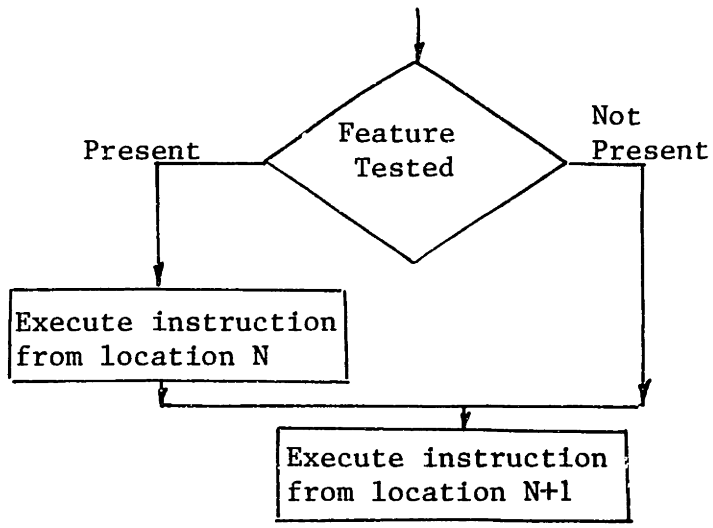
same phoneme undergoes a nasal test the mask:

$\overline{E}$	0	1	0	0

produces

0	0	0	0

and the next instruction is skipped since the phoneme is not nasal. To summarize, the subsequent instruction is skipped if the feature tested is not present.

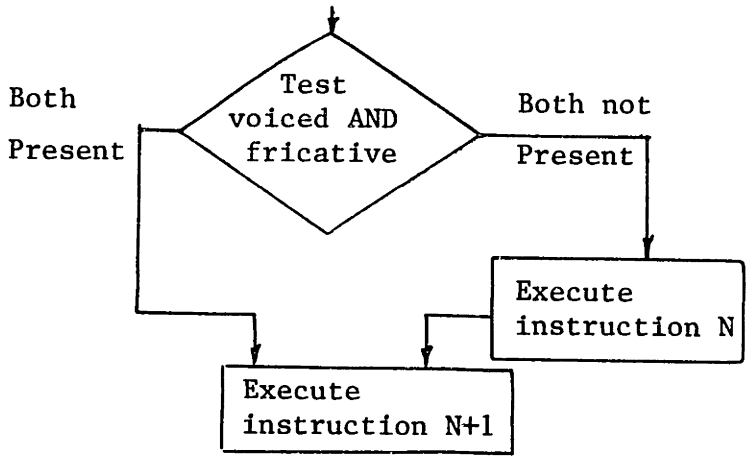


It is possible to test more than one feature at a time. To test whether a phoneme is "voiced OR fricative", in this example one would use the mask:

1	0	1	0

causing the next instruction N to be executed.

Testing for "voiced AND fricative" involves including one bit in Reg. D for $\overline{\text{voiced}}$ and another for $\overline{\text{fricative}}$. The actual test performed, "$\overline{\text{voiced}}$ OR $\overline{\text{fricative}}$" is logically equivalent to "voiced NAND fricative". In this example, the features tested ($\overline{\text{voiced}}$ OR $\overline{\text{fricative}}$) are not present so the instruction N+1 is executed. Given the function NAND, AND can be realized by reversing the roles of the two possible branches as shown below.



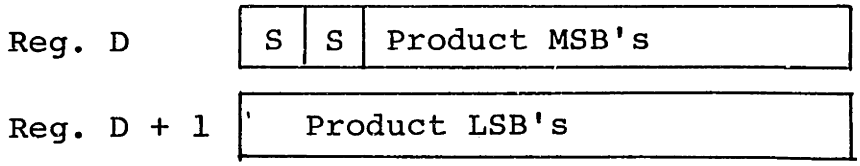
11_8 MULT Multiply the signed numbers (E) and (Reg. D) to generate a double length product. The higher order bits of the product are found in Reg. D and the lower order bits are found in Reg. D+1. Reg. D must be located on an even boundary (i.e. must be an even register). A few extra logic elements together with clever microprogramming provides for a 16x16 bit signed multiplication in slightly over 3 microseconds.

In performing signed multiplication of 16 bit numbers (1 sign bit and 15 magnitude bits), the magnitude of the product is 30

instead of 31 bits long. The actual algorithm used produces two equal sign bits in the two MSB's of Reg. D (bits 14, 15) corresponding to the sign of the product

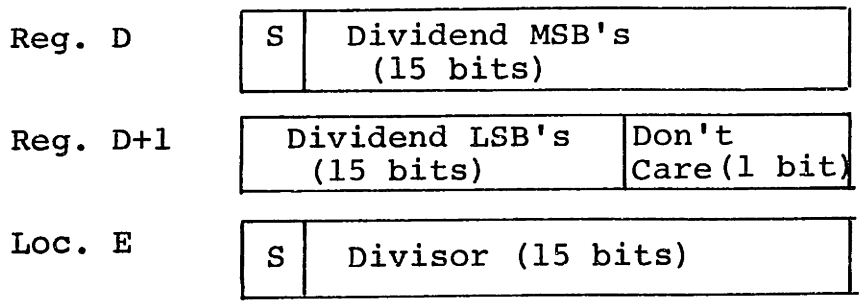
(E)X(Reg. D) → (Reg. D, Reg. D+1)

Final conditions:



12_8 DIV Initially the double length dividend is found in Reg. D (high order bits) and Reg. D+1 (low order bits). The divisor is found in (E).

Initial conditions:

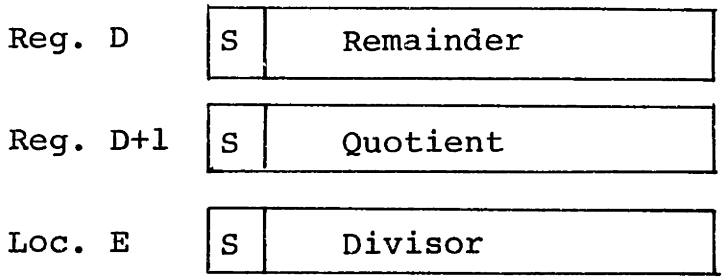


By convention, the absolute value of the dividend must be smaller than the absolute value of the divisor to prevent overflow. Before the division is performed a test for overflow, if affirmative sets the OVERFLOW Flag. As in all arithmetic operations, the programmer must keep track of the binary point. For certain operations it may be necessary to scale operands to maintain enough significant bits.

After the division is complete, Reg. D contains the 16 bit quotient. The signs are as follows:

Original Dividend	Original Divisor	Remainder	Quotient
+	+	+	+
+	-	+	-
-	+	-	-
-	-	-	+

Finally:



Once again Reg. D must be an even register.

13_8 JMP(JSR) Jump-Jump to Subroutine. There are two kinds of jumps depending upon whether Reg. D=0000. When Reg. D≠0000, the address of the next instruction (PC+1) is first stored in Reg. D and then the subroutine's starting address is placed into the PC. This provides for a simple yet effective means of jumping to and returning from subroutines. If Reg. D=0000 the PC is not saved and an ordinary unconditional jump is performed. In both cases the effective address is computed and specifies the address of the next instruction to be executed.

To summarize the effects of

JMP Reg. D, [Base Reg, Displacement / or / LABEL] : Effective Address

If Reg. D = 0000 then E → (PC)

If Reg. D ≠ 0000 then (PC)+1 → (Reg. D)
E → (PC)

In the following example, when the first JMP instruction is encountered, the return address NEXT is saved in Reg. 11 and the subroutine is then executed:

```
JMP 11, SUBROUTINE   (PC)+1=NEXT→(Reg. 11)
                     E=SUBROUTINE→(PC)

NEXT:    .
         .
         .

SUBROUTINE:  .
             .
             .
         JMP 0,11,0  E=(Reg. 11)=NEXT→(PC)
```

At the conclusion of the subroutine, by jumping to the address contained in Reg. 11, program execution continues at location NEXT. If many subroutines are being used or there is a shortage of working registers, the return may be stored in a temporary location as shown below:

```
SUBROUTINE:  STORE 11, TEMP   (Reg.11)=NEXT→
                                (E)=(TEMP)
                  :
             JMP INDIRECT 0, TEMP
                              E =(TEMP)=NEXT→
                                   (PC)
```

TEMP: NEXT

14_8 OUTPUT This special instruction provides for easy, rapid transfer of blocks of data from the data memory to the output buffer. E specifies the first word and (Reg. D) contains the number of adjacent words to be transferred out. With OUTPUT, somewhere between 1-2 million words/second can be moved directly from the memory to the output buffer without passing through the CPU. Comparing codes for outputting data:

Using OUTPUT

```
LOAD 8, THIRTY8     (E)=(THIRTY8)=38→(Reg.8)
OUTPUT 8, START
          :
THIRTY8:  38(10) # of words to be transferred
START:    STARTADR Starting address
```

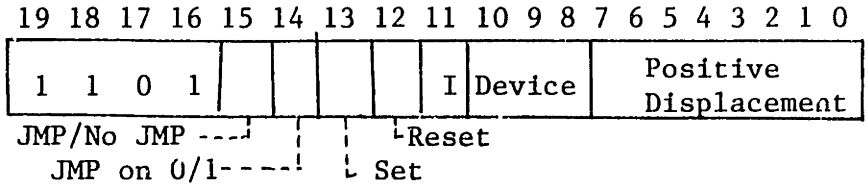
Conventional

```
LOAD 7, NTHIRTY8   (E)=(NTHIRTY8)=-38→(Reg. 7)
LOAD 6, END        (END)=LASTWRD→(Reg.6)
BACK:  LDX 9,6,7   ((Reg. 7)+(Reg. 6))→(Reg. 9)
STORE 9, OUTPUT    (Reg. 9) → OUTPUT Buffer
INCZ 7             (Reg. 7)+1→(Reg. 7) and skip
                                on zero.
JMP 0,BACK    E=BACK→(PC) (e.g. loop 38 times)
      :
NTHIRTY8:  -38(10)
END:  LASTWRD      Location of last word
```

In Klatt's program, it is necessary to output 38 parameters every 5 milliseconds. The OUTPUT instruction operates approximately

5 times faster than conventional code and saves 20 milliseconds of computing time for every second. The 2% reduction in overall execution time does not require extra hardware, only microcode. The OUTPUT instruction is a useful feature to have, but is not essential. Its utility should be reviewed as new algorithms and programs for speech synthesis are developed.

15_8 FLAG The FLAG instruction has a special format:



The device selcts one of the eight status flip-flops.

Bit 10	9	8	
0	0	0	OVERFLOW
0	0	1	RUN
0	1	0	CARRY
0	1	1	IN
1	0	0	OUT
1	0	1	X1
1	1	0	X2
1	1	1	X3

Bits 14,15 cause a conditional jump based on the state of the selected flip-flop.

Bit 15	14	-	Mnemonic
0	0	No Jump	-
0	1	" "	-

Bit 15	14	-	Mnemonic
1	0	Jump on flip-flop=0	Z(ero)
1	1	Jump on flip-flop=1	O(ne)

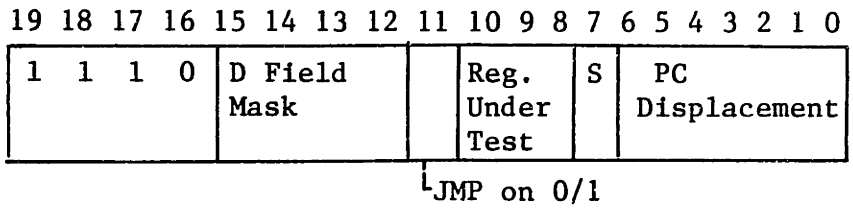
The address jumped to is restricted to locations $0\text{-}377_8$ unless indirection is used. With indirection, any address in the memory can be accessed through these locations.

After the jump test has been performed, regardless of whether a jump actually occurred, the selected flip-flop can be set, reset or complemented according to bits 12,13.

Bit 13	12	(Mnemonic)	
0	0	Flip-Flop (FF) unchanged	(-)
0	1	Reset FF to 0	(R)
1	0	Set FF to 1	(S)
1	1	Complement FF	(C)

The machine can be HALTed by resetting the RUN flip-flop, in which case the computer waits until the FF is set by an external switch.

16_8 JM Jump on Mask. Special instruction well suited for testing the state of individual bits within a 16 bit word. This function is particularly useful in feature testing. The format is:



The register whose bits are to be tested is specified with bits 8-10. The D field in this case does not represent a register, but specifies which bit should be tested. Bit 11 determines whether to jump when the tested bit equals 0 or 1. As an example, the instruction to jump back 25 locations relative to the PC (recalling that the PC always points two locations beyond the current instruction) when bit #3 of base register 5 equals 1 would be as follows:

With JM

OP Code	Mask	JMP on	Reg.	Displacement
1110	0011	1	101	$-27_{10} = 110\ 0101_2$

Convential Code

```
LOAD 8, MASK   (E)=(MASK)→(Reg. 8)
ANDZ 8,5       (Reg. 5)∧(Reg. 8)→(Reg. 8) and
                 skip if result=0
JMP 0,1, -27₁₀ E=(PC)-27₁₀→(PC)
  :
MASK:  0000  0000  0000  1000
```

If bit #3 equals 0, the jump does not occur and the next instruction is executed. When compared to conventional code, JM saves one memory reference (this is important since the memory is relatively slow), saves instruction memory by replacing 3 instructions with one and saves data memory by eliminating storage of the masks. The JM instruction also

operates almost 3 times faster than the conventional code. In Klatt's code, as it currently stands, JM implies a potential saving on the order of 500 macroinstructions together with higher speed.

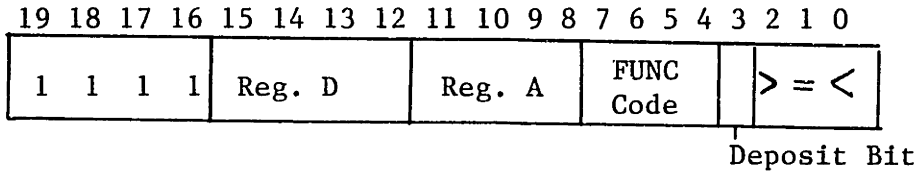
17_8 Class II Dispatch

This OP code causes the computer to find the required operation from the FUNC field (bits 4-7). This simple form of variable length coding makes efficient use of the instruction word.

In most instructions the destination is a register and can be represented by 4 bits. The source in Class II instructions, however, is a register (again 4 bits), whereas in Class I instructions it is an effective memory address requiring 12 bits. It is possible to break the instruction into shorter words, but the additional number of memory fetches would slow the machine substantially. The disparity in instruction size is handled by using a 4 bit OP code for Class I instructions and 8 bits (OP code-4 bits, FUNC code-4 bits) plus a 4 bit modifier field for Class II instructions.

3.3 Class II Instructions

With the exception of LDX and STX, Class II instructions involve register to register operations. All Class II instructions have the OP code 1111_2 which enables another 4 bit field, called the FUNC field. The FUNC field is decoded to represent an additional 16 instructions. Most Class II instructions follow this format:



All Class II instructions except SHIFT, LDX and STX contain a four bit modifier field in bits 0-3.

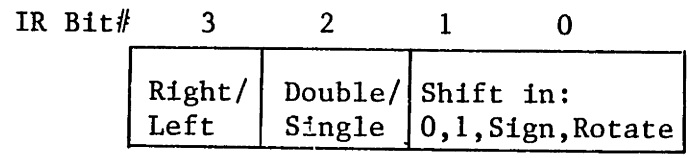
Bits 0-2 cause the next instruction to be skipped if the condition is met.

Bit 2	1	0	Skip if result	Mnemonic
0	0	0	Never	-
0	0	1	<0	LT
0	1	0	=0	Z
0	1	1	<=0	LE
1	0	0	>0	GT
1	0	1	≠0	NE

Bit 2 1 0	Skip if result	Mnemonic
1 1 0	>=0	GE
1 1 1	Always	A

	Mnemonic	
Bit 3 = 1	-	The result is deposited in the destination register.
= 0	#	The result is not deposited. This feature allows testing without affecting either source register.

FUNC Code	Function Mnemonic	Explanation
00_8	ADD	(Reg.A)+(Reg.D)→(Reg.D)
01_8	SUB	(Reg.D)-(Reg.A)→(Reg.D)
02_8	OR	(Reg.A)$\cup$(Reg.D)→(Reg.D)
03_8	AND	(Reg.A)$\cap$(Reg.D)→(Reg.D)
04_8	MASK	$\overline{\text{(Reg.A)}}\cap$(Reg.D)→(Reg.D)
05_8	MOV	(Reg.A)→(Reg.D)
06_8	INC	(Reg.D)+1→(Reg.D)
07_8	DEC	(Reg.D)-1→(Reg.D)
10_8	COM	One's complement. $\overline{\text{(Reg.D)}}$→(Reg.D)
11_8	NEG	Two's complement. $\overline{\text{(Reg.D)}}$+1→(Reg.D)
12_8	SHIFT	The format for SHIFT is slightly different from the standard:



Bits 0-3 are a modifier field to provide for all combinations of shifts and rotates. The A field is interpreted as a literal and specifies the number of bits to be shifted.

Bit 3	Mnemonic	
1	R	Right shift or rotate
0	L	Left shift or rotate

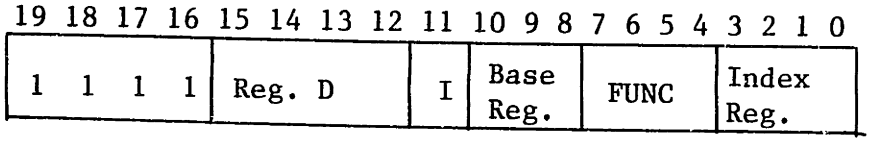
Bit 2	Mnemonic	
0	S	Single word (Reg.D)
1	D	Double word. Reg.D-higher order bits Reg.D+1-lower order bits Reg.D must be on an even boundary.

Bit 1	0	Mnemonic	
0	0	Z	Shift in zeros
0	1	O	" " " ones
1	0	S	" " " the sign of Reg.D (0 if Reg.D is (+) 1 " " " (-))
1	1	R	Rotate

Example: Double shift Reg.10 and Reg.11 to the right 5 places and fill the vacant bits with zeros (SHIFTRDZ).

OP code	Reg.D	Reg.A	FUNC	Modifier
1111	1010	0101	1100	0 1 00

13_8 LDX Load Indexed. LDX and STX use the format:



Normally the code to access the n^{th} entry of a table starting at memory location TBSTRT would be as follows:

Assume initially:

(Reg.7) = N

(Reg.8) = TBSTRT

and that the contents of these registers must be maintained.

```
MOV     10, 8   (Reg.8)=TBSTRT→(Reg.10)
ADD     10, 7   (Reg.7)+(Reg.10)=N+TBSTRT→
                (Reg.10)
LOAD    10, 10,0  (E)=((Reg.10))=(N+
                TBSTRT)=DATA→(Reg.10)
```

```
      :
TBSTRT:
      :
TBSTRT+N  DATA
```

The LDX and STX instructions, in addition to including indirection, a destination register and a base register, also have an index register. The effective address is equal to the sum of the contents of the base register and the contents of the index register. The code for the above example now is reduced from three lines to one:

```
LDX Reg. 10, Reg.8, Reg.7   ((Reg.7)+(Reg.8))=
                            (N+TBSTRT)=DATA→
                            (Reg.10)
```

The high frequency of this kind of memory reference in Klatt's program was a strong motivation for its inclusion in the instruction set. Several hundred instruction memory locations are saved with a corresponding decrease in execution time.

LDX

I=0 ((Base Reg.)+(Index Reg.))→(Reg.D)

I=1 (((Base Reg.)+(Index Reg.)))→(Reg.D)

14_8 STX Store Indexed.

I=0 (Reg.D)→((Base Reg.)+(Index Reg.))

I=1 (Reg.D)→(((Base Reg.)+(Index Reg.)))

15_8-17_8 These FUNC codes are currently vacant and allow room for new instructions.

3.4 Status Flags

OVERFLOW Set if the result of previous operation resulted in an arithmetic overflow. Reset with FLAG instruction.

RUN Set externally to start the machine. Can be reset under program control to halt the machine.

CARRY Set by the previous operation if the operation caused a carry out of its most significant bit. CARRY is useful for double precision arithmetic.

IN Set by external device when contents of input buffer is valid. When program requests data from the input buffer, the flag is tested. Data is transferred only if the IN flag is set (valid input). If not set, the program will wait. The flag is reset after data transferral.

OUT Set when contents of output buffer has been transferred to an external device and therefore the buffer is free. Flag

	is tested when output buffer is requested. Data is transferred from the common bus to the buffer only if the OUT flag is set. Program will wait if the buffer is already full. The flag is reset after the buffer is filled.
X1	Controls the write enable line of the instruction memory.
X2,X3	Currently these flags are not used.

3.5 Input/Output

The input and output buffers are treated as data memory locations. They are accessed using the ordinary memory reference instructions. The machine currently has a 4Kx16 data memory which requires 12 address lines (2^{12}=4096). Since the data word is 16 bits, the 4 MSB's (bits 12-15) are available for other purposes. Setting bit 15 of the address to ONE causes data from the input buffer rather than the data memory to be placed on the bus. If the buffer has been already loaded by the external source (the host computer) and the flag is set to indicate the data is valid, the data is transferred immediately. On the other hand, if the flag has not yet been set, the computer will wait until valid data is

available. For Klatt's program this is not a limitation; the machine cannot proceed without having the latest phoneme in memory. The fact that once input is requested the machine must wait eliminates the need for an interrupt system with its complexity of hardware and software.

Examples of inputting data:

1. LOAD 11,7,0 (E)=((Reg.7))→(Reg.11)

   Reg.7 must contain

   1000 0000 0000 0000

2. LOAD INDIRECT 11, INPUT (E)=((INPUT))→(Reg.11)

   INPUT: 1000 0000 0000 0000

It is possible to test the input flag with the FLAG instruction and continue processing other data until the input data is valid.

```
N:      FLAG 0  INPUT,50        Jump to N+50 if INPUT
          .                     flag is set to ONE.  ELSE
          .                     continue processing
          .
N+50: LOAD INDIRECT 11,INPUT    Load data from input
                                buffer into Reg. 11.
```

Output is handled in a similar fashion. Data can be transferred to the output buffer with either the STORE, STX, or OUTPUT instructions. Once again the buffer is treated as a data memory location. Bit 14 of

the effective address must equal ONE for the output to be the destintion rather than the data memory. The flag is tested before the buffer can be loaded; the machine waits until the flag is reset indicating the buffer is free.

Examples of outputting data:

1. STORE 11,7,0 (Reg.11)→(E)=(Reg. 7)
   Reg. 7 contains 0100 0000 0000 0000
2. STORE INDIRECT 11, OUT (Reg.11)→(E)=((OUT))
   OUT: 0100 0000 0000 0000
3. OUTPUT 12, START If Reg. 12 contains the number N, N adjacent memory locations starting at location START are transferred to the output buffer. The machine checks that the data has been recieved by the external device before transferring more data.

The FLAG instruction can be used to test the output flag and continue processing other data until the output buffer is free to accept more data.

CHAPTER 4

Computer Architecture

Computer architecture refers to the organization and interconnection of the components of a computer system. Some of the basic building blocks include memories, registers, Arithmetic Logic Units (ALU's), multiplexers, and buses. These blocks can be configured in a wide variety of ways to implement a given task. Certain design constraints such as cost or speed help the computer architect choose the best way to select, connect, and control the various modules to perform their functions most efficiently. Very often there is a direct tradeoff between performance and price, so to minimize cost the machine should be designed to just meet the given design specifications. These constraints force the computer architect to be creative in his design.

The overall architecture for the speech synthesis processor is shown in Fig. 6. Only after designing and reviewing dozens of alternatives was this particular structure chosen for its modularity, simplicity, speed, low cost, and flexibility.

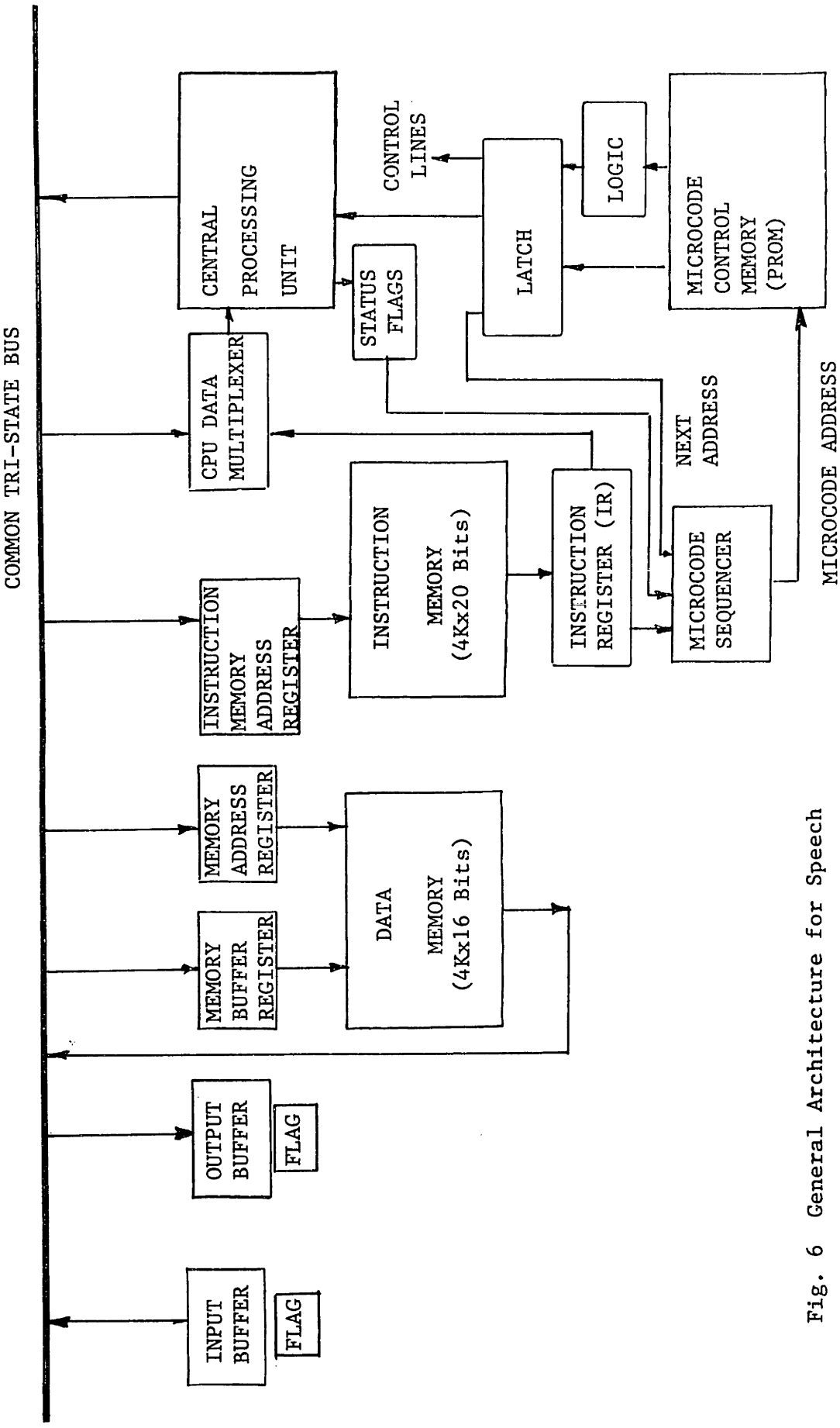


Fig. 6 General Architecture for Speech Synthesis Computer

4.1 Bus Organization

The processor uses a single bi-directional data/address bus which is shared by the input buffer, output buffer, data memory, instruction memory, and the CPU. The bus contains 20 wires (wide enough for the 16 bit data words as well as the 20 bit instruction words), so information is transferred in parallel. Many machines have two or more buses and all data must first pass through the CPU, however the single bus arrangement allows direct communication among all devices. For example, data can be transferred from the data memory to the output buffer while the CPU is busy performing an entirely different operation. This bus structure is extremely modular and easily expandable. The processor as a research tool will almost certainly undergo changes, and other memories, processors, and peripherals can be attached directly to the main bus with only minor hardware alterations.

In a single bus system, during a given cycle, only one device can transmit data while one or more devices can receive data. A problem that frequently arises in a single bus system is one of arbitration, that is, two devices may request to transmit data on the bus at the same time. In this machine, only the input buffer, data memory, and CPU can supply data to the bus. The micro-

program control has been designed in such a way that there is never a conflict; the machine never wastes a cycle while a device requests the use of the bus. The use of single bus combined with tri-state logic (the three stable states are 0, 1, and a high impedance disabled state) maximizes the number of direct data paths while eliminating the need for all but one multiplexer.

The one multiplexer shown controlling the source of data for the CPU was included for the case when the displacement field must be obtained from the Instruction Register (IR), altered by the CPU, and transmitted to the bus all in one machine cycle. If this multiplexer were omitted and the IR fed the instruction directly onto the bus an extra cycle would be required: during the first cycle the contents of the IR would be entered into the CPU via the bus, and during the second the CPU resultant address would be transmitted on the bus. This savings is significant since it is realized for every macroinstruction executed. The overall processor speed is increased by approximately 20% at the cost of only four additional integrated circuits.

4.2 Input/Output

All instructions and data are entered into the

machine via the input buffer which momentarily holds information being transferred into the computer. This buffer is loaded asynchronously from either the front panel or an external source (i.e. another computer). A separate output buffer provides the interface to the vocal tract model. A single bit flag for each buffer is set when the buffer is busy and reset when it is free. Both buffers appear to the programmer as data memory locations and can be used in any memory reference instruction. The need for special I/O instructions is replaced by the ordinary LOAD and STORE instructions. When loading from input the processor checks the status flag and waits for valid data. Similarly, when storing data in the output buffer the machine waits until the buffer is ready to receive more data.

In machines where the external devices operate several orders of magnitude slower than the computer, it is desirable to incorporate interrupts so that the machine can continue processing as it waits for data. This is not a problem in the speech processor, so an interrupt system is not included. Interrupts require significant overhead since the state of the processor must be saved before handling the interrupt request and restored afterwards.

4.3 Data and Instruction Memories

The computer contains separate data and instruction memories. The data memory (4K X 16 bits) has a Memory Address Register (MAR) which is loaded with the address of the location to accessed, and a Memory Buffer Register (MBR) which contains any data to be written into the memory. These registers hold the address and data for the duration of a memory cycle which is almost three times slower than the CPU cycle time. Once they are loaded the bus is freed so the machine can continue processing concurrently with the memory reference.

The instruction memory is currently 4K X 20 bits. Together with the data memory this is large enough to hold present and projected speech synthesis programs. The instruction memory has its own address register, but no data buffer register. Referring back to Fig. 6, the output of the instruction memory goes directly to the IR without first passing through a memory buffer register. Elimination of this buffer reduces the chip count without limiting the machine's power (since the particular memory chip used contains internal data buffers). During program loading (a relatively infrequent operation) the instructions are passed directly from the Input Buffer to the instruction memory. This is the only time the instruction memory is written into: instructions

cannot be altered during normal program execution.

The basic storage device for both memories is a 4K X 1 bit static NMOS random access memory (RAM). The memories can easily be expanded in width or length by simply extending the buffer registers and using more memory chips. The length of the instruction word can also be increased by single bit increments, to increase the power of the instruction set. The memory length can be quadrupled with a 16K X 1 bit RAM that is pin-for-pin compatible with the 4K's and is expected to be introduced into the commercial market during the next year. To use these, the two extra leads that are not connected for the 4K RAM would become the two most significant address lines. The control signals for such a memory would remain unchanged.

It is important to realize that the data and instruction memories are not only separate but of different widths. The reasons for this somewhat unconventional arrangement are:

1. Separate memories allow for concurrent reading and/or writing of data and instructions. Although a high speed memory (215 nanosecond access, 400 ns cycle time) is used, the CPU cycle time is only 150 ns. Since the memory is the slowest part of the system, it is desir-

able to be able to read or write in the data memory while reading the next instruction from the instruction memory. Performing these functions together can save up to 400 ns from the macroinstruction cycle, depending on the particular instruction.

2. Different size words maximize the efficiency of each memory. It appears that 16 bit precision is sufficient for all data, while being too restricting for the instructions. The extra four bits makes each instruction more powerful and versatile, thus reducing the number of instructions and memory references in a program. The wider word therefore reduces the effect of the relatively slow instruction memory.

   Even faster execution times are obtained and less memory is required when several ordinary assembly language macroinstructions are combined into a single "super" instruction. Super instructions often require more than 16 bits to specify all the operands and conditions.

3. Ultimately when the synthesis program is perfected, the instruction memory will be replaced by a Read Only Memory (ROM). Advantages over a read/write memory include non-volatility, higher

speed, lower cost, and higher bit density per chip. For the time being, the read/write capability allows for testing, debugging, and altering programs.

4. The inability to change the contents of the instruction memory during program execution ensures the use of pure code. Most agree that pure code is easier to understand, debug, and document.

4.4 The Central Processing Unit

The CPU is constructed with four high speed 4-bit bipolar microprocessor slices cascaded to form a 16 bit machine. The basic architecture of the composite machine is shown in Fig. 7.

The processor contains essentially a 16 word by 16 bit two-port RAM and a high speed Arithmetic Logic Unit (ALU) with the associated decoding and shifting capabilities and data paths. In a single machine cycle (which lasts approximately 150 ns) data can be

1. read simultaneously from the A and B ports of the RAM,
2. routed through the ALU, with the option of placing the result on the bus,

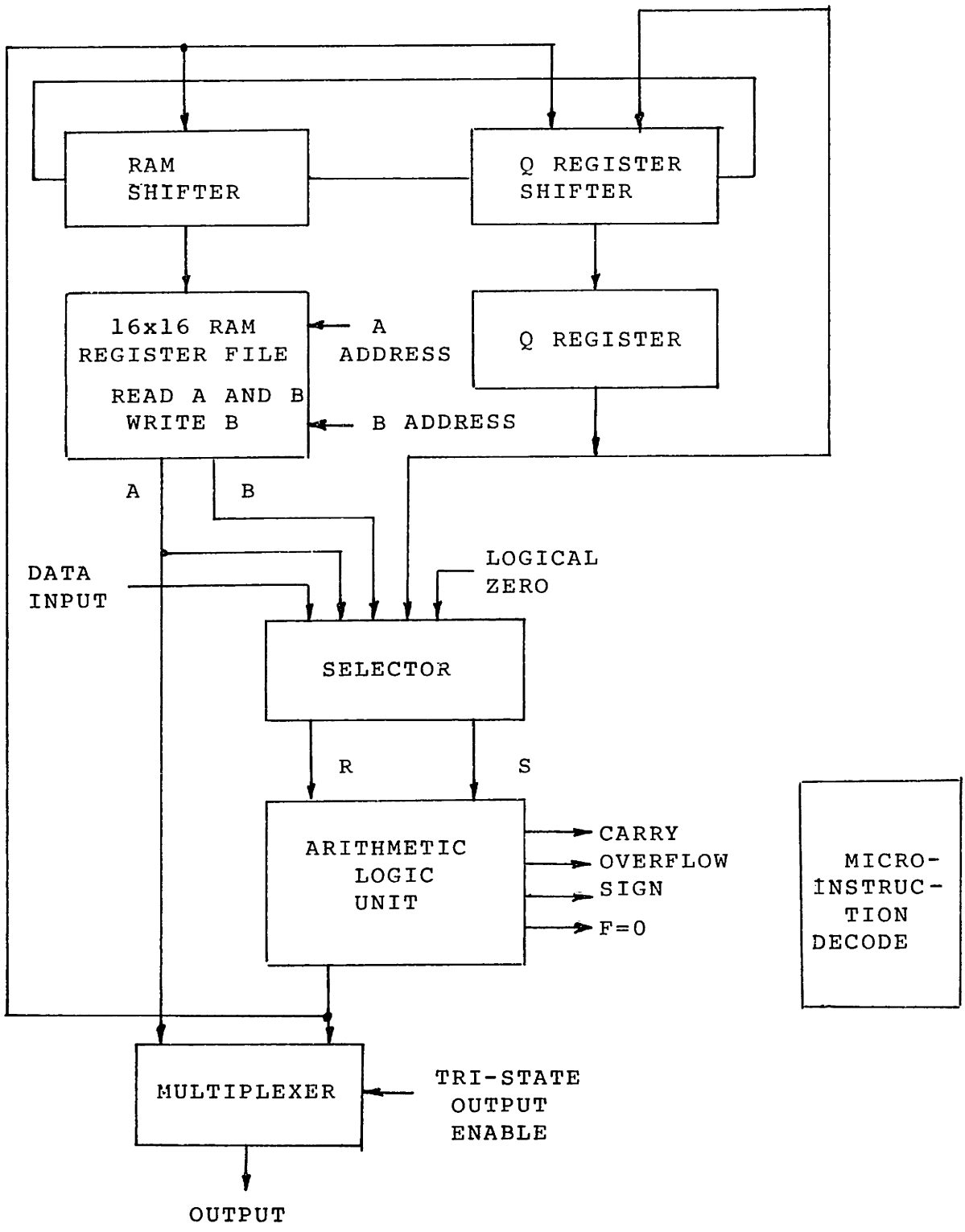


Fig. 7 CPU Architecture

3. shifted one bit to the left or right if desired,
4. and finally be written into the register specified by the B address.

The processor has a two-address architecture. The selector chooses the two source operands simultaneously from the following and passes them to the ALU:

1. the contents of the RAM register specified by the A address
2. the contents of the RAM register specified by the B address
3. data from the input port
4. logical zero
5. the contents of the Q register, a separate register intended primarily for multiplication, division, and shifting.

The ALU is a combinatorial logic circuit that can perform three arithmetic and five logical functions:

1. R + S	5. R AND S
2. S - R	6. $\overline{R}$ AND S
3. R - S	7. R EX-OR S
4. R OR S	8. R EX-NOR S

The ALU output can be routed to several destinations including the RAM, Q register, or the tri-state output.

The data can be shifted left, right, or not at all before being written back into the RAM or Q register.

The CPU architecture is clean and conceptually simple, yet extremely versatile. By controlling the CPU in a proper fashion, virtually any digital process can be performed. Since the sequence of operands and operations is determined by a microprogram resident in the control unit, the functions performed by the computer can be drastically altered by simply changing the control program, while maintaining the same hardware structure.

4.5 Microprogram Control

The Controller orchestrates the functions of all the other blocks: its commands activate data paths and determine the function (if any) to be executed by the ALU. Traditionally the control information is permanently built into the system with dedicated hardwired logic for each macroinsturction. With microprogramming, a technique employed in the processor's control section, the control is implemented in "firmware" which lies conceptually somewhere between hardware and software.

Each macroinstruction is broken down into a number of smaller, more basic steps called microwords, and the execution of a given machine instruction could be though of as the execution of a small program written

in microcode. A microinstruction is the lowest level of instruction the computer can execute. The microinstructions are stored in a high speed 256 x 44 bit Programmable Read Only Memory (PROM). Once fetched from this memory by the microcode sequencer, the microword issues the appropriate signals to units internal and external to the CPU for initiating proper processor action. The microinstructions control the actual transfer of information from one register to another. The path can be either direct, through the ALU or through other logical networks. They specify not only which path will be activated, but also the function performed by the ALU, the address of the next microinstruction, and other special operations such as control of the memory read/write lines. The microinstruction provides exact timing and designates which operations occur simultaneously and which ones occur sequentially.

Microwords tend to be quite wide, often ranging between 20-100 bits. The highest speed and flexibility can be achieved with a microword having one bit for each control line -- a type of coding commonly called horizontal. This allows one to issue many control functions in parallel. Unfortunately, pure horizontal microprogramming requires a very wide word and does not use the microcode memory efficiently.

The microword can be shortened however by encoding mutually exclusive bits. A good example is the control of the ALU, whose eight possible functions can be expressed in three bits. In each machine cycle, only one ALU function is performed, so this type of coding, vertical coding, uses three bits instead of eight. The cost of vertical coding is the expense of the decoder and the time to perform the decoding. Another good application of vertical coding is in the control of the tri-state devices sharing a common bus. Only one device should be enabled at any given time: enabling more than one will short the outputs and destroy the devices. Vertical code ensures that only one device is enabled.

The speech processor has been designed with a 44 bit microword that combines vertical and horizontal coding to exploit the best features of each. (See Fig. 8) Mutually exclusive control functions are vertically coded, whereas all functions that can be performed concurrently are horizontally encoded. Details of the microcode are given in Chapter 5 and Appendix B.

The microcode is stored in PROM's rather than RAM's for several reasons. PROM's are non-volatile: they retain information even when power is removed. This feature eliminates the need to reload the microcode each time the machine is turned on. At current prices,

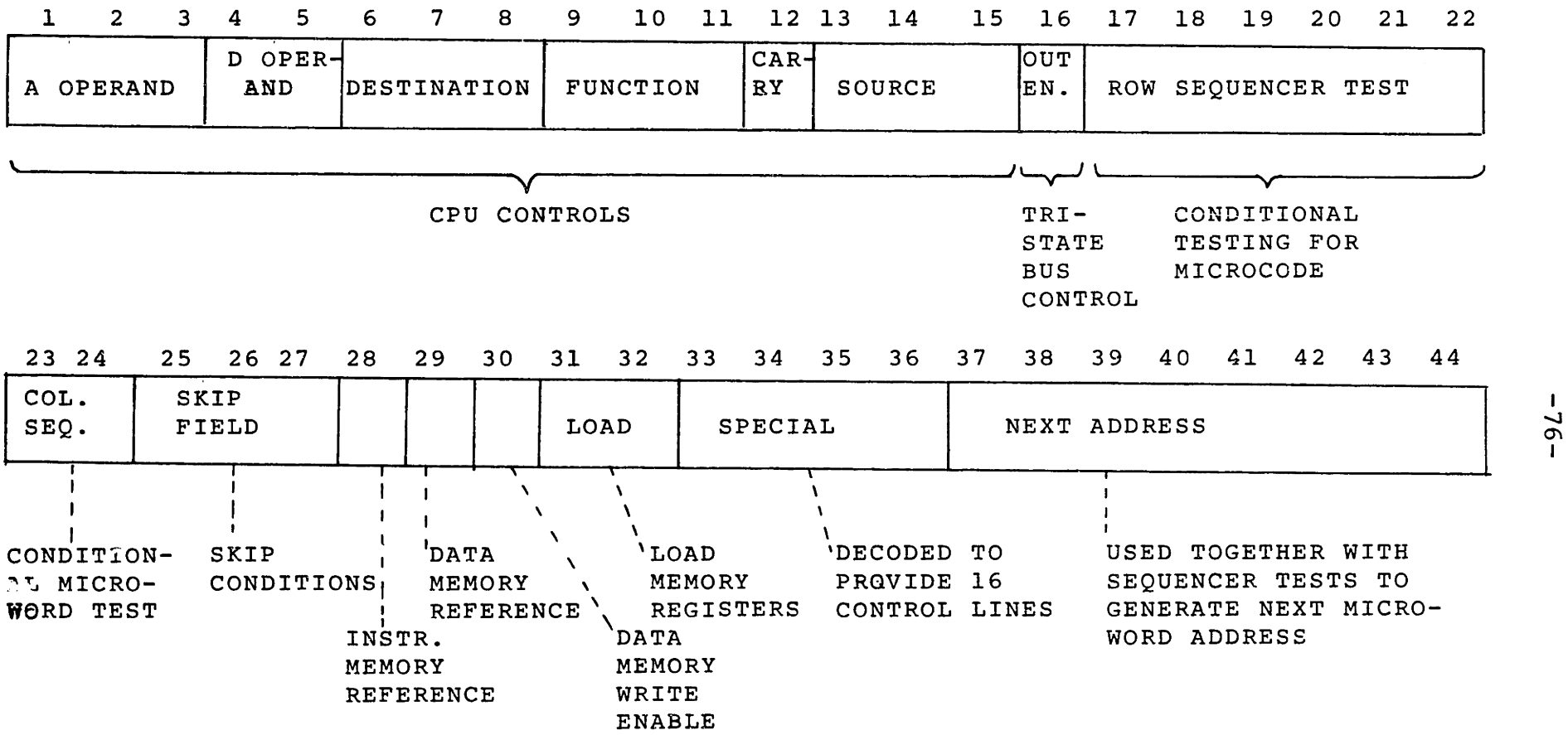


Fig. 8 44 bit microword format

PROM's are over seven times cheaper than comparable RAM's -- a strong incentive to use PROM's. Of course, once PROM's are programmed they cannot be erased (actually erasable PROM's do exist but they are much too slow for the speech processor and they cost more). Since it is not intended to make frequent changes in the microcode, PROM's are the logical choice for the control memory.

The microcode has been designed so that minor changes can be made without having to discard the PROM's. Many of the 256 possible locations are vacant. A microcode sequence can be altered simply by changing the next address field to one of these free locations and inserting the new microword there.

As more involved functions are microcoded to further simplify the task of the programmer, it may be desirable to increase the size of the control memory. The microword can be easily expanded in width by increments of four bits since the control memory is fabricated from 256 x 4 bit PROM chips. The board space and connections for a twelfth chip already exist, and to expand the microword from 44 to 48 bits one must simply insert the extra chip. The microinstruction memory can also be expanded in length; 512 x 4 bit PROM's are available and are pin-for-pin compatible with the ones being used. To

use these longer memories involves the connection of one additional address line.

The sequencing of the microwords is done in an interesting and simple way. Due to the large percentage of microinstructions that require either conditional or unconditional branching, the next address is explicitly designated as part of the current instruction (rather than having a program counter). For unconditional jumps an 8 bit field ($2^8 = 256$) allows the next instruction to be accessed anywhere in the PROM, whereas for conditional branching information from the IR and a variety of other bits are combined with the next address field to determine the next microinstruction. This scheme eliminates the need for explicitly specifying a next address for each condition.

These addresses are generated by replacing a bit (or bits for multi-way transfers) of the next address field with the value of the condition bit(s) under test. Included in the microword is a test field which specifies the conditional test to perform. As a simplified example, consider the circuit to execute one microinstruction if the contents of a flip-flop is ZERO, but the following microinstruction if the flip-flop is ONE (see Fig. 9). When no conditional branching is indicated the next address is obtained unmodified from the next address

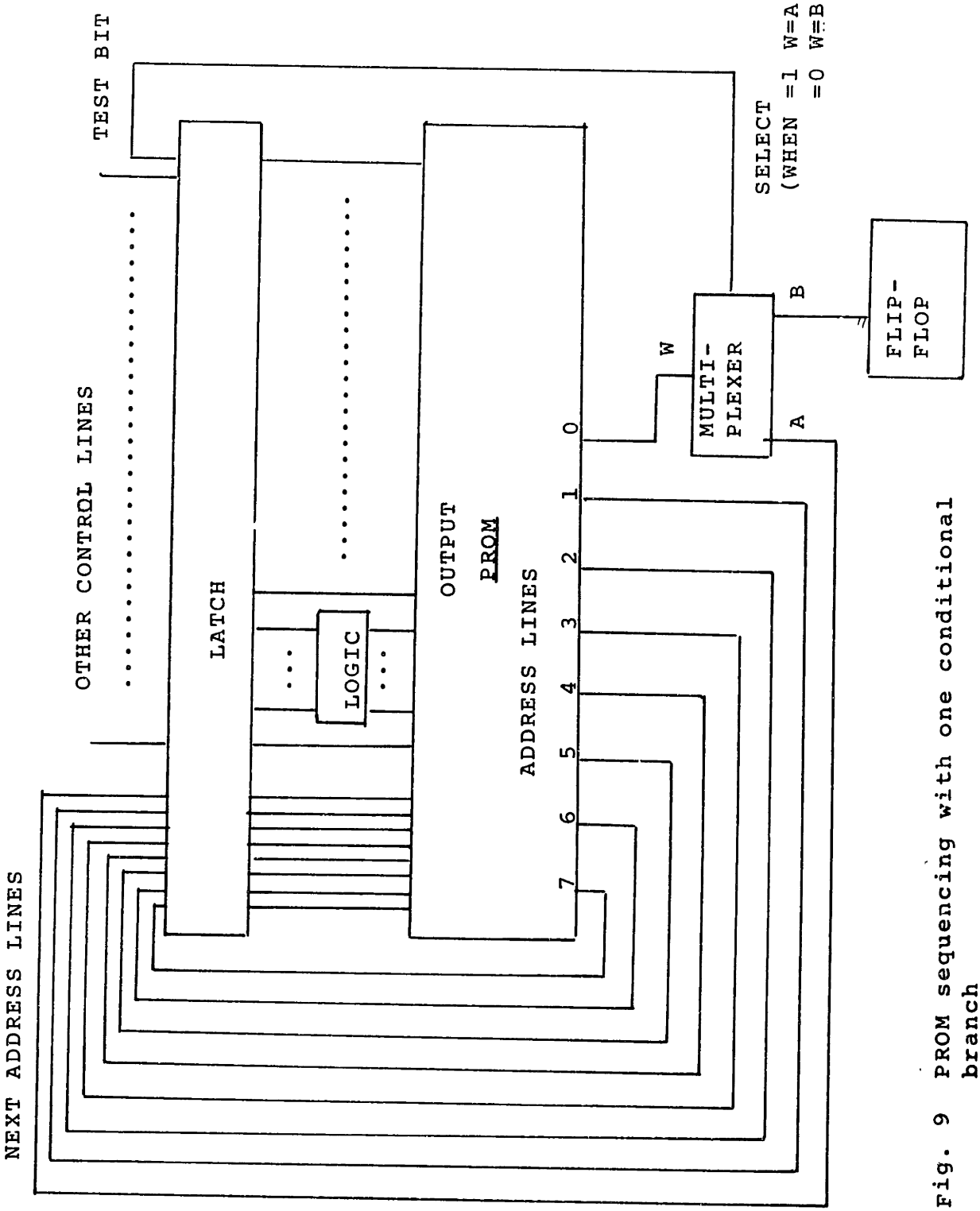


Fig. 9 PROM sequencing with one conditional branch

field. When there is conditional branching the test is enabled by setting the appropriate test bit in the microword. This bit controls the select line of the multiplexer, and when equal to ONE routes the output of the flip-flop directly to the PROM's 0-address line. So for example, if the next address field of the current instruction contains $0101\ 0110_2$, the actual next address will be either $0101\ 0110_2$ or $0101\ 0111_2$, depending on the contents of the flip-flop.

Once the microword has been accessed it can be modified by the logic situated between the PROM outputs and the latches. For example, in Class II instructions, the result is deposited back into a register if the IR deposit bit (bit 3) is set to ONE, and not written back of it is ZERO. If the deposit bit is tested in the manner shown in the previous example, one of two separate microwords would be accessed depending on the outcome of the test. These microwords would be identical except for the bits in the destination field: in each case the function performed is the same. Rather than accessing two microwords, one word can be retrieved and its destination field can be altered depending on the deposit bit. Some simple logic and a test enable signal is all that is required. Conditional modification of microwords is also employed in the shift, multiply, and

divide routines to save slightly under fifty locations in the control memory.

The latch of Fig. 9 serves two purposes. If not for the latch which is clocked sequentially, the addressing circuit would be unstable and would oscillate with a period equal to the address-to-input delay time through the PROM. The latch also allows for overlap fetching of microinstructions, a technique described later.

In actuality, the sequencing scheme is somewhat more elaborate. (See Fig. 10) The eight PROM address lines are divided into two groups of four, where one group can be thought of as a row address (bits 4-7) and the other as a column address (bits 0-3). Rather than appearing as a vector 256 locations long, the microcode now appears as a 16 x 16 matrix. Conditional branching for the row address is similar in concept to the example of Fig. 9, but now two 8-input multiplexers control the PROM's bit 4 and bit 5 address lines. Any one of eight tests can be performed on each bit independently. This requires 6 bits for the row test field (3 bits for each address line) and allows either unconditional, or 2- or 4-way conditional branching. In the 4-way branch two different condition bits can be examined concurrently. Some possible row branches are shown in Fig. 11b.

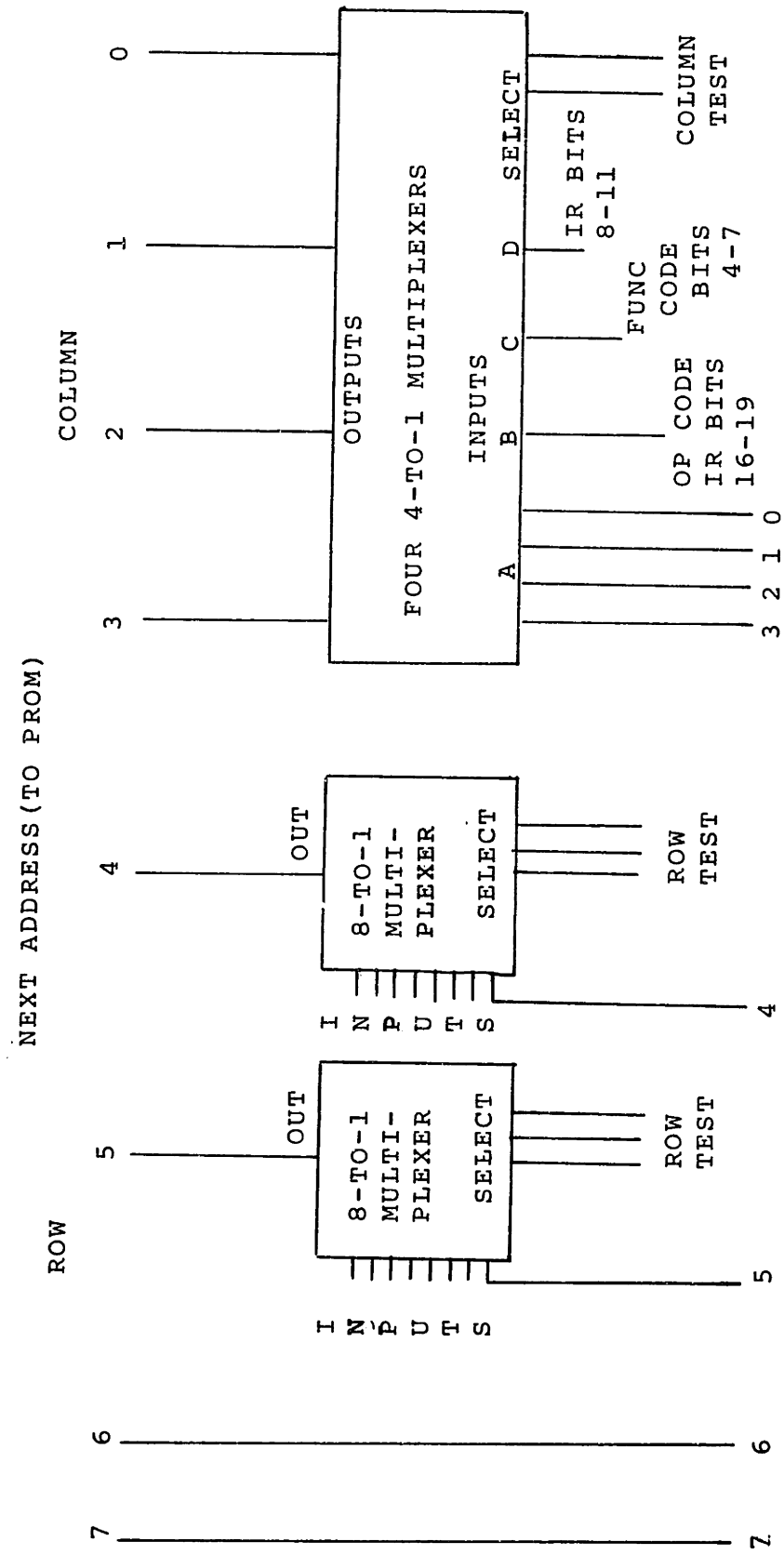
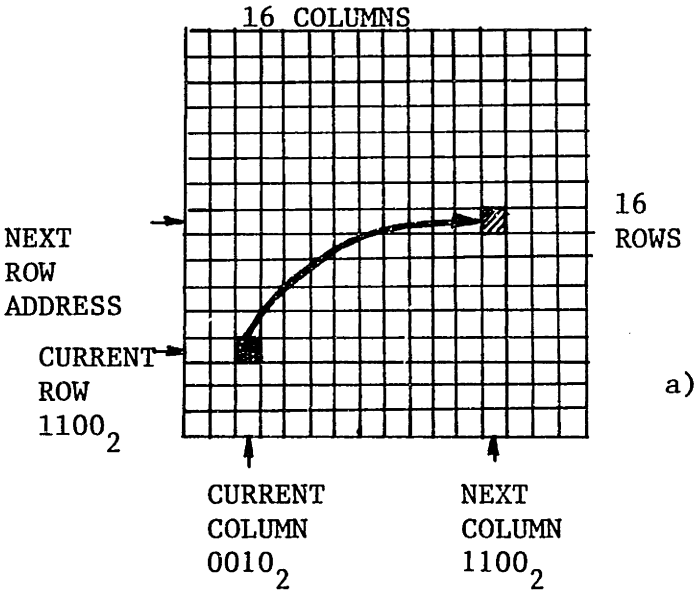
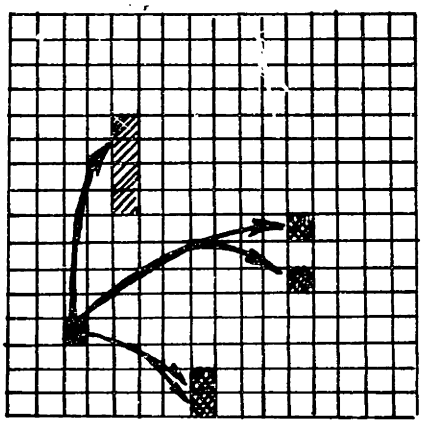


Fig. 10 Actual PROM Sequencer



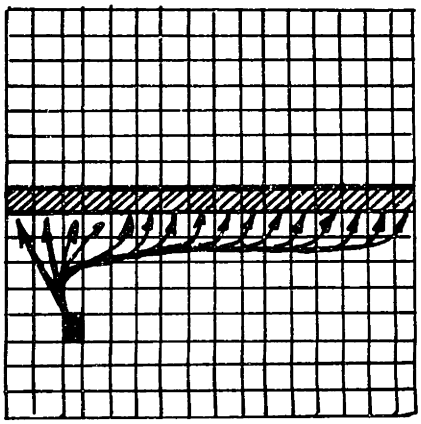
a) Unconditional Branch



b) Conditional Row Branches

4-WAY

2-WAY



c) 16-Way Column Branch

Fig. 11 Microcode Branches. The crosshatched areas are examples of microprogram locations that may be selected as the next address.

The column sequencer consists of four 4-to-1 multiplexers, one for each address line. This provides a method for decoding the OP code. During the OP code dispatch, the 4 bit OP code from the IR becomes the column address. For example, the OP code 0000_2 causes the zeroth column to be selected, 0001_2 the first column, and so forth. In the case of a Class II instruction, a second 4 bit FUNC field serves as the OP code, and can be routed from the IR to the PROM. Bits 8-11 from the IR can also be selected as input to the multiplexer in the special case of shift instruction. Finally, the fourth possible source of the column address is the unmodified next address field. The 16-way column branch is illustrated in Fig. 11c. Note that row and column branches could be combined for 32- or 64-way conditional branches.

Using the configuration of Fig. 6 together with the proper microcode almost any digital machine can be emulated efficiently. Simply by rewriting the microcode, the macro instruction set can be drastically altered with only minor changes in hardware: the entire system need not be redesigned. For example, provision for floating point computations could be added using existing hardware and a new sequence of microinstructions, without special floating point equipment. The inherent

flexibility of a microprogrammable machine means that the detailed microcode need not be written until late in the design process; the system and microcode development can proceed together. Specialized, more complex macroinstructions can be developed and microcoded to replace several ordinary instructions or subroutines with a substantial improvement in execution time.

The advantages of modularity and flexibility afforded by microcoding cannot be overemphasized. The processor which is described here is very versatile and its software can be easily updated to meet future requirements.

4.6 Parallelism

Techniques such as pipelining and parallel processing have been used throughout the machine to assure high-speed operation. Pipelining involves splitting a process into several sequential tasks and subdividing the work over a number of concurrently operating units. The concept of pipelining can be best explained with an example. Suppose a particular task can be subdivided into two distinct sequential processes, P1 and P2, both requiring 1 unit of execution time. Assuming only one process can be performed at a given time, it takes 2 units of time to process both pieces of data as shown

in Fig. 12a.

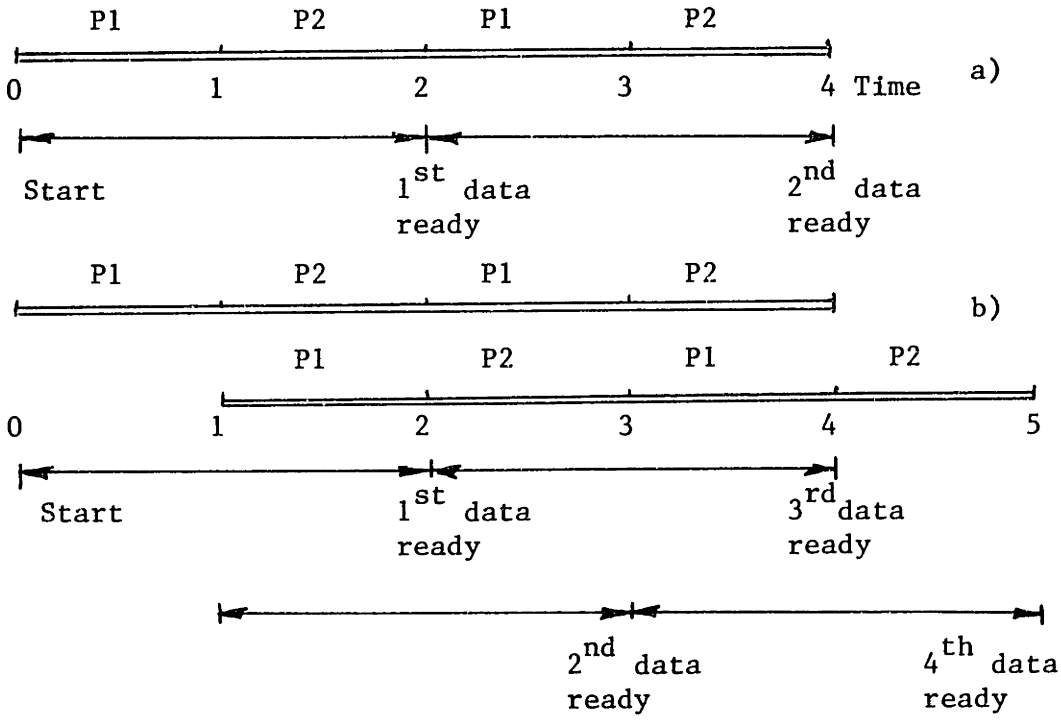
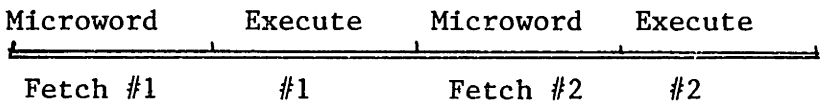


Fig. 12 Simple example of pipelining.

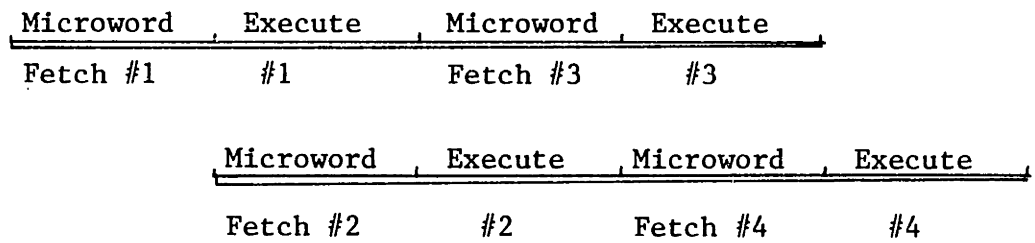
In Fig. 12b both processes are performed concurrently. Overall execution proceeds in an assembly line fashion. Each piece of data takes 2 units to process as before, but notice that once the pipe is started, the throughput is twice as fast. Pipelining is most efficient when the processes each take approximately the same amount of time, and the pipe is kept full.

In the speech processor, both micro and macro

instructions are overlap-fetched or piplined. Fig. 13 shows how the next microword fetch is performed concurrently with the execution of the current microword. The pipelined system operates twice as fast as the ordinary approach. The additional hardware consists merely of a latch to store the current microword while the next one is being accessed.



a) Serial processing



b) Pipelining. Microinstrucitons are overlap fetched during the CPU execurion cyle.

Fig. 13 Overlap fetching of microinstructions.

Macroinstructions are overlap-fetched from the instruction memory: the next instruction is accessed while the current one is being executed. The saving in

execution time for both a typical Register-to-Register instruction (RR) and a Memory-Reference instruction (MR) is one cycle as shown in Fig. 14b. Notice that the memory access time (the time required to read valid data) is 2 microcycles, whereas the cycle time (the time before another memory reference can be made) is 3 microcycles. Instruction overlapping minimizes the effects of a slower instruction memory. When a skip or jump is encountered in an overlapped section, the next instruction which is being fetched is incorrect. The correct instruction can be accessed by updating the PC, performing an additional fetch, and then returning to the normal sequence for a total of three extra steps (see Fig. 14c). The only extra hardware to allow for instruction overlapping is the IR. If the processing were strictly serial, the instruction at the output of the instruction memory would not change during execution and would not have to be latched.

Fig. 14d shows how concurrent memory referencing from the instruction and data memories saves another microcycle for MR instructions.

The average execution time for an ordinary RR macroinstruction (made up of a sequence of macroinstructions) using overlapping is dependent upon the number of skips or jumps:

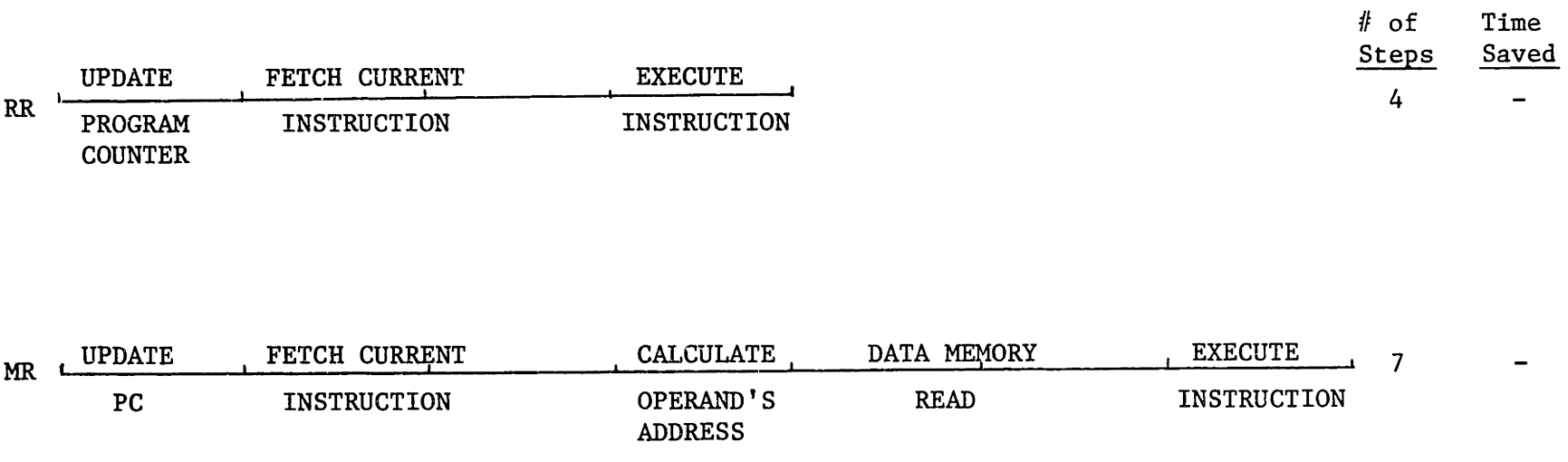
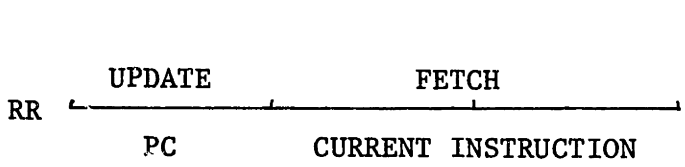
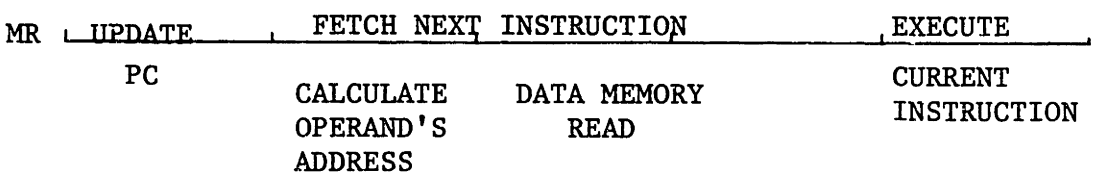


Fig. 14 Effects of parallel processing on macroinstruction execution speed



	# of Steps	Time Saved
RR	3	-

c) Correction cycle for overlap fetching for skip or jump



	# of Steps	Time Saved
MR	5	29%

d) Overlap fetching and concurrent memory referencing

Fig. 14 (Continued)

$$\text{Average RR Execution Time} = \left\{\begin{array}{c}\#\text{ of microcycles in sequence} \\ \text{without branches}\end{array}\right\}$$

$$+\left\{\begin{array}{c}\#\text{ of cycles in correction} \\ \text{sequence}\end{array}\right\} \cdot \left\{\begin{array}{c}\#\text{ of skips or jumps} \\ \underline{\text{actually performed}} \\ \text{total \# of macro-} \\ \text{instructions}\end{array}\right\}$$

To be worth implementing, this time must be less than the RR execution time for serial processing. Replacing the variables with actual numbers:

$$\text{Average RR Execution Time} = 3 + 3 \left\{\begin{array}{c}\text{fraction of skips or} \\ \text{jumps in program}\end{array}\right\}$$

As a very rough estimate, 20% of the instructions involve some type of branching. Assuming a two-way conditional branch where a jump or a skip is actually performed 50% of the time, the fraction in the above equation equals 0.1. Statistically speaking, RR average execution time equals $\sim$3.3 microcycles. Instruction overlapping, in this case, saves approximately 18% in execution time.

For an ordinary MR instruction with overlapping and concurrent memory referencing, average execution time can be calculated in a similar way:

$$\text{Average MR Execution Time} = 5 + 3\ (0.1) = 5.3$$

Using parallel techniques saves almost 25% in execution time.

CHAPTER 5

Algorithms and Microcode

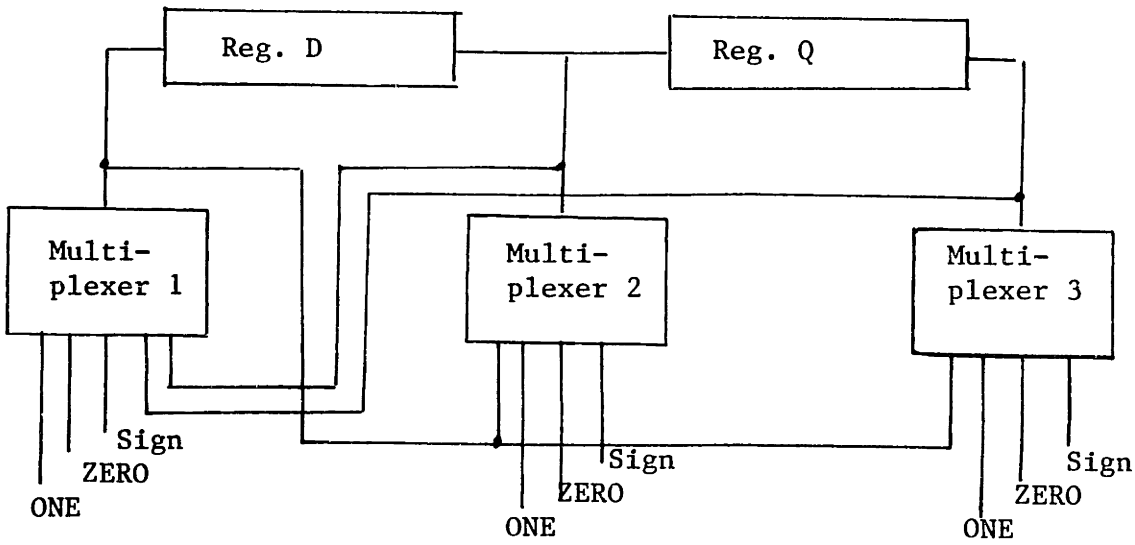
Certain functions of the speech synthesis computer are implemented directly in hardware, but the majority are implemented in microcode. Microprogrammable machines have shifted the emphasis in computer design from pure hardware to the development of microprogram algorithms.

Data flow graphs for the various functions performed by the machine and a detailed description of the microword are included in Appendix B. In the data flow graphs the boxes correspond directly to individual microinstructions and the arrows indicate the sequence in which the operations are performed. Most of the algorithms are self-explanatory, but the ones for shifting, multiplying, and dividing are described below.

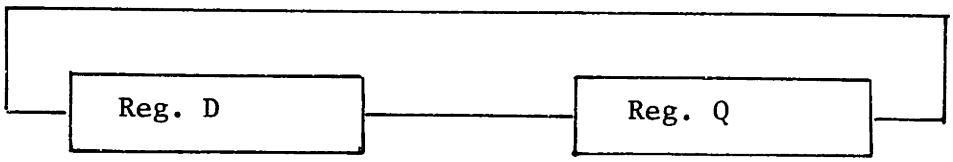
5.1 SHIFT

The SHIFT instruction provides for both single and double word shifts, where the bits shifted into a register can be either: logical ONE, logical ZERO, the sign bit of the chosen register, or the bits shifted out (this is a rotate). The conventional approach involves the use of tri-state multiplexers to select the proper bit to

be shifted into the vacated bits (see Fig. 15A). In the case of any right shift, for example multiplexer 1 is enabled and the others must be in the high impedance state.



a) Conventional shift logic. Not shown are the multiplexer select and output enable lines.



b) In the speech processor all shifts are implemented as a double register rotate.

Fig. 15 Shift logic

In the case of rotate, the shifted bit must pass through a multiplexer. Even if high-speed Schottky logic is used, the propagation delay is approximately 15 ns and unless a variable speed system clock is used, the en-

tire processor must be slowed for this one case of rotate.

A completely different approach has been devised to eliminate the multiplexers altogether, with a corresponding 10% overall speed increase. The algorithm performs all shifts as a double register rotate using any one of the working registers in the CPU register file together with the Q register (see Fig. 15B) To effect a single register shift of register D, register Q is first loaded with one of four possible bit patterns to be shifted into register D as illustrated:

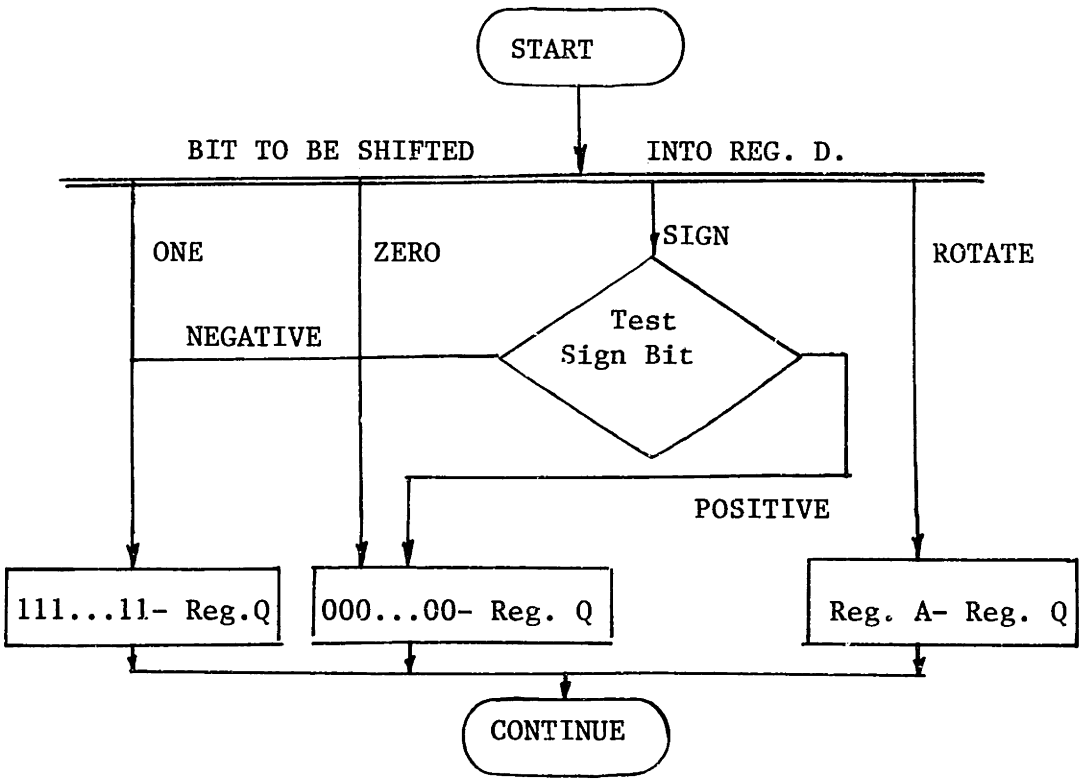
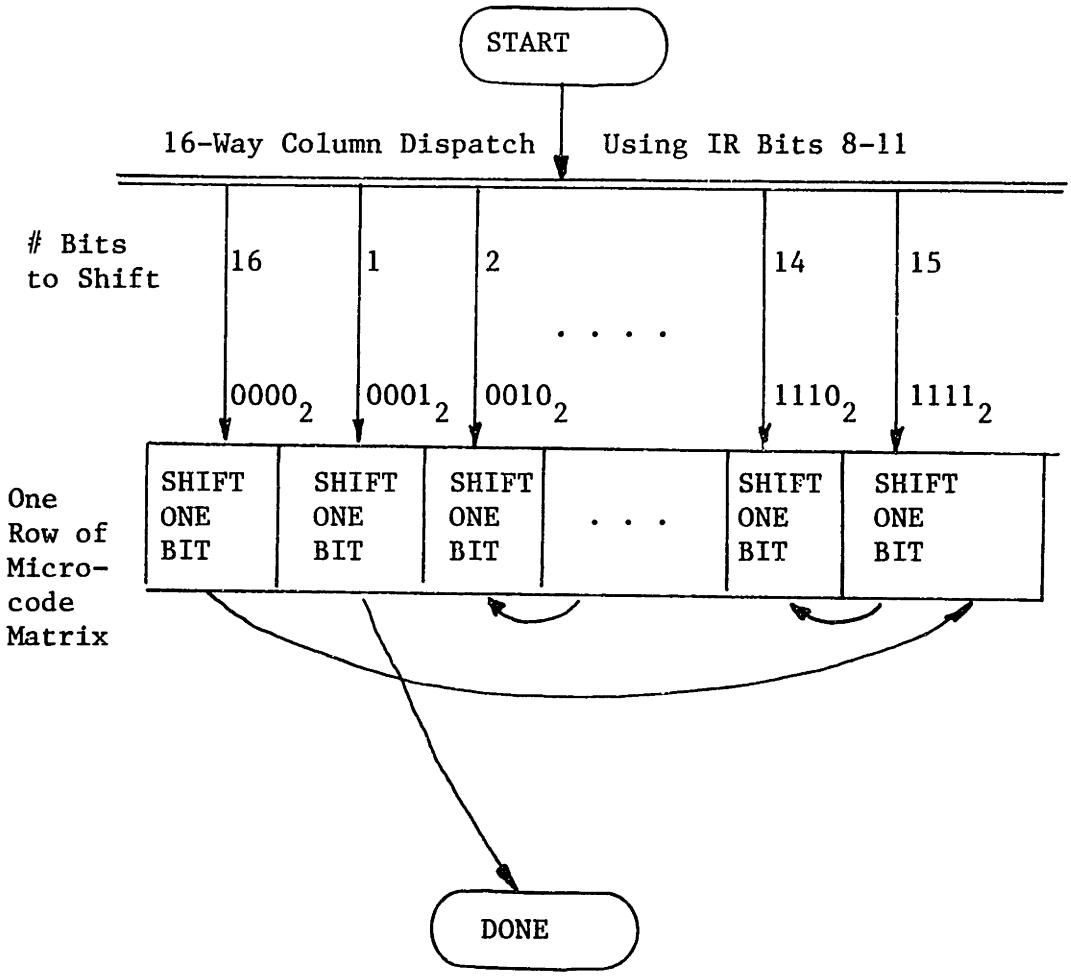


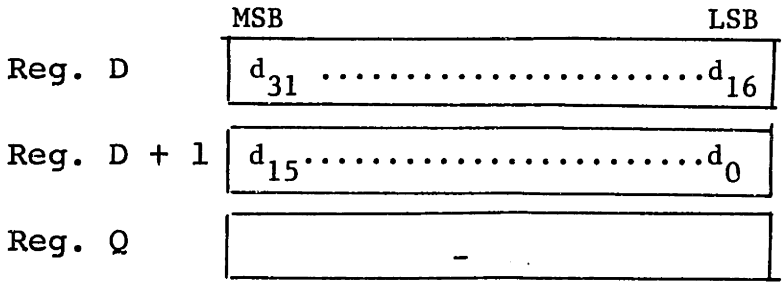
Fig. 16 Initialization of register Q for shift

Once register Q is loaded, both registers are shifted either right or left up to 16 positions as determined by the microinstruction. The four bits from the A register field (IR bits 8-11) specify the number of positions to shift. They are used in the microcode to produce a 16-way conditional branch as shown below:

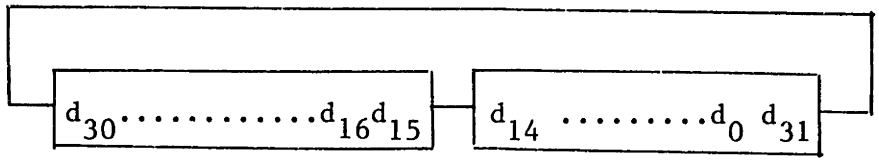


Each microinstruction in the given row shifts Reg. D and Reg. Q one bit and points to the next microinstruction to be executed. Multiple shifts are performed by executing the number of microinstructions specified by IR bits 8-11. For example to shift two places, the first shift is performed by the microinstruction in column 0010_2 and the second by the microinstruction in column 0001_2. This type of algorithm occupies 16 positions in the control memory but eliminates the need for a separate counter. The hardware is simplified and the speed is increased.

Double shifts are somewhat more involved. They could be implemented easily if an extra register is added to Fig. 15B and all three could be shifted together in a manner analogous to the single shift example. However, the CPU allows only for 2 register shifts. The algorithm is best described with a specific example. Given registers D and D + 1 with contents d, perform a double left shift inserting zeros in the LSB of register D + 1. Starting with:

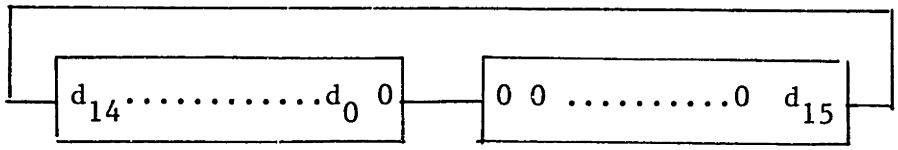


Register D + 1 is copied into register Q and shifted left 1 bit with register D:



Reg. D Reg. Q

Register D now contains the correct result. Generation of the correct bits for register D + 1 is accomplished by setting register Q to zero and now shifting with register D + 1:



Reg. D Reg. Q

The elimination of the shift multiplexers and the corresponding increase in the processor speed has been made at the expense of added complexity in the SHIFT rountine. This can be justified by the relatively infrequent usage of the SHIFT instruction. The microprogram algorithm is not only responsible for reducing the hardware, but increasing overall processor performance.

5.2 MULTIPLICATION

All multiplication can be performed as repeated

addition. The example below makes this clear:

0110	multiplicand	$5_{10} = 0101_2$
x 0101	multiplier	
0110		$6_{10} = 0110_2$
0000	partial	
0110	products	$5 \times 6 = 30_{10}$
0000		
0011110	product	$= 0001\ 1110_2$

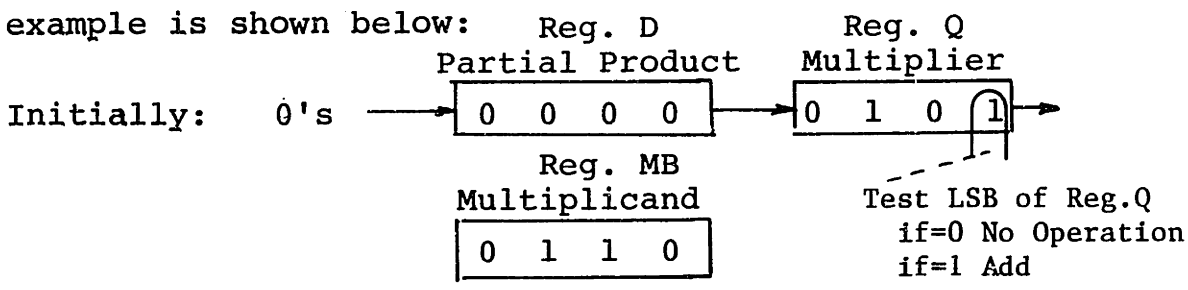
Note that the product of two 4-bit numbers (each with one sign bit and three magnitude bits) is only seven bits long (one sign bit and 6 magnitude bits). The determination of each partial product is based on the multiplier bit being used to generate that partial product. If the multiplier bit is:

ONE, the multiplicand is added into the partial product

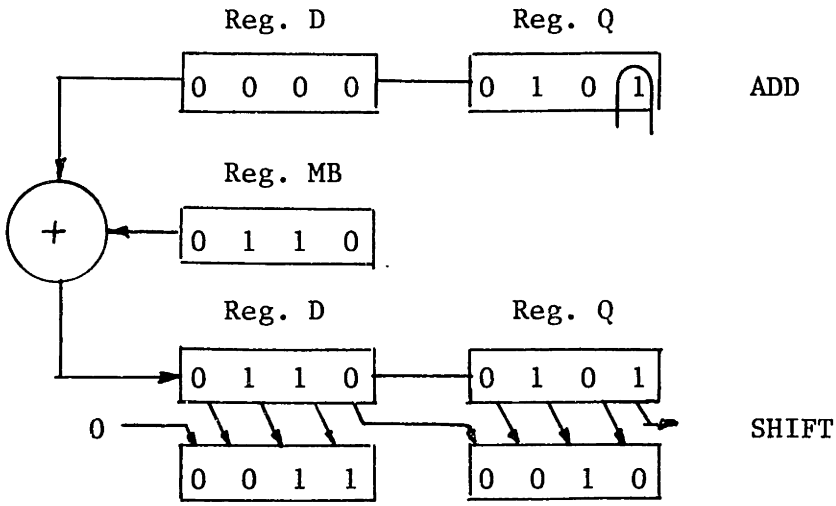
ZERO, the addition is suppressed.

Rather than summing all the partial products as the last step, they can be summed two at a time as they are generated; each partial product is added to the sum of the previous partial products.

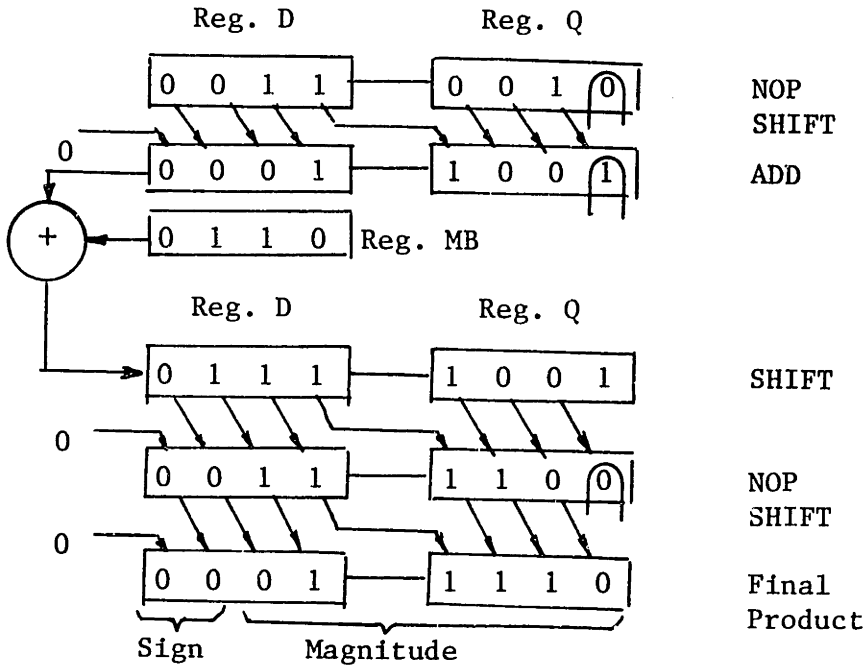
A possible register implementation of the above example is shown below:



After testing the LSB of register Q (= 1), the multiplicand is added to the partial product (initially = 0). Register D and register Q are now shifted together one place to the right with ZERO inserted to produce:



The partial product has been shifted to be in proper position for the next cycle. The MSB of register Q at the moment is ZERO and corresponds to the LSB of the final product, thus will not be altered by further partial product summations. The three subsequent add and shift cycles are:



By convention, the sign of the product is duplicated in its two most significant bits.

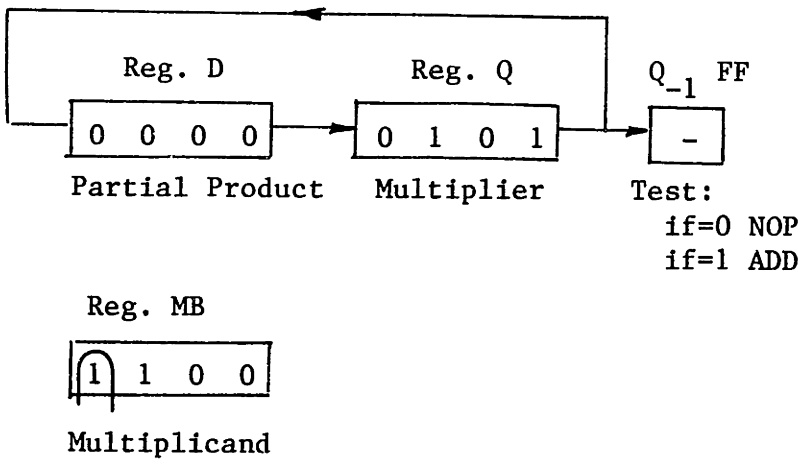
An algorithm for two's complemented negative numbers is similar to the example above but requires that the bit shifted into the partial product is the sign bit of the partial product. A correction cycle is necessary when the multiplier is negative.

The actual hardware contains the registers shown above, but as mentioned in connection with SHIFT, the LSB of register Q is wired directly to the MSB of register D. Unfortunately, the insertion of any logic in this

path acts to slow the entire microprocessor significantly. Thus at first it seems that the algorithm for multiply cannot be made to work properly with the multiplier bit being shifted around into the partial product.

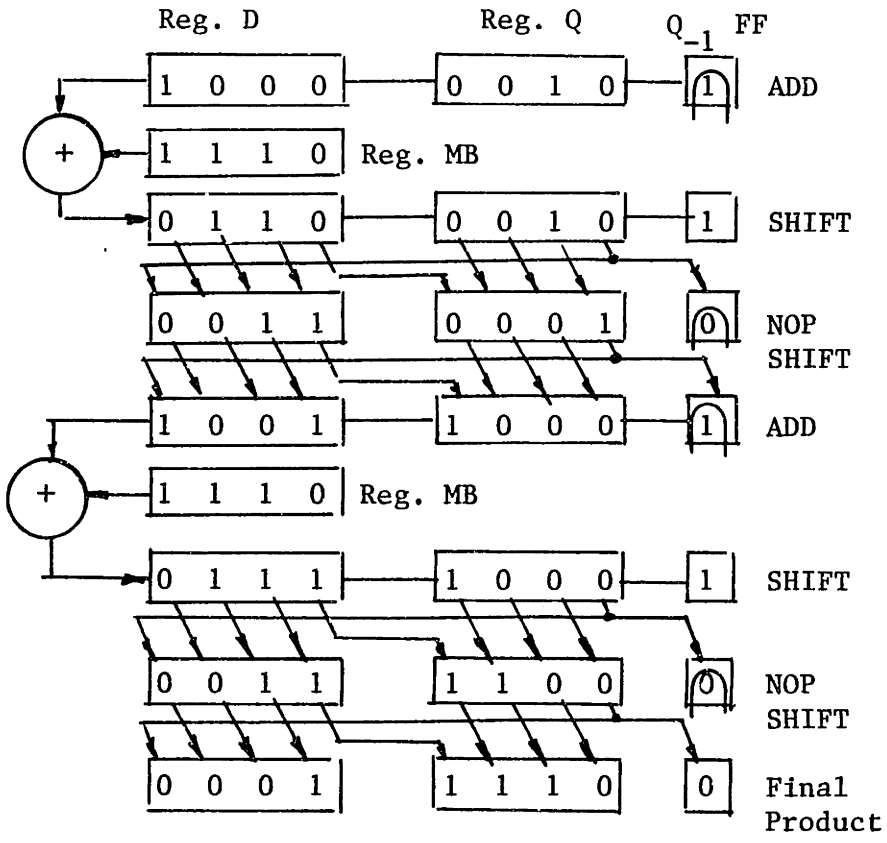
An interesting algorithm, however, has been developed to circumvent this problem. The algorithm works only for positive numbers, so it is assumed that the multilier and multiplicand are made positive, and the sign of the multiplicand and product is restored at the end. This function is handled by the microcode and is not the concern of the programmer.

Modifications to the previous example include an extra flip-flop (Q_{-1} FF) and the setting of the MSB of the multiplicand to ONE:



The first step now is a double register right rotate to move the LSB of the multiplier into the Q_{-1} FF where the conditional test for addition is performed. Four more

add/shift cycles are repeated in an analogous fashion to the previous example. By setting the MSB of the multiplicand to ONE, all ONE's shifted into the partial product are converted to ZERO's after addition; ZERO's that are shifted into register D are unchanged. The overall effect is to make a double right rotate appear as a double right shift zero as required by the first algorithm.



5.3 Division

This section covers two basic methods for signed binry division: restoring division and nonrestoring division. For both algorithms the binary point is located just to the right of the sign bit. To prevent overflow, the absolute value of the dividend must be less than the absolute value of the divisor. Since division is fixed point, it may be necessary for the programmer to scale the operands to maintain sufficient precision.

Restoring division closely follows binary long division. A flowchart is shown in Fig. 17. In the example below of restoring division the dividend, $+13_{10} \times 2^{-6}$, is divided by a four bit divisor, $+5_{10} \times 2^{-3}$, to produce a four bit quotient and remainder:

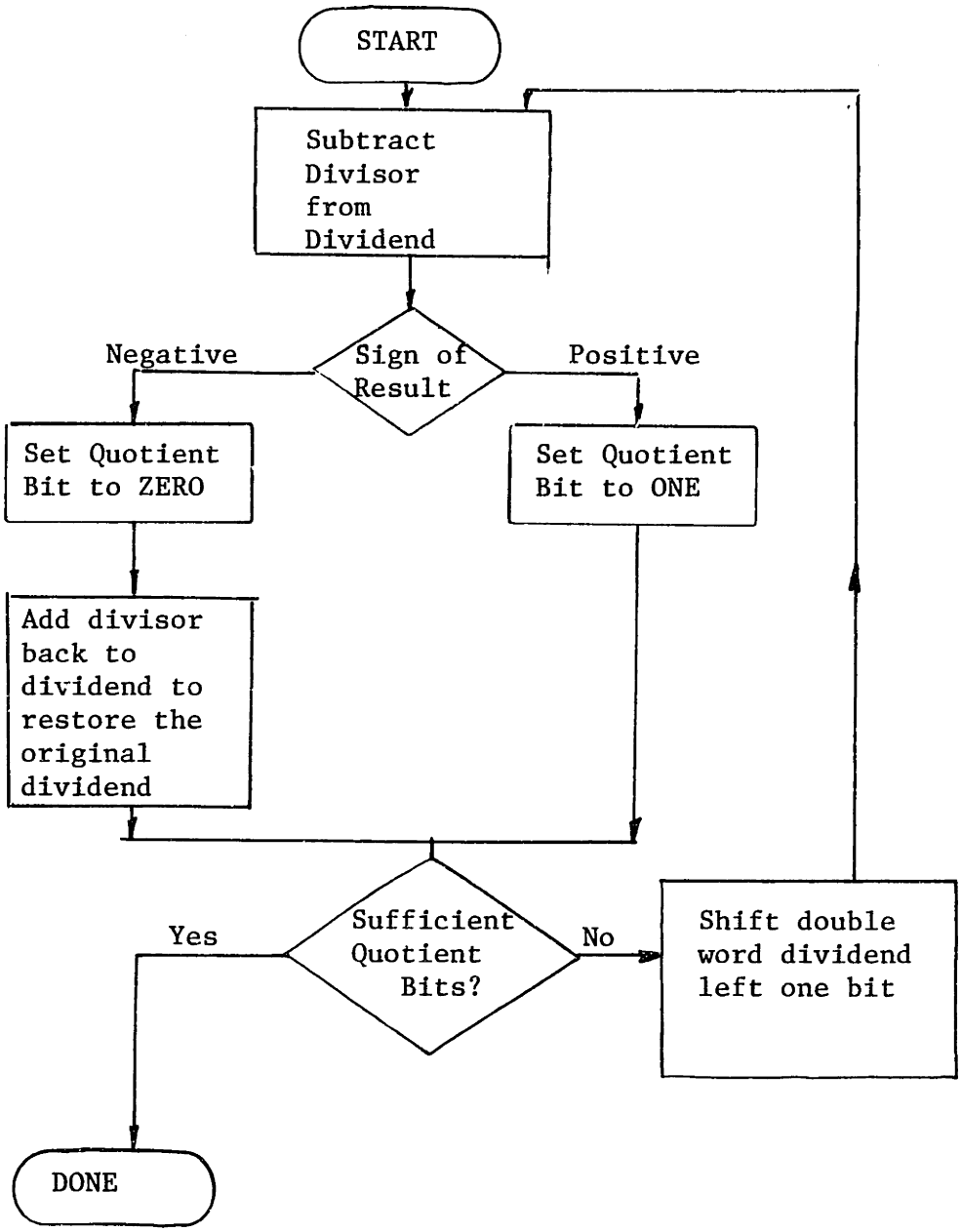


Fig. 17 Flowchart for restoring division of positive numbers

$+13_{10} = 1101_2$

$+5_{10} = 0101_2$

$-5_{10} = 1011_2$

$$\frac{13 \times 2^{-6}}{5 \times 2^{-3}} = 2 \times 2^{-3} + 3 \times 2^{-3}$$

Quotient Remainder

$Q_0\ Q_{-1} Q_{-2} Q_{-3}$

```
          0. 0  1  0
      ------------------
0. 1  0  1 )0. 0  0  1  1  0  1  -
```

+1 0 1 1	Subtract 5 (Add 5)
(1) 1 0 0	Negative, set Q_0=0
+0 1 0 1	Restore dividend
0 0 0 1 1 0 1 -	Shift
0 0 1 1 0 1 -	
+1 0 1 1	Subtract 5
(1) 1 1 0	Negative, set Q_1=0
+0 1 0 1	Restore
0 0 1 1 0 1 -	Shift
0 1 1 0 1 -	
+1 0 1 1	Subtract 5
(0) 0 0 1 1 -	Positive, Q_{-2}=1
0 0 1 1 -	Shift
+1 0 1 1	Subtract 5
(1) 1 1 0 -	Negative, Q_{-3}=0
+0 1 0 1	Restore
0 0 1 1	Remainder

Non-restoring division is similar but the step of restoring the dividend is eliminated. The same results are generated with fewer steps. The process is best illustrated with the flowchart in Fig. 18. For positive partial products the two methods perform the same steps, but in non-restoring division with negative partial products, the steps of restoring, shifting, and subtracting are replaced simply by shifting and adding. The effects of these steps are shown below:

restoring	non-restoring
1. restore - add divisor (DVR) to negative partial product (NPP)	1. shift dividend generating $\frac{DVR}{2}$
2. shift - shifting the dividend (DVD) to the left has the same effect as dividing the divisor by 2	2. add divisor to the partial product
3. subtract the divisor from the partial product	
new partial dividend $= NPP + DVR - \frac{DVR}{2}$	$= NPP + \frac{DVR}{2}$
$= NPP + \frac{DVR}{2}$	

The same example using the non-restoring algorithm looks like:

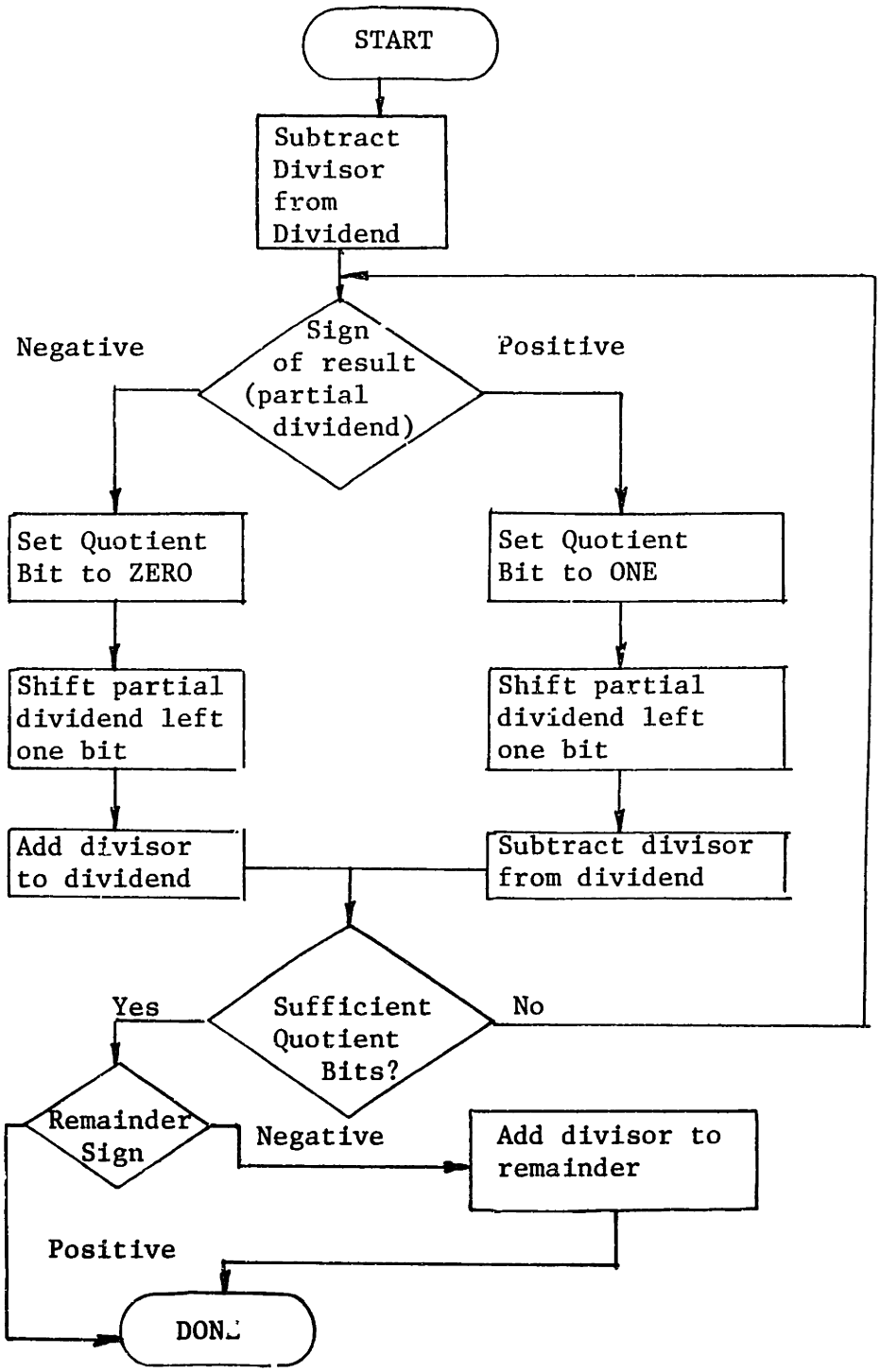
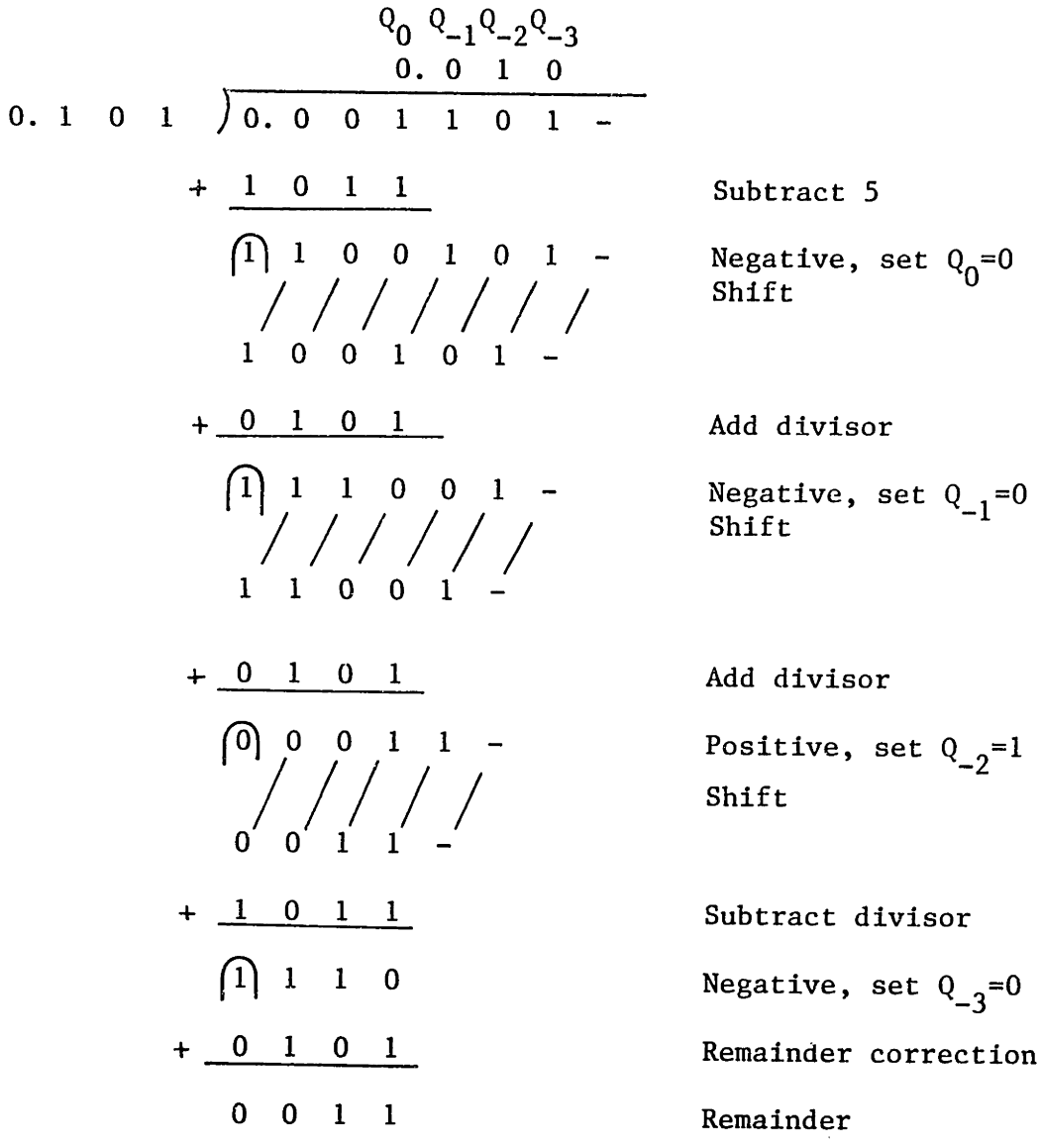
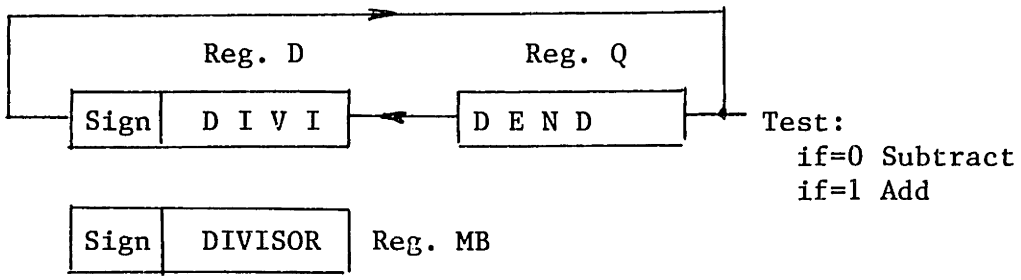


Fig. 18 Flowchart for nonrestoring division of positive numbers


$Q_0\ Q_{-1}Q_{-2}Q_{-3}$
0. 0 1 0
0. 1 0 1) 0. 0 0 1 1 0 1 -
+ 1 0 1 1
Subtract 5
1 1 0 0 1 0 1 -
Negative, set Q_0=0
Shift
1 0 0 1 0 1 -
+ 0 1 0 1
Add divisor
1 1 1 0 0 1 -
Negative, set Q_{-1}=0
Shift
1 1 0 0 1 -
+ 0 1 0 1
Add divisor
0 0 0 1 1 -
Positive, set Q_{-2}=1
Shift
0 0 1 1 -
+ 1 0 1 1
Subtract divisor
1 1 1 0
Negative, set Q_{-3}=0
+ 0 1 0 1
Remainder correction
0 0 1 1
Remainder

The actual implementation uses non-restoring division for its higher execution speed. The registers are initialized as follows:

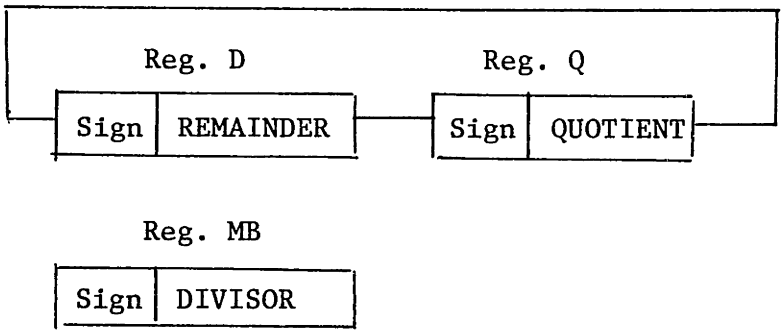


The divisor is subtracted from the dividend according to the flowchart in Fig. 18. Assuming the divisor and dividend are positive numbers, when the sign of the result (partial dividend) is positive, the quotient bit should be set to ZERO. Since the sign bit of the partial product (positive = 0, negative = 1) is passed directly to the LSB of the Q register during the left shift, the ones's complement of the quotient is formed in the Q register. To obtain the proper answer after 16 quotient bits have been generated, the contents of the Q register are one's complemented.

The signs of the remainder and quotient are determined from the following table:

original dividend	original divisor	quotient	remainder
+	+	+	+
+	-	-	+
-	+	-	+
-	-	+	-

Testing the sign of the partial dividend, setting the quotient bit, shifting left one bit, and finally subtracting or adding are all performed in a single microinstruction. After 16 reduction steps, and the appropriate remainder and quotient corrections, the registers contain:



CHAPTER 6

Hardware Implementation

The computer has been built and tested. Complete schematics are included in Appendix A. At the time of this writing, the microcode was being tested by entering each microword manually through the front panel. Provisions are now being made to "burn in" the PROM's so that full speed testing can be performed.

6.1 Logic Families

The machine uses a variety of logic families including Low power Schottky Transistor-Transistor Logic (LS/TTL), Schottky TTL and NMOS with miniaturization ranging from Small to Large Scale Integration (SSI - LSI). Low power Schottky (LS) logic has been used wherever possible primarily for its combined high speed and low power. It requires one-fifth the power of standard TTL and therefore reduces the cost, size, and weight of the power supplies, and eliminates the need for forced air cooling. Lower power implies increased packing density and enhanced reliability. LS follows the same wiring rules as standard TTL and can be directly interfaced with other TTL types.

In sections where the propagation delay through the device affected a critical timing path, regular Schottky logic was used. Schottky logic is almost twice as fast as LS, but uses ten times more power and must follow stricter wiring rules. Since the rise time is 2 - 3 times steeper than standard TTL, ringing and noise are more prevalent with Schottky logic and wires must be kept short and close to the ground plane. Every chip must have a decoupling capacitor. All registers that drive the bus are Schottky for reasons of speed and driving capability. The Schottky tri-state outputs have a high fanout that make bus drivers unnecessary. Special bus drivers would have been required had LS registers been used. The microprocessors achieve their high speed through the use of Schottky LSI. Emitter-Coupled-Logic (ECL) was considered for its high speed but was not used based on its high power dissipation and incompatibility with TTL.

The memories are 4K x 1 statis NMOS RAM's. Memories of this size are not currently available in other logic families. NMOS seems to be well suited for large memory chips due to its high packing density and low power consumption. NMOS unfortunately requires three supply voltages: +12, +5, and -5, and is fundamentally slower than bipolar devices.

6.2 CPU Implementation

A wide range of alternatives was considered in the choice of the CPU. The major design constraints called for high speed operation, versatility, and low cost. Until recently, random logic combined with some Medium Scale Integration (MSI) devices (e.g. ALU's and registers) would have been the only reasonable choice.

During the last two or three years there has been a great increase in the use of microprocessors as a replacement for discrete logic. Complete computers on a chip have been developed, most notably the Intel 8080, an 8-bit machine. The 8080 uses NMOS technology to achieve a 2 microsecond cycle for its simplest instructions. This cycle time does not include any reference to memory, and it assumes the instruction has already been fetched.

After careful analysis of the requirements of Klatt's algorithm it became clear that the 8080 was not suitable. The machine is much too slow for real-time processing of speech. This is due not only to the relatively slow cycle time, but also to the fixed instruction set. The computer cannot be tailored to the specific functions performed in speech synthesis. The other dominant drawback is the size of the data word: 8 bits does not provide the precision demanded by the speech programs.

More recently, bit-slice microprocessors have been made commercially available. They essentially contain an ALU, registers, and data paths for either 2 or 4 bits. Any number of these slices can be cascaded to create a larger machine. Bit-slice machines generally do not have a fixed instruction set and must be microprogrammed.

The first bit-slice machine considered was the Intel 3001. It is a 2-bit slice single address machine with 11 general purpose registers. Such a processor uses Schottky bipolar LSI technology and is capable of typical register-to-register add cycle times of less than 125 nanoseconds (fifteen times faster than the 8080).

The architecture of the Monolithic Memories 6701 seemed better suited to the task. The 6701 is a 4-bit slice two address machine with 17 internal registers. Being a 4-bit slice, only four microprocessors are needed to configure a 16-bit machine. The 6701's cycle time is 200 nanoseconds, considerably slower than the Intel 3001.

In mid-1975, Advanced Micro Devices introduced the 2901, a 4-bit slice machine with an architecture closely resembling the 6701. The Am 2901, however, has a cycle time of 100 nanoseconds, draws half the power of the 6701, and costs less. It was finally chosen as the most

attractive solution. The current cost in quantities of 100 is $30 and the projected cost two years hence is $15. The overall architecture is straightforward and flexible. It is likely that faster versions incorporating the same architecture will soon be introduced. Motorola has just announced the MC10800, an ECL microprocessor slice with a 55 nanosecond cycle and similar architecture to the Am 2901. As faster, pin-for-pin compatible chips are developed, the performance of the speech processor can be improved simply by replacing the old chips with the newer versions.

6.3 Selection of the Memory Chip

The SEMI 4200 4096 x 1 bit NMOS RAM's were chosen over numerous other memory chips for several reasons. These chips have a fully static memory cell which eliminates the need for the refresh circuitry found in dynamic memories. The memory timing and control circuitry is thereby simplified. At the time the chips were purchased, the SEMI 4200 were the fastest 4K static RAM's commercially available. They have an access time of 215 nanoseconds and a cycle of 400 nanoseconds. They are mounted in a 22 pin package and are inherently faster than a 16 pin RAM which requires external multi-

plexing. Projections indicate that lower cost, high speed memories are on the way and will be pin-for-pin compatible. Even the 16K x 1 RAM's being developed will be compatible and will only require connecting two more address lines.

In comparison with bipolar memories, the NMOS chip has higher density and lower power consumption. Bipolars are faster (30-100 ns are typical cycle times), but the largest ones available are 1K x 1 bit. Use of a 4K RAM instead reduces the chip count by a factor of 4. This savings is important since board space and wiring contribute significantly to the overall cost and reliability. Furthermore, the cost of 4K of NMOS is approximately ten times less than the cost of 4K of bipolar memory. In this machine, more than $5000 was saved by using the SEMI 4200 over a higher speed bipolar RAM. The detrimental effects of the relatively slow NMOS RAM's have been minimized by overlap instruction fetching and concurrent accessing of the data and instruction memories.

Referring to the memory schematics in Appendix A, reading from or writing into memory requires the 12 address lines, the Read/Write line, and the data to be stable before the Chip Select (CS) goes from 1 to 0. The CS negative going edge clocks these data into an

internal buffer register. To prevent device overheating, there is a maximum chip enable pulse width of 1 millisecond. For flexibility allowing the machine to run with a cycle slower than 1 ms (for example in the testing phase), it was necessary to include an extra buffer to capture the valid data before the memory is disabled. Ultimately when the machine is run at optimal speed these buffers can be removed to reduce chip count.

Also for reasons of flexibility, the chip select pulse width is determined by a monostable flip-flop. The memory cycle time can thus be altered independently of the system clock. Monostable timing is intended for the prototype only; to maximize reliability it should be replaced with a synchronous circuit such as a counter clocked by the system clock.

6.4 Timing

For simplicity and ease of design, the computer has a single phase system clock that drives the logic synchronously. A worst case clock period is shown in Fig. 19. Referring to the diagram, the following operations are performed during the indicated subintervals:

1. On the positive edge of the system clock pulse the microword to be executed is latched. The

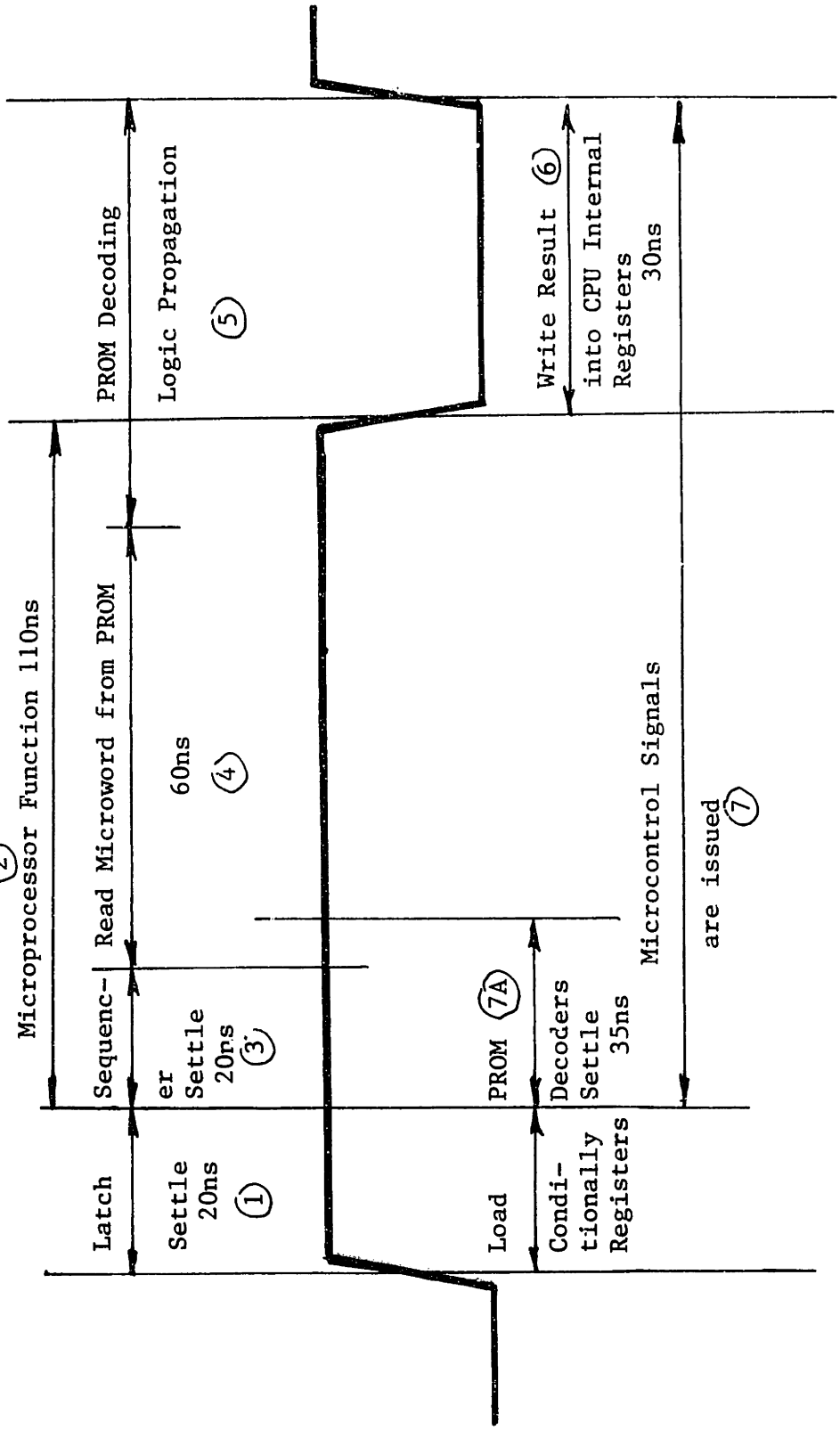


Fig. 19 Worst case system timing

correct signals appear at the latch outputs in less than 20 nanoseconds. Other data registers are conditionally loaded, depending on the details of the previous microinstruction.

2. During this interval the CPU fetches its two operands and passes them through the ALU. The 110 ns includes the time for the carry to propagate from one slice to another. This time would be significantly longer if not for the use of a carry lookahead chip.
3. Information from the current microinstruction is passed to the PROM sequencer which generates the next microword address. The next microword is fetched concurrently with the execution of the present one. The Sequencer uses Schottky multiplexers with a worst case settling time of 18 ns, more than twice as fast as the Low power Schottky counterpart.
4. Actual reading from the PROM takes less than 60 ns.
5. The PROM output is decoded and modified to produce the next correct microword. In the case of multiplication and division, the result of the CPU operation in (2) can modify certain

bits of the microword. At the end of (5) the next microword has been generated and is ready to be latched.

6. 30 ns are required to write the result of the function just executed into one of the internal registers.
7. The microcode control signals are issued during this interval. Control signals that are vertically encoded must be decoded and are not valid until the end of (7A).

On the next positive edge, the process repeats.

6.5 Component Layout and Chip Count

Photographs of the computer are shown in Figures 20 to 26. Fig. 20 is a rear veiw of the processor. Three rather large power supplies were used because they were readily available in the laboratory. Particularly the +12 and -5 volt supplies for the memories can be reduced in size since the NMOS memories draw very little current.

The computer is constructed on two universal wirewrap boards. The boards are specially designed for Schottky logic: they have two ground planes with a voltage plane sandwiched in between. This arrangement minimizes the problems of noise, ringing and crosstalk.

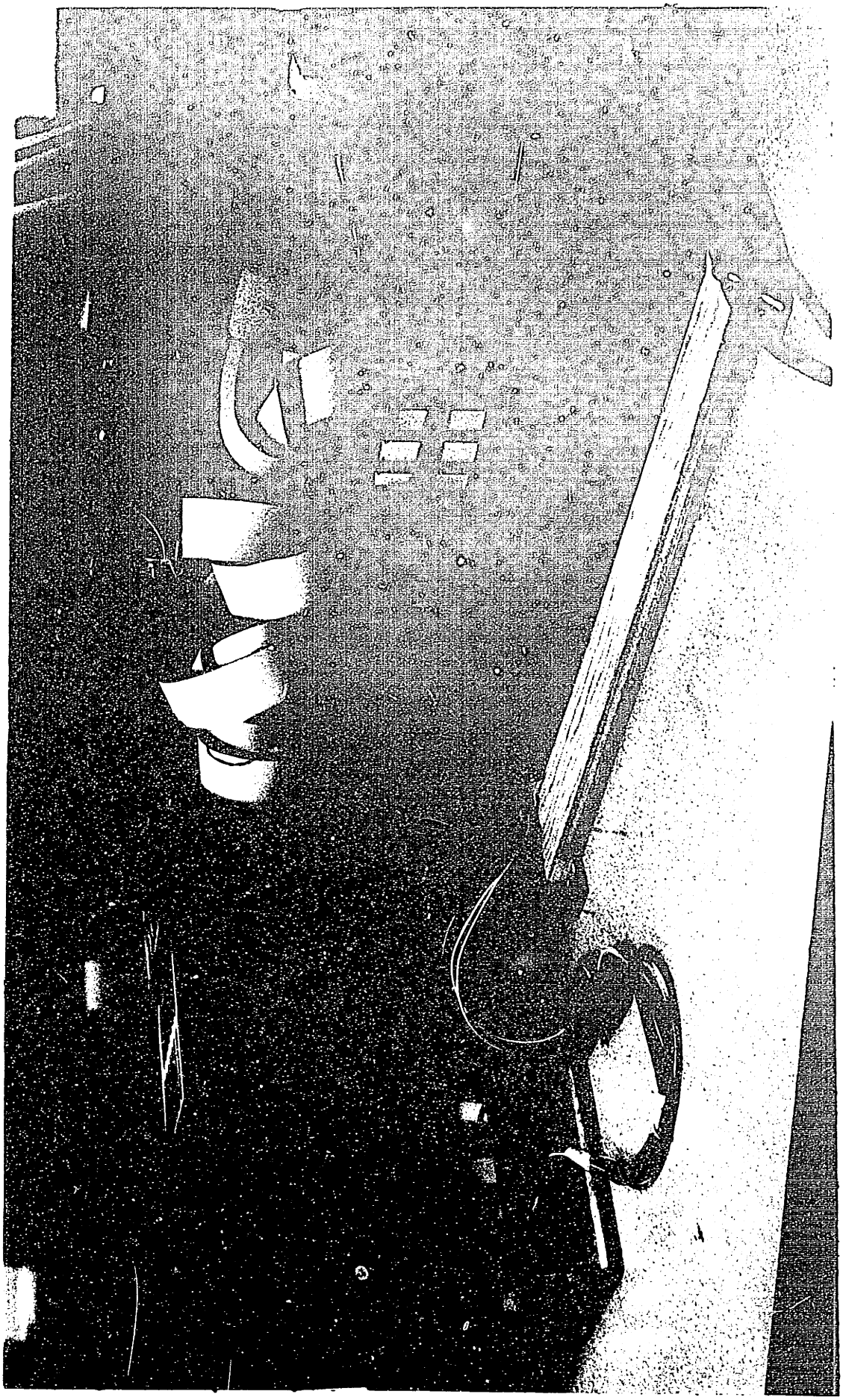


Fig. 20 Rear view

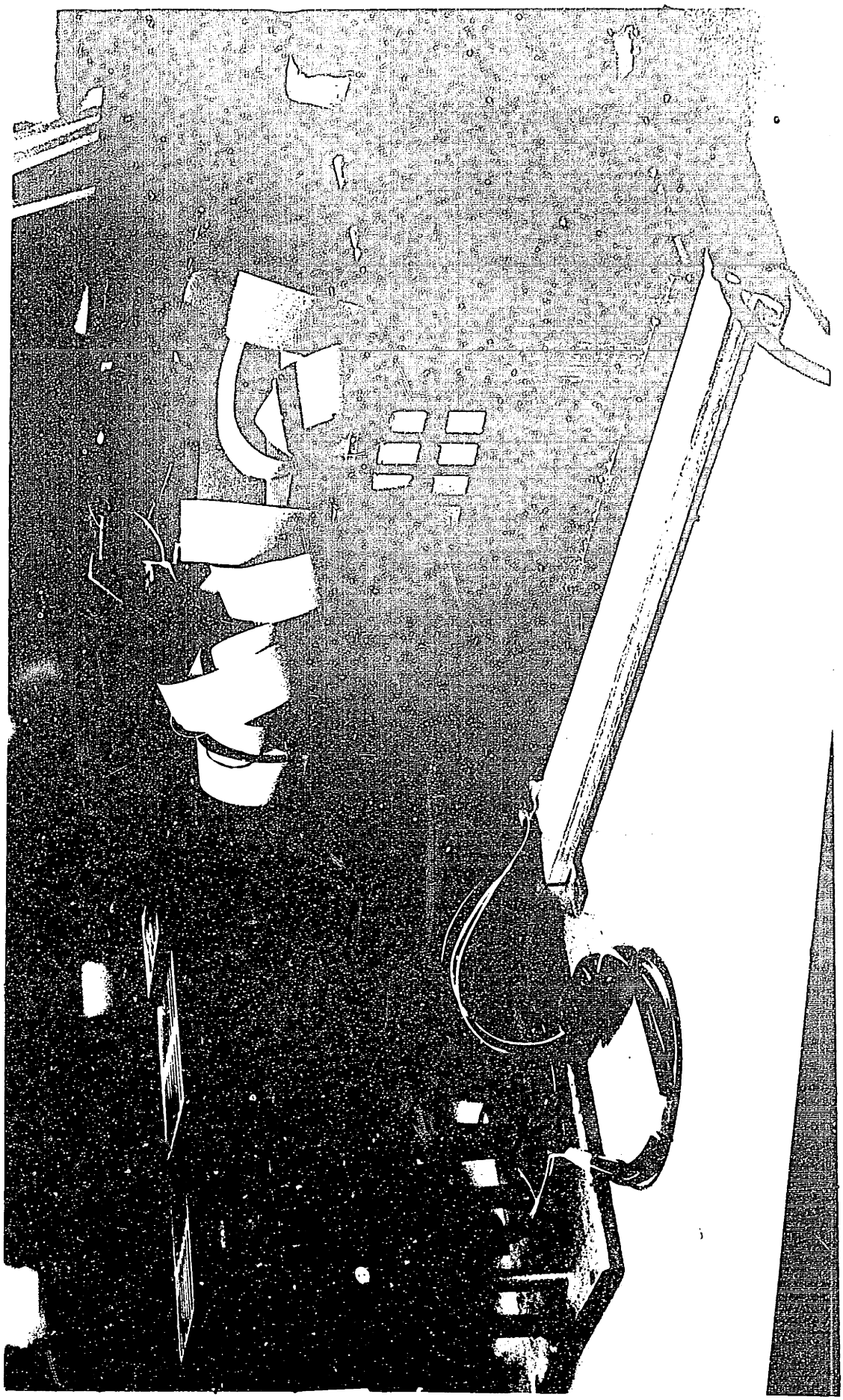


Fig. 20 Rear view

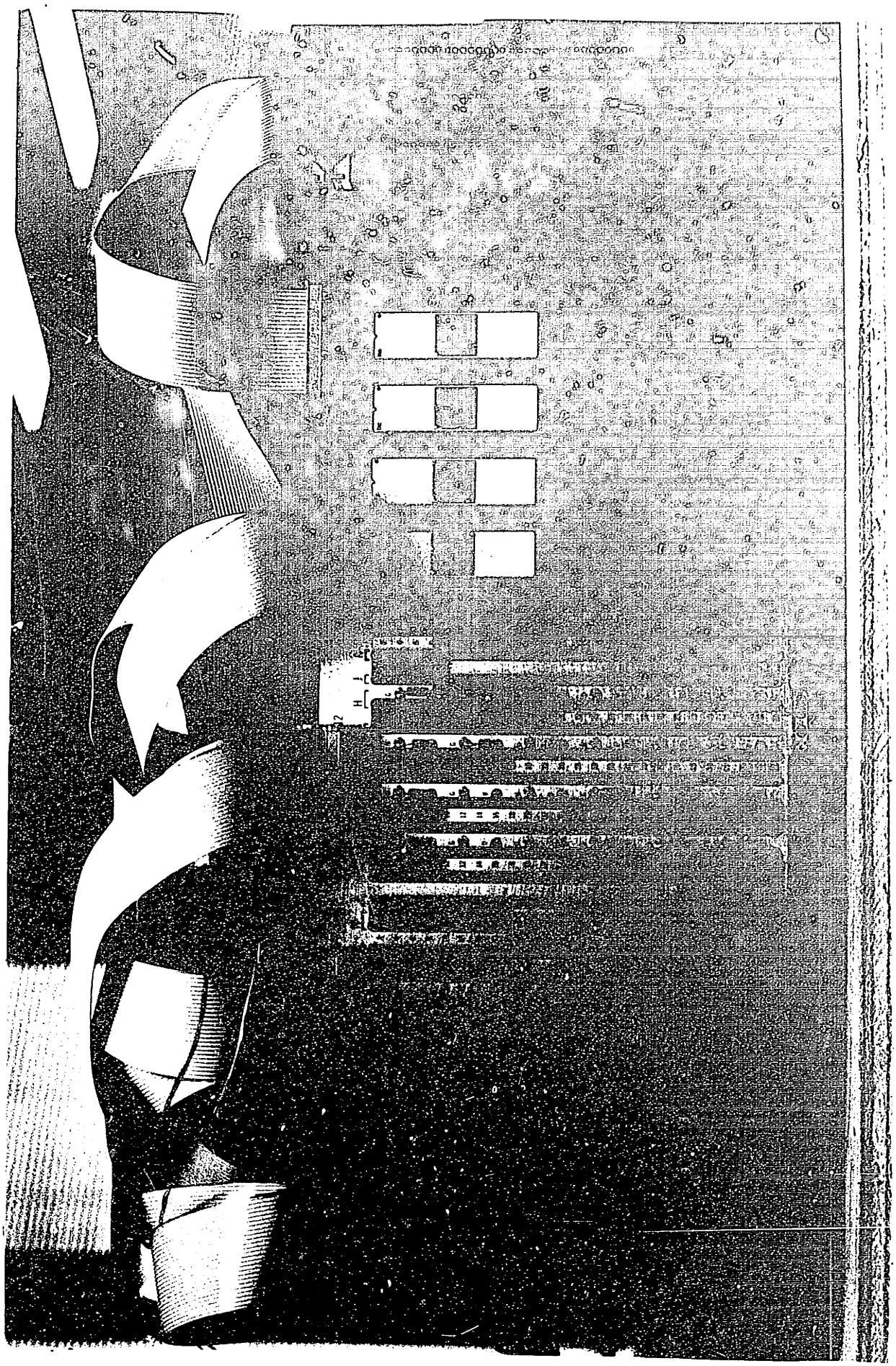


Fig. 21 CPU Board

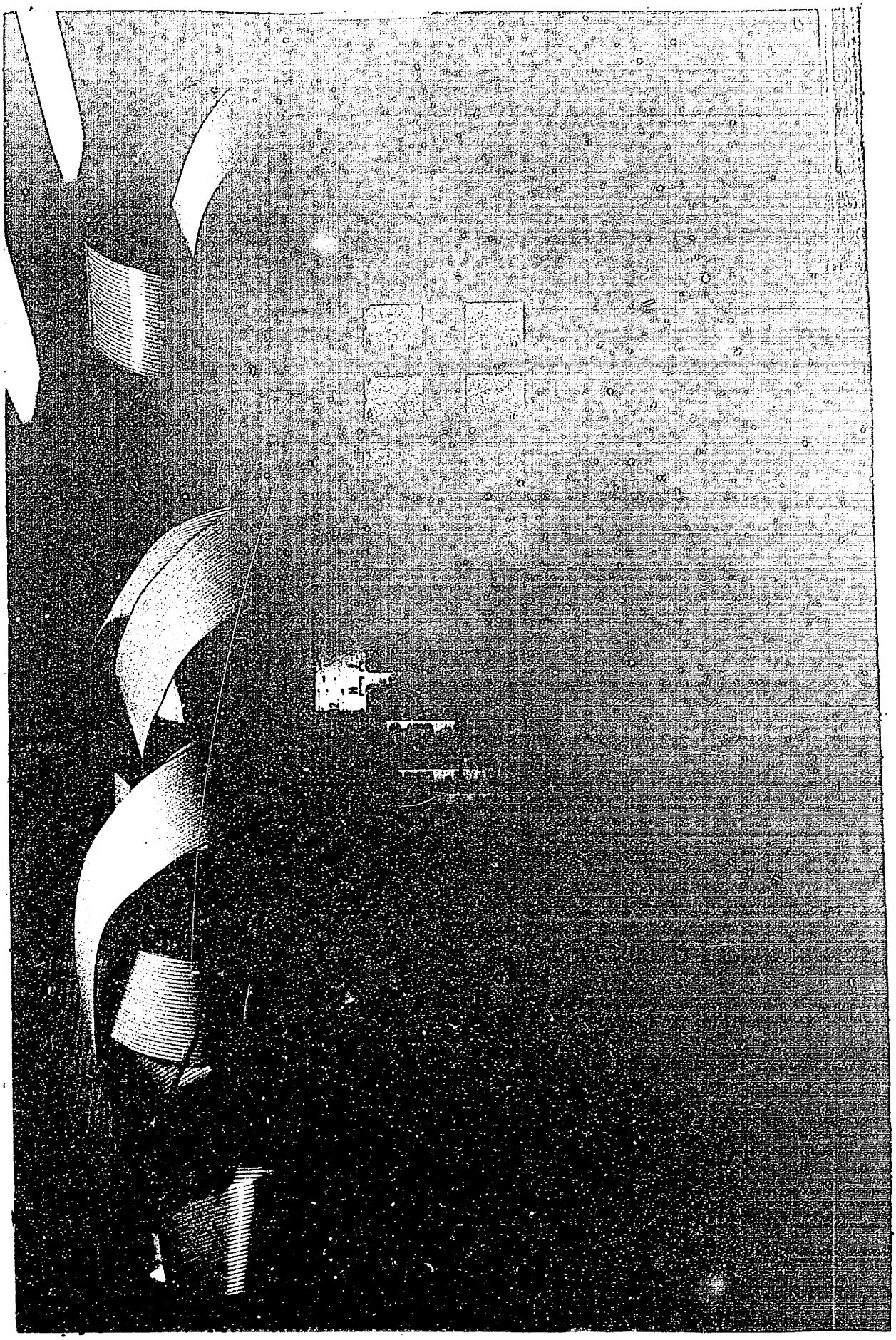


Fig. 21 CPU Board

Fig. 22 Memory Board

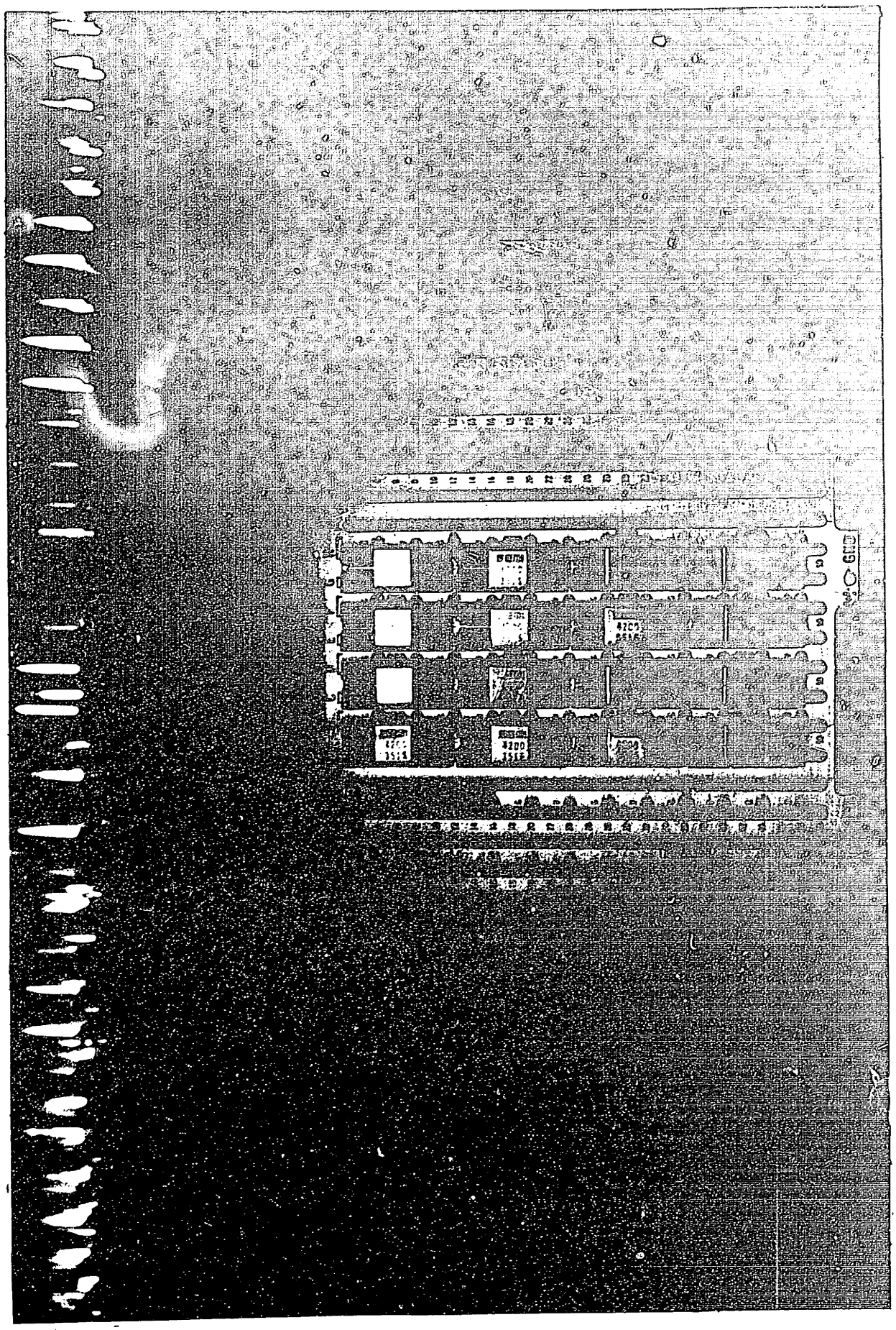


Fig. 22 Memory Board

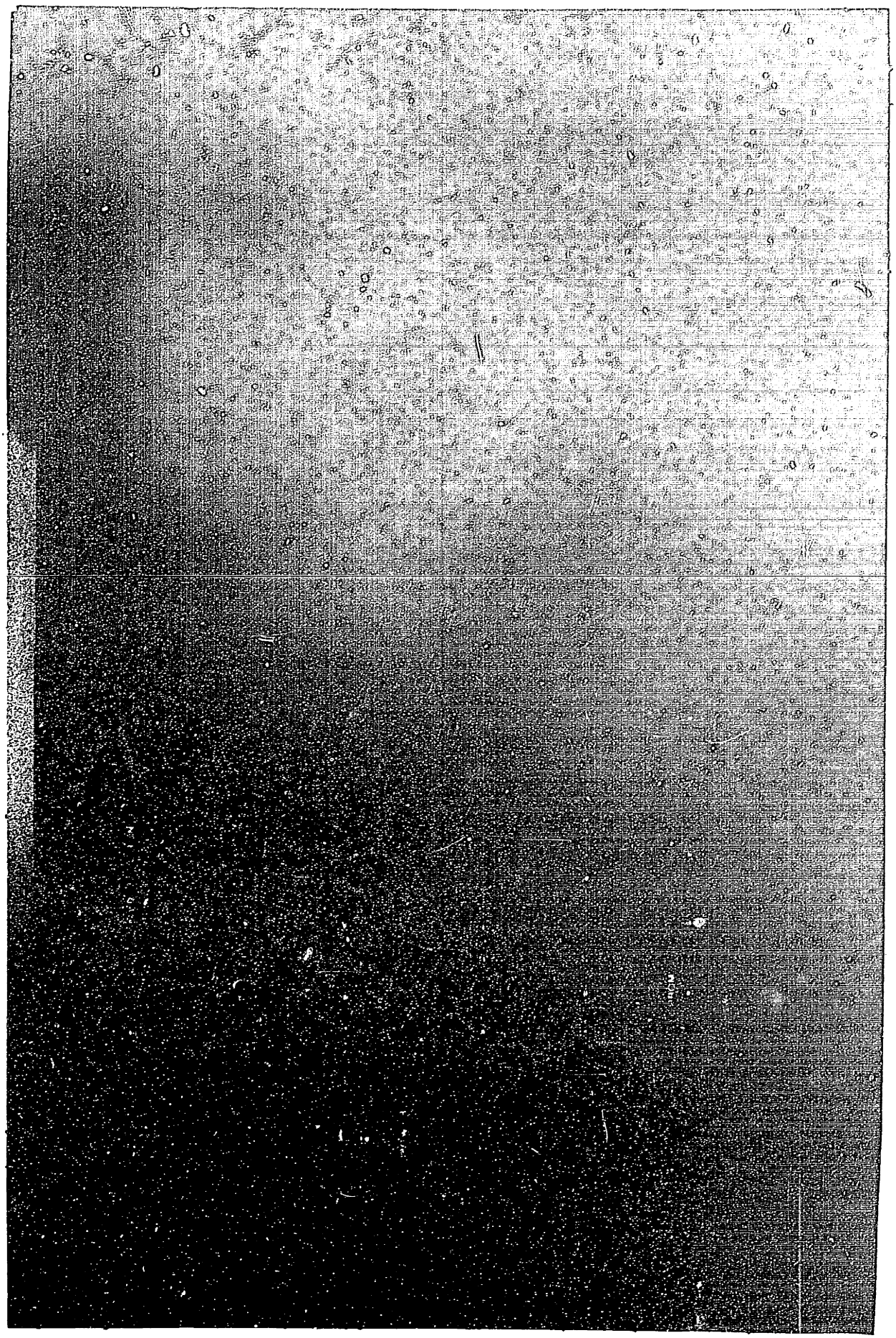


Fig. 23 Front Panel

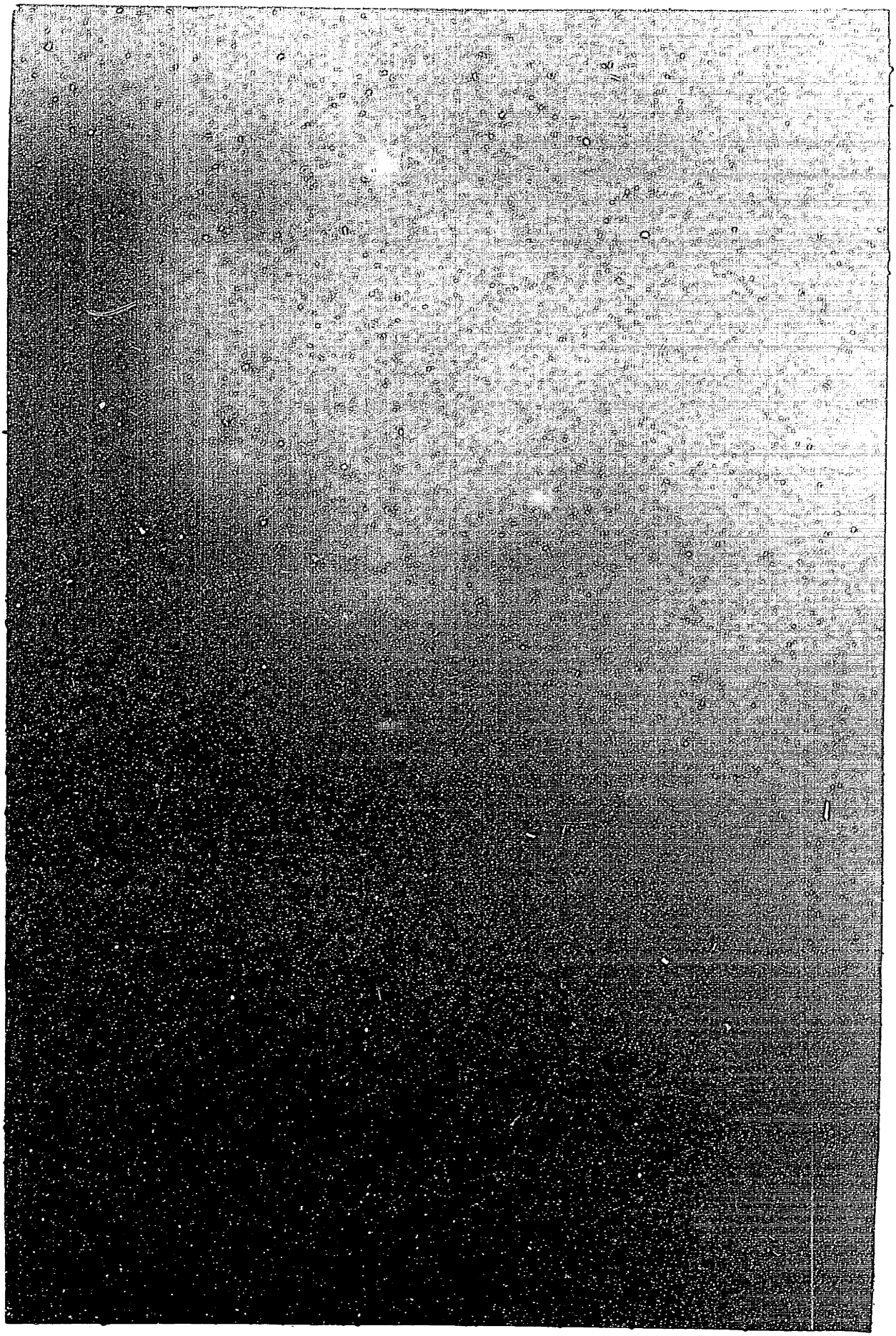


Fig. 23 Front Panel

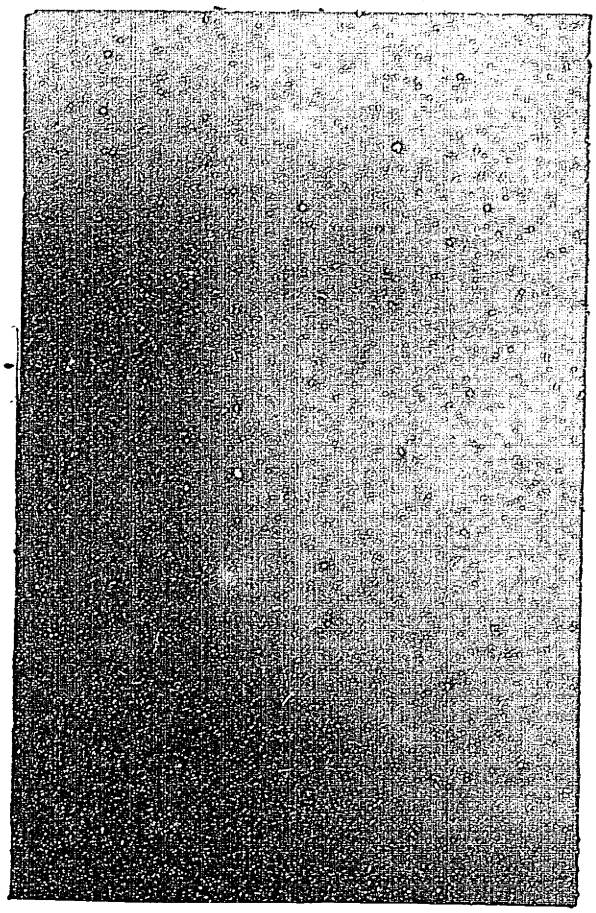


Fig. 24 Underside of wirewrapped board

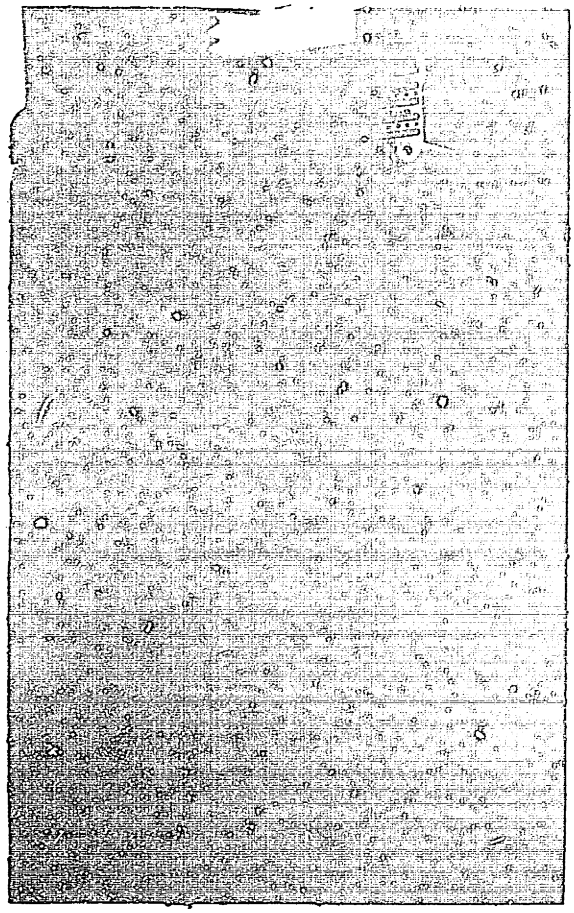


Fig. 25 Boards hinged for testing

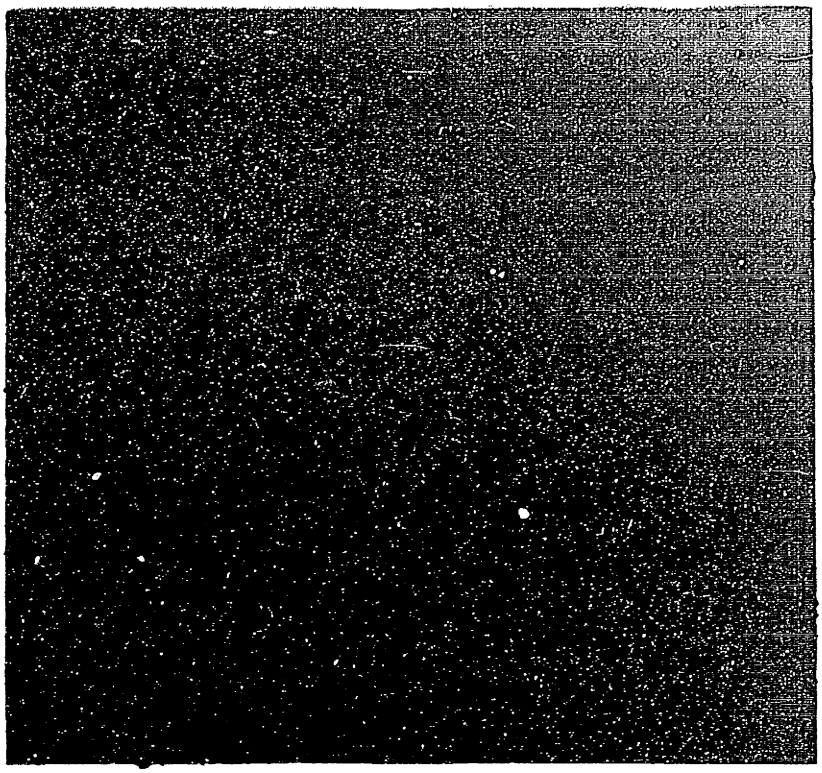


Fig. 26 Back view of front panel

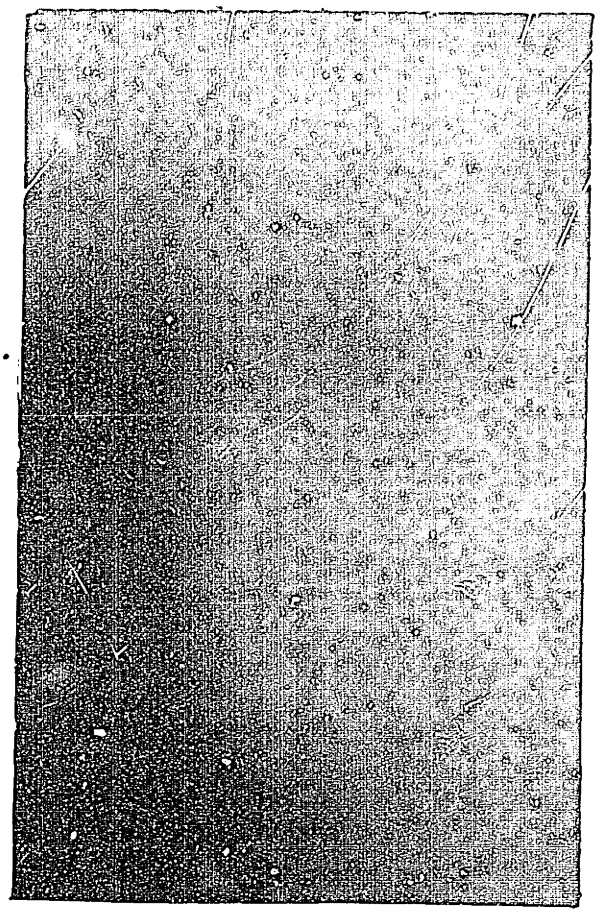


Fig. 24 Underside of wirewrapped board

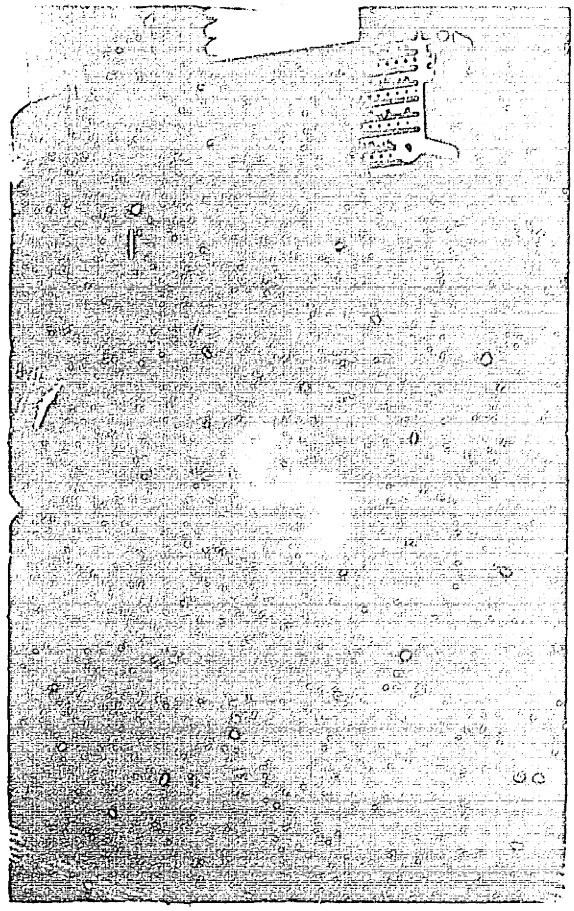


Fig. 25 Boards hinged for testing

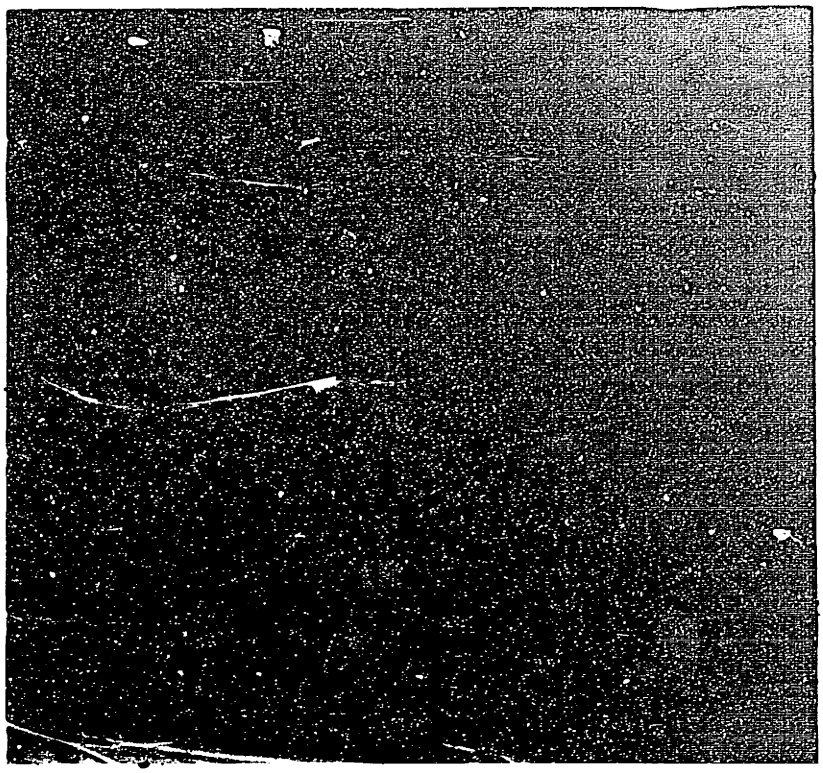


Fig. 26 Back view of front panel

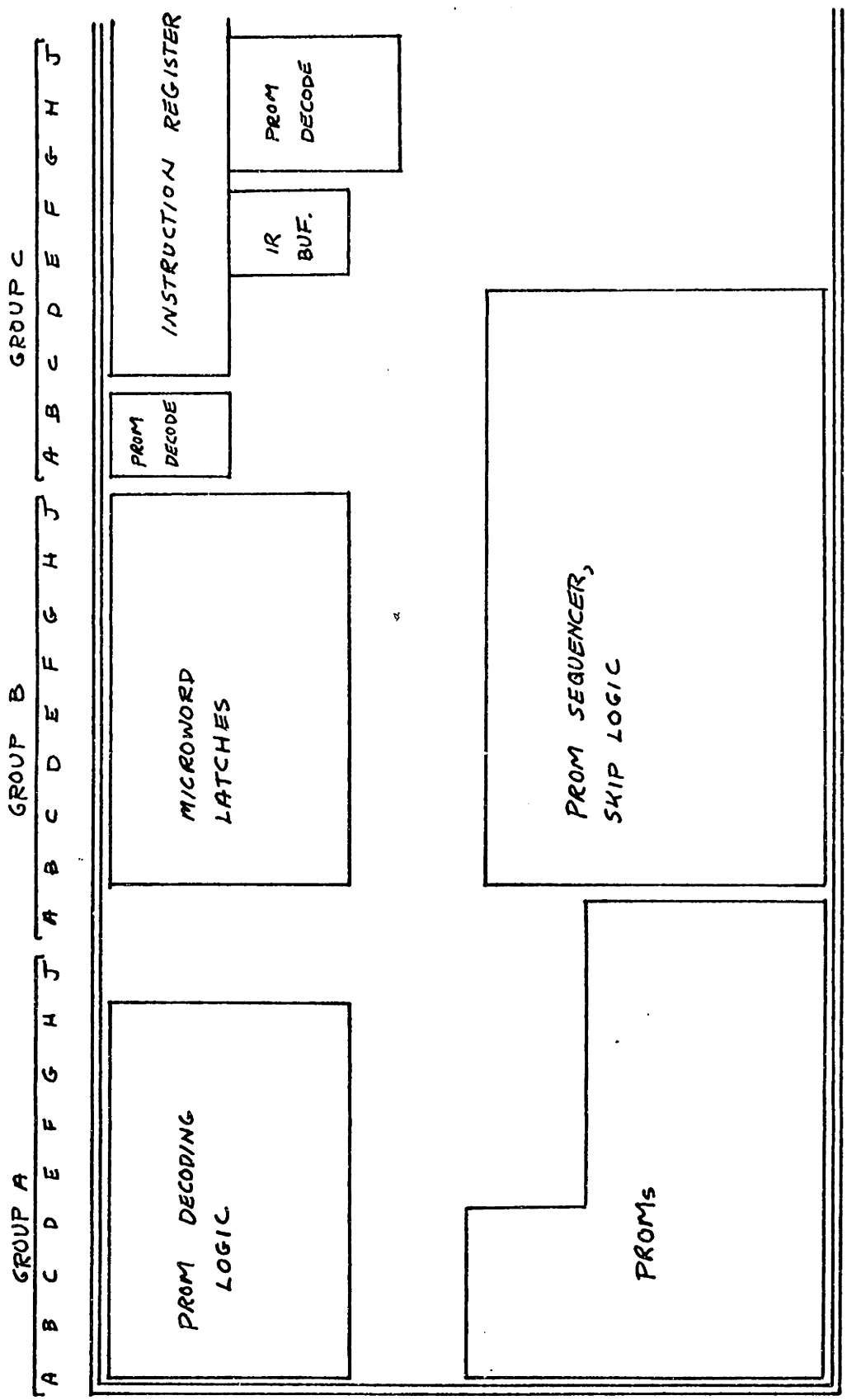


Fig. 27a Chip layout for CPU board (Groups A-C)

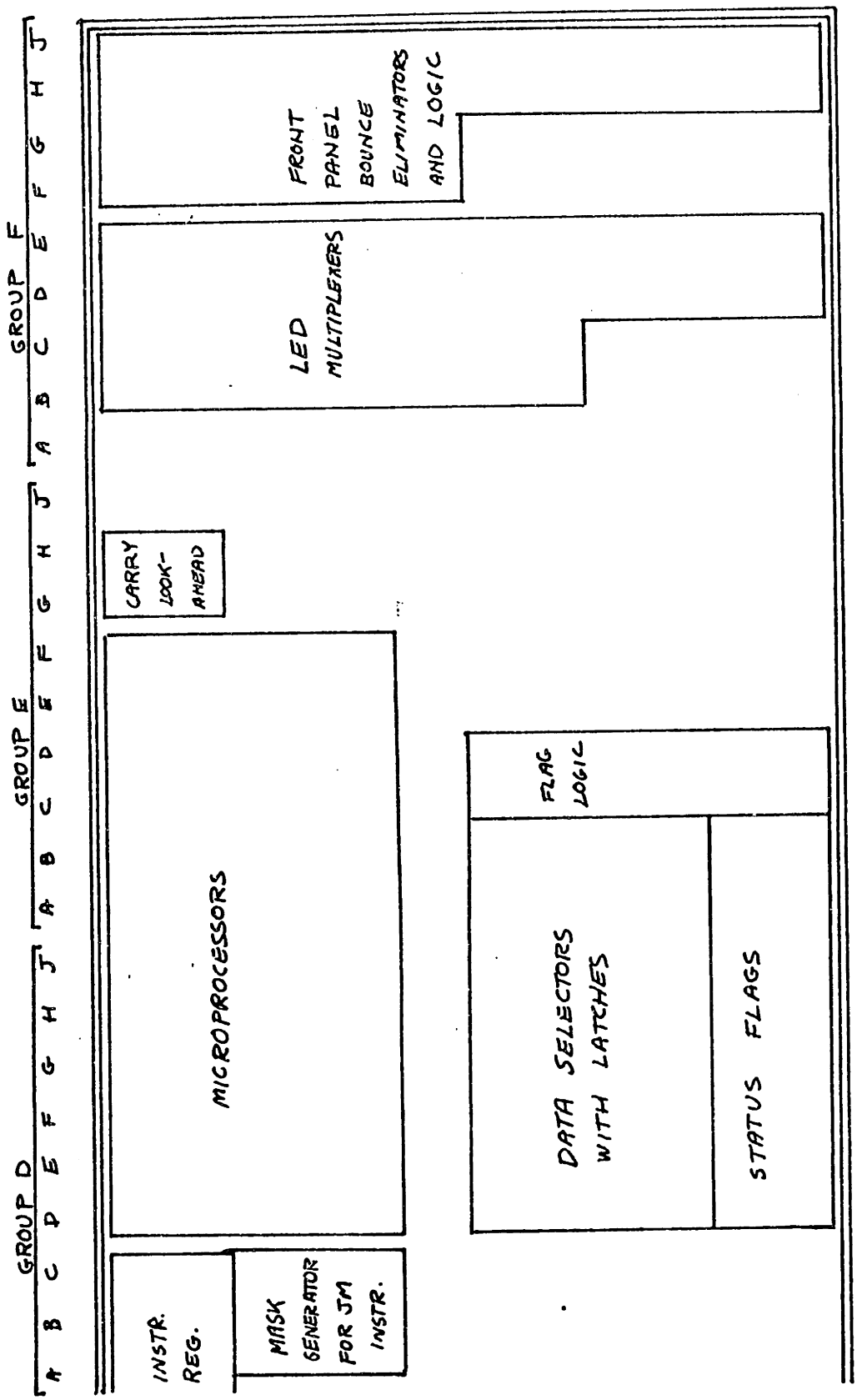


Fig. 27b Chip layout for CPU board (Groups D-F)

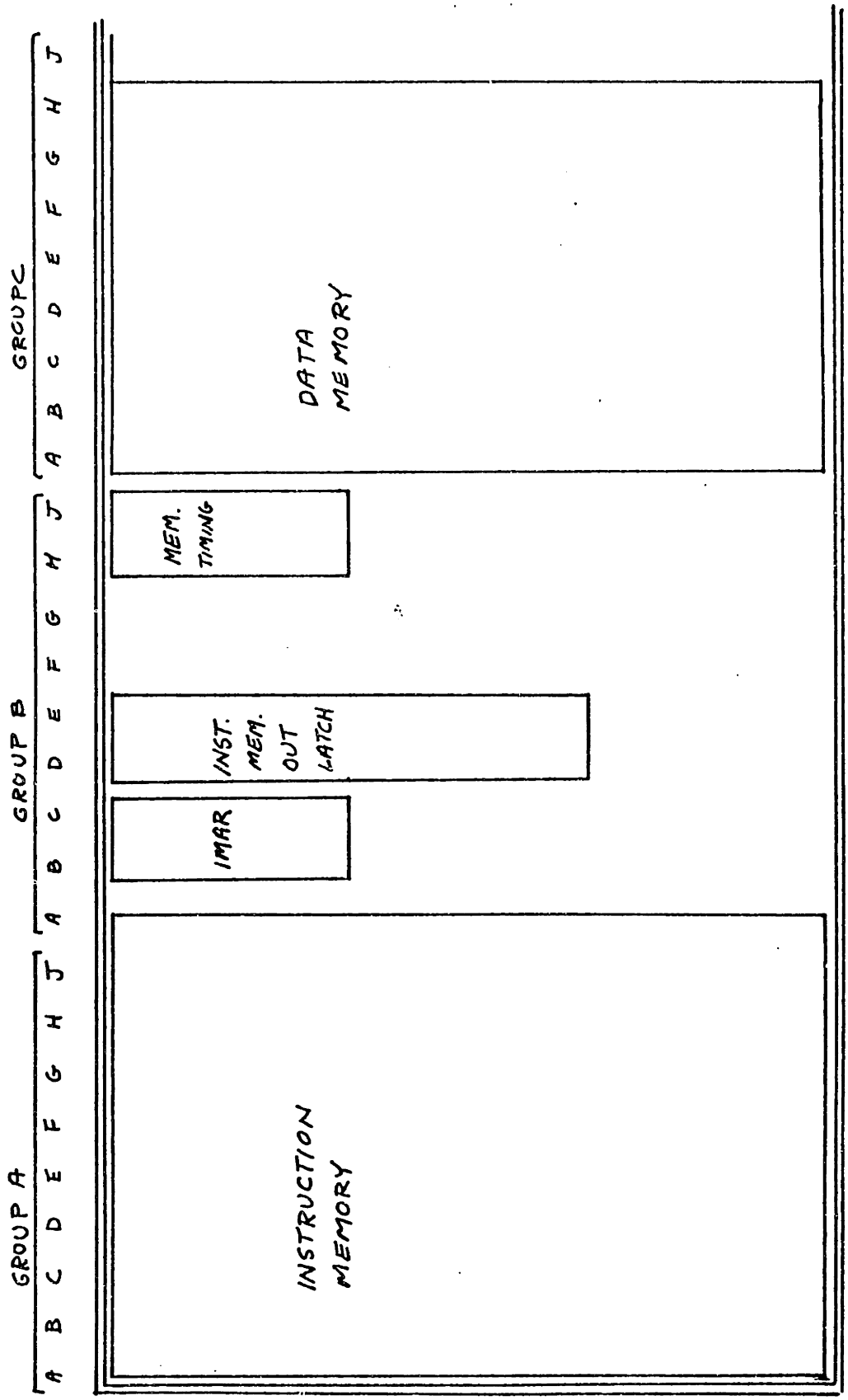


Fig. 28a Chip layout for memory board (Groups A-C)

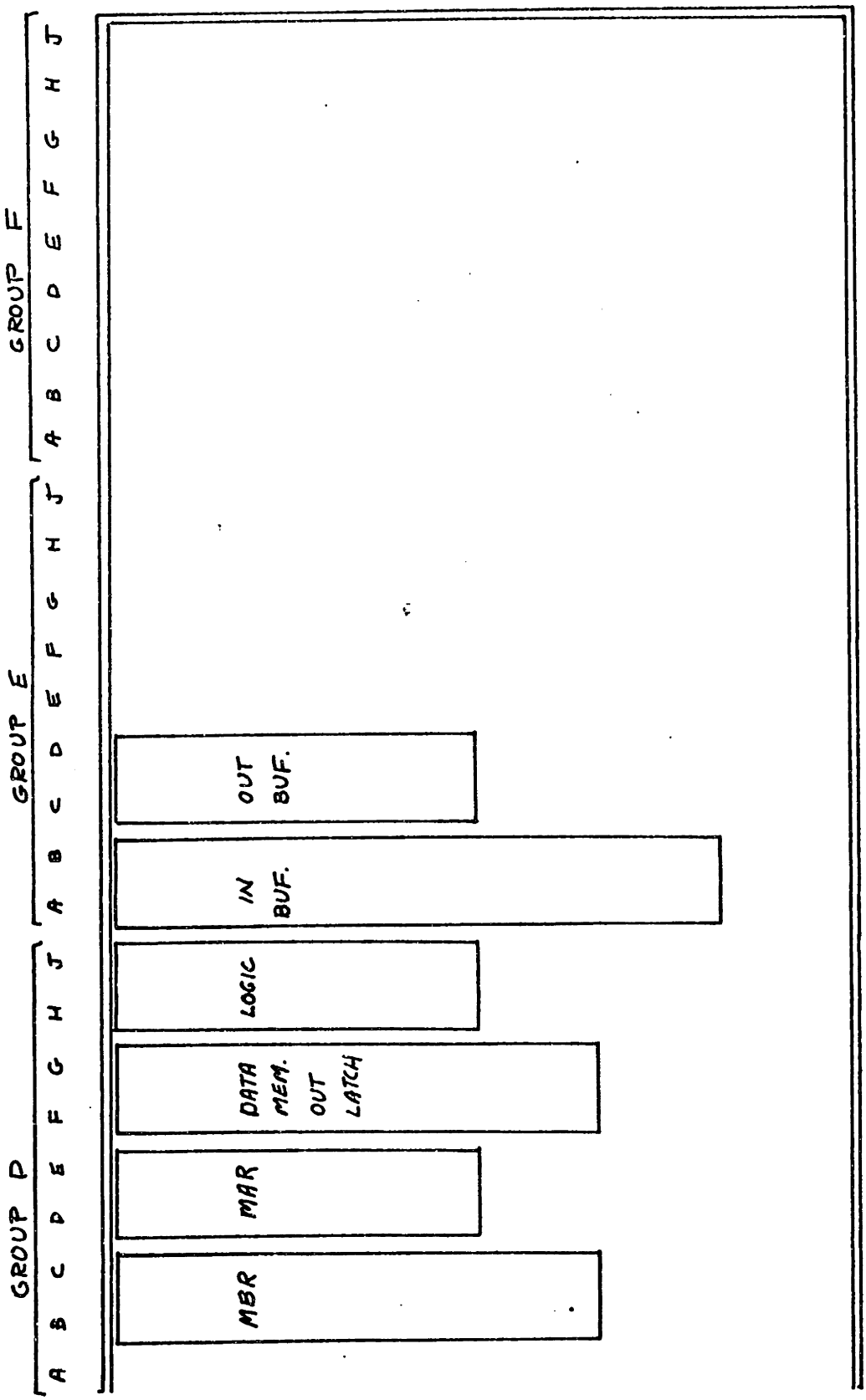


Fig. 28b Chip layout for memory board (Groups D-F)

Generous use of decoupling capacitors contributed to further noise reduction.

The chips have been placed to minimize the wire lengths while maintaining their modularity. Chips comprising the same logical unit such as the data memory, the CPU or PROM sequencer are all located together. Ample space has been left for further modifications and expansion. One board contains the data and instruction memories, their associated buffers and control, and the I/O buffers (see Fig. 22 and Fig. 28); the other contains the CPU and microcode controller (see Fig. 21 and Fig. 27).

Wires common to both boards are connected with ribbon cable, alternating signal wires with ground to minimize crosstalk. The boards themselves are mounted in the vertical plane to enhance convection cooling and obviate the need for a fan. Fig. 25 shows how the boards are hinged to allow easy access for testing.

Great care was taken to minimize the total chip count; more chips require more board space and connections with a corresponding cost increase. Furthermore, since most integrated circuit system failures occur at the interconnections between chips it is good design practice to minimize their number. The cost of using a single 16 pin chip on the universal wirewrap board is approximated

by:

Board space	$2.75	
Wiring @ $.075/wire	$1.20	
Decoupling capacitor	$ .15	
	$4.10	Subtotal

To this subtotal one must add the cost of the chip, power supply (approximately $3/watt), testing, and general overhead (i.e. cabinet, fans, etc.). The cost of most standard small scale integration logic chips ranges from $.25 to $1.00, while MSI chips are usually less than $5.00/chip. One can see that the cost of the chip itself is frequently overshadowed by the costs associated in using it in a hardware system. These figures provide strong motivation to reduce the number of chips and the power requirements.

The processor contains a total of 160 chips, including 8K of memory and the microprocessors. Total power consumption is under 30 watts. Nineteen chips provide for the panel functions: bounce eliminators and pulse synchronizers, multiplexers to drive the Light Emitting Diodes (LED's), and some random logic. These chips can be eliminated once the machine is prepared to stand alone with panel controls. It is within the current state-of-the-are to reduce the entire machine, with

the exception of the 36 4K memory chips and the 4 microprocessors, to one or two LSI chips, bringing the total chip count down to 42. Advances in LSI technology is certain to increase packing densities and it is reasonable to predict that the entire machine could be reduced to 15 chips within 2 years.

6.6 Front Panel

The computer was designed with a relatively elaborate front panel so that most functions of the machine could be tested directly with front panel switches and LED's. The rationale behind this decision is that test equipment designed especially for debugging parallel-mode logic (such as the Hewlett-Packard 1600A and 1607A) was not available.

At the top of the front panel (see Fig. 23) are 8 LED's for the status flags. The next set of 20 LED's represent data from the Bus, Instruction Register, Output Buffer or the microcode inputs to the CPU, depending on the position of the Display Select Switch. The ability to view all four of these sources by simply turning a switch has greatly facilitated the testing and the debugging of the machine. Below the LEDs are 20 switches to input data to the Input Buffer. The data is actually written into the buffer when the Load Input

pushbutton at the bottom of the panel is depressed. Eventually, these 21 switches will be replaced by a buffer to the host machine.

The next row contains 6 pushbutton switches whose functions are described below:

1. PROG. LOAD. This is the first switch to depress when starting the computer. It causes the PC to be set to zero and causes all data entered through the Input Buffer to be written sequentially in the Instruction Memory. There is no need for a bootstrap loader since the micro-instructions for loading both instruction and data memories are resident in the PROM's. The LED above the switch verifies the machine is in fact in program load mode.
2. DATA LOAD. Once the program has been loaded, depressing the DATA load button causes data from the Input Buffer to be written into the Data Memory starting from location zero. An LED lights when the computer is in data loading mode.
3. START. The START switch causes both the PROG and DATA LEDs to be turned off and initiates the program execution. The first instruction is fetched from the Instruction Memory at location

zero. The RUN flag is turned on.

4. HALT stops execution of the machine by turning the RUN flag off. The system clock continues to run but the microcode enters a wait cycle until the RUN flag is turned on again. The machine can also be halted under program control.
5. CONTINUE resets the RUN glag so the machine resumes execution where it was last HALTed.
6. INST. STEP. (Instruction Step) causes the next instruction to be executed after which the machine is halted.

Forty-four microcode switches are included on the panel to enter and test the microcode without the PROMs. This feature allows microprograms to be simulated and debugged before dedicating them to the PROM which is an irreversible process. In the UP position, the corresponding bit of the microword is logical zero. Once the PROMs are inserted and the microcode source switch is moved from SWITCH to PROM, all 44 switches must be in the DOWN position to prevent damage to the PROM outputs. The 8-LEDs in the lower righthand corner indicate the address of the next microword to be fetched and are essential in single step simulation of the microcode.

The CLEAR OUT pushbutton resets the OUT flag indicating that the data has been copied from the Output Buffer into the post-processor. This function will eventually be handled by the Vocal Tract Model.

The last two switches control the system clock. Normally the clock in ON, however it must be turned OFF to single step through the microcode.

6.7 Wirewrapping

Generation of the wirelists and the actual wirewrapping is a monumental task since the machine contains over 5000 individual connection. Despite its complexity, wirewrapping is less time consuming to implement, and mistakes are more easily corrected when compared with soldering or printed circuit techniques. A computerized system to aid in the conversion of logic diagrams to hardwired boards greatly facilitated this task. The program, WIREWRAP, developed at the University of Michigan and imported to M.I.T. by Jack Allweiss, not only generates a wirelist with the name and length of every wire, but also prepares a paper tape for automatic and semi-automatic wirewrapping.

The first step in using the program is writing a logic series description in which each IC is entered

into a macro dictionary with its cost, power requirements, load and drive factors and pin assignment. Once this information is entered, the programmer need never be concerned with pin locations. In the next part of the program, (device description) a unique signal name (net name) is assigned to each output and all its associated inputs. Each line of code fully describes all the connections for that particular logic element.

Chips are positioned by the programmer with a preset command which specifies the location of the first pin. Every pin is assigned a unique name based upon its location. The board is subdivided into 6 groups or bays labeled A through F. Within each group there are 9 columns (A-J excluding I) and 50 rows (numbers 1-50). Pin names are designated by: Group - Column - Row (i.e. A - J - 29). The program, referring back to the macro dictionary, then proceeds to connect all inputs and outputs that share the same net name. WIREWRAP generates a net name cross-reference, a net name dictionary that lists all the pins that comprise each net, a list of single pins, a wire list with the net name and length of every wire, and finally a backplane plot. This information saves time and errors, and provides accurate documentation.

Once the circuit was entered into the program and

debugged a tape for semi-automatic wirewrap was punched. All wiring is point-to-point to minimize wire lengths. In less than one week from the creation of the tape, the entire machine was wired and debugged. Approximately a half dozen wiring mistakes were found and corrected. The modularity of design greatly sped the debugging procedure. Each module was fully tested separately before combining them and testing the overall machine.

6.8 Performance

The computer has been constructed to meet all the design requirements. The average RR instruction execution time is approximately 500 ns and for MR instructions is approximately 800 ns. These times include calculating the instruction's address, fetching it from memory, and performing the specified operation. Certain special operators take more time. For example, both fixed point multiplication and division require slightly over 3 microseconds.

These statistics compare very favorably with existing minicomputers. For the given task, the speech processor is faster than most general purpose minicomputers available today.

A state-of-the-art machine of roughly similar cost

and size is Digital Equipment's LSI-11, the smallest member of the PDP-11 family of computer systems. The execution of a typical MR instruction (of the type discussed in Chapter 3) on the LSI requires over 8 microseconds or ten times longer than the speech processor. The worst case multiply on the LSI is 64 μs (versus 3.3 μs for the speech processor) and divide is 78 μs (versus 3.3 μs). The LSI-11 is more flexible for general use and costs about $1000 (single quantity price with 4K of RAM), but runs more than ten times slower than the special purpose processor. As a result it is not suitable for real time speech processing. The higher cost of the speech processor can be justified in terms of its performance, and can be reduced significantly if produced in large quantities. For its particular application, the speed of the speech computer is comparable with minicomputers costing well over $10,000.

6.9 Future Development

The speech computer has been designed specifically for speech synthesis programs. Only by actually implementing the complete speech synthesis system can its performance and efficiency be truly tested. It must be

connected to the host computer and the vocal tract model. As new requirements become evident they should be implemented in hardware and/or firmware. New instructions can be added simply by rewriting parts of the microcode. During these development stages it may be desirable to write an assembler to facilitate programming. Once the programs are perfected, they should be written into ROM's. Advances in LSI technology will be instrumental in further reducing the cost and size of the processor so that a practical speech synthesis system can be built in the near future.

APPENDIX A

Hardware Schematics

In the following schematics note that:

1. All logic gates, unless otherwise specified are 7400 series low power Schottky TTL.
2. The location of the first pin of each device on the wirewrap board is given with the notation:

   Group-Column-Row (i.e. A-J-9)

3. The wire net names corresponding to those used in the wirewrap program are shown next to the appropriate wires. Any wire can be located quickly by using the schematics and the wirewrap documentation (Steingart, Jan. 1975) together.
4. The detailed schematics of the memories have been omitted. For each memory, all address and control lines are connected together. Data input and output lines are specified on the schematics.

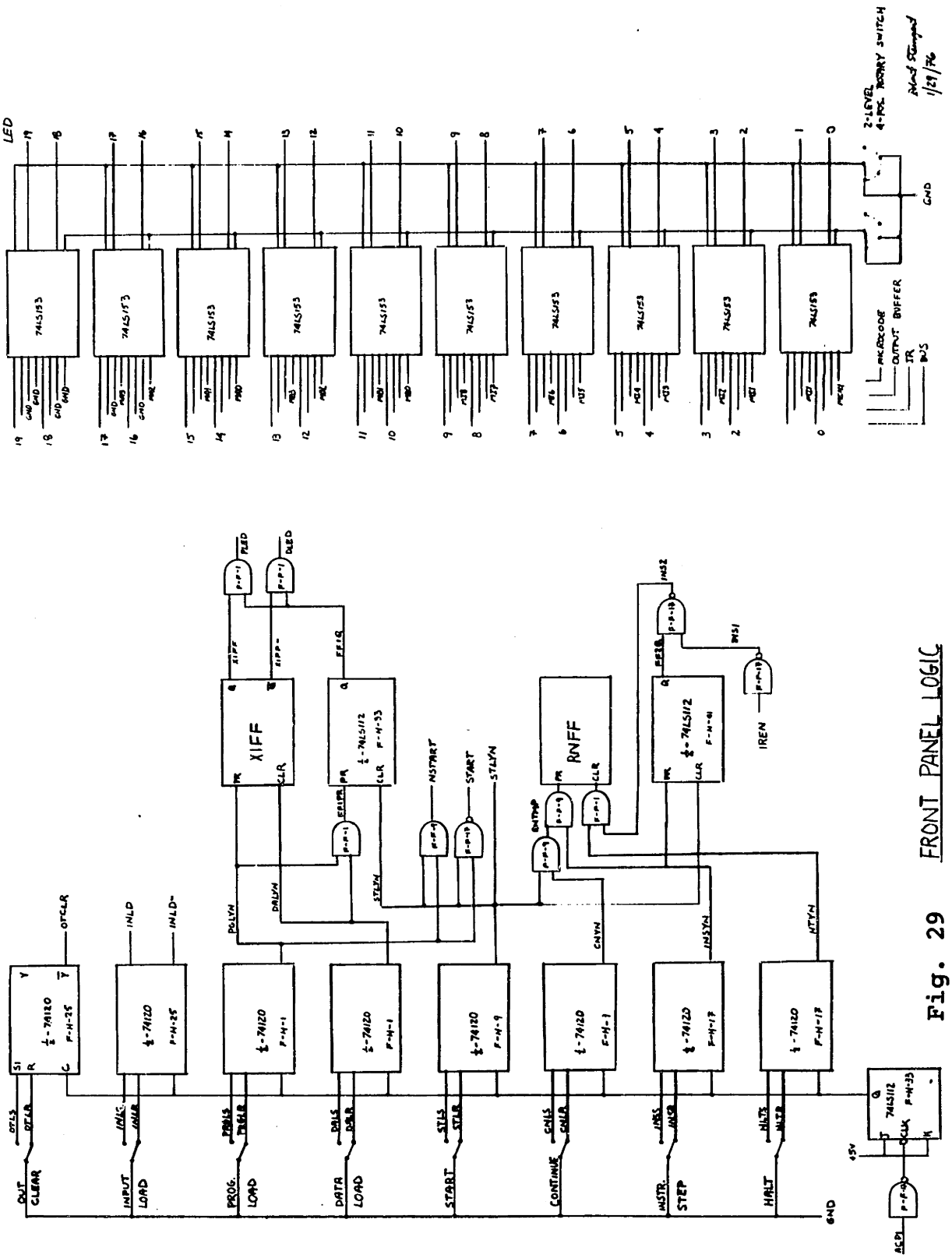


Fig. 29 FRONT PANEL LOGIC

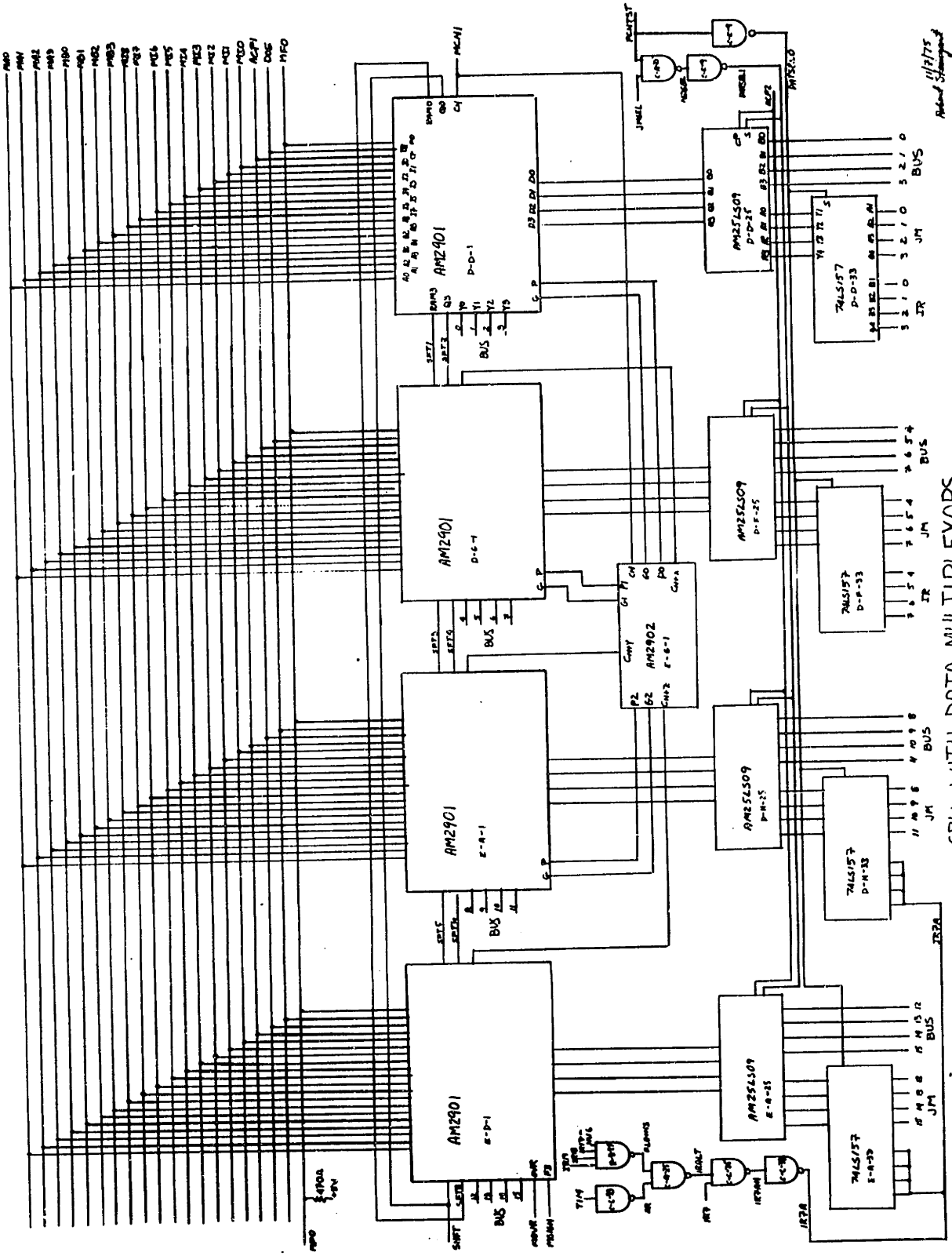

AM2901
AM2902
AM25LS09
74LS157
BUS
CPU WITH DATA MULTIPLEXORS

Fig. 30

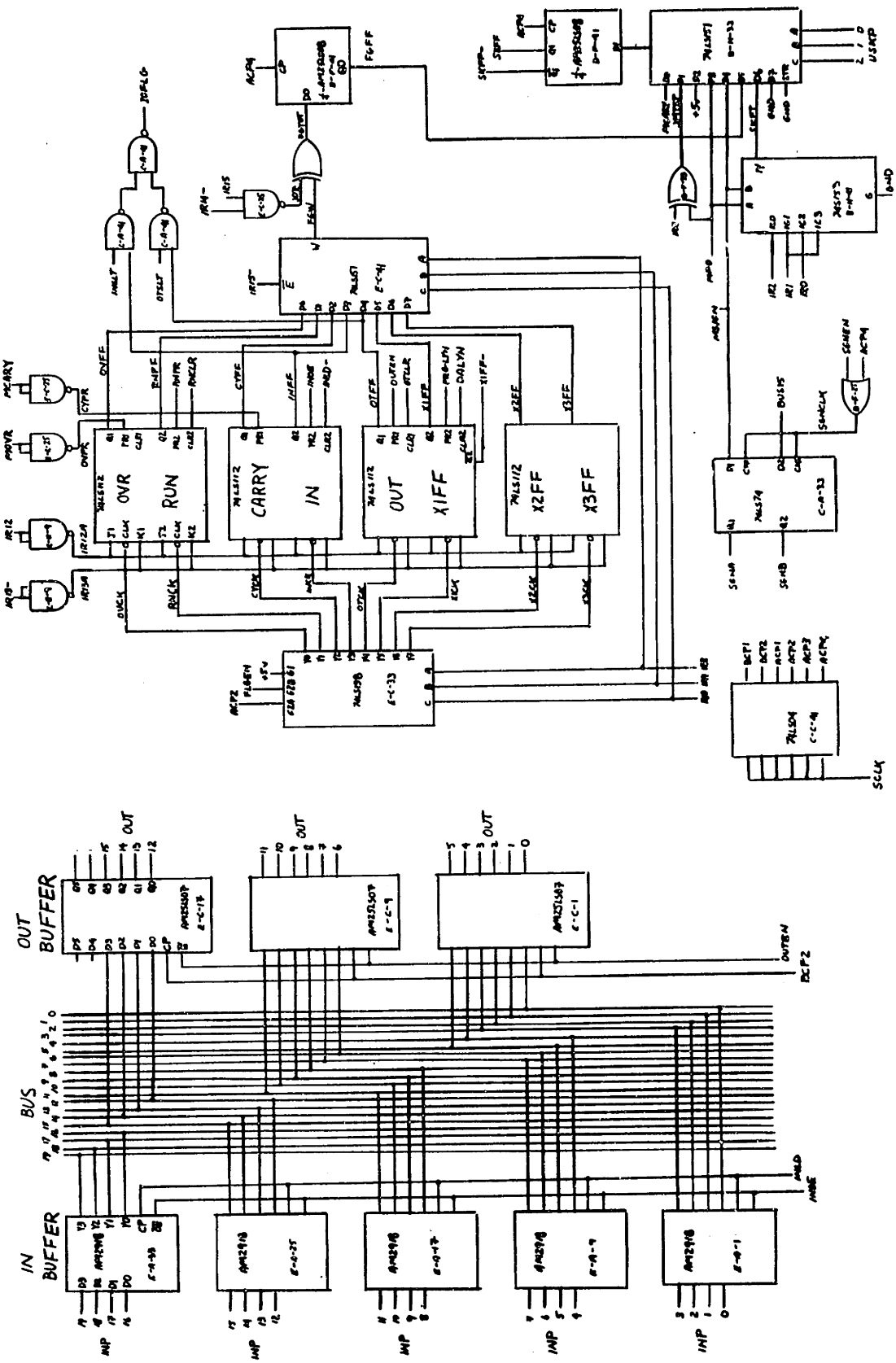


Fig. 31. I/O BUFFERS, FLAG LOGIC, SKIP LOGIC

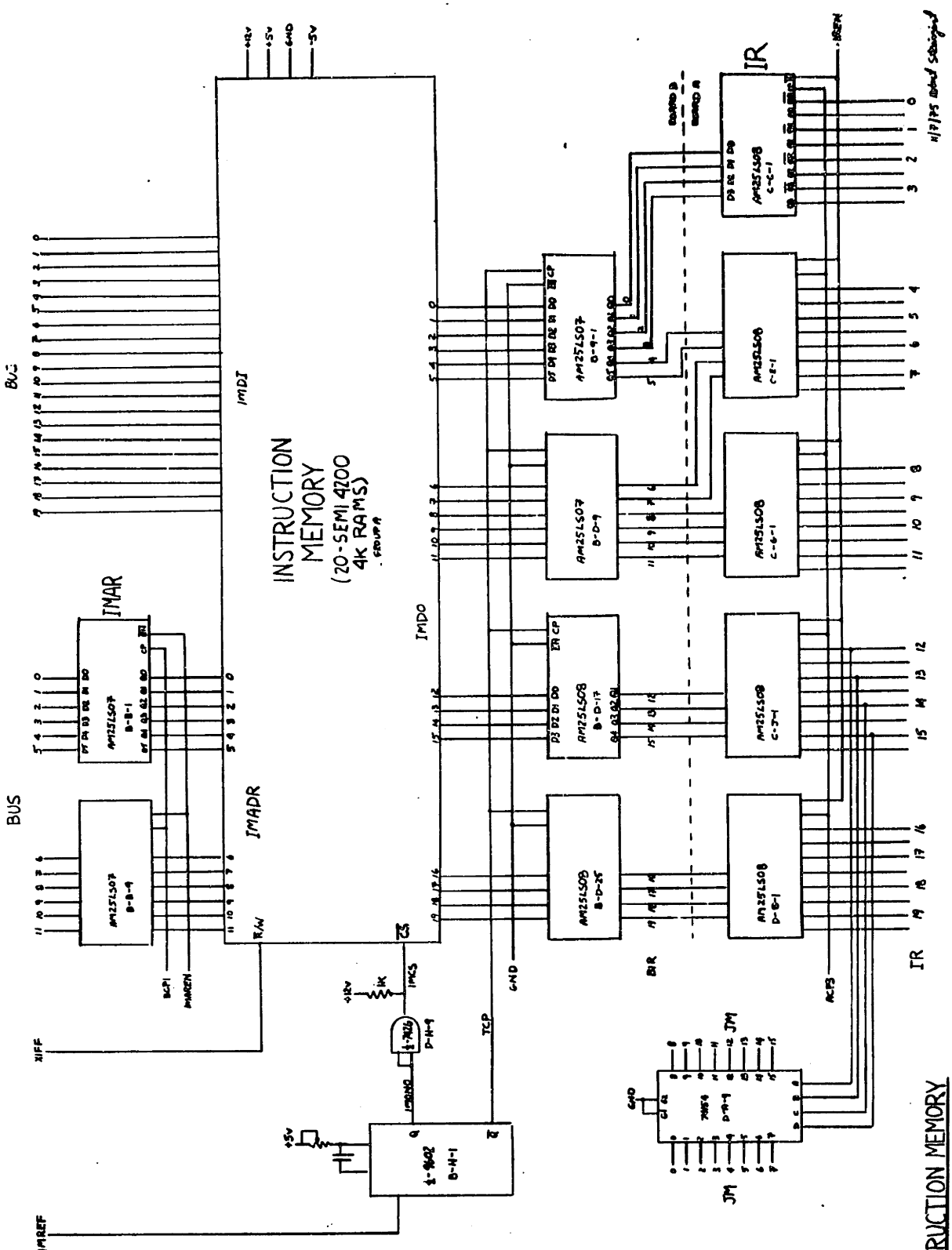

INSTRUCTION MEMORY
(20-SEMI 4200 4K RAMS)
IMDI
IMDO
IMADR
IMAR
BUS
IR
JM
INSTRUCTION MEMORY

Fig. 32

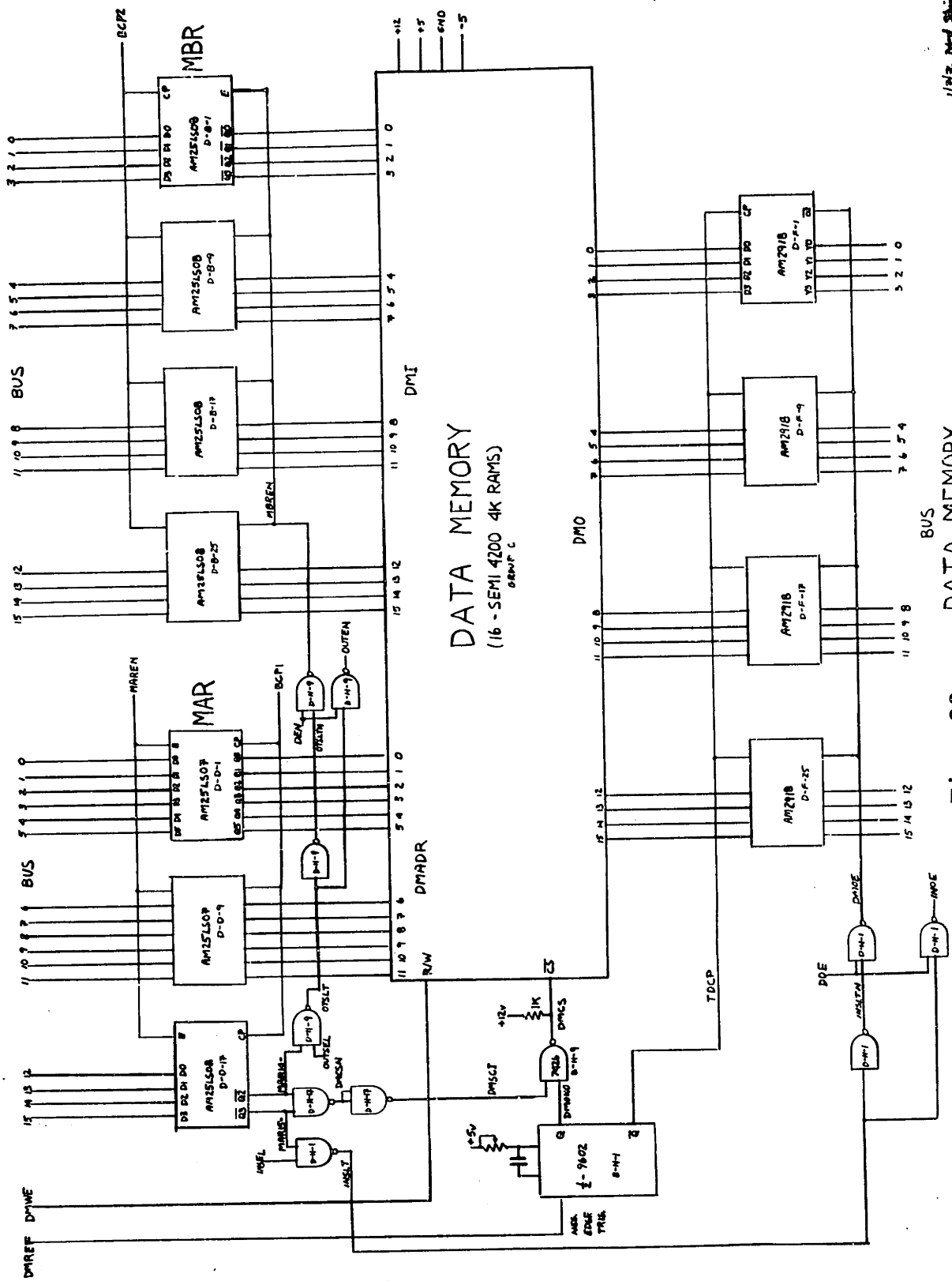

DATA MEMORY
(16 - SEMI 4200 4K RAMS)
MBR
MAR
DMADR
DMI
DMO
BUS
DATA MEMORY

Fig. 33

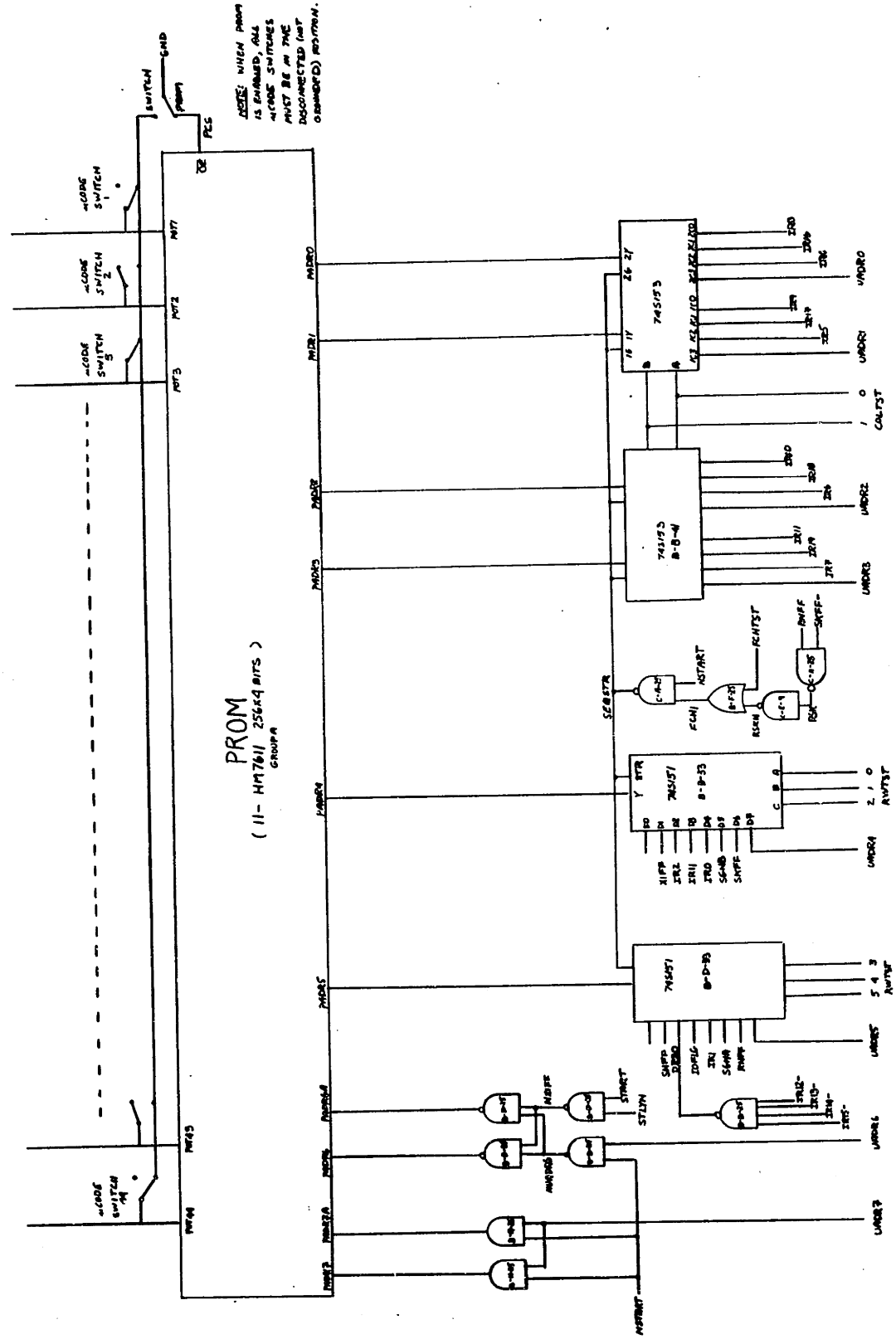


Fig. 34 PROM WITH SEQUENCER

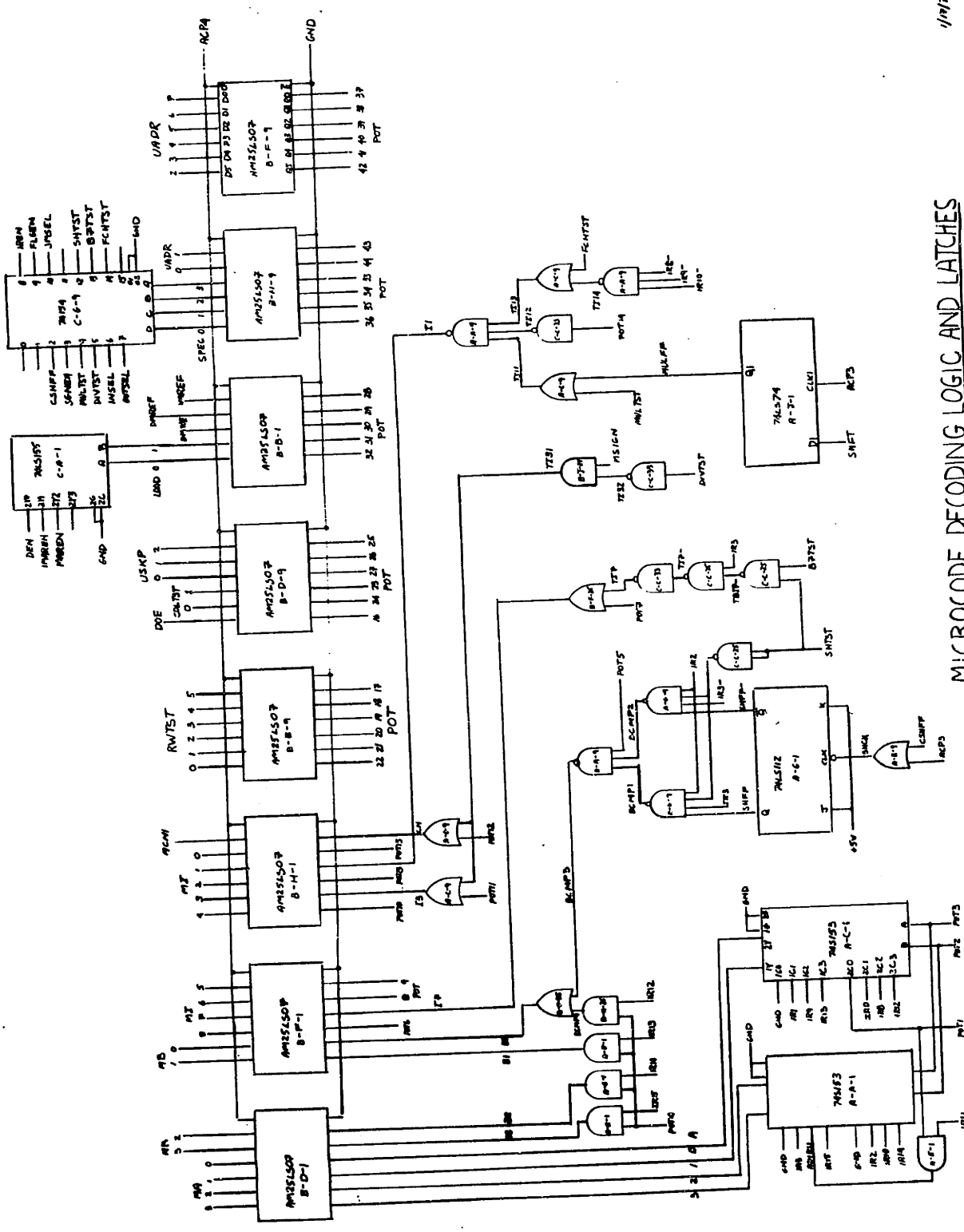


Fig. 35 MICROCODE DECODING LOGIC AND LATCHES

APPENDIX B

Microword and Data Flow Graphs

Details of Microword (see Fig. 8)

Bit# 1	2	3	A OPERAND
0	0	0	Reg. MB
0	0	1	Index Reg. Field
0	1	0	Base Reg. Field
0	1	1	Destination Reg. (Reg.D)
1	0	0	PC
1	0	1	Index Reg. Field
1	1	0	Reg. A Field
1	1	1	Reg. D Field

Bit# 4	5	D OPERAND
0	0	PC
0	1	Reg. MB
1	0	Reg. D+1
1	1	Reg. D

Bit# 6	7	8	DESTINATION
0	0	0	Reg. Q
0	0	1	NOP
0	1	0	Reg. A ← CPU Output
0	1	1	Internal RAM
1	0	0	" " Double Left Shift
1	0	1	" " Single " "
1	1	0	" " Double Right "
1	1	1	" " Single " "

Bit# 9	10	11	FUNCTION
0	0	0	R+S
0	0	1	S-R
0	1	0	R-S
0	1	1	R OR S
1	0	0	$\overline{R}$ AND S
1	0	1	$\overline{R}$ AND S
1	1	0	R EXCLUSIVE OR S
1	1	1	R EXCLUSIVE NOR S

Bit # 12	CARRY IN
0	ZERO
1	ONE

Bit# 13	14	15	SOURCE
0	0	0	AQ
0	0	1	AB
0	1	0	0Q
0	1	1	0B
1	0	0	0A
1	0	1	DA
1	1	0	DQ
1	1	1	D0

Bit# 16	OUTPUT ENABLE
0	CPU
1	DATA MEMORY or INPUT BUFFER

Bit #17	18	19	ROW SEQUENCER TEST
0	0	0	-
0	0	1	SHFF (Shift Flip-flop)
0	1	0	Reg. D= ZERO
0	1	1	I/O Flag- Busy and Selected
1	0	0	IR bit 1- Shift
1	0	1	FF A- Sign
1	1	0	RUN FF
1	1	1	NEXT ADDRESS (From Latch)

Bit#20	21	22	ROW SEQUENCER TEST
0	0	0	-
0	0	1	X1FF (X1 Flip-flop)
0	1	0	IR bit 2 (Shift D/S)
0	1	1	IR bit 11 (Indirect bit)
1	0	0	IR bit 0 (Shift)
1	0	1	FF B- Sign
1	1	0	SKIP FF
1	1	1	NEXT ADDRESS (From Latch)

Bit#	23	24	COLUMN SEQUENCER
	0	0	IR bits 8-11 (For SHIFT)
	0	1	OP Code Dispatch
	1	0	FUNC Dispatch
	1	1	NEXT ADDRESS (From Latch)

Bit#25	26	27	SKIP FIELD
0	0	0	CARRY OUT
0	0	1	JM Test
0	1	0	Always
0	1	1	ZERO
1	0	0	SIGN
1	0	1	FLAG Conditions
1	1	0	IR Class II Skip Conditions
1	1	1	NEVER

Bit#	28	INSTRUCTION MEMORY REFERENCE
	0	READ or WRITE
	1	NOP

Bit#	29	DATA MEMORY REFERENCE
	0	READ or WRITE
	1	NOP

Bit#	30	DATA MEMORY WRITE ENABLE
	0	WRITE
	1	READ

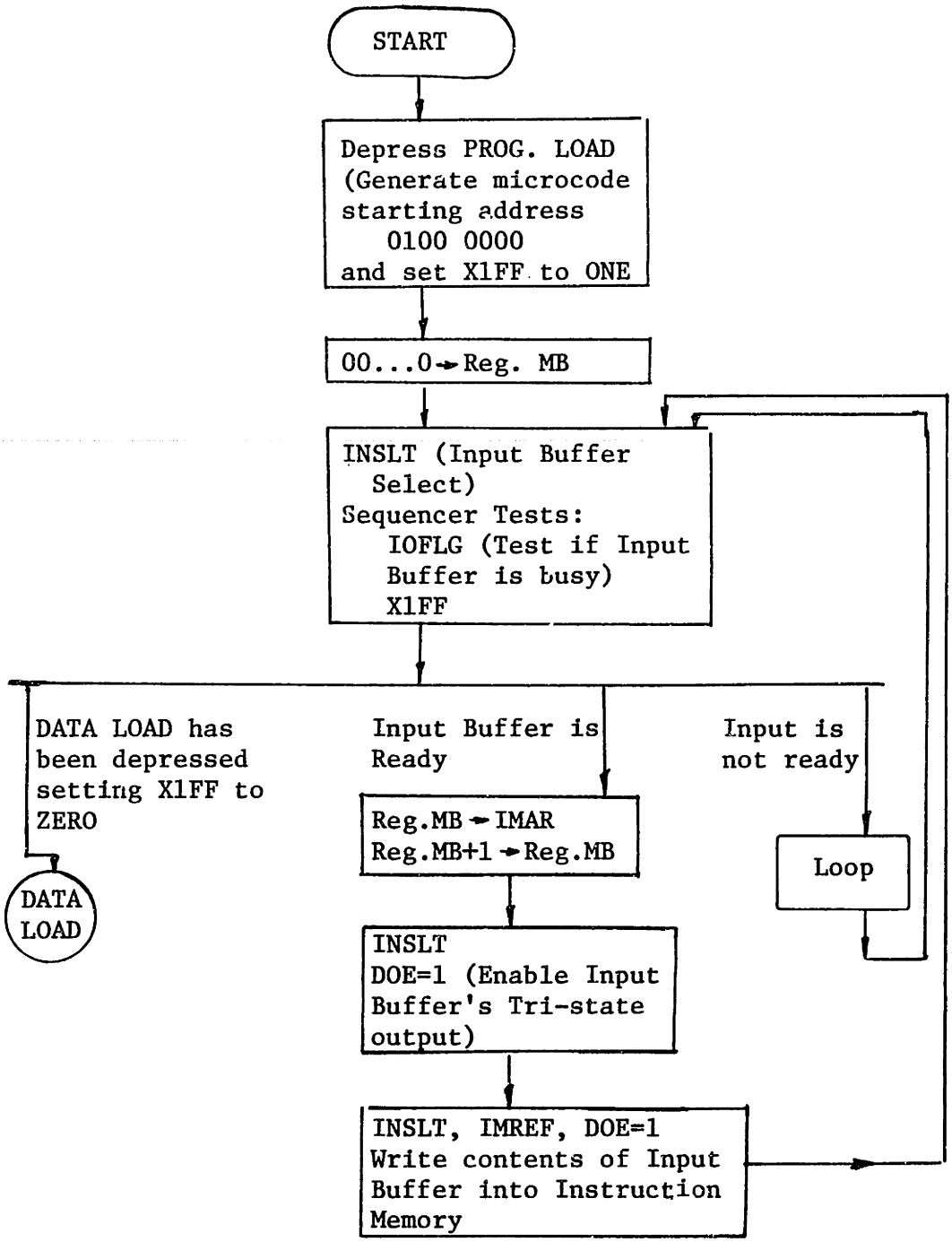
Bit#31	32	LOAD
0	0	Data Memory Buffer Register (MBR) or OUTPUT Buffer
0	1	Instruction Memory Address Register (IMAR)
1	0	Data Memory Address Register (MAR)
1	1	NOP

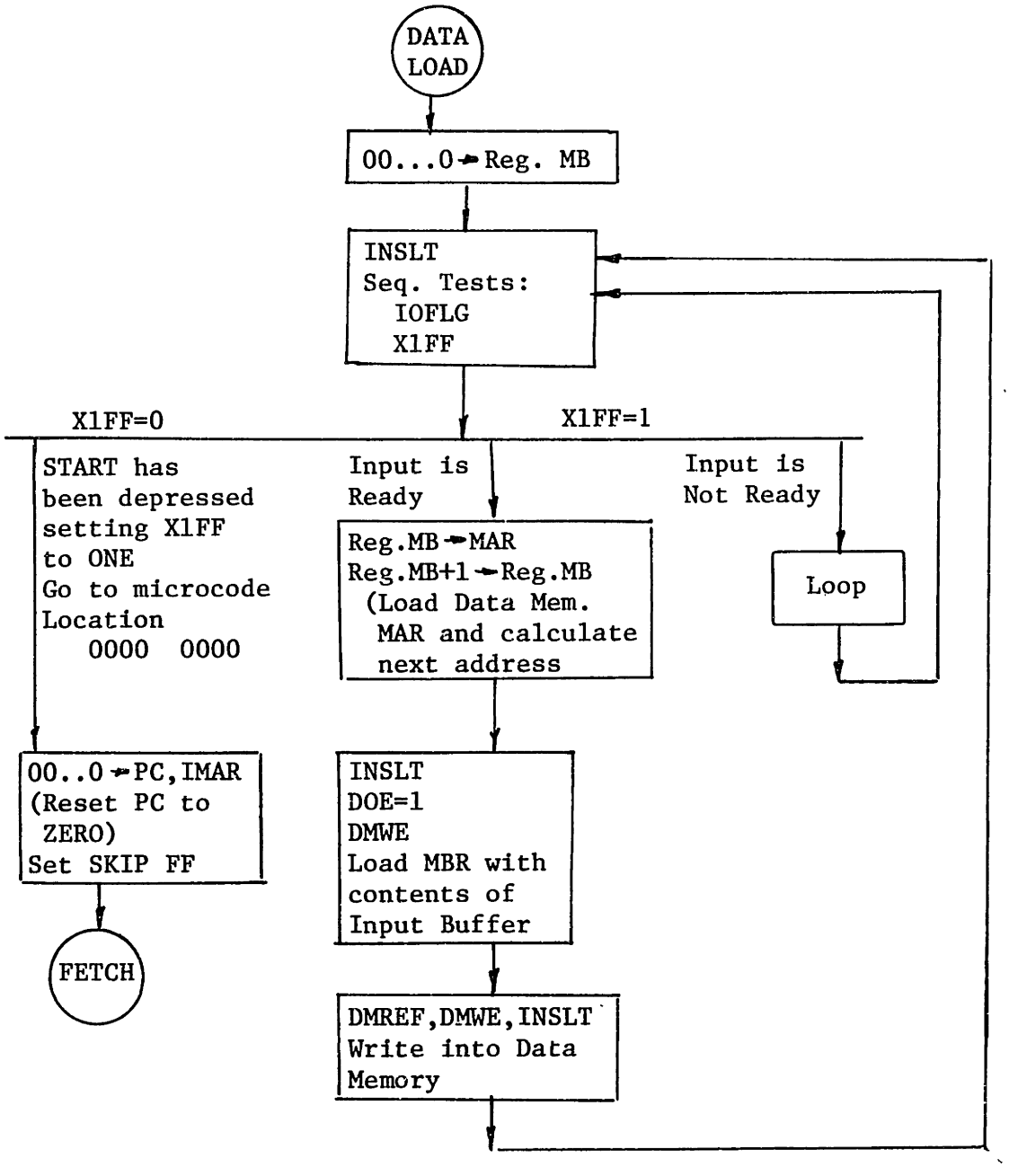
Bit#33	34	35	36	SPECIAL
0	0	0	0	-
0	0	0	1	-
0	0	1	0	CSHFF (Complement SHFF)
0	0	1	1	SGNEN (Load Signs into FF A and B)
0	1	0	0	MULTST (Enable Multiplication Test)
0	1	0	1	DIVTST (" Division ")
0	1	1	0	INSEL (Select INPUT Buffer)
0	1	1	1	OUTSEL (" OUTPUT ")
1	0	0	0	IREN (Load IR)
1	0	0	1	FLGEN (Enable FLAG Logic)
1	0	1	0	JMSEL (" JM ")
1	0	1	1	-
1	1	0	0	SHTST (SHIFT Test)
1	1	0	1	B7TST (Modify bit I7 of Microprocessor Microword- Use for SHIFT and Deposit)
1	1	1	0	FCHTST (Fetch Test)
1	1	1	1	-

Bits 37-44 NEXT MICROWORD ADDRESS

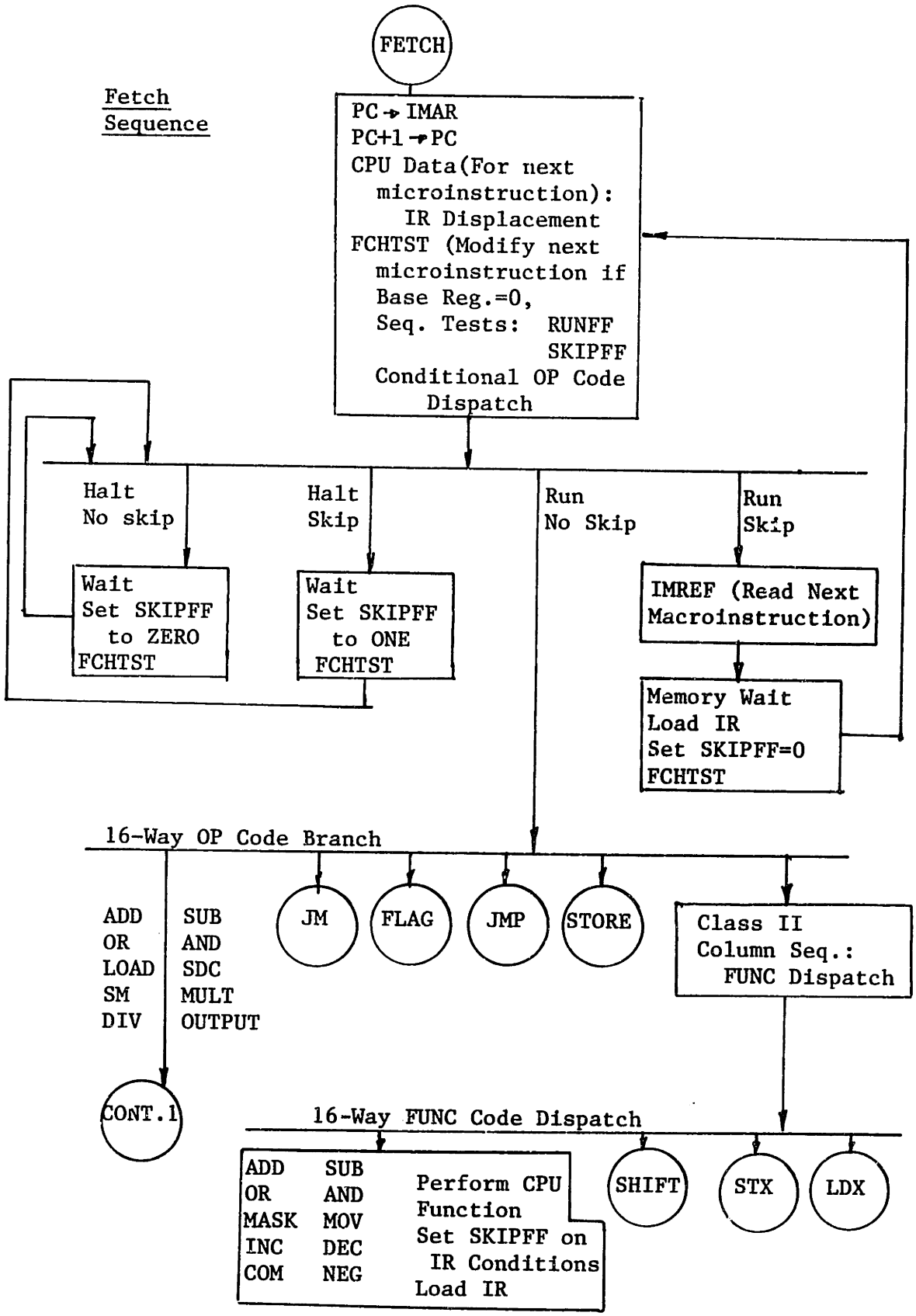
MICROCODE DATA FLOW GRAPHS

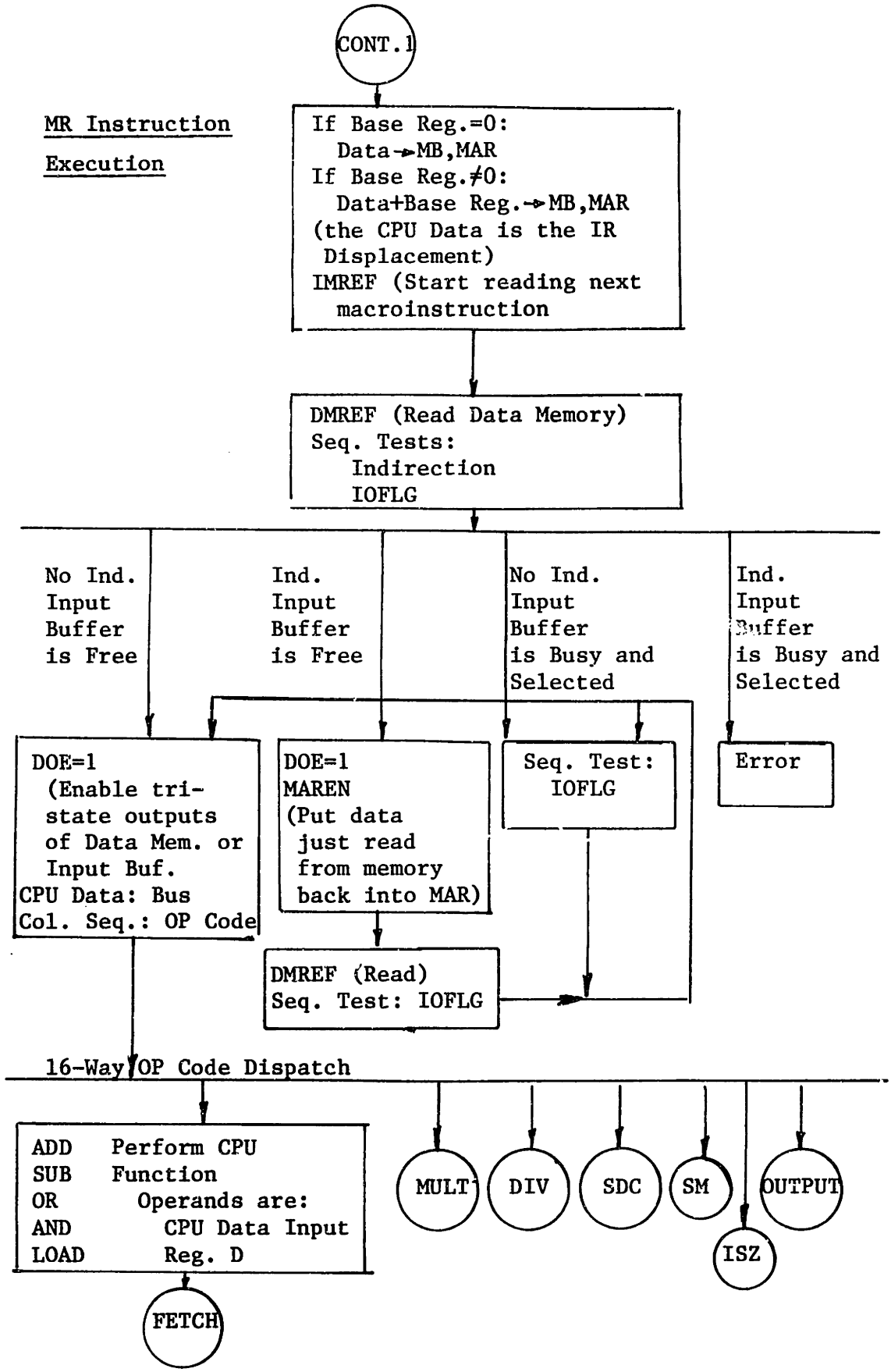
Program Load

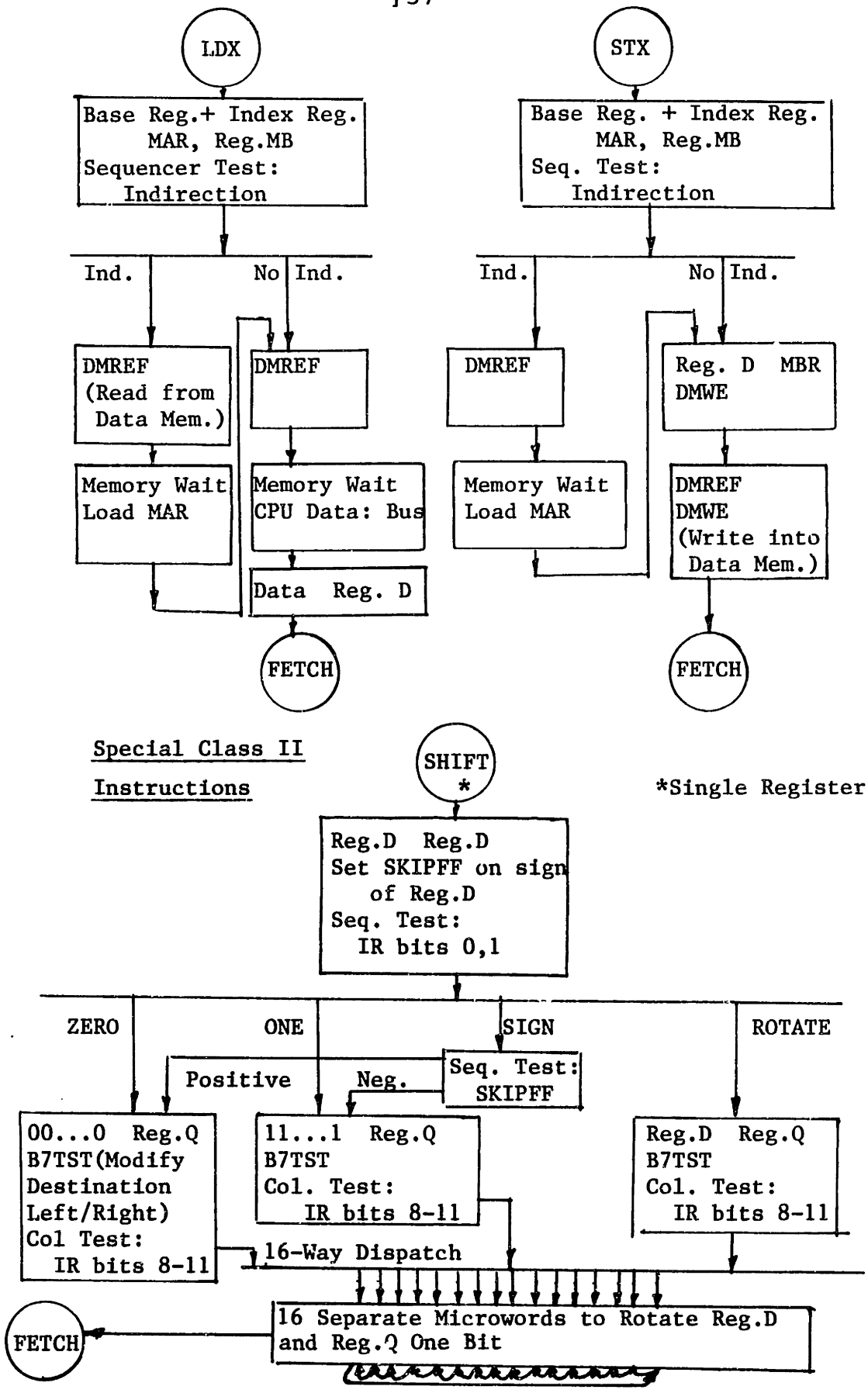


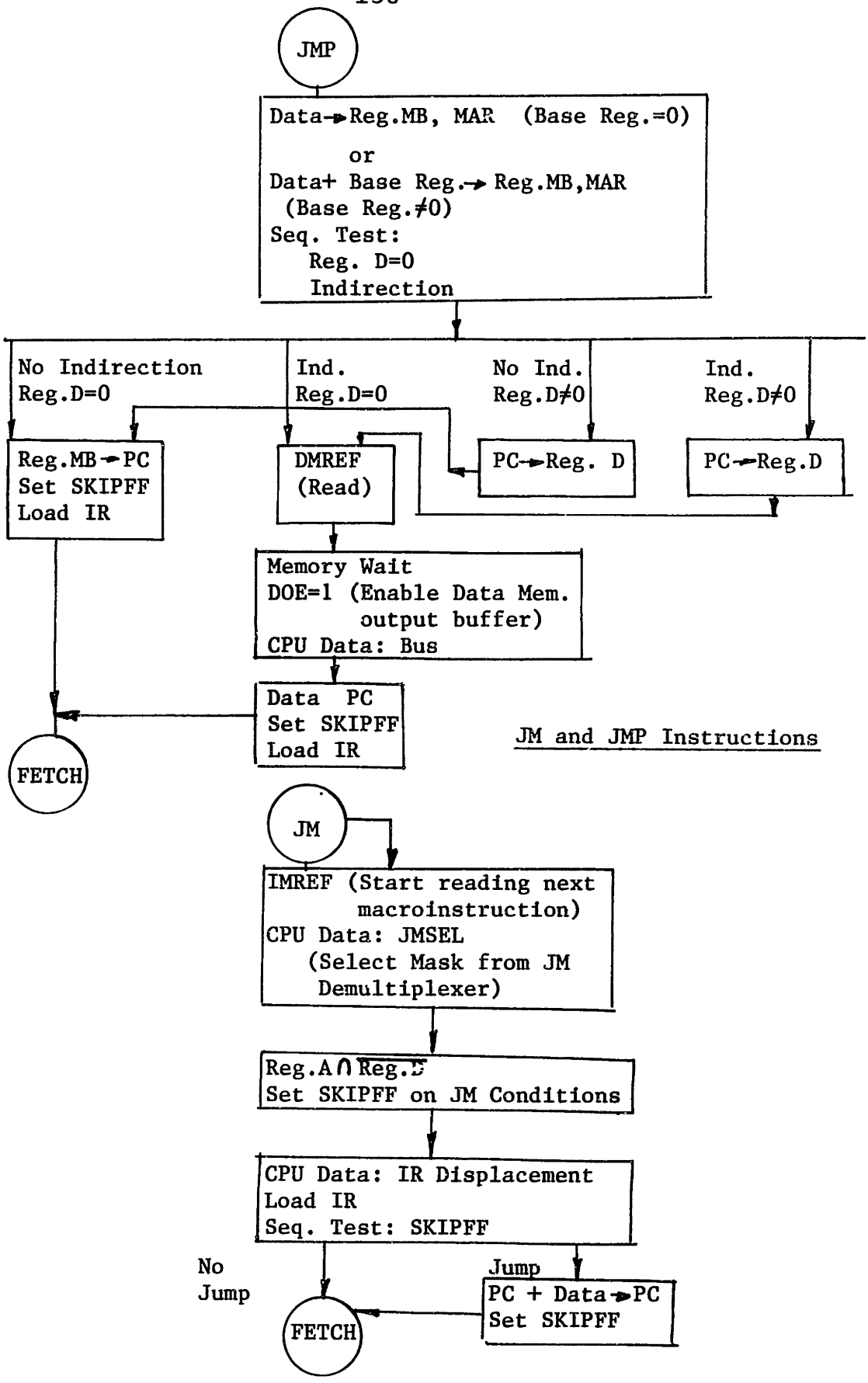


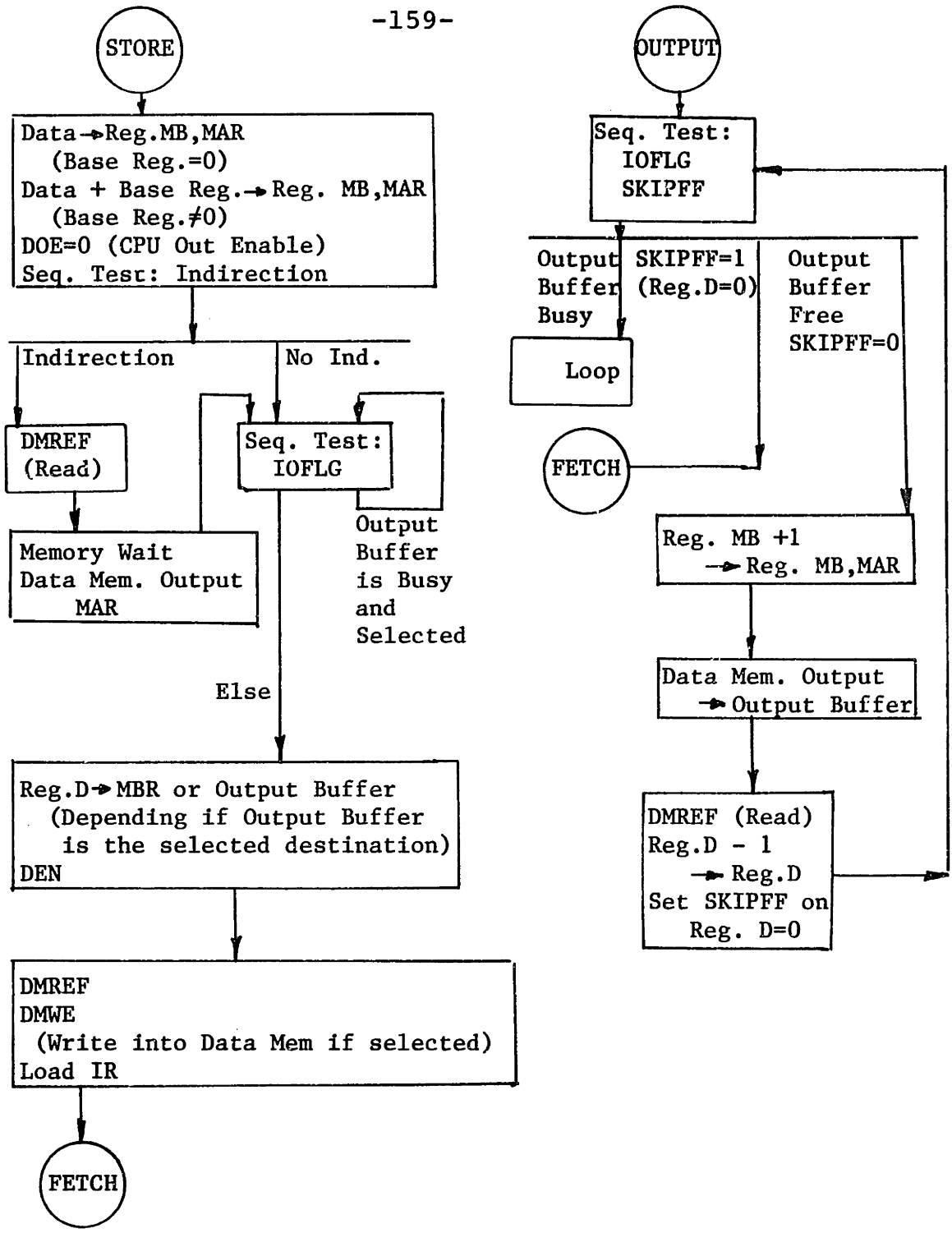
Data Memory Load and Start Sequence


FETCH
Fetch Sequence
PC → IMAR
PC+1 → PC
CPU Data(For next microinstruction):
IR Displacement
FCHTST (Modify next microinstruction if Base Reg.=0,
Seq. Tests: RUNFF
SKIPFF
Conditional OP Code Dispatch
Halt No skip
Halt Skip
Run No Skip
Run Skip
Wait
Set SKIPFF to ZERO
FCHTST
Wait
Set SKIPFF to ONE
FCHTST
IMREF (Read Next Macroinstruction)
Memory Wait
Load IR
Set SKIPFF=0
FCHTST
16-Way OP Code Branch
ADD
OR
LOAD
SM
DIV
SUB
AND
SDC
MULT
OUTPUT
JM
FLAG
JMP
STORE
Class II
Column Seq.:
FUNC Dispatch
CONT.1
16-Way FUNC Code Dispatch
ADD
OR
MASK
INC
COM
SUB
AND
MOV
DEC
NEG
Perform CPU Function
Set SKIPFF on IR Conditions
Load IR
SHIFT
STX
LDX


CONT.1
MR Instruction Execution
If Base Reg.=0:
Data→MB,MAR
If Base Reg.≠0:
Data+Base Reg.→MB,MAR
(the CPU Data is the IR Displacement)
IMREF (Start reading next macroinstruction
DMREF (Read Data Memory)
Seq. Tests:
Indirection
IOFLG
No Ind. Input Buffer is Free
Ind. Input Buffer is Free
No Ind. Input Buffer is Busy and Selected
Ind. Input Buffer is Busy and Selected
DOE=1
(Enable tri-state outputs of Data Mem. or Input Buf.
CPU Data: Bus
Col. Seq.: OP Code
DOE=1
MAREN
(Put data just read from memory back into MAR)
Seq. Test: IOFLG
Error
DMREF (Read)
Seq. Test: IOFLG
16-Way OP Code Dispatch
ADD
SUB
OR
AND
LOAD
Perform CPU Function
Operands are:
CPU Data Input
Reg. D
MULT
DIV
SDC
SM
OUTPUT
ISZ
FETCH

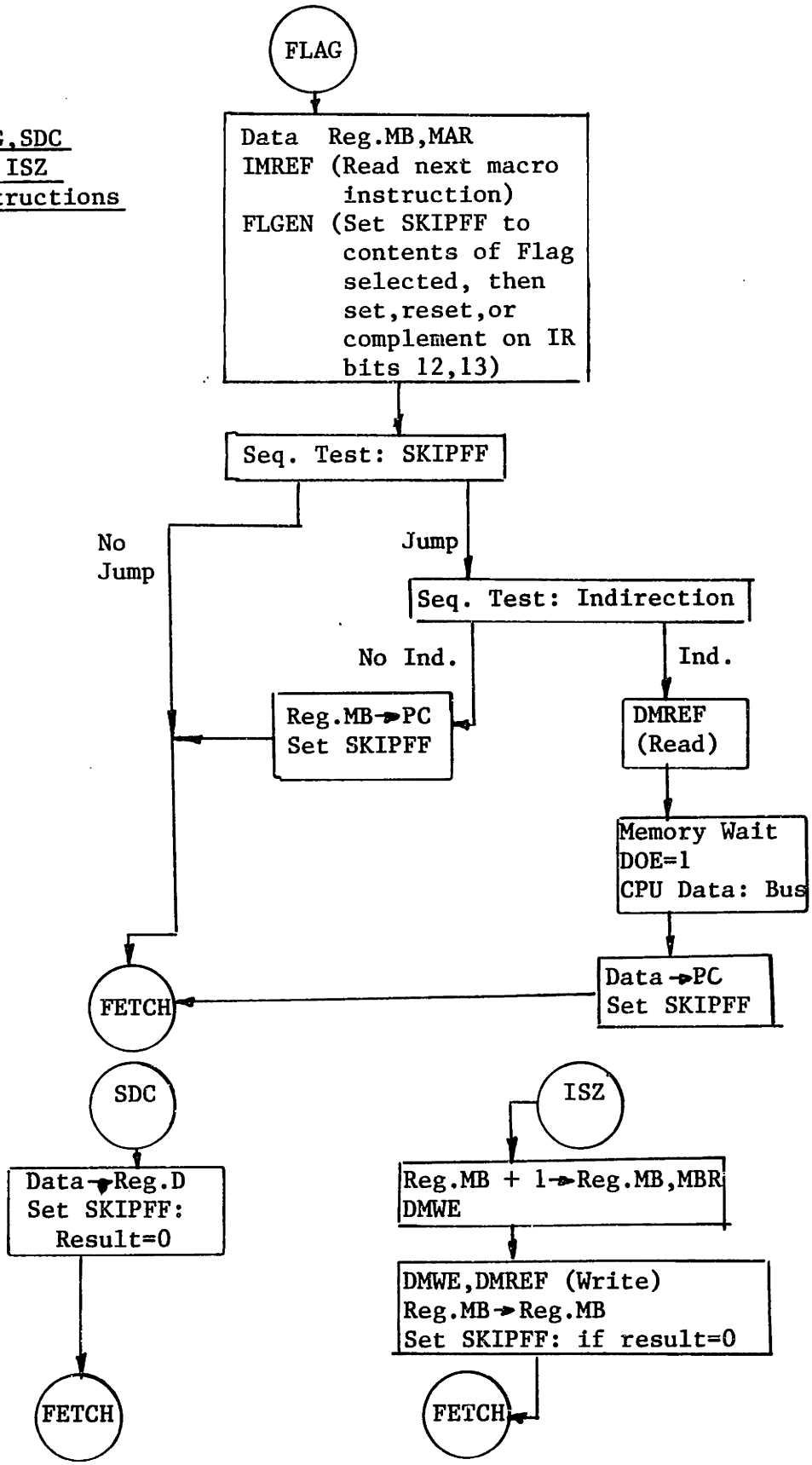

LDX
Base Reg.+ Index Reg.
MAR, Reg.MB
Sequencer Test:
Indirection
Ind.
No Ind.
DMREF
(Read from
Data Mem.)
Memory Wait
Load MAR
DMREF
Memory Wait
CPU Data: Bus
Data Reg. D
FETCH
STX
Base Reg. + Index Reg.
MAR, Reg.MB
Seq. Test:
Indirection
Ind.
No Ind.
DMREF
Memory Wait
Load MAR
Reg. D MBR
DMWE
DMREF
DMWE
(Write into
Data Mem.)
FETCH
Special Class II
Instructions
SHIFT
*
*Single Register
Reg.D Reg.D
Set SKIPFF on sign
of Reg.D
Seq. Test:
IR bits 0,1
ZERO
ONE
SIGN
ROTATE
Positive
Neg.
Seq. Test:
SKIPFF
00...0 Reg.Q
B7TST(Modify
Destination
Left/Right)
Col Test:
IR bits 8-11
11...1 Reg.Q
B7TST
Col. Test:
IR bits 8-11
Reg.D Reg.Q
B7TST
Col. Test:
IR bits 8-11
16-Way Dispatch
FETCH
16 Separate Microwords to Rotate Reg.D
and Reg.Q One Bit

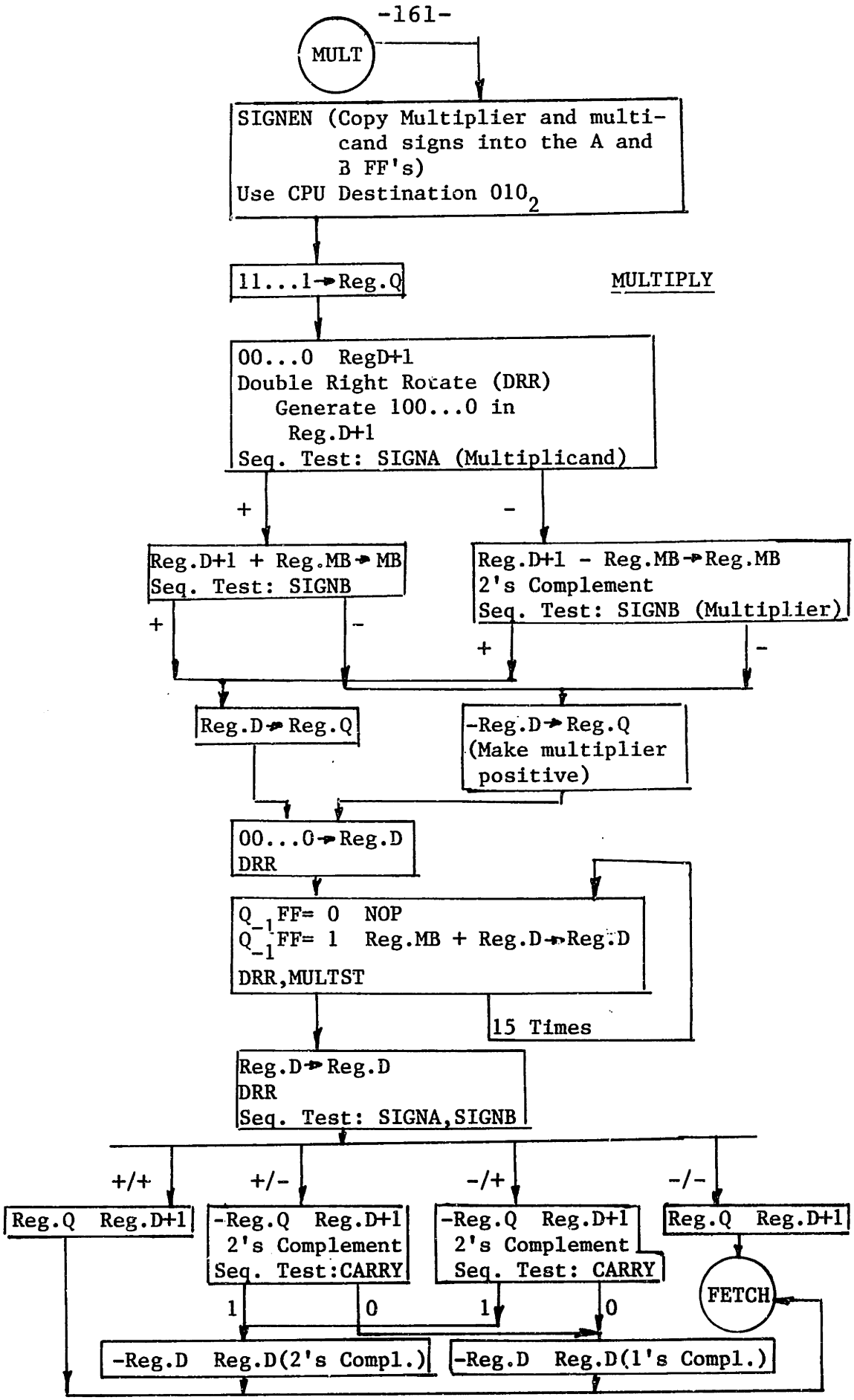

JMP
Data→Reg.MB, MAR (Base Reg.=0)
or
Data+ Base Reg.→ Reg.MB,MAR
(Base Reg.≠0)
Seq. Test:
Reg. D=0
Indirection
No Indirection
Reg.D=0
Ind.
Reg.D=0
No Ind.
Reg.D≠0
Ind.
Reg.D≠0
Reg.MB→PC
Set SKIPFF
Load IR
DMREF
(Read)
PC→Reg. D
PC→Reg.D
Memory Wait
DOE=1 (Enable Data Mem.
output buffer)
CPU Data: Bus
Data PC
Set SKIPFF
Load IR
FETCH
JM and JMP Instructions
JM
IMREF (Start reading next
macroinstruction)
CPU Data: JMSEL
(Select Mask from JM
Demultiplexer)
Reg.A∩Reg.D
Set SKIPFF on JM Conditions
CPU Data: IR Displacement
Load IR
Seq. Test: SKIPFF
No
Jump
Jump
PC + Data→PC
Set SKIPFF
FETCH

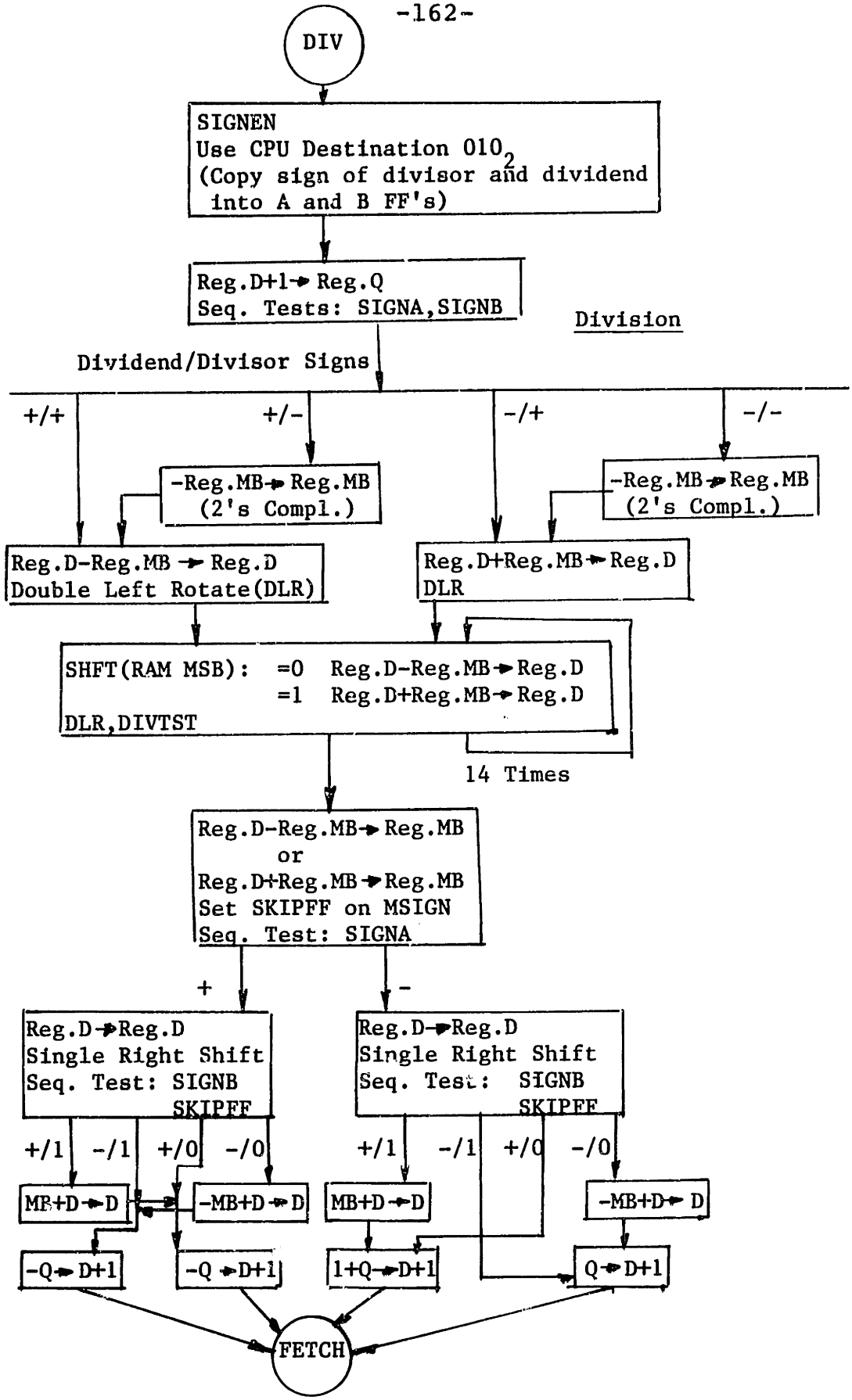


STORE and OUTPUT Instructions

FLAG, SDC and ISZ Instructions



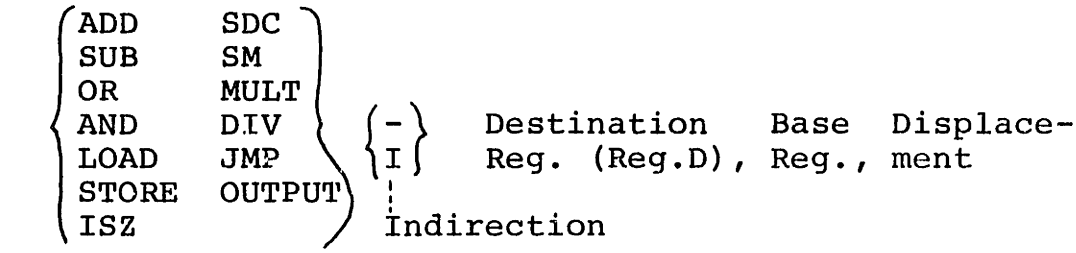

MULT
SIGNEN (Copy Multiplier and multi-
cand signs into the A and
B FF's)
Use CPU Destination 010₂
11...1→Reg.Q
MULTIPLY
00...0 RegD+1
Double Right Rotate (DRR)
Generate 100...0 in
Reg.D+1
Seq. Test: SIGNA (Multiplicand)
+
-
Reg.D+1 + Reg.MB→MB
Seq. Test: SIGNB
Reg.D+1 - Reg.MB→Reg.MB
2's Complement
Seq. Test: SIGNB (Multiplier)
+
-
+
-
Reg.D→Reg.Q
-Reg.D→Reg.Q
(Make multiplier
positive)
00...0→Reg.D
DRR
Q-1 FF= 0 NOP
Q-1 FF= 1 Reg.MB + Reg.D→Reg.D
DRR,MULTST
15 Times
Reg.D→Reg.D
DRR
Seq. Test: SIGNA,SIGNB
+/+
+/-
-/+
-/-
Reg.Q Reg.D+1
-Reg.Q Reg.D+1
2's Complement
Seq. Test:CARRY
-Reg.Q Reg.D+1
2's Complement
Seq. Test: CARRY
Reg.Q Reg.D+1
FETCH
1
0
1
0
-Reg.D Reg.D(2's Compl.)
-Reg.D Reg.D(1's Compl.)

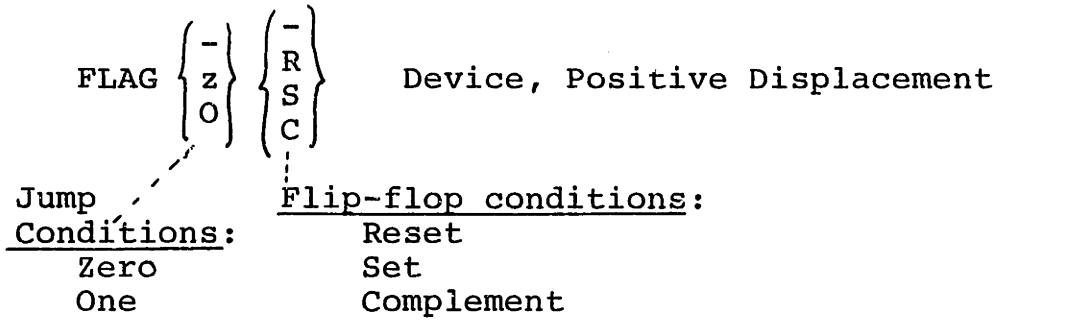

DIV
SIGNEN
Use CPU Destination 010₂
(Copy sign of divisor and dividend into A and B FF's)
Reg.D+1→Reg.Q
Seq. Tests: SIGNA,SIGNB
Division
Dividend/Divisor Signs
+/+
+/-
-/+
-/-
-Reg.MB→Reg.MB
(2's Compl.)
-Reg.MB→Reg.MB
(2's Compl.)
Reg.D-Reg.MB→Reg.D
Double Left Rotate(DLR)
Reg.D+Reg.MB→Reg.D
DLR
SHFT(RAM MSB): =0 Reg.D-Reg.MB→Reg.D
=1 Reg.D+Reg.MB→Reg.D
DLR,DIVTST
14 Times
Reg.D-Reg.MB→Reg.MB
or
Reg.D+Reg.MB→Reg.MB
Set SKIPFF on MSIGN
Seq. Test: SIGNA
+
-
Reg.D→Reg.D
Single Right Shift
Seq. Test: SIGNB
SKIPFF
Reg.D→Reg.D
Single Right Shift
Seq. Test: SIGNB
SKIPFF
+/1
-/1
+/0
-/0
+/1
-/1
+/0
-/0
MB+D→D
-MB+D→D
MB+D→D
-MB+D→D
-Q→D+1
-Q→D+1
1+Q→D+1
Q→D+1
FETCH

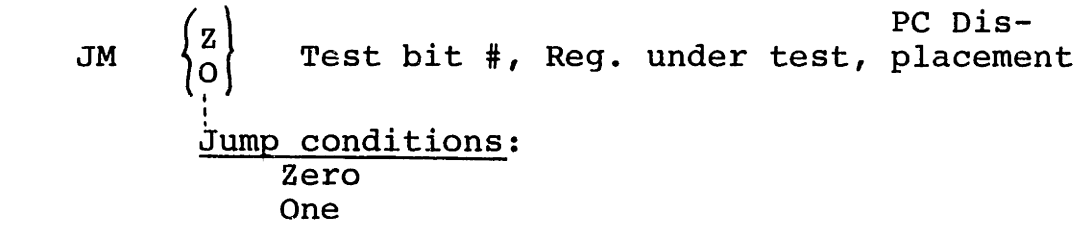
APPENDIX C

Instruction Mnemonics and Formats

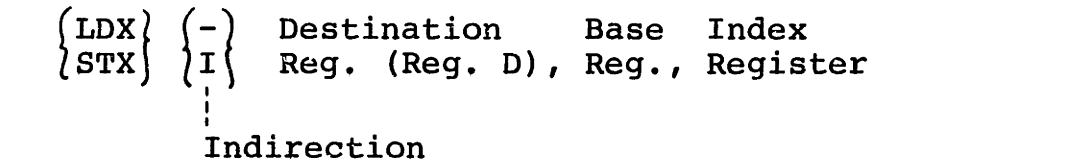
Class I Instructions

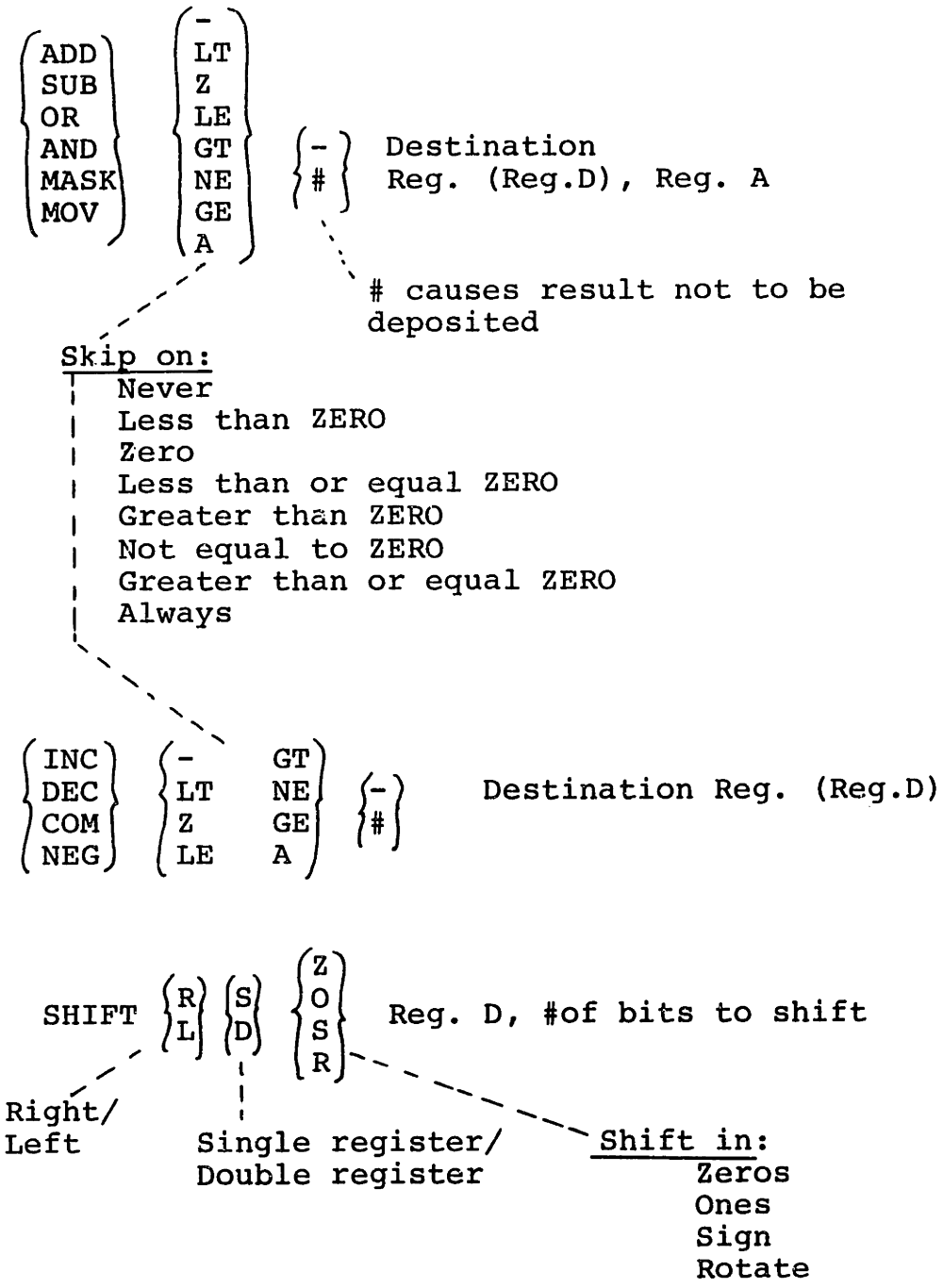






Class II Instructions




ADD
SUB
OR
AND
MASK
MOV
-
LT
Z
LE
GT
NE
GE
A
-
#
Destination
Reg. (Reg.D), Reg. A
causes result not to be deposited
Skip on:
Never
Less than ZERO
Zero
Less than or equal ZERO
Greater than ZERO
Not equal to ZERO
Greater than or equal ZERO
Always
INC
DEC
COM
NEG
-
LT
Z
LE
GT
NE
GE
A
-
#
Destination Reg. (Reg.D)
SHIFT
R
L
S
D
Z
O
S
R
Reg. D, #of bits to shift
Right/
Left
Single register/
Double register
Shift in:
Zeros
Ones
Sign
Rotate

BIBLIOGRAPHY AND REFERENCES

Speech Synthesis

Fant, C.G.M., Acoustic Theory of Speech Production, Mouton and Co., The Hague 1960.

Flanagan, J.L., "Voices of Men and Machines", JASA 51, 1375-1387, 1972.

Flanagan, J.L., Coker, C.H. et al, "Syntehtic Voices of Computers", IEEE Spectrum 7 No. 10, Oct. 1972.

Flanagan, J.L., Rabiner, L.R., ed., Speech Synthesis, Dowden, Hutchinson and Ross 1973.

Klatt, D.H., "Acoustic Theory of Terminal Analog Speech Synthesis", Proceedings, 1972, International Conference on Speech Communication and Processing, April 24-26, 1972, Boston, MA.

Klatt, D.H., "An Acoustic Theory of Terminal Analog Speech Synthesis and A Control Strategy for the Replication of a Natural Utterance" (unpublished) 1975.

Mattingly, I.G., "Synthesis by Rule of General American English", Supplement, Haskins Laboratory Status Report on Speech Research, April 1968.

Steingart, R.J., "Klatt's Voice Synthesis Programs", M.I.T. Natural Language Processing Group, Memo #95, July 1975.

Hardware and Software Design

Advanced Micro Devices
Am2900 Bipolar Microprocessor Family, 1975.
Low-Power Schottky Data Book, 1975.

Allen, J., M.I.T. Course Notes 6.032 Computation Structures, 1975.

Data General Corporation
How to Use the Nova Computer, 1972.

Digital Equipment Corporation
LSI 11 PDP11/03 Processor Handbook, 1975.
Small Computer Handbook, 1973.
PDP10 Timesharing Handbook, 1970.
PDP9 User's Handbook, 1967.

Electronic Memories and Magnetics Corporation
SEMI 4200 Preliminary Data Sheet, 1975.

Fairchild Semiconductor
Full Line Condensed Catalog, 1975.
Low Power Schottky and Macrologic TTL, 1975.
TTL Applications Handbook, 1973.

Flores, I., The Logic of Computer Arithmetic, Prentice-Hall, Inc., 1963.

Harris Semiconductor
Integrated Circuits Data Book, 1975.

Hewlett Packard
2100 Computer Microprogramming Guide, 1971.

Intel Corporation
Intel 8080 Microcomputer Systems Users Manual, 1975.

Signetic Corporation
Signetics Data Book with Applications Supplement, 1974.

Sloan, M.E., Computer Hardware and Organization, Science Research Associates, Inc. 1976.

Steingart, R.J., Wirewrap Programs and Documentation for Speech Processor on file with Prof. J. Allen, M.I.T., Room 36-581, Cambridge, MA. Jan. 1975.

Stone, H.S., et al, Introduction to Computer Architecture, Science Research Associates, Inc., 1975.

Digital Signal Processing

Oppenheim, A.V., Schafer, R.W., Digital Signal Processing, Prentice-Hall Inc., 1975.

Rabiner, L.R., Gold, B., Theory and Applications of Digital Signal Processing, Prentice-Hall Inc., 1975.