Data Processing Division
Product Development Laboratory
Box 390, Poughkeepsie, New York

International Business Machines Corporation

Telephone: Globe 4-1000

November 12, 1959

Mr. Harlan E. Anderson, Chairman
EJCC Publication Committee
Digital Equipment Corporation
Maynard, Massachusetts

Dear Mr. Anderson:

Enclosed are four copies of the manuscript by Mr. Leo Hellerman of this Laboratory, entitled "A Computer Analytic Method for Solving Differential Equations."

Included are a set of the figures for reproduction, with copies, an abstract, and the author's biographical information. The figures are identified by number, and a separate sheet contains the captions for all figures.

I trust that this material meets your specifications. If we can supply any further information, please don't hesitate to let us know.

Sincerely yours,

P. J. Nelson

P. J. Nelson, Editor
Publications Department
Laboratory Communications

PJN:am
Enc.
cc: Mr. Leo Hellerman

File Copy

BIOGRAPHY

Leo Hellerman (IRE Member, 1959) was born on February 8, 1924 in Brooklyn, N. Y. He received the BEE degree from the College of the City of New York, N. Y. in 1946, the MA degree in mathematics from George Washington University, Washington, D. C. in 1952, and the PhD in mathematics from Yale University in 1958. His Doctoral thesis was concerned with minimal sets on a circle, in topological dynamics.

He was an Assistant in Instruction in Mathematics at Yale University from 1954 to 1946, and a non-resident Instructor in Electrical Engineering MIT, in 1957 and 1958. Since 1956 he has been with the Data Systems Division of IBM, Poughkeepsie, N. Y., where he has studied techniques for reliable design of electronic circuitry, developing some machine programs using Monte Carlo methods in design and analysis.

He is a member of the American Mathematical Society.

A COMPUTER ANALYTIC METHOD FOR SOLVING DIFFERENTIAL EQUATIONS

by

Leo Hellerman

ABSTRACT

The basic idea in mathematical symbol manipulation on a stored-program digital computer is the correspondence of symbols with storage locations, and of functions with computer programs. This principle underlies our analytic method for finding the derivative of a function. The nature of a term, whether constant, the variable, or a dendrite (composed from other terms, its "branches"), is recognized by the location of the storage cell corresponding to this term. The derivative found is not a number equal to the quotient of finite differences, as in numeric methods; it is a program for the derivative function. The method is a close parallel to differentiation by hand, applying well-known rules of elementary calculus. It uses the dendritic property of functions to keep track of results as the differentiation proceeds.

The method described is the heart of an algorithm for obtaining the formal solutions of ordinary differential equations, by generating their Taylor series. Such solutions allow many evaluations with different sets of numeric data without the necessity of solving the differential equation over and over again. We describe the implementation of this algorithm as an IBM 704 program, and give a simple illustration of its use.

Product Development Laboratory, Data Systems Division
International Business Machines Corporation, Poughkeepsie, New York

A COMPUTER ANALYTIC METHOD FOR SOLVING DIFFERENTIAL EQUATIONS

Leo Hellerman

INTRODUCTION

In recent years numeric analysis has been claiming an increasing share of overall mathematical research activity. The reason for this is apparently the need to have answers - numeric answers - to problems of modern technology, along with the development of the stored program digital computer for carrying through the computations of numeric methods. But this emphasis on numbers is also an indication of the attitude of problem solvers: to use the computer, use numeric methods. And yet these methods are not always adequate. It may be more important to know how x depends on other variables than to know that $x = 3$.

The inadequacy of numeric techniques for the solution of differential equations is highlighted by the following engineering problem. In the design of a transistor switching circuit for a high speed computer we wish to know the output current level at a particular time after the start of an input pulse. This information is contained in the solution of a non-linear differential equation, which can be solved very nicely by numeric methods on a computer in, say, ten minutes. In evaluating the reliability of this circuit with respect to component deviation and drift, we want to know the statistical distribution of outputs. A simple method for finding this is synthetic sampling, or Monte Carlo[1], but at ten minutes per solution this may not be practical. However,

---

1 L. Hellerman and M. P. Racite, "Reliability Techniques for Electronic Circuit Design" Transactions I.R.E. PGRQC, September 1958; pp. 9-16.

Monte Carlo is known to be practical in estimating distributions associated with analytic expressions. This suggests we first obtain the analytic solution of the differential equation, and then apply Monte Carlo to the solution. The methods are compared schematically in Figure 1.

In the numeric method we must solve each case anew, starting with the data and differential equation. In the analytic method, enough information is contained in the solution, so that we need solve the differential equation only once, and evaluate the solution for each case. Since a major portion of machine time is taken up with solving differential equations, there may be problems in which (1) is not practical and (2) is, provided (2) can be carried through by the computer.

The purpose of this paper is to call attention to a basic principle of analytic technique on a stored program digital computer, and to illustrate this principle by a computer algorithm, and "address calculus," for finding solutions of ordinary differential equations by analysis. We also describe the implementation of this algorithm in an IBM 704 program. We see no reason why the same technique might not be applied to a host of other mathematical problems.

THE PRINCIPLE AND GENERAL APPROACH

The principle of numeric computation in a stored program digital computer is well known: numbers are represented by the contents of storage cells, and computation is accomplished by arithmetic manipulation on these contents. Functions are represented by a finite table of numeric values. The principle of analytic computation may be stated thus: algebraic symbols are represented

by the locations of storage cells, and analysis is accomplished by manipulating addresses. Functions are represented by machine programs. An algorithm for the analytic solution of a problem is an assignment of the correspondence of algebraic symbols with addresses, and a description of the way the addresses are manipulated.

In the description of the following programs, in referring to the address of some location corresponding to some symbol S, we will say "address S." Address and symbol are equivalent, and we may use the symbol to designate the address. On the other hand "address of S" refers to some other location and address, say T, which has the address S as part of its contents. Thus the address T may be the address of S.

Our approach to the analytic solution of ordinary differential equations will be to develop the Taylor series expansion of the solution. If the differential equation is

$$y^{(k)}(x) = f\left(x; y^{(0)}(x), \ldots, y^{(k-1)}(x)\right) \tag{1}$$

then the formal solution is

$$y(x) = y(0) + y^{(1)}(0)\, x + y^{(2)}\frac{(0)x^2}{2} \ \ldots . \tag{2}$$

where the $y^{(j)}(0)$ for $j = 0, \ldots\ldots, k-1$ are assigned initial values, and for $j = k, k+1, \ldots$ are determined from f and the derivatives of f.

Thus the heart of the problem is to develop analytic differentiation on a computer. In this connection we mention the work of H. Kahrimanian[2], and the LISP Programming System[3]. However, our approach is a bit

---

2 H. G. Kahrimanian, "Analytical Differentiation by a Digital Computer," M. A. Thesis, Temple University, May 1953.

3 J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine" Quarterly Progress Report No. 53, Research Laboratory of Electronics, M.I.T., April 15, 1959

different from both of these, being a close parallel to differentiation "by hand". The SHARE routine PE PARD[4] for differentiation and partial differentiation of rational functions is a prototype of our present program. Recall that the function to be differentiated is, in the computer, a stored program. PARD examines this program as a college sophomore examines a function to be differentiated, and when it finds it to be the sum of two parts it applies the rule: the derivative of a sum is the sum of the derivatives. Or, if it finds a product, it uses D(uv) = uDv + vDu, and similarly for other differentiable combinations. Eventually the derivative of a function is expressed in this way in terms of the derivatives of constants and the independent variable, and the differentiation process is complete. The problem in doing this on a computer is doing it in a uniform and orderly way, so that the method may be applied to arbitrary differentiable functions, and so that the results of the differentiation of each term can be combined in the end to one expression (program) for the derivative.

THE DENDRITE NATURE OF FUNCTIONS

In this section we examine a stored program aspect of functions. Some notions will be defined which will facilitate the description of the differentiation algorithm.

A *binary operation* is an operation on two quantities. Addition, subtraction, multiplication, division and exponentiation are binary. *Unary operators* operate on a single quantity. Exp, log, sin, and cos are unary.

---

4 M. R. Dispensa and L. Hellerman, "Differentiation and Partial Differentiation of Rational Functions" PE PARD, SHARE distributed Program D2-445.

Let the symbol a * b have this meaning: * is a binary or unary operation. If * is binary it operates on a and b; if unary it operates on a, and b is ignored. Thus a * b may be, for example, a + b, a ÷ b, a^b, or log a, or sin a. We will say a mathematical expression is a finite dendrite if it is composed by a finite number of binary and / unary operations from a set of starting terms. We call a starting term an elementary term, or an end: it is not composed from other terms.

For example, consider the dendrite y = (x + a) b - sin x. Its branching nature is shown in Figure 2.

The elementary terms are a, b, and x. The dendritic terms are, besides y itself, (x + a) b, x + a, and sin x.

Note that the dendritic picture of y may serve as a flow chart for a program for its computation. First x is added to a, and the result is multiplied with b. Then x is operated on by some sine routine, and the result of this is combined by subtraction with (x + a) b to give y. Thus a stored program for evaluating a function is essentially a sequence of binary and unary operations, starting with operations on elementary terms. That is, a program is a dendrite.

Blocks of 1's and 2's are a convenient notation for the branches of a dendrite. If $\alpha_1 \ldots \alpha_n$ is such a block representing some dendritic term, then $\alpha_1 \ldots \alpha_n = \alpha_1 \ldots \alpha_n 1 * \alpha_1 \ldots \alpha_n 2$. The branch designations of the above example are shown in Figure 3.

A set of branches of the form $\alpha_1, \alpha_1 \alpha_2, \ldots, \alpha_1 \alpha_2 \ldots \alpha_n$, where the last branch is an elementary term, will be called a chain. All the

chains of the example are

1, 11

1, 12, 121

1, 12, 122

2, 21

A finite dendrite has a finite number of chains.

THE DIFFERENTIATION ALGORITHM

Let us suppose we are given a y-program, that is, a stored program for computing y. We wish to extend this to a program for its derivative, a Dy-program, by adding additional instructions. The block of storage for the Dy - program will include the block for the y-program. Since y is a dendrite, $y = 1 * 2$ and

$$Dy = A_1 \; D \; (1) + A_2 \; D \; (2) \tag{3}$$

Where A_1 and A_2 are functions determined by the operation * and the branches 1 and 2. That is, $A_\alpha = A_\alpha$ (*, 1, 2) where $\alpha = 1, 2$. These functions are specified in Table 1. It may happen that an A_α is the number 0, or 1, or some function which is known to exist in the y-program. This is the case for $y = u + v$ and $y = \exp u$, and in these cases it is unnecessary to place any instructions in the block reserved for the Dy-program. If an A_α is not of this type, say $A_\alpha = - u \div v^2$, then we do construct the program for this function and place it in the first available locations in the Dy-program block. Whether A_α is constructed or not, we save the addresses A_1 and A_2, in locations L (1) and L (2). Since we can only differentiate one term at a time, we also save the instruction to find D (2), in a location I (2).

TABLE 1

A_1 and A_2 in $D(u * v) = A_1 \, Du + A_2 \, Dv$

u * v	A_1	A_2	I
u + v	1	1	v
u - v	1	-1	v
u · v	v	u	v
u ÷ v	1 ÷ v	$-u \div v^2$	v
u^v, v constant	$v u^{v-1}$	0	v
exp u	exp u	0	0
ln u	1 ÷ u	0	0
sin u	cos u	0	0
cos u	- sin u	0	0

We may now go on to find D (1). Suppose 1 is dendritic and 1 = 11 * 12. Then

$$D(1) = A_{11} \quad D(11) + A_{12} \quad D(12)$$

The functions A_{11} and A_{12} are constructed as A_1 and A_2, again by Table 1, the addresses A_{11} and A_{12} are stored in L (11) and L (12), and after storing in I (12) the instruction to find D(12) we continue to find D (11).

In general, if $\alpha_1 \ldots \alpha_n$ is dendritic, then

$$D(\alpha_1 \ldots \alpha_n) = A_{\alpha_1 \ldots \alpha_n 1} \; D(\alpha_1 \ldots \alpha_n 1) + A_{\alpha_1 \ldots \alpha_n 2} \; D(\alpha_1 \ldots \alpha_n 2) \tag{4}$$

The coefficients are constructed if necessary, and the addresses $A_{\alpha_1 \ldots \alpha_{n+1}}$ stored in L ($\alpha_1 \ldots \alpha_{n+1}$); the instruction to find D ($\alpha_1 \ldots \alpha_n 2$) is saved in I ($\alpha_1 \ldots \alpha_n 2$); then we go on to D ($\alpha_1 \ldots \alpha_n 1$).

Eventually, since y is finite, we come to an elementary term, 11 ... 11. This will be a constant, the independent variable, or an initial condition $y^{(j)}(0)$, j = 0, ..., k - 1. Thus D (11 ... 11) = $A_{11 \ldots 111}$ is known, and we store this in L (11 ... 111).

At this point we have traversed one chain of the dendrite y. We may now examine the I cells for some deferred differentiation instruction, and proceed with the differentiation of this new term, until another end is reached. Continuing in this way, all chains will be completed, for there are a finite number.

It is clear from (3) and (4) that Dy is simply the sum of all products of A's, where the subscripts of the factors of each product range over a complete chain. If $A_{\alpha_1 \ldots \alpha_j} = C_{ij}$, where i stands for the i-th chain, we may write

$$D\, y = \sum_{i=1}^{s} \prod_{j=1}^{k(i)} C_{i1} \quad C_{i2} \quad \ldots C_{ij} \tag{5}$$

where i ranges over the set of chains of y, s in number, and where $C_{ik(i)}$ is the derivative of the end of the i-th chain.

Thus the Dy-program is completed by construction of a program for evaluating (5). This can be done because the addresses $A_{\alpha_1 \ldots \alpha_n}$ are at hand in the locations $L(\alpha_1 \ldots \alpha_n)$.

The algorithm as it stands requires excessive storage. To differentiate any function composed with n operations, we should allocate 2^n locations for storing the addresses $A_{\alpha_1 \ldots \alpha_n}$, for there are as many of these as n-blocks of 1's and 2's. But consider the situation when the first chain has been completed.

At that point we know

$$\begin{aligned} Dy = {} & A_1 A_{11} \ldots A_{11\ldots11} A_{11\ldots111} \\ & + A_2 D(2) \\ & + A_1 A_{12} D(12) \\ & + \ldots \\ & + A_1 A_{11} \ldots A_{11\ldots1} A_{11\ldots12} D(11\ldots12) \end{aligned} \tag{6}$$

The addresses of the A's and D's are consecutive cells in three blocks of storage, called the A_1-block, A_2-block, and I-block. The storage arrangement is shown schematically in Figure 4.

The lines in this figure indicate the formation of the products in (6).

But we do not actually need B in the accumulator, for we do not really intend to add numbers. The program is used only to recognize that B and C are composed by the binary operation addition, to form A. Thus a more compact code is possible, and desirable if the information we need is to be easily available. The code we use for A = B + C is

A : P Z E B , , C

That is, location A contains the addresses B and C in the address and decrement portion of the word, and the prefix P Z E is used to indicate that these are composed with the binary operation of addition. The code for other operations is shown in Table 2.

Table 2 also shows the detailed 704 version of Table 1. When a function K = u * v is differentiated, the construction of A_1 and A_2 and the updating of certain blocks of storage is specified by this table. After K = u * v is differentiated, the next step is to find Du. In the flow chart of Figure 6 "new K' refers to u.

The 704 program flow chart is clarified by a description of the roles played by certain blocks of storage.

(a) Constants block. All constants are given addresses of storage locations in this block.

(b) Initial conditions block. This contains the locations $y^{(j)}(o)$, $j \leq k-1$, as consecutive storage cells. When an end $y^{(j)}(o)$, $j < k-1$, is recognized, its derivative is the address $y^{(j+1)}(o)$.

(c) Variable of differentiation. This is a single storage cell.

(d) Function program block. This contains the sequence of pseudo instructions defining the function to be differentiated. The last pseudo instruction is in $y^{(k)}$ (o). New terms for the construction of A_1 and A_2, as shown in Table 2, are placed in the first available locations following $y^{(k)}$ (o), as needed. The program of pseudo instructions for formula (5) is also stored here, when all its terms have been constructed.

(e) Derivative block, D. The derivative of the initial condition $y^{(k-1)}$ (o) is $y^{(k)}$ (o), which is not an initial condition but an address in the function program block. The D-block cells contain addresses of $y^{(j)}$ (o), $j \geq k$, stored in order, so that these may be treated in a manner similar to initial conditions.

(f) A_1-block, A_2-block, and Instruction block I. The roles played by these blocks are as described in connection with Figures 4 and 5. In up-dating we add new terms as prescribed by Table 2. In down-dating we eliminate terms that are no longer needed.

(g) Factor block, F. This saves all completed non-trivial (no zero factors) A_1 chains. In transplanting an A_1 chain into the F-block, all ones and minus ones are boiled down to a single sign for the entire product. All ones are omitted from the F-block, unless the particular product contains nothing but a single one.

In the flow chart of the 704 program, K stands for the address of some pseudo order of the y-program currently under examination. The program starts with examination of the last K, $y^{(k)}$ (o). A tag bit in K will indicate that $Dy^{(k)}$ has been found, and is in the D-block, so that it need not be found over again when constructing higher derivatives.

TABLE 2

Up-dating of Dy-Program, A_1-Block, A_2-Block, and I-Block

Function K =	Code at Location K	Dy-Program New Terms for Construction of A_1 and A_2	A_1	A_2	I
u + v	PZE u , , v	None	1	1	v
u - v	PON u , , v	None	1	-1	v
u . v	PTW u , , v	None	v	u	v
u ÷ v	PTH u , , v	L + 0: MON v , , MFL1*	L	-(L+1)	v
		L + 1: PTW K , , L			
u^v	MON u , , v	L + 0: PON v , , FL1*	L+2	0	0
		L + 1: MON u , , L			
		L + 2: PTW N , , L + 1			
exp u	MZE u , , 1	None	K	0	0
in u	MZE u , , 2	L:MON u , , MFL1*	L	0	0
sin u	MZE u , , 8	L:MZE u , , 16	L	0	0
cos u	MZE u , , 16	L: MZE u , , 8	-L	0	0

*MFL1 is the address of -1; FL1 is the address of 1.

The series construction, which involves multiplying each derivative by the appropriate power of $x - x_0$ and dividing by factorials, is straight-forward, and is not shown in the flow chart.

The Taylor series solution of the differential equation which is finally obtained is in the form of a sequence of pseudo instructions. It is always possible to convert these and print them on paper using familiar mathematical symbols, but we do not do this and will hardly ever want to. If a differential equation is sufficiently complicated to warrant using the program, the chances are that any significant information in the expression for the solution will be hidden in its complexity. If w is some complicated function of x, y, and z, $w = f(x, y, z)$, and we want to find out how w depends on x, it will do no good to inspect the expression f. Instead, we picture w versus x by evaluating f for a range of y values. Similarly, if we wish to study w versus y, we evaluate f with a range of y values. The point is, we need find the program for f only once. We may then evaluate it numerically as many times as we wish, illuminating the dependence on any desired variables.

To evaluate the solution obtained it is necessary to convert the pseudo code program to a regular machine language code, and to supply numeric data. This is done by interpretive and output routines. The flow of information is shown in Figure 7.

An example of the solution of a differential equation, showing the kind of information that can be obtained from these solutions, is shown in Figures 8, 9, and 10.

CONCLUSION

We have described an analytic method for finding a series solution of ordinary differential equations on a stored program digital computer. Note, however, that the present IBM 704 program for implementing this method has room for improvement. Indeed, in the present program little attempt is made to simplify the generated derivative expression. This is a severe waste of storage capacity, and unduly limits the number of series terms that can be found. Further, the unsimplified expression, containing redundant and irrelevant terms, increases the machine time for evaluating a solution. For this reason we cannot now obtain a significant estimate of the merit of the analytic method in comparison to conventional numeric techniques.

The method should be useful, in illuminating local properties of solutions. It also appears to lend itself to extending solutions by analytic continuation, but this is a problem that has not yet been attacked.

Another needed improvement, if we are to handle the differential equations of electrical engineering practice, is the capability of handling simultaneous equations. The obvious modification to do this is to provide a separate function program and D-block for each differential equation of the system.

ACKNOWLEDGEMENT

The research reported in this paper has been sponsored by the Electronics Research Directorate of the Air Force Cambridge Research Center, Air Research and Development Command, under contract AF 19(604)-4152.

The idea of using the computer to develop the series solution of a differential equation occurred in conversation with Ramon Alonzo.

The author also wishes to express his gratitude to Albert G. Engelhardt for substantial help in the planning and development of this work.

CAPTIONS for L. Hellerman, "A Computer Analytic Method for Solving Differential Equations"

Fig. 1. Comparison of numeric and analytic methods of solving a differential equation many times.

Fig. 2. The dendrite $y = (x + a) b - \sin x$.

Fig. 3. Branch designations for the dendrite $y = (x + a) b - \sin x$.

Fig. 4. Contents of A_1-Block, A_2-Block, and I-Block of storage, at completion of a chain.

Fig. 5. Contents of storage blocks after down-dating of Fig. 4 arrangement.

Fig. 6. Flow chart for successive differentiation of $y^{(k)}(x) = f(x; y^{(o)}(x), \ldots, y^{(k-1)}(x$

Fig. 7. Data flow for solution of differential equation.

Fig. 8. Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $y(o) = 1$, $a = 2$.

Fig. 9. Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $x = 1$, $a = 2$.

Fig. 10. Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $y(o) = 1$, $x = 1$.

IBM

Data Processing Division
Product Development Laboratory
Box 390, Poughkeepsie, New York

International Business Machines Corporation

Telephone: Globe 4-1000

November 12, 1959

Dr. Harlan E. Anderson, Chairman
EJCC Publication Committee
Digital Equipment Corporation
Maynard, Massachusetts

Dear Dr. Anderson:

Enclosed are four copies of the manuscript entitled "Use of a Computer to Design Character Recognition Logic" by Mr. R. J. Evey of this Laboratory.

Included are a master set of the illustrations, with copies, and an abstract. The illustrations are identified by figure number, and a separate sheet lists the captions for all the illustrations.

I trust that this material meets your specifications. Don't hesitate to let us know if you need any further information or material. Please direct any such requests directly to me for the speediest handling.

Sincerely yours,

E. F. Boomhower

E. F. Boomhower, Editor
Publications Department
Laboratory Communications

EFB:am
Enc.
cc: Mr. R. J. Evey
Dr. J. P. Lazarus

USE OF A COMPUTER TO DESIGN CHARACTER RECOGNITION LOGIC

by

R. J. Evey
International Business Machines Corporation
Product Development Laboratory
Poughkeepsie, New York

ABSTRACT

Character recognition logic for the IBM 1210 Sorter/Reader was developed with the aid of an IBM 704 computer. Characters printed with magnetic ink are quantized by the Sorter's scanning system into a 7 x 10 binary matrix which drives recognition logic of the AND and OR type. To develop this logic, computer programs which simulated the quantizing and the logic were used in conjunction with hardware that provided "real-life" character degradation. These programs and the procedure for developing the recognition logic are discussed.

USE OF A COMPUTER TO DESIGN CHARACTER RECOGNITION LOGIC

R. J. Evey

International Business Machines Corporation
Poughkeepsie, New York

I. THE SYSTEM

The IBM 1210 Sorter/Reader recognizes characters printed in a specified location on paper with magnetic ink.[1] A schematic diagram of the machine system is given in Fig. 1. The characters first come to a writing head which induces a magnetic field in the special purpose ink with which the characters are written. Next this magnetic field is sensed by a multi-channel reading head. The output of the reading head is a set of ten time-dependent voltage waves.

Actually (as Fig. 2 shows) there are thirty channels in the reading head. However, every tenth channel is "OR'ed" together (e.g., 1-11-21, 2-12-22, etc.) so there are only ten outputs. These waveforms are time-sampled and changed into binary pulses by the quantizing circuits. The output of each quantizer is seven bits of binary information per character. The outputs of the ten quantizers (one per output channel of the reading head) are stored in a 10 x 7 trigger matrix.

The final section of the system is a set of 14 logical circuits (one

---

[1] K. R. Eldredge, F. J. Kamphoefner, P. H. Wendt, "Automatic Input for Business Data Processing Systems", Proceedings of Eastern Joint Computer Conference (December 10-12, 1956), p. 69

for each character of the ABA alphabet[2]) made up of standard digital computer AND and OR components. These circuits are driven directly by the trigger matrix and operate in parallel. If a pattern in the trigger matrix satisfies any one of the logical circuits (called logics in the sequel), the corresponding character trigger is set. Recognition occurs if one and only one of these character triggers is set; otherwise the pattern is rejected.

It was mentioned previously that the thirty channels in the reading head are OR'ed together in groups of three. This means that the registration of the pattern in the matrix is unknown. So the system looks for recognition ten times per pattern; that is, it tries to recognize the pattern in the position in which it first appears in the matrix. Then the whole pattern is moved up one row at a time, with any bits in the top row being brought down into the bottom row. Thus each pattern really presents ten different patterns to the logics. Only after a pattern has "rolled" through all ten positions are the fourteen character-triggers examined for recognition or rejection.

This paper deals only with the design of the fourteen logics in this final part of the machine. It will attempt to make clear the problems which we tried to solve in this design and the methods we used to develop these circuits.

---

[2] Bank Management Commission: American Bankers Association, "The Common Machine Language for Mechanized Check Handling", Bank Management Publication 147, Automation of Bank Operating Procedure, 12 East 36 Street, N.Y. (April, 1959).

II. THE PROBLEM

The total number of different patterns possible in a 70-bit matrix is 2^{70}; and the total number of logics that can be designed for this input is $2^{2^{70}}$. The size of these numbers requires that some simplification be found to make the logical design tractable. Much of the required simplification lies in the two-dimensional correlation of bits in the matrix; that is, most of the logically possible patterns do not look anything like a possible character pattern. We found that a basic set of about 20 to 30 different patterns are obtained 90% of the time a given character is scanned. Almost all of the rest of the time a pattern is obtained which differs in one, two, or three bit positions from one of the patterns in the basic set. If these noisy bit positions are treated as don't-care positions, logical combinations of the common logical characteristics of the patterns in the basic set can be formed which will recognize virtually all the patterns obtained from scanning a character. Noisy bit positions for a given character account for over half the matrix, but this is not serious because four bits actually overdetermine the entire set of 14 characters.

The problem is thus reduced to that of finding the stable combinations of bits for a given character. At this point, however, we must consider the problem of registration -- a problem which is present in all character recognition systems. Some are designed from the point of view that this is the major problem of character sensing and must be eliminated entirely; that is, an attempt is made to design the system so that once the first character is found there is no further problem occasioned by registration.

In the 1210 system, however, even after the character has been scanned by the reading heads, the registration of the pattern in the matrix is unknown. A solution to the horizontal problem is the requirement that the leading edge of the character be located in the right-hand column of the matrix. The E13B font, with its strong leading edges, is designed for this.

The problem of vertical registration reduces to the "roll" problem, and the main problem here is that of cross-recognition. A degraded two, for example, may "roll" around to make a pretty good five (it should be noted that in the 1210 system this situation would result in a reject rather than a substitution, because both the "two" and "five" character triggers would be set). Part of the solution to this problem lies in the fact that the normal pattern is only eight rows high. Therefore, a condition which required at least one blank row at the top or at the bottom of the matrix was made a part of each logic. Once the pattern has been restricted so that it can move only a few rows vertically in the matrix and cannot roll completely around, the problem of design of the logics has been reduced to the required degree.

III. SOLUTION

A. Theory

We assumed that the set of patterns to be recognized could be approximated by the union of two other sets of patterns which we could construct. The first of these would be the set of all admissible patterns assuming ideal printing and machine operation; that is, if the edges of characters were not ragged, there were no voids and no splatter, the

magnetic field induced was uniform over the whole character, etc.

This set was generated by a program which we called the Theoretical Shape Program (TSP). Details of its operation can be found in Appendix 1. Let us say briefly here that the input to the program was a coding of each ABA character into binary bits. Each bit represented one square mil of ink. Hence, E13B characters, which are nominally 117 mils high and 91 mils wide, were entered into the 704 in the form of about 500 36-bit binary words (allowing for some blank border). This "micro-matrix" was then "scanned" by a program which simulated the operation of the reading head and quantizers. The output was a set of 10 x 7 "macro-matrices" (i.e., simply a set of patterns for each character) which were written directly onto 704 tape. The program assumed that registration, variations of magnetic density from character to character, timing across the character, fringing of the magnetic field, printing tolerances, etc. (see Appendix 1 for complete list of parameters), cannot be held firm. Hence, these "theoretical variables" were varied in the program and used to generate a set of different patterns for each character. This set was called the theoretical shapes.

We resorted to experiment to get a feel for the less systematic problems (such as voids). A hardware model of the scanning and quantizing part of the system was constructed and tied into an IBM 519 Reproducing Punch. This "print tester" scanned single characters from checks run at 1210 S/R speed and punched the resulting pattern into an IBM card. A small sample (about 10, 000 checks per character) of printing chosen to

cover the range of ABA printing specifications[3] was scanned and punched into one card per pattern. The resulting patterns (called "real-life" shapes) were transferred from cards to tape and used to indicate the types of "noise" which might be expected to degrade the theoretical shapes.

We now had two sets of patterns (each stored on its own IBM 704 tape). Each of these was now reduced to a set which was composed of only the unique shapes of the original. These patterns were now examined by a second 704 program called the Logic Processing Program (LPP -- see Appendix 2 for complete details). This program accepted, as input, logics (i.e., logic statements) punched into cards in a "Boolean" notation. It interpreted each logic and stored it in core memory; then one pattern at a time was read from tape and tested against the logic. If a pattern which represented a two, say, were being tested against a logic which was supposed to recognize two's (self-test), and if the pattern was not recognized by the "two" logic, but met a preset number of conditions (see Appendix 2), the pattern was printed. If it met the logic, that fact was simply noted in summary tables printed at the end of a run. If a two were being tested against a logic which was supposed to recognize, say, fives (cross-test), the criteria for printing the pattern or entering the tables were nearly the reverse of those for self-testing.

B. Method of Designing Logics

With these tools at hand, the following method was used to design the logics. A simple trial logic consisting of single black (1) or white (0) bits was tried against the set of theoretical shapes for that character

---

3 Bank Management Commission, op. cit.

(i.e., a self-test was run against theoretical shapes). After several trials it is possible to determine a set of 10 to 15 positions consisting of single black bits inside the character outline and single white bits close to the character outline. It must be emphasized that it is always a set of "sure bits" which is found. For different criteria a different set will be found. For example, a program was written which determined the maximal set of sure bits for each theoretical character. However, in some cases, a more desirable set of sure bits would be one which distinguished sharply a given character from that character (or characters) which looked the most like it. These "sure bits" were then used as a trial logic for running a cross-test against the rest of the theoretical shapes. The result of this run would be a reduced set of "sure-bits", which were useful in telling this character apart from the other theoretical characters. Then these "useful sure bits" were used as conditions for a trial logic for the given character.

First, this trial logic would be self-tested against corresponding real-life shapes. Samples of real-life shapes would not be recognized because of voids, ink-splatter, skew, etc. By examining the tabulations and patterns printed by LPP, the designer would attempt to modify the single-bit trial logic by OR'ing a more complex condition to the sure-bits which gave trouble. This new logic would again be real-life self-tested. After a number of trials, a logic would be obtained which would recognize all of the real-life shapes the designer felt were realistic. Then the logic was real-life cross-tested and modified using a similar procedure. Here, however, the criterion for final acceptance was that no character

should be misrecognized by the logic (this was due, of course, to the system's more stringent requirements on substitutions than rejections). A flow-diagram of the above procedure is shown in Fig. 3.

Several modifications of each logic would usually have to be made at each step in the process before the logic would be considered satisfactory. Sometimes it was necessary to start from the very beginning with a search for a new set of useful sure-bits. In all cases a complete, transmissible record of the design of each trial logic, the results obtained in testing it against the trial shapes, and the reasons for modification existed in the summaries kept by LPP.

IV. CONCLUSION

Only two other methods of designing logics of this type are known to the author. One of these consists of building hardware which allows the engineer to shift wires in the model quickly (somewhat like IBM plug-boards for EAM equipment). In this way logics can be wired directly into the machine and paper can be fed through an actual model of the system. This method has the advantage that the engineer knows the logics are trying to recognize patterns which are produced under field conditions. It has the great disadvantage that there are no records of patterns successfully recognized by the logic. When a change is made in logic wiring and a retest is run, the engineer has no way of knowing whether the same patterns as before are being presented to the logic. Hence, he has no assurance that he is really comparing the new logic against the old. The new logic may work better; but it may be because it is seeing more easily recognizable patterns. This method of designing logics has been tried at IBM and has

not been as successful as the subject method in either time, cost of logics, or reliability. However, using the procedure described in this paper, a set of statements for each of the fourteen characters was developed with an expenditure of six man-months for the 704 programs (which are of an exceedingly general nature and have been used in whole or part for other applications) and two man-months for the design of the logics. Further, it was found that two different designers working independently on the same statement tended to produce logics that were equivalent in cost, performance, and the bit positions used (see Fig. 4). The best proof of the method, however, lies in the fact that the initial set of statements developed through its use have been wired into models of the 1210 S/R and have remained there unchanged after more than a year of rigorous testing.

Another method known to the author is that of devising an automatic procedure to design these logics. Most exhaustive procedures can be ruled out due to the astronomical number of possible logics, but useful procedures have been developed by limiting the complexity of the conditions used in the statements.[4] However, possibly because of this limitation, statements so produced have never been as successful in practice as those designed by people.

There was a time, nevertheless, when we felt that a definite shortcoming of this method was that it was not automatic. In the many areas in which there is an attempt being made to utilize computers for the solution of complex decision problems (e.g., theorem-proving, language translation, network analysis and synthesis, etc.), the goal is complete automation.

---

[4] P. H. Howard, "A Computer Method of Generating Recognition Logics for Printed Characters," IBM Technical Note, TN 00.10070.357 (May 5, 1959)

However, this was not our goal. We needed a reliable set of logics and we were able to utilize the computer to advantage in completing this task. It processed a trial logic against a controlled set of input patterns. It ran tests and tabulated the results of this processing. Under a variety of sense-switch controls it displayed specific items of interest to the designer. Finally, it kept accurate records of this continuing iterative process of logic design, so that previous work could be re-examined. In this way the human beings in the process were freed from monotonous tasks and could devote their experience and creative judgment to the actual task of designing logics that recognize characters.

ACKNOWLEDGEMENTS

It would be impossible to mention all the people who have helped with this project. But at the risk of omitting equally worthy contributors, I would like to single out D. R. Andrews, G. E. Bartholomew, P. C. Murley, and L. O. Nippe, who wrote most of the programs; A. J. Atrubin and P. H. Howard, whose constructive criticism and helpful suggestions contributed both to the writing of the programs and to the design of the logics; and especially Dr. J. P. Lazarus, without whom neither the project nor this paper could have been completed.

APPENDIX 1 - DETAILS OF TSP

Input to the program consisted of two sets of IBM cards. The first set consisted of a coding of the character into one-mil squares. This was accomplished in this fashion:

A detail drawing of the character was blown up to 50-times life-size. A grid marked off in squares, which represented one square-mil to the same scale, was then laid over the character. Each coordinate on this grid was marked. Hence, a person could quickly see the coordinate where each row started into black and where it left. One card was then punched per row -- with first the coordinate when black was started, when it was left, when it started again (if it did) and so on. Since each ABA character is 117 mils high nominally, this would result in 117 cards per character. A further coding was incorporated, however, in that where the edges of the character are not curving, one card may be the same as a preceding card. Hence, there is no need to repeat the next card; simply punch into the first card the number of times it is to be repeated. Fig. 5 gives the listing of the cards required to code the character 2. These cards were read by the program (actually they were put on tape and read from there) and interpreted into bits where there was black indicated in the character and blanks where the character was white.

The second set of cards (an example may be seen in Fig. 6) contained a complete set of the parameters which could be varied in the program. These parameters (and the card fields in which they were punched) were:

1. The dimensions of the macro-matrix (i.e., the output matrix or trigger matrix of the S/R).
2. The font (this would be varied by changing the first set of input cards).
3. The width of a reading channel to the nearest mil.
4. The width of the dead space between the channels to the nearest mil.
5. The horizontal sampling interval in mils.
6. The clipping level of the quantizing circuits (i.e., the height of the voltage waveform they would have to see to call it above the noise level.)
7. The integration time of the quantizing circuits.
8. The initial registration of the character (that is, whether its leading edge were sensed too soon due to magnetic fringing or other effects, right on time, or late due to missing or low-density ink).
9. Printing tolerance.

The program would first read a set of character coding input cards, interpret them, and position the coded character in storage in such a way that it simulated a character with its bottom edge on the bottom edge of a reading channel. Then a parameter card would be read and the character "scanned" in accordance with the parameters punched therein. The result of this "scan" would be a pattern (or macromatrix) which was written on tape immediately. Then the character would be "moved" (or "rolled") up one mil in its relation to the channel and land (dead space) and again scanned

in accordance with the same set of parameters. This process would continue until the character had rolled up to the position in which its bottom edge just rested on the bottom edge of the next higher channel. At this point it is obvious that we would begin to see the same set of patterns all over again. So another parameter card would be read and this process repeated for that card. This would continue until all the parameter cards for a given character were read, at which point a new set of character coding cards for the next character would be read and the whole process repeated. This process is illustrated in the simplified flow-chart of the program shown in Fig. 7.

There was one parameter which does not appear in a parameter card. That is the system of quantizing used. This was varied by re-programming. That part of the program was made into a closed sub-routine and reprogrammed whenever the engineers changed their quantizing circuits. About five different types of quantizers were tried and they had so little in common we felt this was better than attempting a general program. It should be mentioned here that after the program was used a couple of times, it was so successful in simulating the scanning that the engineers would try a new idea for quantizing here before they would try it in hardware.

APPENDIX 2 - DETAILS OF LPP

The input to LPP consisted of two parts also. First, of course, was a set of cards into which were punched the logic to be tested. These were punched in this manner:

The character for which the logic was written was punched in column 1, (A, B, C, D being used for the four special symbols of the ABA alphabet). The number of conditions was punched in columns 2 and 3. A condition is a multistaged logical AND'ing and OR'ing of trigger matrix bits which, when AND'ed with other conditions, forms the logic for the given character. No assumption of minimal form is made, so that the same logic may be decomposed in different ways into conditions. For example, if A and B are two conditions, the total logic consists of A · B and 02 is the number of conditions. AB may be taken as a condition and the total logic then has one condition. Suppose A = C+D, then there are two conditions, (C+D) ∘ B; or the logic can be written BC+BD, which is only one condition. Hence reference is most easily made to a logic picture to see what was constituted as a condition. Fig. 8, which shows a simple logic and what would be punched into the logic card, may make this clear.

Starting in column 4, a cycle of symbol-row-column started and kept up until column 72 or until all the logic was punched. If the logic had to extend over to a second card the same sequence was used; that is, character, total number of conditions, symbol, row, column, etc., starting where one left off on the preceding card. The symbols used were numerals 1 to 9, "+" for OR, a comma "," for AND, and the letter "S", which also

symbolized a logical OR but had a larger scope than the plus sign. The numerals indicated that a new condition was starting and told how many of the subconditions following it were to be satisfied (2 out of three for example). A subcondition is one bit specified by the row and column location. If the bit were to be a blank, a negative sign (X over-punch) was punched over the row.

These cards were read by the program, interpreted, and stored in memory. (See Fig. 9 for a flow-chart of LPP.) Then the program reads one character pattern (the second element of input) from tape. This pattern was tested against the logic. As we have said, if the character were being tested against its own logic and met all the conditions this was noted in a final summary table. Actually more was done with it. The whole pattern was added, a bit at a time, into a frequency table (Fig. 10). That is, this table kept track of how many times the characters had bits in each matrix position when considered in the roll position in which they were recognized by the logic. Now, if the pattern was not recognized, it was rolled through all ten roll positions, and the program kept track of the roll position in which it missed the fewest number of conditions (or the first position in case of a tie). Then the whole pattern would be printed out (these printouts were called printed patterns, or PRAT's -- Fig. 11). Further, the pattern was added into a table called a best position frequency table (Fig. 12). Further, a table was made up of the conditions which were missed. These were called condition-not-met-maps (Fig. 13) and told the conditions which kept patterns from recognizing.

If the pattern came within one condition of recognition, the count was printed on one line; but if it were two or more conditions from recognition, the count was printed on a different line.

As has been said previously, if a pattern were being tested against a logic for a different character all the above tables were entered but the criteria for entrance were simply reversed. Further, entrance was made in the tables for every roll position. In this way the CNMM (condition-not-met-maps) told what conditions were actually keeping characters from being recognized. All of these tables were printed at the end of each character run. Only a final summary table was printed at the end of the complete run (Fig. 14). This told for each character how many patterns came within 0 (i.e., complete), 1, or 2 conditions of being met.

Complete control of entrance into each of the summary tables and printing of the summaries was maintained by using a combination of control cards and sense switches. The control cards specified whether or not a certain summary was to be kept and, if so, gave a limit of conditions. If a pattern missed recognition by more than this number of conditions, the summary table for that character was not entered. Then, as the program ran, the logic designer could choose to see certain tables (or even change the course of the program) by a selection of sense-switch settings. In this way the program displayed only that data the designer thought would be helpful at any given time.

List of Illustrations

Fig. 1. System Schematic

Fig. 2. Roll Problem

Fig. 3. Statement Writing Procedure

Fig. 4. Two "2" Logics

Fig. 5. Coding for Character 2 for TSP

Fig. 6. Parameter Card for TSP

Fig. 7. TSP Flow Chart

Fig. 8. Simple Logic

Fig. 9. Logic Processing Program

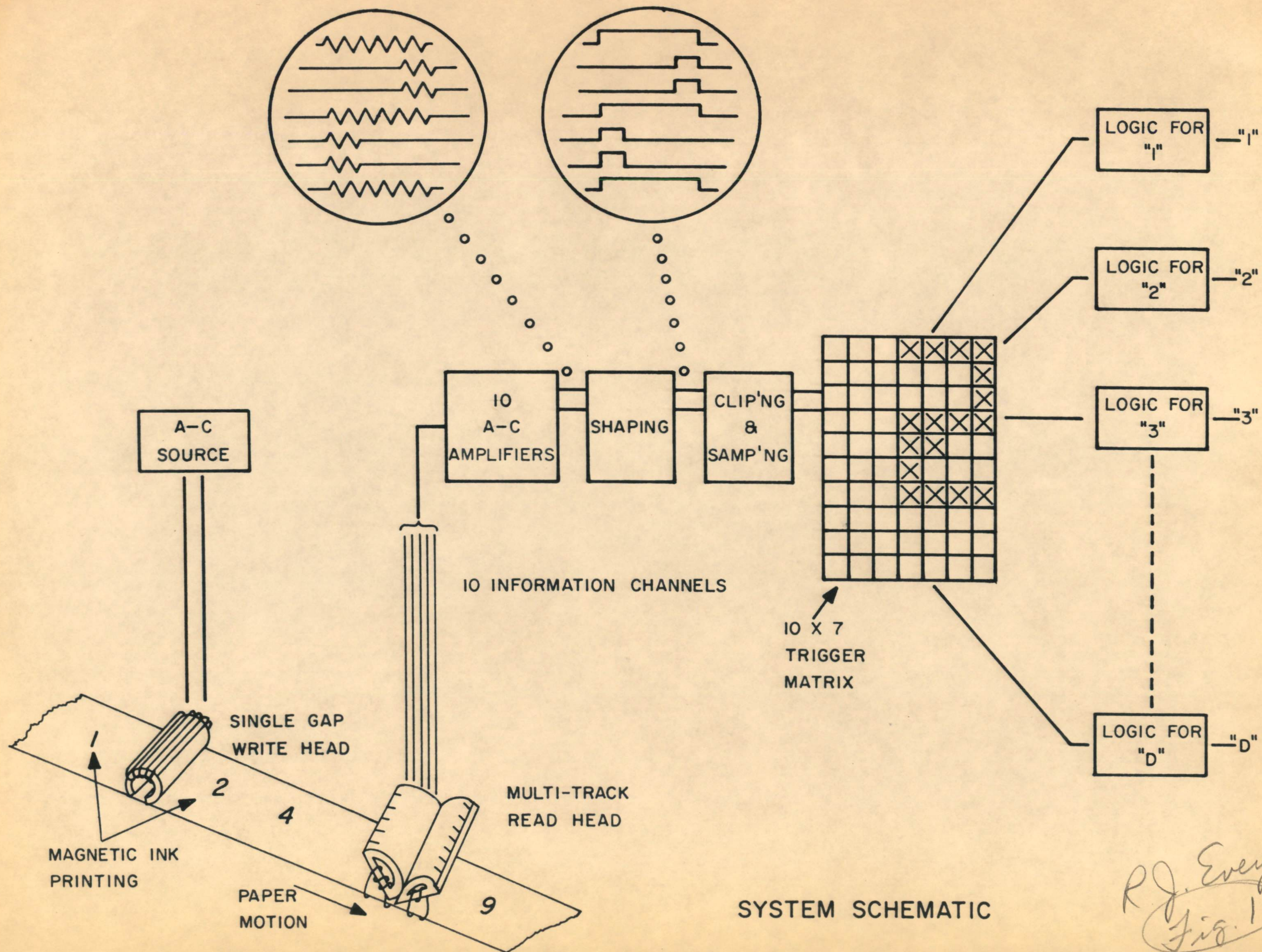
Fig. 10. Frequency Table (FT)

Fig. 11. Printed Pattern (PRAT)

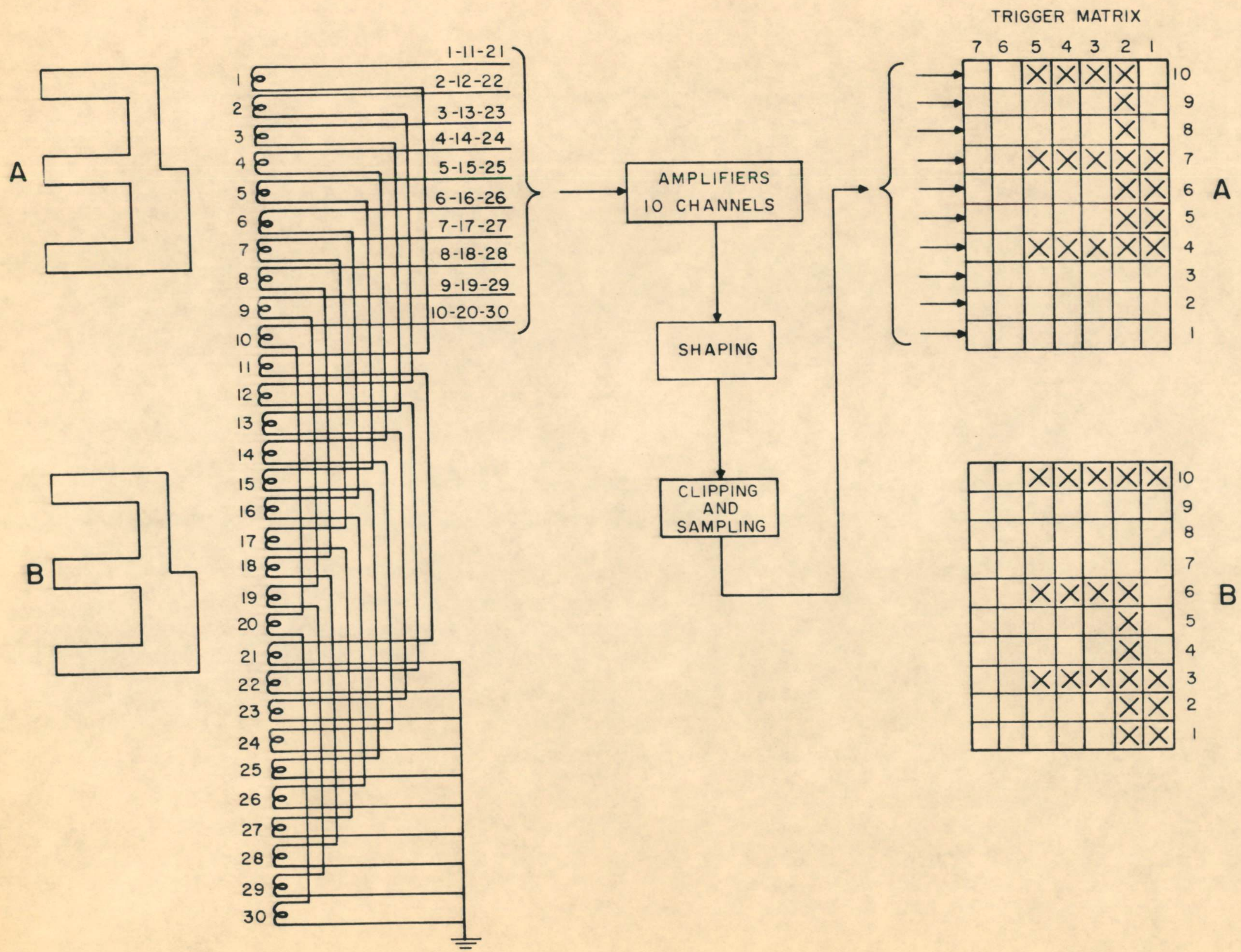
Fig. 12. Best Position Frequency Table (BPFT)

Fig. 13. Condition-Not-Met Map (CNMM)

Fig. 14. Summary

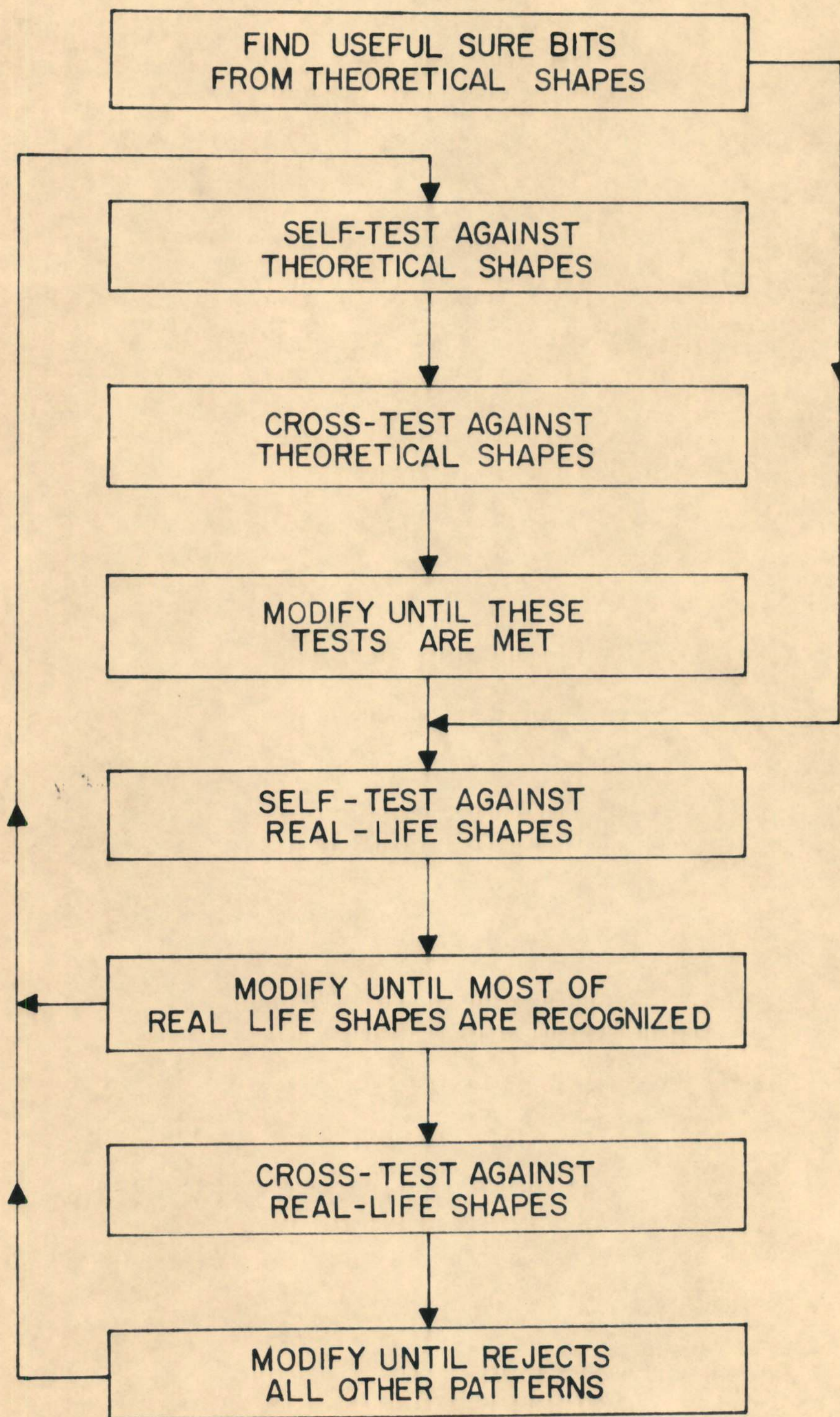


SYSTEM SCHEMATIC



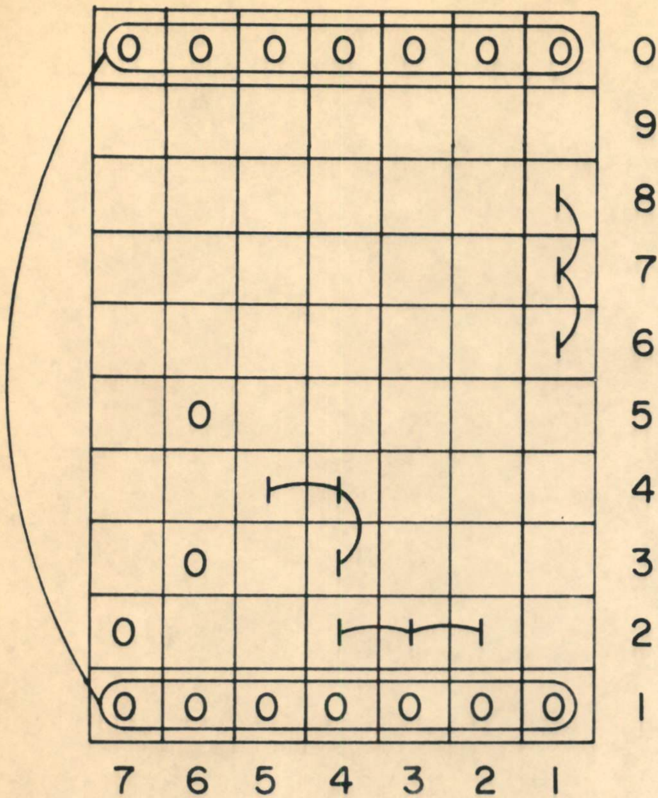
ROLL PROBLEM

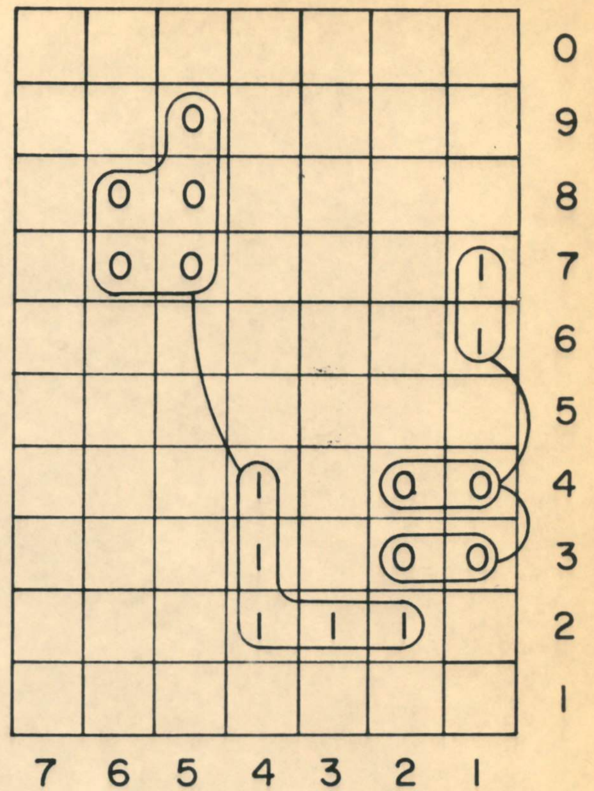
R. J. Evey
Fig. 2

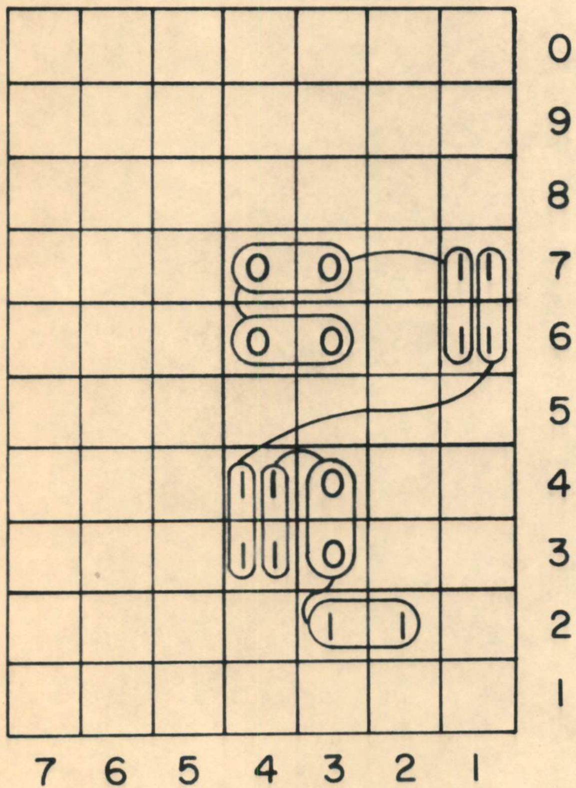


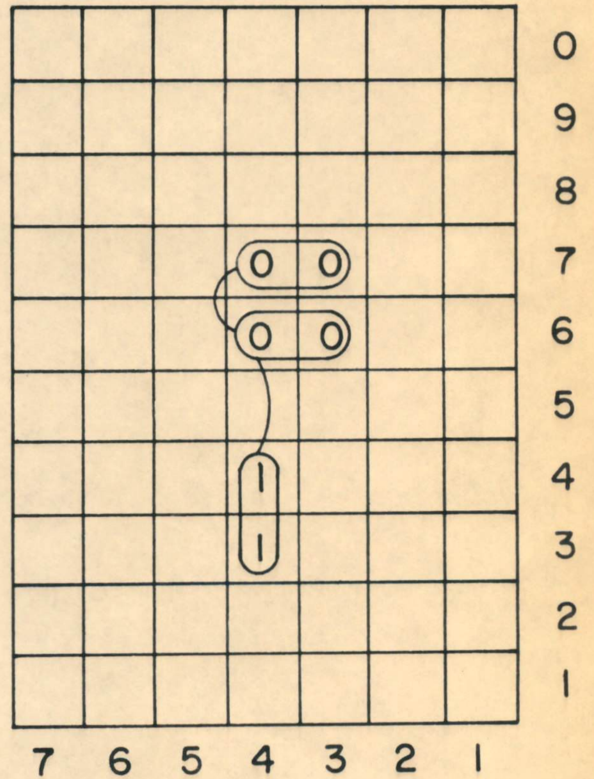
STATEMENT WRITING PROCEDURE

R. J. Evey
Fig. 3



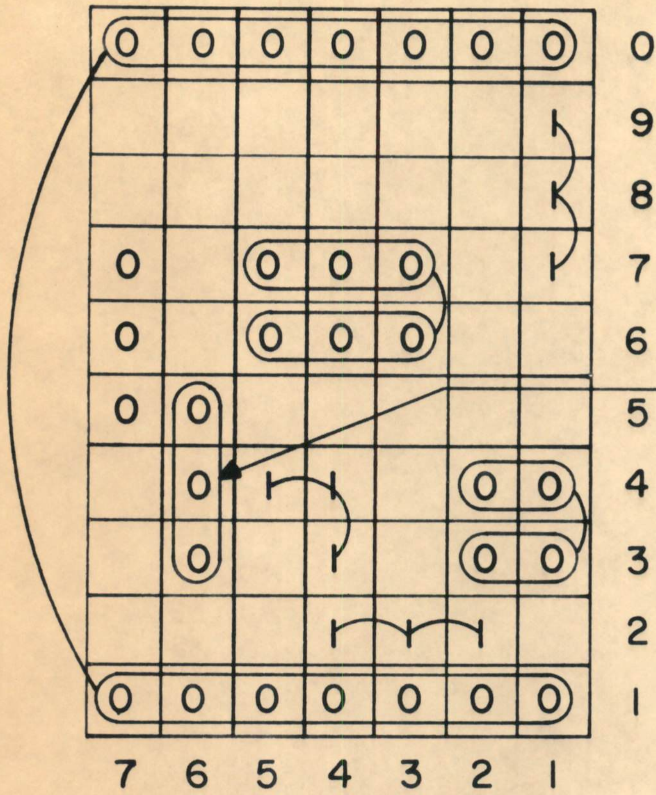


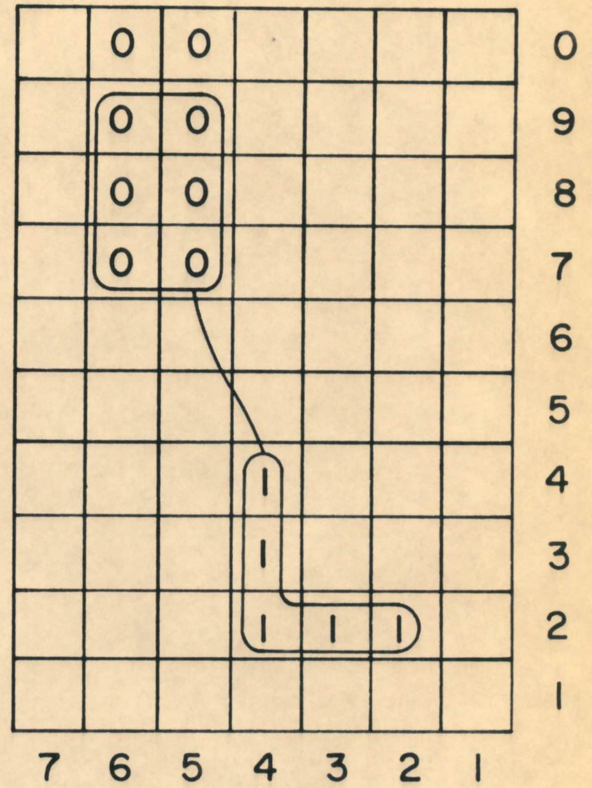




FINAL LOGIC STATEMENT FOR CHARACTER 2

R. J. Evey
Fig. 4a





FINAL 2 LOGIC BY DIFFERENT DESIGNER

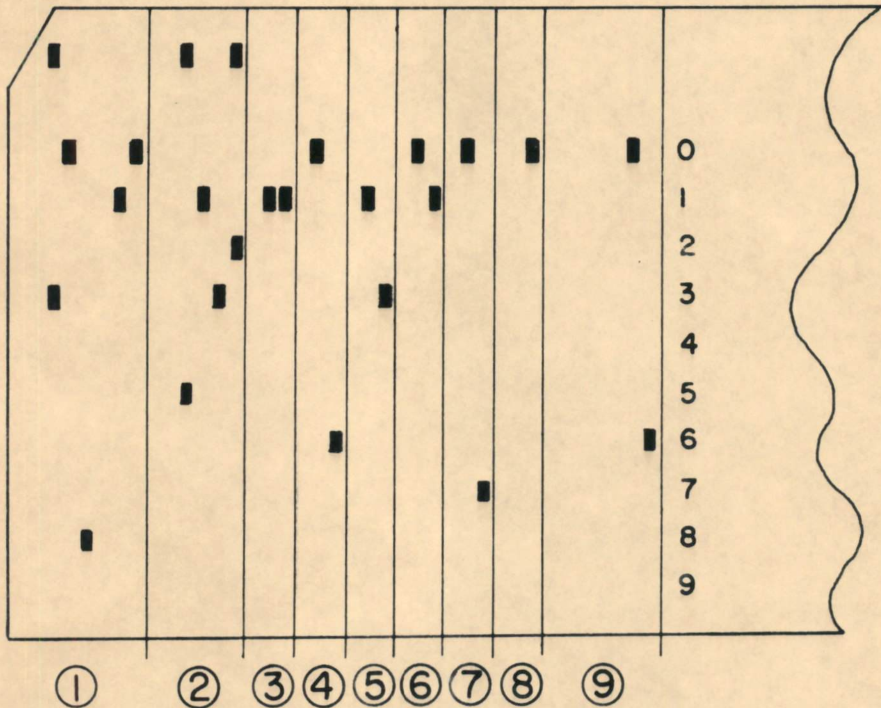
R. J. Evey
Fig. 4b

	IDENTIFICATION	ROW	NO. OF REPEATS	BLACK STARTS	BLACK STOPS	BLACK STARTS	ETC.
CARD COLS.	1,2	3-5	6-8	9-11	12-14	15-17	ETC.
	D2	117	001	005	048		
	D2	116	001	003	050		
	D2	114	002	002	051		
	D2	109	005	001	052		
	D2	107	002	001	051		
	D2	106	001	001	050		
	D2	105	001	001	048		
	D2	104	001	001	017		
	D2	103	001	001	015		
	D2	101	002	001	014		
	D2	070	031	001	013		
	D2	068	002	001	014		
	D2	067	001	001	015		
	D2	066	001	001	017		
	D2	065	001	001	048		
	D2	064	001	001	050		
	D2	062	002	001	051		
	D2	057	005	001	052		
	D2	055	002	002	052		
	D2	054	001	003	052		
	D2	053	001	005	052		
	D2	052	001	036	052		
	D2	051	001	038	052		
	D2	049	002	039	052		
	D2	018	031	040	052		
	D2	016	002	039	052		
	D2	015	001	038	052		
	D2	014	001	036	052		
	D2	013	001	005	052		
	D2	012	001	003	052		
	D2	010	002	002	052		
	D2	005	005	001	052		
	D2	003	002	002	051		
	D2	002	001	003	050		
	D2	001	001	005	048		

END OF DATA FOR GIVEN CHARACTER.

R. J. Evey

Fig. 5

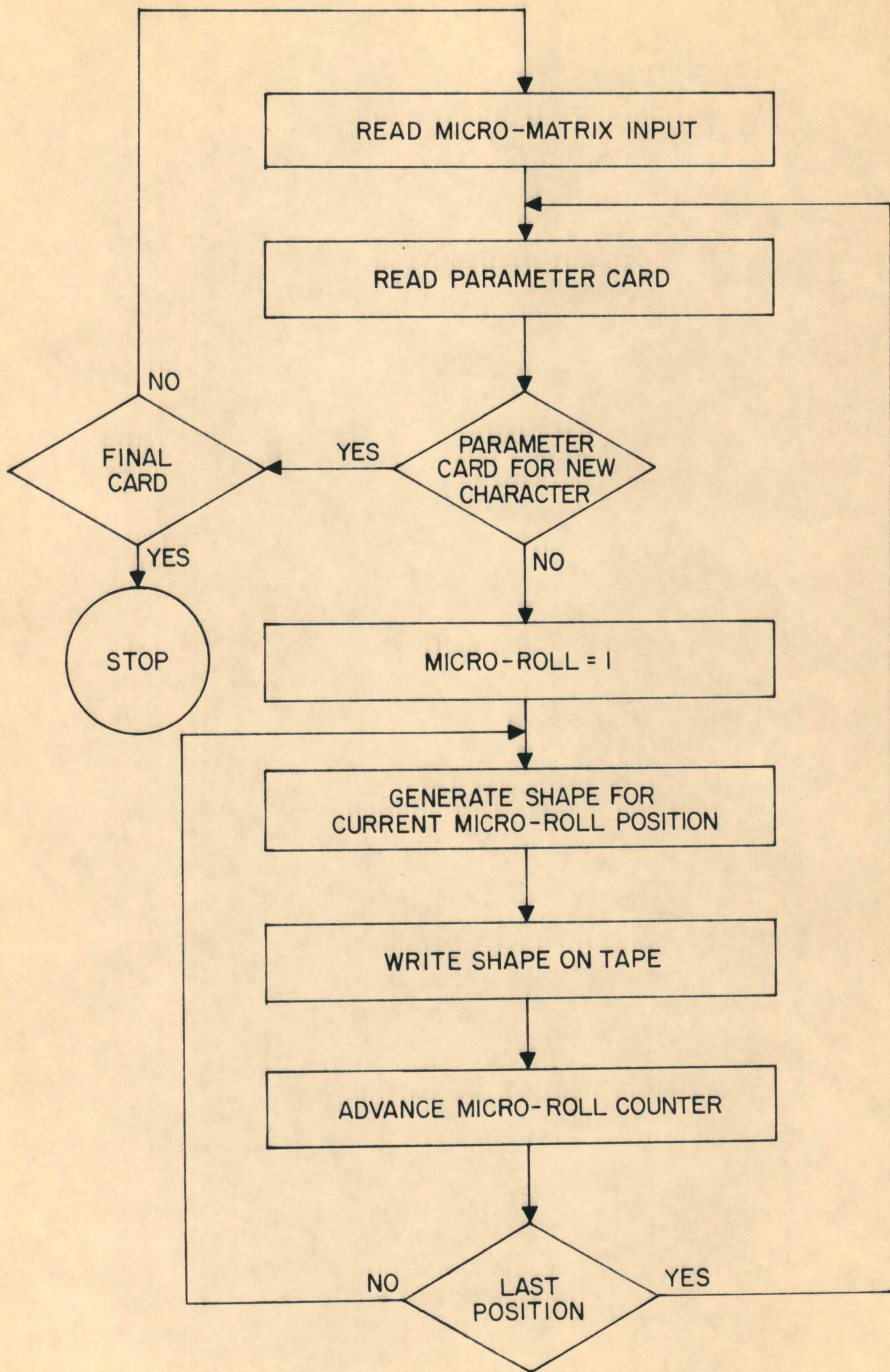


FIELD	COLS.
①	1–6
②	7–12
③	14–15
④	17–18
⑤	20–21
⑥	23–24
⑦	26–27
⑧	29–30
⑨	35–37

NOTE: SEE PAGE 12 FOR FIELD DEFINITIONS

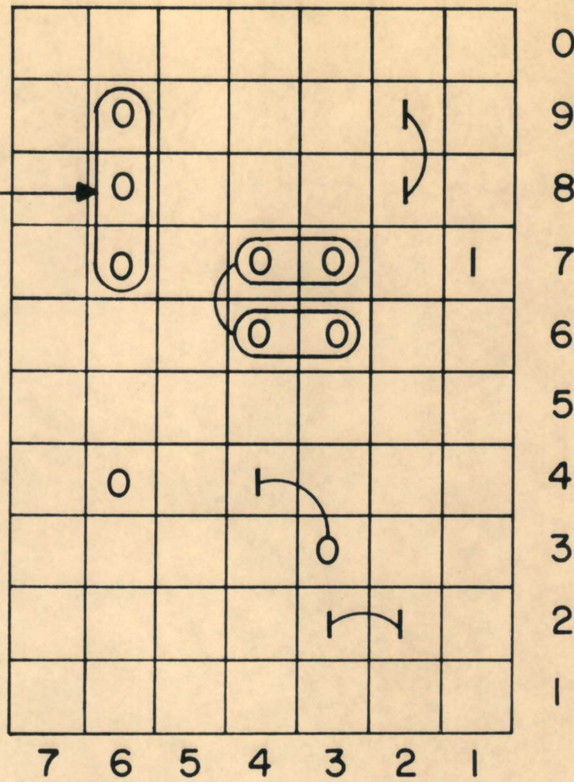
PARAMETER CARD FOR TSP

R. J. Evey
Fig. 6

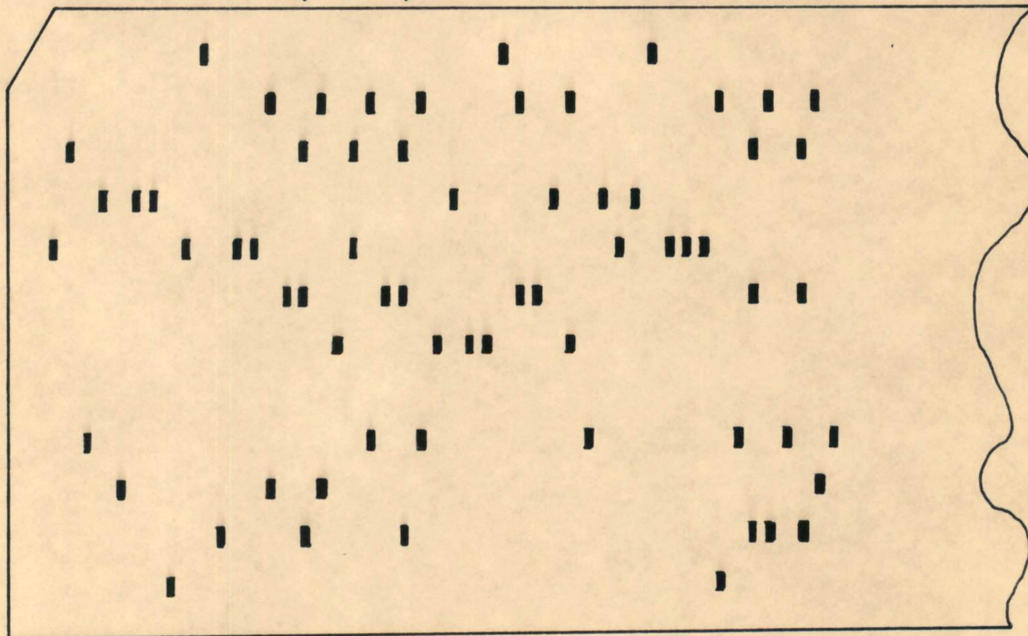


TSP·FLOW CHART

R. J. Evey
Fig. 7

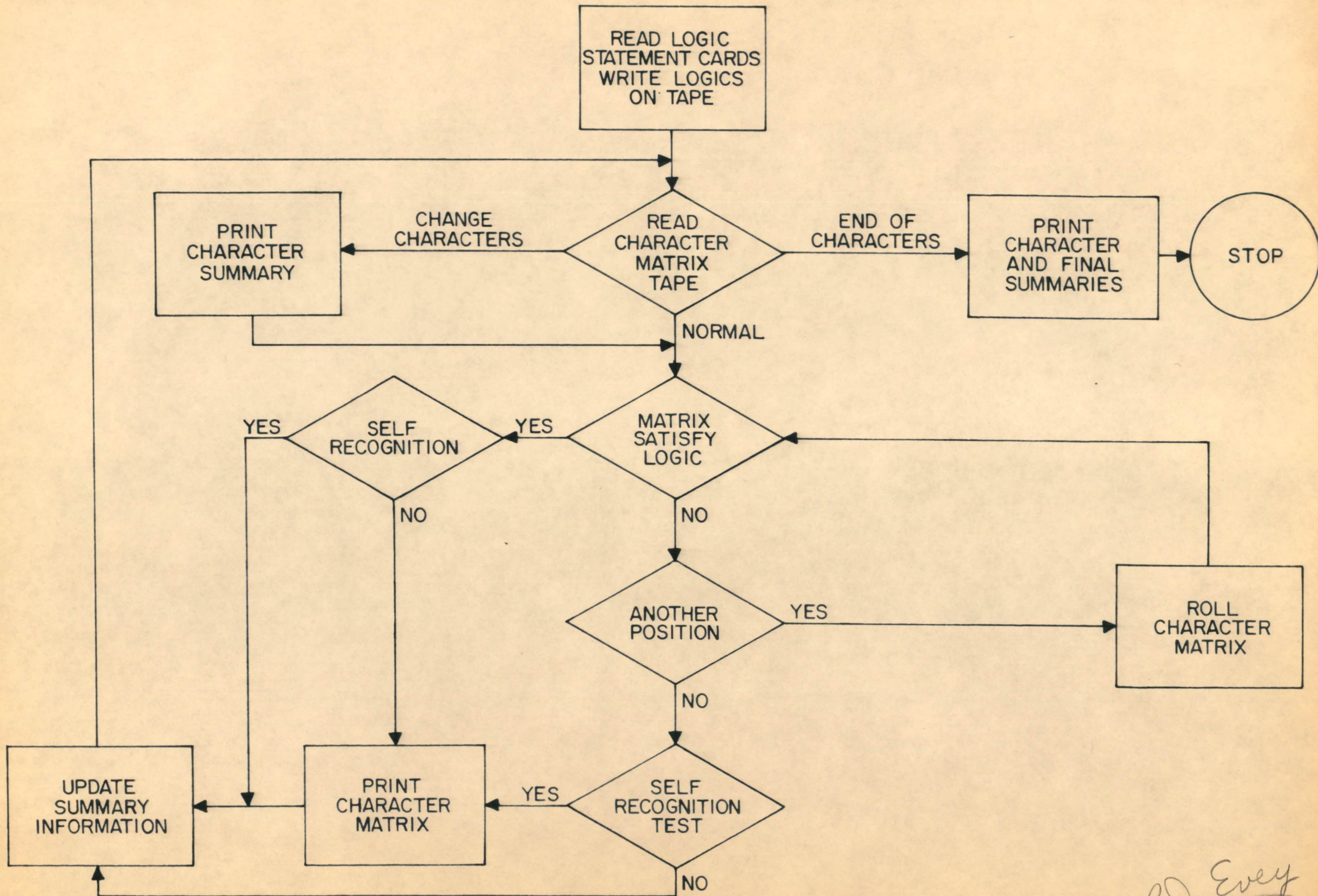


207171192+822$\bar{7}$3,$\bar{7}$4S$\bar{6}$3,$\bar{6}$4144+$\bar{3}$31$\bar{4}$6121+222$\bar{9}$6,$\bar{8}$6,$\bar{7}$6



SIMPLE LOGIC

R. J. Evey
Fig. 8



LOGIC PROCESSING PROGRAM

FREQ. TABLE 2 LOGIC CHAR. 2 490 NCR MFC 486 CNML 2

								A
			10	54	55	55	44	9
			12	86	93	96	99	8
			0	9	10	39	100	7
			10	56	58	64	100	6
			13	92	98	97	97	5
			13	98	8	4	4	4
			13	93	69	60	48	3
			12	91	100	100	100	2
								1
8	7	6	5	4	3	2	1	

POS / FREQ. 1 / 486,

R. J. Evey

Fig. 10

									A
									9
				1		1	1	1	8
2 LOGIC								1	7
CHAR. 2								1	6
CNMLXX					1	1	1		5
1 CNM / 22					1				4
POS 1				1					3
27B515051528					1	1	1		2
* 040									1
	8	7	6	5	4	3	2	1	

R. J. Evey

Fig. 11

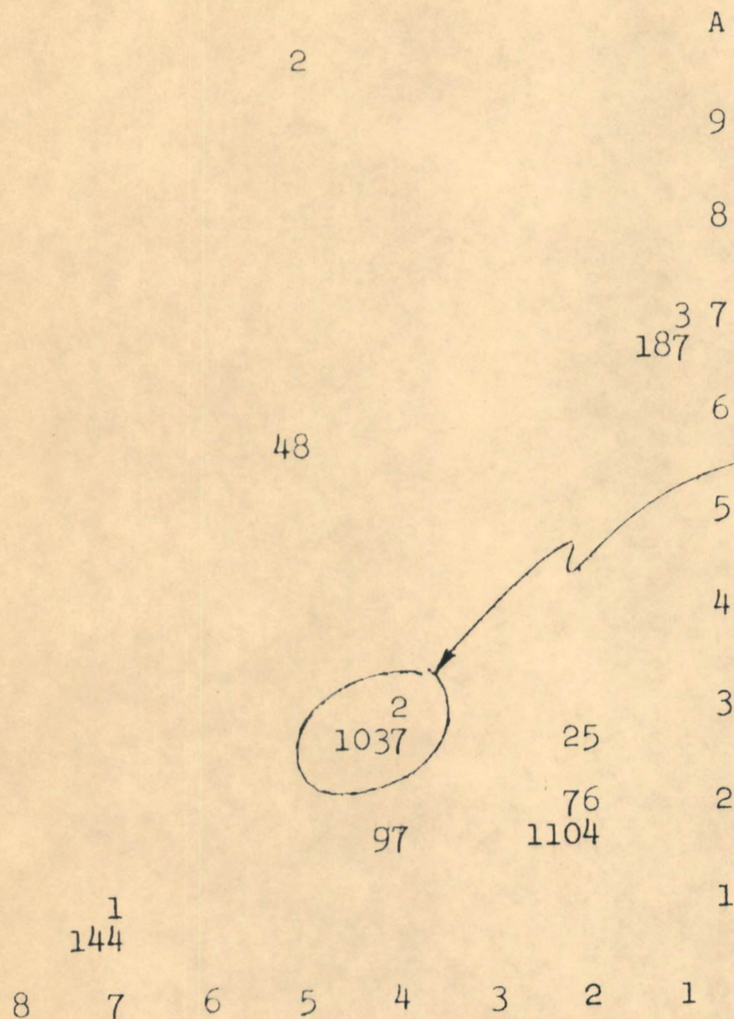
BPFT 1 2 LOGIC CHAR. 2 1000 NCR MFC 3 CNML 2

								A
				100	66	66	66	9
			33	33	66	33	100	8
							100	7
				33		66	100	6
				33	66	66	66	5
				33				4
				66	33	33		3
			33	33	66	33		2
								1
8	7	6	5	4	3	2	1	

R. J. Evey

Fig. 12

CNM MAP CNML 2 992 NCR 2 LOGIC CHAR. 7



... 2 TIMES WAS THE ONE CONDITION KEEPING A 7 PATTERN FROM SATISFYING THIS 2 LOGIC.

...1037 TIMES WAS ONE OF TWO CONDITIONS KEEPING A 7 PATTERN FROM SATISFYING THIS 2 LOGIC.

R. J. Evey

Fig. 13

CNM TABLE 2 LOGIC CARD TOTAL 6482

CHAR. / CNM	0	1	2	3	4	5	6	7	8	9	A	B	C	D
0			708					1						
1		1	5	5		17		48		1				
2		60		209	7	186	5	265	3	118				

CNML 2

R. J. Evey

Fig. 14

File Copy

INTERNATIONAL BUSINESS MACHINES CORPORATION
FEDERAL SYSTEMS DIVISION
SYSTEMS ENGINEERING OFFICE
3104 FARNAM STREET
OMAHA 31, NEBRASKA

November 13, 1959

Dr. Harlan E. Anderson
EJCC Publication Committee
Digital Equipment Corporation
Maynard, Massachusetts

Dear Dr. Anderson:

In accordance with the instructions contained in Dr. Felker's letter of September 21, 1959, I am enclosing four copies of the manuscript entitled, "The Virtual Memory in the STRETCH Computer", by John Cocke and Harwood G. Kolsky for inclusion in the Proceedings of the E. J. C. C. December 1 - 3, 1959.

Also enclosed is a one-hundred word biography of the speaker. For visual aids I plan to use 35 mm slides.

Sincerely,

H. G. Kolsky

H. G. KOLSKY

HGK/jiv
Enclosures

File Copy

Biography of Dr. Harwood G. Kolsky

Dr. Kolsky obtained his bachelor's degree from the University of Kansas and his doctorate in Physics from Harvard University. During World War II he did work in cryptography and communications in the U.S. Signal Corps. He spent 7 years on the staff of the Los Alamos Scientific Laboratory in the field of digital computer applications as an Associate Group Leader in the Theoretical Division. He has at various times held part-time teaching positions at Kansas, Harvard, and New Mexico Universities.

In 1957 he joined IBM as a member of the STRETCH Project Product Planning group. He was recently made Assistant Manager of IBM's SAC Intelligence Project in Omaha, Nebraska.

Paper to be printed in the Proceedings of the Eastern Joint Computer Conference, December 1-3, 1959

The Virtual Memory in the STRETCH Computer

John Cocke
and
Harwood G. Kolsky

International Business Machines Corporation
Poughkeepsie, New York

I. INTRODUCTION

Early in the planning of the Stretch computer it was seen that by using the latest solid state components in sophisticated circuits it would be possible to increase the speed of floating point arithmetic by almost two orders of magnitude over that in existing computers. However, there seemed to be no possibility of developing on the same time-scale economically feasible large memories with more than a factor of ten or perhaps twenty increase in speed. As a result, the proposed system appeared to be in danger of being seriously memory-access limited.

Moreover, as the speed of the floating point operations increases, a larger and larger percentage of the computer's time is spent on "parasitic operations", i. e., operations whose only function is program control and data selection. It was obvious that a radically new machine organization

was necessary in order to capitalize upon the possibilities opened up by the high arithmetic speeds in the presence of relatively slow memories.

At this time, a number of persons were considering the possibility of a "look-ahead" device in which an independent indexing arithmetic unit would prepare the effective addresses of instructions and initiate memory references to a multiplicity of memory boxes. The data thus fetched would be held in high-speed buffer registers until needed by the arithmetic unit. This device would serve two desirable purposes: (1) some of the parasitic operations would be done in parallel and thus not delay the principal calculations, and (2) several memory boxes could be running simultaneously, giving the effect of higher memory speed.

Since our original work on the virtual memory and simulation in 1957-58, a large number of detailed changes have been made in the actual hardware design of Stretch. These necessitated several modifications in the simulation program to estimate their effect on the overall system performance. In this report we are omitting many of these changes for expository reasons, since our purpose is to describe the virtual memory and timing simulation concepts, not to describe the Stretch hardware exactly. The result is that the system described below embodies a more general system than that found in the simulator, which in turn is more general than that found in the actual computer.

II. GENERAL DESCRIPTION OF THE SYSTEM

The major logically-independent blocks of the Stretch computer are shown in Figure 1. Each of the units pictured may be considered as operating asynchronously. That is, each does its tasks as fast as possible independently of the others. In theory, each box could have its own clocking circuits and still operate properly. In practice, for economy's sake they are all timed by the same master oscillator, but this does not destroy their logical independence.

simultaneously the control unit assigns priorities in the following order: (1) High-speed Exchange, (2) Basic Exchange, (3) Virtual Memory, and (4) Indexing Arithmetic Unit.

The Indexing Arithmetic Unit fetches instructions, performs all necessary indexing operations and sends the instructions to be executed to the Virtual Memory.

The Virtual Memory fetches and receives the data required by the instruction and holds this data until the arithmetic unit is ready for it. The virtual memory also performs all store operations. It holds the data generated by the arithmetic unit or indexing arithmetic unit until the memory to which the data must be sent is available. Thus the virtual memory acts not only as a "look-ahead" for instructions to be fed to the arithmetic unit, but also acts as a "look-behind" storage buffer.

The actual design of such a "look-ahead" device posed a number of logical problems, particularly in connection with conditional branches.

However, a machine organization of this complexity requires a detailed timing analysis in order to determine the value of adding hardware in the form of the virtual memory . This is especially true since the sole function of the virtual memory is to increase machine speed, by increasing the efficiency of other devices. It was also felt that the timing analysis could not be made on the basis of a few trivial examples

(e.g. matrix multiply). Machine performance obtained in this fashion can be extremely deceptive. Since a detailed timing analysis of a computer of this complexity is extremely tedious to carry out by hand, it became clear that if the job were to be done, it would be necessary to simulate the proposed machine on another computer. This prompted us to write the simulation program to be described later.

With the above general organization in mind, let us discuss some of the logical problems posed by such a system. The first problem is a result of the very concept which enables us to obtain such great benefits from the stored program computer -- the ability to treat instructions as data. In a system such as we have proposed there is a large amount of simultaneous operation. For example, the indexing arithmetic unit may be busy preparing an instruction before previous instructions have been completed or even started by the arithmetic unit. One of these previous instructions may modify the instruction which is presently being indexed. The virtual memory must recognize this situation and allow the intervening instructions to be completed before doing the modified instruction.

A similar problem exists with respect to ordinary data. In order to operate several memories simultaneously, it is necessary to start obtaining data from these memories before the preceding operations have been completed. Yet, one of these operations may be a store into one of the

data locations. The virtual memory must make provisions to insure that each instruction obtains the most up-to-date data as implied by the order of the program.

One of the novel features of the Stretch computer is its elaborate interrupt system. Under this system, whenever some unexpected occurrence arises, the program will be interrupted and control will pass to a special routine which is designed to take care of the case in question, then return control to the original program. In this situation the virtual memory must have provisions to retain enough information so that when an interrupt occurs we can resume the computation exactly where we left off. It must be able to recognize which of the changes that have been made in advance are not desired and should be obliterated, and which are exact solutions that must be restored.

Another special case arises when a conditional branch on arithmetic results occurs. Here we will not know which of the two branches we should have taken until the preceding instruction is executed. In the case where the wrong path has been selected, the virtual memory must be prepared to drop the intermediate results which have been computed and pick up the correct branch in a way very similar to that of an interrupt.

Summing up all these logical problems, we may state that the fundamental rule for the virtual memory is that it must make the asynchronous and non-sequential computer give results identical to those which would be obtained by performing the program one instruction at a time in the order in which they are written.

III. DETAILED DESCRIPTION OF VIRTUAL MEMORY OPERATION

A. General Conditions to be Considered

The conditions which occur in the following situations must be considered in some detail:

1. The fetching of instructions by the Indexing Arithmetic Unit (IAU).
2. The indexing of instructions and modification of Index registers.
3. The loading of the virtual memory and the setting of its conditions by the IAU.
4. The action of the virtual memory in fetching data.
5. The action of the virtual memory in storing data.
6. The communication between the virtual memory and the main arithmetic unit.
7. Special situations such as conditional branching on arithmetic results, etc.

B. Definitions

Some of the terms we will use are defined as follows:

1. Operations

Operations are considered to be of three types:

(1) Bring or Fetch Type - All instructions requiring

data to be transmitted from external memory to the virtual memory.

(2) Store Type - Instructions requiring the transmission of data from the virtual memory to external memory or index memory.

(Note: We consider all indexing instructions to be of the store type, although the store may be to either external memory or index memory.)

(3) Immediate Type - All operations not requiring data transmission.

2. <u>Virtual Memory Quantities</u>

(1) Virtual Memory - A number of virtual memory (or look-ahead) levels (numbered 0 to N-1).

(2) Level of Virtual Memory - A collection of registers and control bits. The contents of the j th level are shown in Figure 2.

(3) Instruction Address Register (I_j) - Contains the address of the instruction currently in the j th level.

(4) Operation Code Register (OP_j) - Contains the operation to be performed by the arithmetic unit.

(5) Store Bit (S_j) - a one-bit trigger which indicates the level, contains a store type instruction.

(6) Bring Bit (B_j) - A one-bit trigger which indicates the level, contains a fetch type instruction for which the data access has not been started.

(7) Forwarding Bit (F_j) - A one-bit trigger which indicates that the j th level must transmit data to another level.

(8) Forwarding Address (FA_j) - A register which contains the number of the level to which the data must be sent if F_j is set.

(9) O. K. Bit (OK_j) - A trigger which when set indicates that the correct data for the instruction to be executed is present in the j th data field.

(10) Data Field (D_j) - A register which contains the operand data for the instruction.

(11) Data Address (DA_j) - The operand data address (already indexed by the IAU) for D_j.

(12) Compare Bit (C_j) - A trigger which if not set indicates the address in DA_j should not be included in any address comparisons being made.

3. Counters

The virtual memory is controlled by a set of counters which count mod(N), where N is the number of virtual memory levels.

(1) Counter one (C_1) - Indicates the level into which the next instruction may be placed.

(2) Counter two (C_2) - Indicates the level from which the next bring type instruction may be initiated.

(3) Counter three (C_3) - Indicates the level from which the next store type instruction may be initiated.

(4) Counter four (C_4) - Indicates the level from which the arithmetic unit will get its next operation and data.

4. Interlocks

The above counters must be interlocked in the following manner to assure proper sequential operation of the computer (see Figure 3):

(1) Interlock one (I_1): $C_1 = C_3 + N$ Prevents the IAU from placing the next operation into the level

decisions will be made in such a manner as to make sure this is the case.

(2) Addresses can be compared by the IAU with every DA_j address simultaneously. DA_j is not used for any level which does not have its C_j bit set. If a comparison exists between a new DA_j being placed in the virtual memory and an old DA_k, the compare bit C_k is turned off and the address of level j is placed in FA_k. This insures a unique meaning for the comparison. If this were not done, another instruction address DA_e might compare against <u>two</u> levels and thus cause an ambiguity.

2. Instruction Fetch Logic

Figure 4 is a flow diagram of the IAU Instruction Fetch Procedure. The logic is as follows: If the IAU is ready to fetch another instruction, it compares the instruction address with all the DA_j's of virtual memory. If there is no comparison, the instruction fetch is initiated. If there is a comparison, the IAU must take its instruction from the virtual memory provided the OK bit is set; otherwise, it must wait until the OK bit is set.

Note: This procedure prevents the logical difficulty mentioned earlier which would occur if the virtual memory contained a store order into the instruction presently being fetched.

For Example:

a	STORE Address a+2
a+1	LOAD M, i
a+2	ADD N, i
a+3	----

The store to a+2 must be done in sequence or the old value N would be used for the address instead of the quantity being set by a.

3. Indexing Logic

Figure 5 shows the flow for instruction indexing. After determining that an instruction is ready to be indexed, the IAU tests whether or not the index value is available. If it is, the indexing operation is started; if not, the memory reference is started and the IAU waits until the data returns before proceeding. If the index-fetch has not been started, the IAU compares the index address against all the data addresses in virtual memory. If none compare, the index value is fetched normally. If one does

compare, the index fetch is held up until the OK bit is set for the data. This value from the virtual memory is then used for indexing the instruction.

4. Logic of Putting Instructions in the Virtual Memory

(1) Figures 6, 6A, 6B, 6C represent the logical flow for putting instructions into the virtual memory. If the indexing arithmetic unit has an instruction prepared for the virtual memory, it may transmit the instruction into the virtual memory if interlocks one and five do not forbid it. These interlocks prohibit a new instruction from destroying an old one which has not been executed as yet, whether an arithmetic operation (I_5) or an unexecuted store (I_1). The handling of the instructions varies depending on whether they are of the bring type, store type, or immediate type.

(2) The bring type, as described in Figure 6A, proceeds as follows: If the effective data address of the instruction compares with the DA address in some level, the instruction, its op code, and effective data address are loaded into the level

will be required. The level's bring bit and forwarding bit are set to zero; its compare bit is set to one. If on the other hand the addresses do compare, the same procedure is followed; but in addition, the compare bit in the level compared-with is set to zero so that future comparisons will not use it.

The OK bit has not yet been set. It is set to one if the operation is an index store and set to zero if it is an ordinary store. For the ordinary store it is clear that the OK bit should be zero since the data must come from the arithmetic unit after the preceding instruction is executed.

As was mentioned in the definition previously we treat all indexing instructions as store type and place the new value of the indexed quantity into the virtual memory. This is done because the indexing arithmetic unit is going ahead of the normal order of instruction execution and an interruption may occur before this indexing instruction should have been done. In this case, the old value of the index is still in the index register. On the other

hand the indexing arithmetic unit compares with the virtual memory and extracts the most recent value of the index for indexing succeeding instructions. The OK bit is set to one since the appropriate data is in the above level. Both the new and old index values must be carried along to give logically correct conditions in the case of an interrupt.

A situation very similar to interrupt occurs in branches on arithmetic results where the indexing arithmetic unit "guesses" which branch will be taken and proceeds with fetching and processing the instructions on this branch, subject to being wiped out if the guess proves to be wrong. (See the discussion on "Wrong way Branches" below.)

(4) Immediate type instructions are the simplest type because they essentially carry their data with them. Figure 6C shows the logic in this case. The instruction is placed in the virtual memory level marked by C_1. The address field of the instruction is placed in the data field of C_1. The OK bit is set to one indicating the data is present. The bring and

store bits are both set to zero. The compare bit is set to zero since the DA address field has no meaning for immediate type ops. (The data address of the last instruction which occupied this level still remains in DA, so it has no relation to the present D field.)

5. Logic of Data Fetching

See Figure 7: When an instruction of the bring type has been placed in the virtual memory, the data required by the instruction in general will not be present (unless a comparison exists as was described above) and thus the data must be obtained from core storage. The fetch cannot be started if interlock I_3 holds, which means all the fetches corresponding to the instructions presently in the virtual memory have been started. If a fetch is possible, the bring bit at level C_2 indicates whether or not a fetch is necessary. If necessary the fetch may be started if the memory bus and memory unit corresponding to the data address are not already being used. When the fetch is started, the bring bit for level C_2 is set to zero. The counter C_2 is then stepped forward to the next level.

6. Logic of Data Storing

Figure 8 shows the Data Store Logic, which is very similar to that for data fetching just described. The only significant difference is that the O. K. bit must be set before the operation can be started.

7. Logic for Placing Data into the Virtual Memory

In Figure 9, we see the logical conditions which must be satisfied by the data returning from memory addressed to the virtual memory. The return address which was supplied when the fetch was started selects the level into which the data will be placed. The O. K. bit is then set to one, indicating that the proper data is in the level. The operation is complete at this point unless the forwarding bit is set. In this case, the data must be forwarded to the level designated by the forwarding address. This procedure continues from level to level as long as the data continues to arrive into a level whose forwarding bit is set. This procedure automatically supplies all operands present having identical data addresses with the proper data, without additional memory references.

8. <u>Logic of Removing Instructions from the Virtual Memory</u>

Observing Figure 10, we notice that as the arithmetic unit completes an instruction it checks to see if the next instruction in the virtual memory is ready to be executed (indicated by interlock I_4). Note: The operation may be an unconditional branch, a conditional branch, or an index type store, as well as a normal bring or store type instruction involving the accumulator. Figure 10 shows only the cases which involve the universal accumulator. The index and unconditional branches and the index store operations are merely ignored at this point. They are carried along only to provide the data for recovery in the event an interrupt occurs. The execution of the conditional branches on arithmetic results are described in the next section.

If the next instruction marked by counter C_4 is ready, it is fed into the arithmetic unit. If it is a store type, the data is gated from the accumulator into the data field of level C_4, and the OK bit is set to one. If the forwarding bit of the level is set, a forwarding procedure in this case is <u>essential</u> for the proper logical operation of the computer, whereas in the bring case it is a time-saver only.

If the instruction is not a store type, the arithmetic unit must hold up until the O. K. bit for the level is set, When the O. K. bit is set, the instruction is gated into the arithmetic unit and executed.

9. Logic of Interrupt Procedure

If for any cause an interrupt (or trap) from a special condition occurs, the instruction which is being executed in the arithmetic unit is completed. However, the next instruction is not executed in spite of the fact all the data preparation for it may have been completed. The address in the IA (instruction address)field will serve as the value to reset the instruction counter if it is desired.

The Virtual Memory is initialized, i. e. , set to the starting conditions of an interrupt, with the exception that all store orders which have already received data from the accumulators must be executed first. Note: If the interrupt is of such a nature that the normal flow of instructions is not resumed, the procedure of storing the modified values of the index registers in the Virtual Memory gives logically correct results, i. e. , the same as if the interrupt had occurred before the indexing took place.

One of the fundamental concepts in the Stretch design is that of asynchronous operation of the components. This means that there are a large number of logical steps being executed at any one time in the computer, each of them proceeding at its own rate. To simulate this flow of many parallel continuous operations, we have broken the continuous time variable into finite time steps. The basic time step is taken as 0.1 microsecond in the simulator.

By taking 0.1 microsecond as our quantum of time, we are automatically setting the scale of the smallest circuit entities which we will consider as being those which accomplish complete functions in 0.1 microsecond or few multiples thereof. Thus, by using this philosophy, and considering many of the components of the computer as "black boxes", we greatly simplify the details which must be considered without introducing serious timing inaccuracies.

Our experience has indicated that more information was gained by making a large number of fast parameter studies using different configurations and programs than could have been obtained by a very slow, detailed simulation of a few runs with more precision per run. Even so, our time scale is too fine to make serious input-output application studies. These would require a simpler simulator having at least a factor of 10 coarser basic time interval.

It is interesting to note that since the simulator simulates timing only, not the arithmetic or indexing functions, the sequence of instructions to be executed must be furnished as a "string" with all loops unwound. However, to make the computer behave as it actually would, the loops must be furnished with "wrong way" paths given for the cases where the computer would take such paths. Also one must furnish more than enough information along such paths since it is difficult to predict in advance how far the computer will get down the wrong path before it is called back.

Parameters are changed from one run to another by use of control cards. The control cards are set up in such a way that any number of parameters may be changed between runs. Results are given either as detailed timing charts or as summary listings for each problem. The usual procedure has been to print only summary results while making a series of parameter studies. The detailed timing charts as printed on the 704 for most problems would be about 50 feet long for each run. Since over 1000 cases have been run, it is clear that only a few cases could be printed in full detail. These are particularly useful in seeking the causes of conflicts which slow the computer.

C. Results of Parameter Studies

When the simulator program was completed, we undertook a series of studies in which the main parameters describing the Stretch system were varied one or two at a time in order to get

a measure for the importance of different effects. After this we began to specialize the studies towards answering specific questions in the STRETCH design.

The simplified flow diagram in Figure 11, indicates the order in which the subroutines for the various logical units are executed at each time step. Using the types of techniques just described above, the logical subroutines simulate the action of the components of the computer such as the Virtual Memory, arithmetic Unit, etc.

V. SOME RESULTS OF THE SIMULATION STUDIES

Figure 12 shows examples of the type of output listings given by the simulator. Figure 12 is a piece of a long timing chart with each line of printing representing 0.1 microsecond of time. The columns represent the various components of the computer. On the left and right are timing counts subdividing each microsecond. On the far right are conflict indicators ("C" on the charts) and waiting indicators, "W", which indicate when interlocks prevent operations from proceeding.

The 2nd column, II, gives the number of the instruction being indexed. The 4th column, AU, gives the number of the instruction using the arithmetic unit. The next four columns represent the instructions using the memory buses. The columns labeled X-, F-, and M- represent the index, fast, and main memories. A string of "X's" in the columns represents the cycle time of the memory. The number indicates the instruction using the memory and the number of times which it is repeated gives the readout time of the memory. The columns L- indicate which instruction is located in the virtual memory levels. The other columns are for details in analysis and need not be considered here.

Five of the test problems used most frequently are described below. Other test problems were used for specific studies, but since the results were similar for all problems of a given type, we gradually discontinued using them. The following were originally selected as being typical of different classes of problems.

1. Mesh Problem - Part of an hydrodynamics problem from Los Alamos. It contains a more or less "average" mixture of instructions for scientific problems: 85% floating point instructions, 14% index modification instructions, and 1% VFL. It is usually arithmetic unit limited.

2. Monte Carlo Branching Problem - Part of an actual Monte Carlo neutron diffusion code. It represents a chain of logical decisions with very little arithmetic in between. It contains 47% floating point, 15% index modification instructions, and 36% branches of the indicator and unconditional types. It is largely instruction-access limited.

3. Reactor Problem - The inner loop of a neutron diffusion problem. It consists of 90% floating point arithmetic (39% of which are multiplys) and 10% index modification instructions. It is almost entirely arithmetic unit limited.

4. Computer Test Problem - The evaluation of a polynominal using computed indices. It has 71% floating point, 10% index modification, 6% VFL and 13% indicator branches. It is usually arithmetic unit limited, but not for all configurations.

5. Simultaneous Equations - The inner loop of a matrix inversion routine 67% floating point and 33% index modification.

Arithmetic and logic are about equally important. It is limited both by arithmetic and instruction-access speeds.

A. Speed vs Number of Levels of Virtual Memory

Figure 13 shows the effect on computer performance of varying the number of levels of virtual memory. Curves for the Monte Carlo and Mesh Calculations with two sets of arithmetic and indexing arithmetic speeds are shown. The AU times given are averages for all operations.

A number of interesting results are apparent from these curves:

(1) There is a tremendous gain to be had in going to the virtual memory organization. The point for "0 levels" means that the arithmetic unit is tied directly to the instruction preparation unit, although simple Indexing-Execution overlap is still possible.

(2) The gain in performance goes up very rapidly for the first two levels then rises more slowly for the rest of the range.

(3) A large number of levels does the Monte Carlo problem less good than the Mesh problem because constant branching in the former spoils the flow of instructions. Notice that the curve for the Monte Carlo problem actually <u>decreases</u> slightly beyond six levels. This phenomenom is a result of memory conflicts caused by extraneous memory references started by the computer running ahead on the wrong-way paths of branches.

(4) The computer performance on a given problem is clearly less for slower arithmetic speeds. However, it is important to note that the sensitivity of the performance is also less for slower arithmetic speeds. The virtual memory improves the performance in either case, but it is not a substitute for a fast arithmetic unit.

B. Speed vs Number of Main Memory Units

Figure 14 shows how internal computer performance varies with the total number of memory units for a particular problem. The entire calculation is assumed to be contained in memory for all cases. The speed gain from overlapping memories is quite apparent from the graphs.

The speed differential between having and not having instructions separated from data arises from delays in instruction fetches caused by the memory units being busy with data. The size of this effect varies from problem to problem, being less pronounced for problems which are arithmetic limited and more for logical problems.

The "X's" on the graph show the effect of replacing the 0.6 usec instruction memories by a pair of 2.0 usec memories. The resulting performance change is small for the Mesh problem, which is arithmetic limited, but large for the instruction-fetch limited Monte Carlo problem.

C. Speed vs Arithmetic Unit and Indexing Arithmetic Unit Times

Although everyone realizes the importance of arithmetic speed on overall computer performance, it was not until the simulator results became available that the true importance of the indexing arithmetic speeds was recognized. Figures 15 and 16 show a two parameter family of curves giving the computer speed as a function of the AU and IAU times.

Figure 16, in which the arithmetic time is the abscissa, shows an interesting "saturation" effect where the computer performance is independent of AU speed below some critical value. Thus it makes no sense to strain AU speeds if the IAU is not improved to match. The curves in Figure 15 show the same effect, i. e., the IAU speed serves as a "ceiling" on performance beyond which the AU speed cannot pass.

D. Arithmetic Unit Efficiency

One fallacy which is frequently quoted is that the goal of improved computer organization is to increase the arithmetic unit efficiency. Actually there are two reasons why this is not the goal in itself. The first is that arithmetic efficiency depends strongly on the mixture of arithmetic and logic in a given problem so that a general purpose computer cannot hope to give equally high percentage utility to all. The second reason is that the simplest way to increase the arithmetic unit efficiency in any asynchronous case is to slow down the arithmetic unit!

The real goal of improved organization is maximum overall computer performance for minimum cost. One will tend to increase the arithmetic unit speed as long as its percent efficiency is reasonable for a variety of problems. One will stop this process when the overall performance gain no longer matches the increase in hardware and complexity. Thus the arithmetic unit efficiency is a by-product of this design process, not the prime variable.

E. Speed vs Concurrent Input-Output Activity

Because of the relative time scales of I/O activity and the CPU processing speeds, the simulator cannot take in account the availability or non-availability of data from I/O on the program being run. However, we can observe the effect on the computation of the I/O devices operating at different rates simultaneously with computing.

Using the Stretch control word philospphy, it is possible to have a number of input-output units operating at the same time the Central Processing Unit is running. The Basic Exchange can reach a peak rate of 1 word every 10 microseconds. The high speed disk normally operates at 1 word every 4 microseconds. Since the mechanical devices take priority over the CPU in addressing memory, the computation slows down because of memory-busy conflicts.

Figure 17 shows an example of how internal computing speed is slowed as the I/O word rates are varied continuously. At the theoretical "choke off" the I/O devices take all the memory cycles available and stop the calculation. Notice that this condition can never arise for any I/O rates presently attainable.

A Stretch system with only 1 or 2 memory units has less performance than a larger one for three reasons: (1) The top speed of the system is reduced by the loss of memory overlap, (2) it has a larger I/O penalty when I/O is run concurrently with the computation, and (3) the smaller amount of data which can be held in the memory at one time increases the amount of I/O activity needed to do the job. Note, however, that increasing the memory size on a computer of conventional organization only improves the third area.

F. A Study of Branching on Arithmetic Results in Stretch

One penalty of the non-sequential preparation and execution of instructions used in STRETCH is that if there is a branch in the problem code it spoils the smooth flow of instructions to the indexing arithmetic unit. Any branch in a program will cause some delay, but the most serious ones are the branches on arithmetic results which cannot be detected by the indexing arithmetic unit in advance.

There are two fundamental ways in which branches on arithmetic unit results can be handled by the computer.

(1) The computer can stop the flow of instructions until the arithmetic unit has completed the preceding operation so that the result is known, then fetch the next correct instruction. This places a delay on every AU result branch whether taken or not.

(2) The computer can "guess" which way the branch is going to go before it is taken and proceed with fetching and preparing the instructions along one path with the understanding that if the guess was wrong, these instructions must be discarded and the correct path taken instead.

A detailed series of simulator runs were made to study this situation and to decide which way Stretch should be designed. Some of the general observations were:

(1) The performance variation in a problem with considerable arithmetic data branching can vary by approximately ± 15% depending on the way in which the branches are handled.

(2) Holding-up on every branch seems to be less desirable than any of the guessing procedures. Some time is lost whenever a branch is executed rather than proceeding to the next instruction. Unless there is an unusual situation in which there is a very large probability that the branch will always be taken, the least time will be lost if one assumes that the branch is <u>not</u> taken.

(3) The theoretically highest performance would be obtained if each branch had an extra "guess bit" which would permit the programmer to specify which way he estimates

each branch will most likely go. However this would place a considerable extra burden on the programmer for the gains promised. (It also uses up many valuable OP codes.)

(4) It is realized that there is a "feedback" in such decisions because the way in which the machine guesses the branches will influence future programmers to write their codes to take advantage of the speed gain. The result is that the statistics of the future will be biased in favor of the system chosen for the machine, and thus "prove" that it was the right decision!

CAPTIONS for John Cocke and Harwood G. Kolsky, "The Virtual Memory in the STRETCH Computer"

Fig. 1. Schematic of stretch computer

Fig. 2. Virtual memory - contents of one level

Fig. 3. Virtual memory interlocks

Fig. 4. Instruction fetch procedure

Fig. 5. Indexing procedure

Fig. 6. Procedure for placing instructions into the virtual memory

Fig. 6a. Logical conditions for bring type operations

Fig. 6b. Logical conditions for store type operations

Fig. 6c. Logical conditions for immediate type operations

Fig. 7. Data fetch procedure

Fig. 8. Data store procedure

Fig. 9. Procedure for placing data into virtual memory

Fig. 10. Procedure for removing instructions from virtual memory

Fig. 11. SIM - 2 simplified flow diagram

Fig. 12. Listing of simulator print-out

Fig. 13. Computer speed vs. no. of levels of Look-ahead registers: 4 main mems. 2.0 μs; 2 fast mems. 0.6 μs; for two sets of arith. speeds

Fig. 14. Computer speed vs. number of main memory boxes: 4 levels LA; 0.6 μs I AU time; 0.64 μs AU time

Fig. 15. Computer speed vs. indexing arith. times for various arithmetic unit times: 4 main mems. 2.0 μs; 2 fast mems 0.6 μs; 4 levels of look-ahead

Fig. 16. Computer speed vs. arithmetic times for various indexing arithmetic unit times: 4 main mems. 2.0 μs; 2 fast mems. 0.6 μs; 4 levels of look-ahead

Fig. 17. Internal computing speed. Percentage reduction in speed caused by input-output devices referencing memory at different rates while the calculation is proceeding.

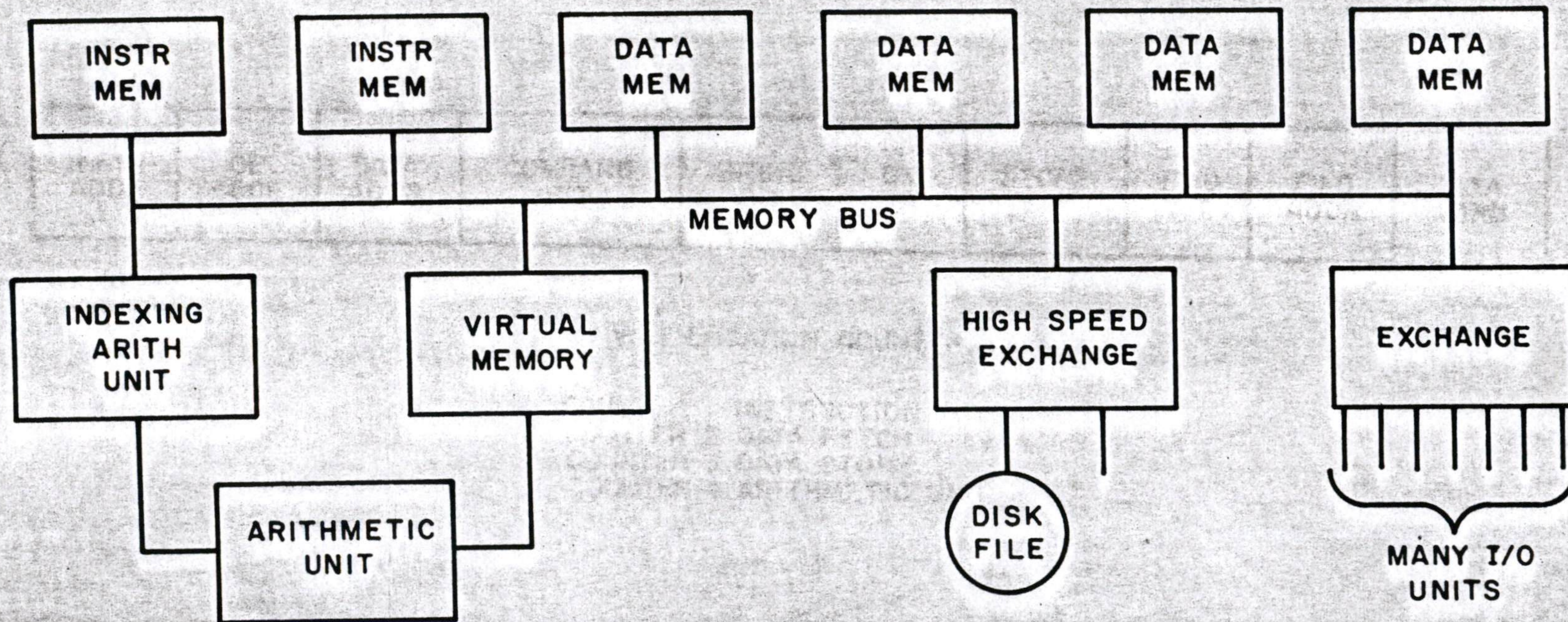
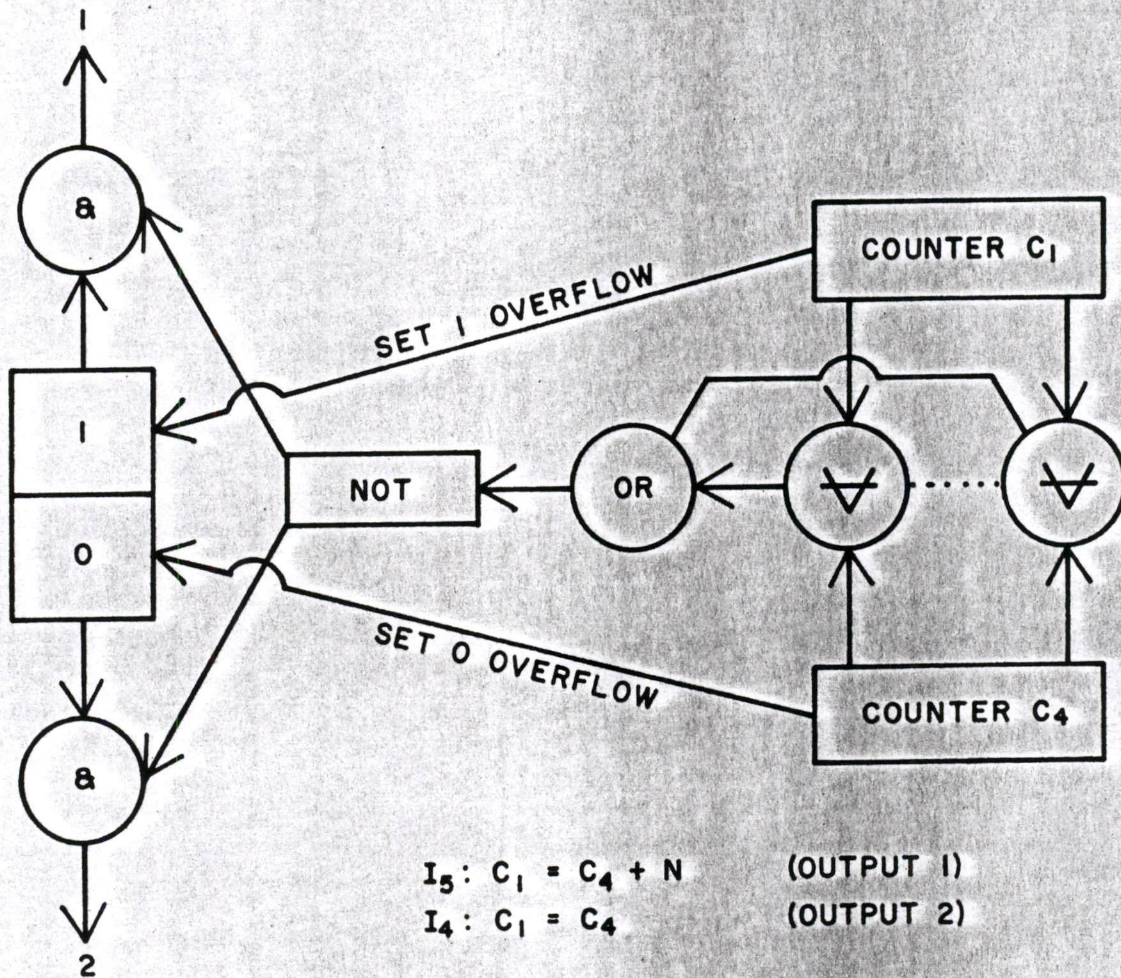


Fig 1

Fig. 1



INTERLOCKS I_4 AND I_5 ARE AS SHOWN; THE OTHER INTERLOCKS ARE DONE IN A SIMILAR MANNER.

Fig. 3

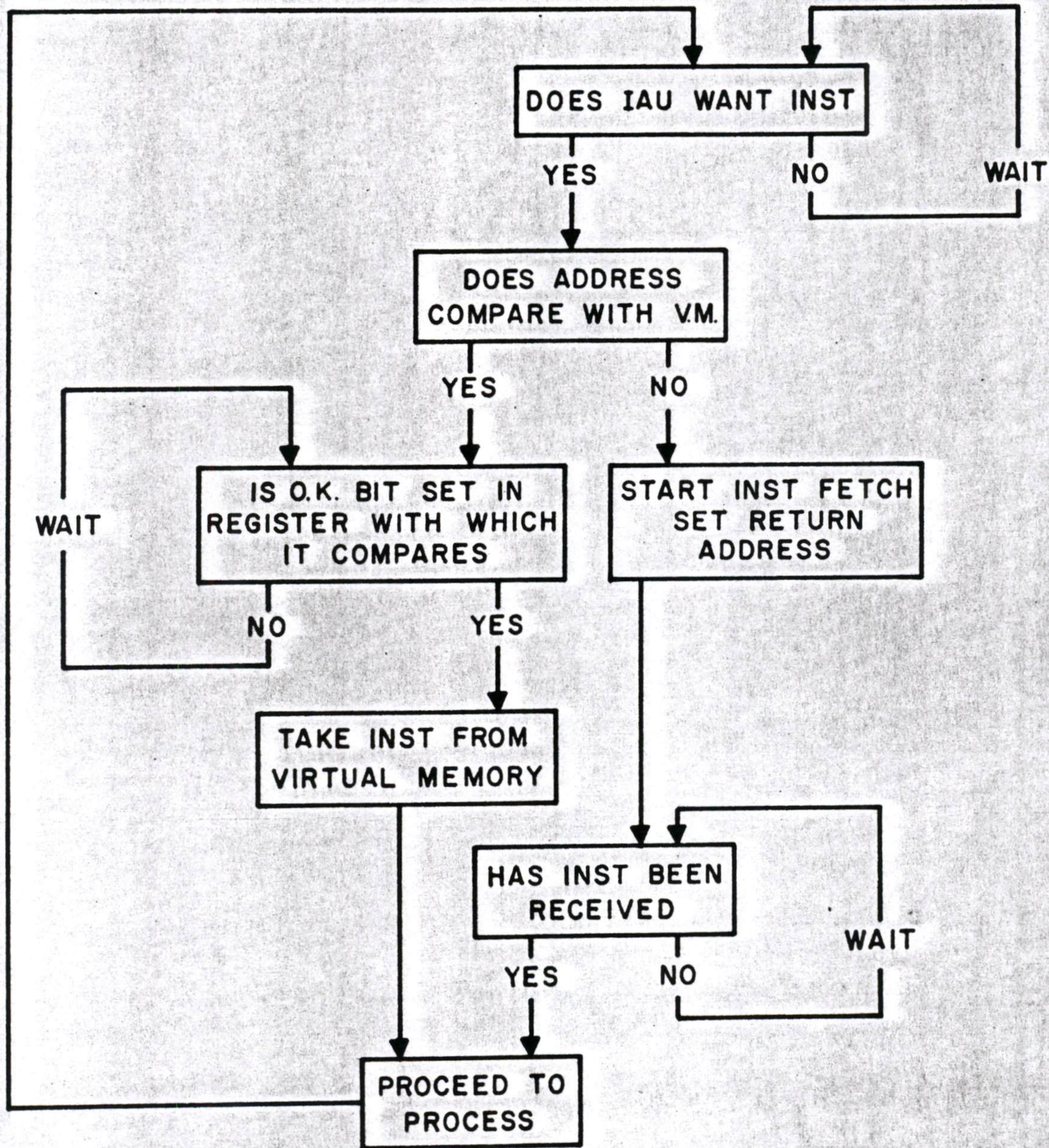


Fig. 4

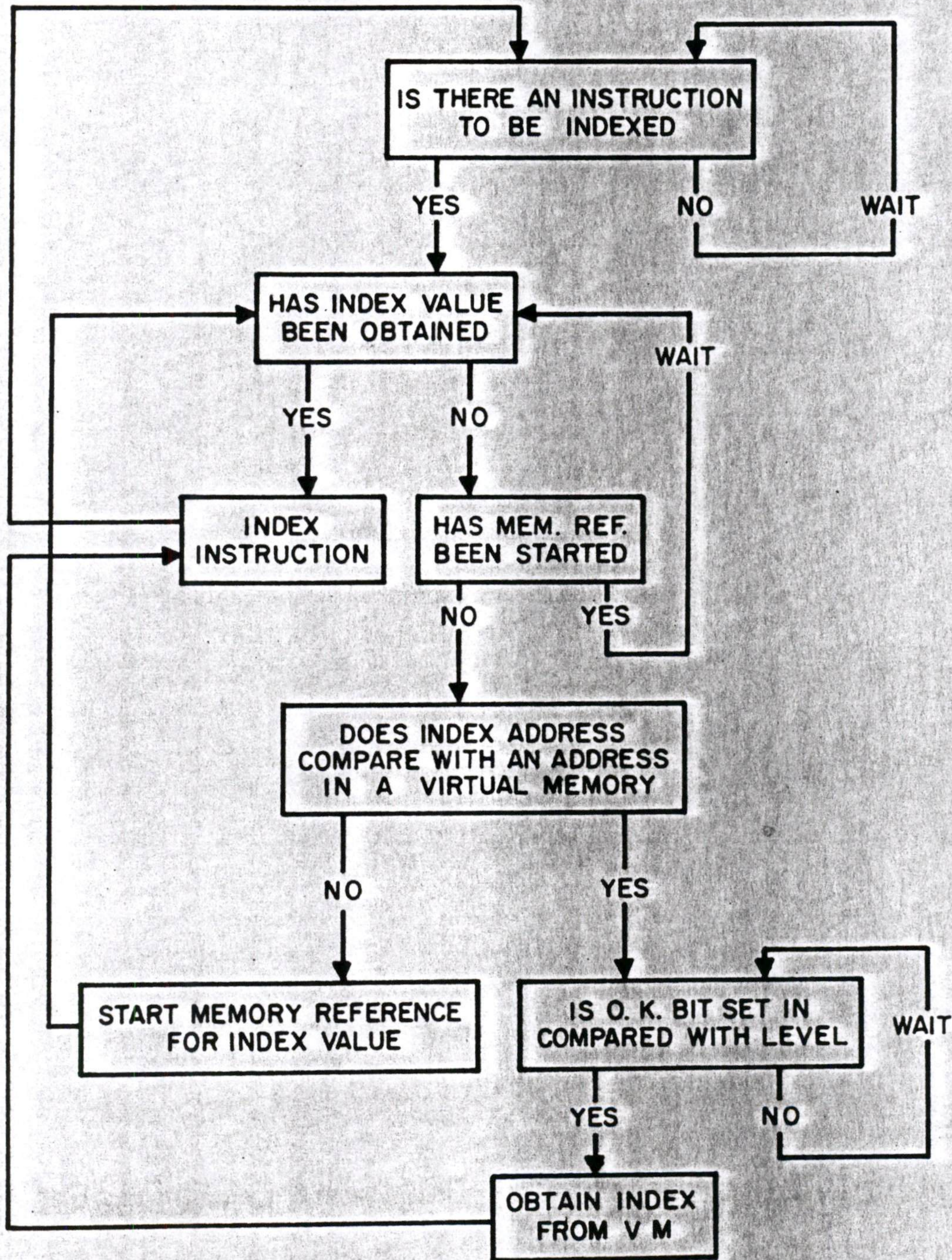


Fig. 5

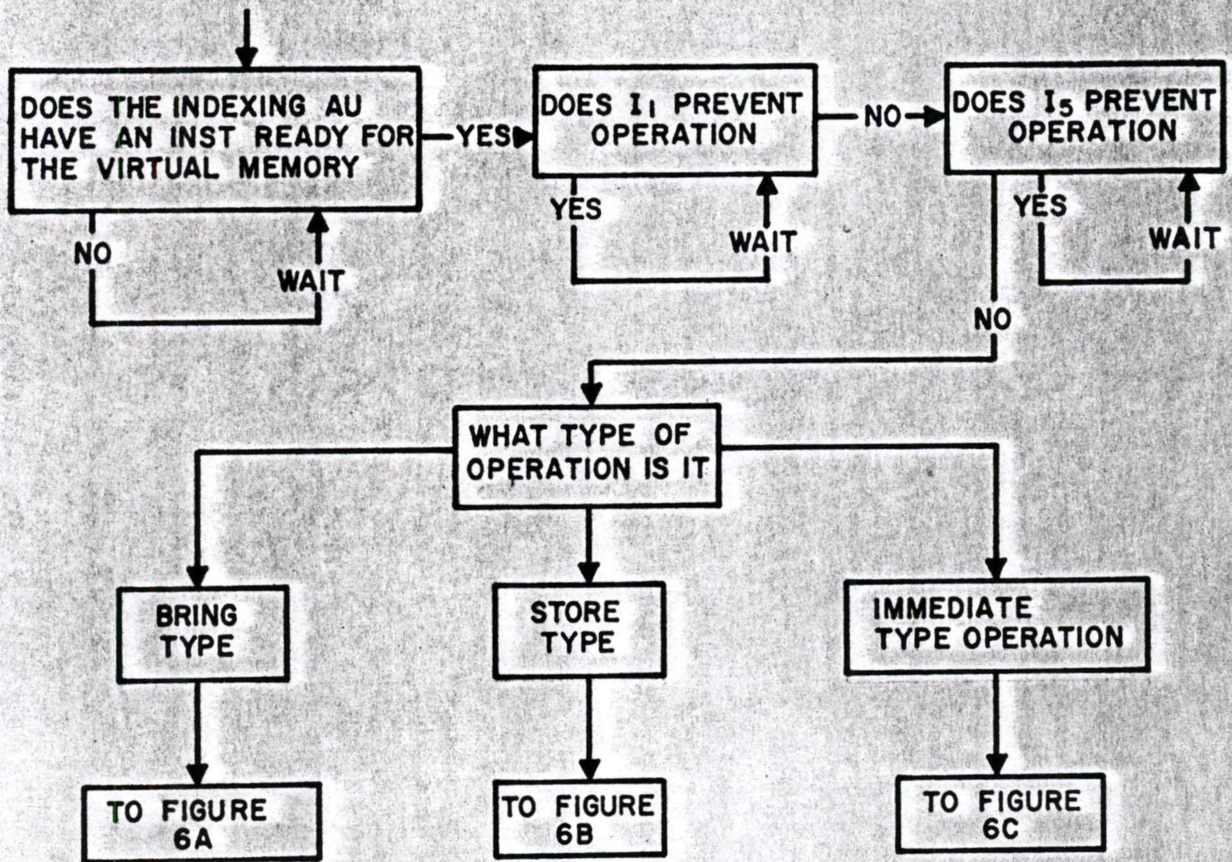


Fig. 6

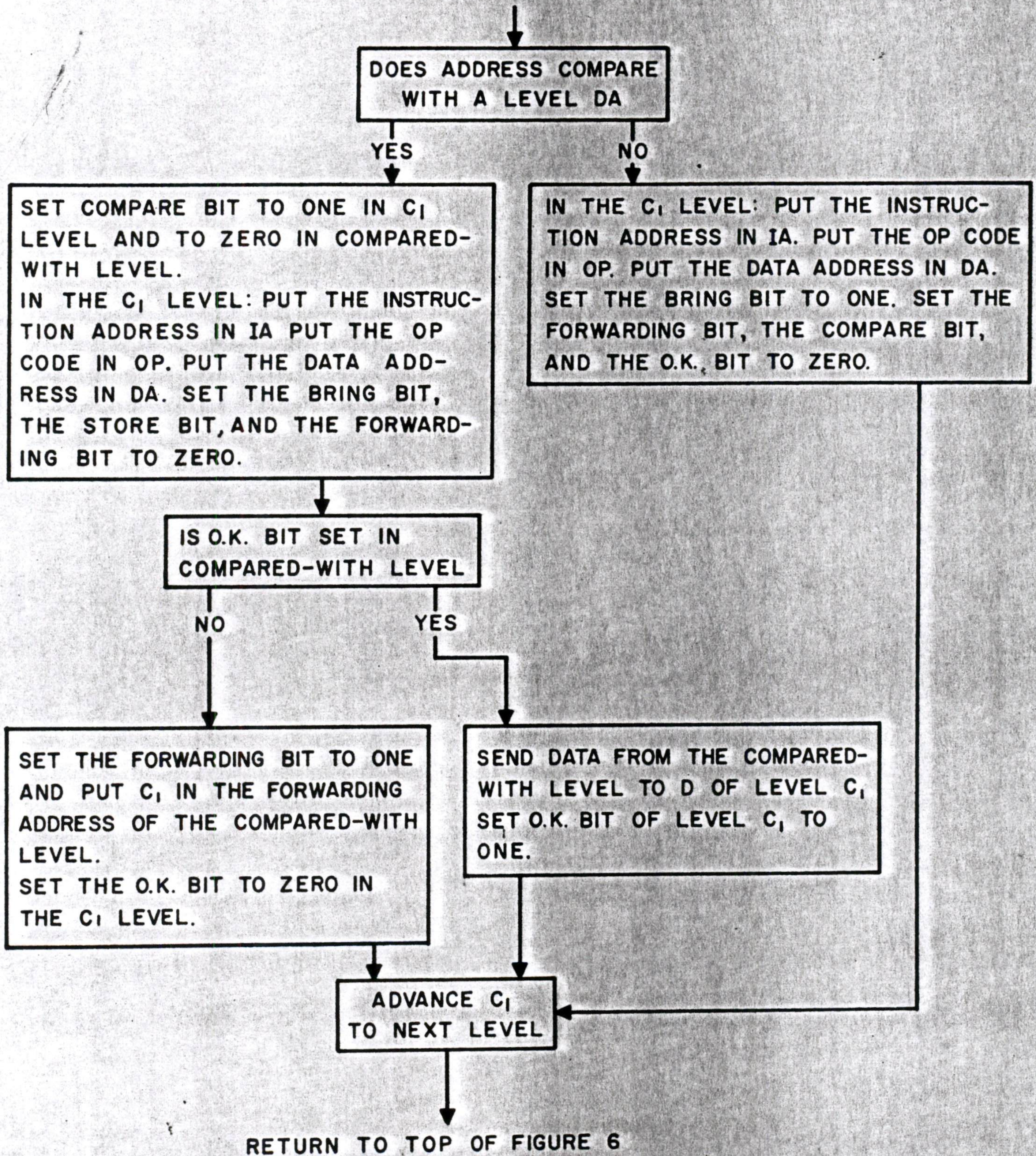


Fig 6A

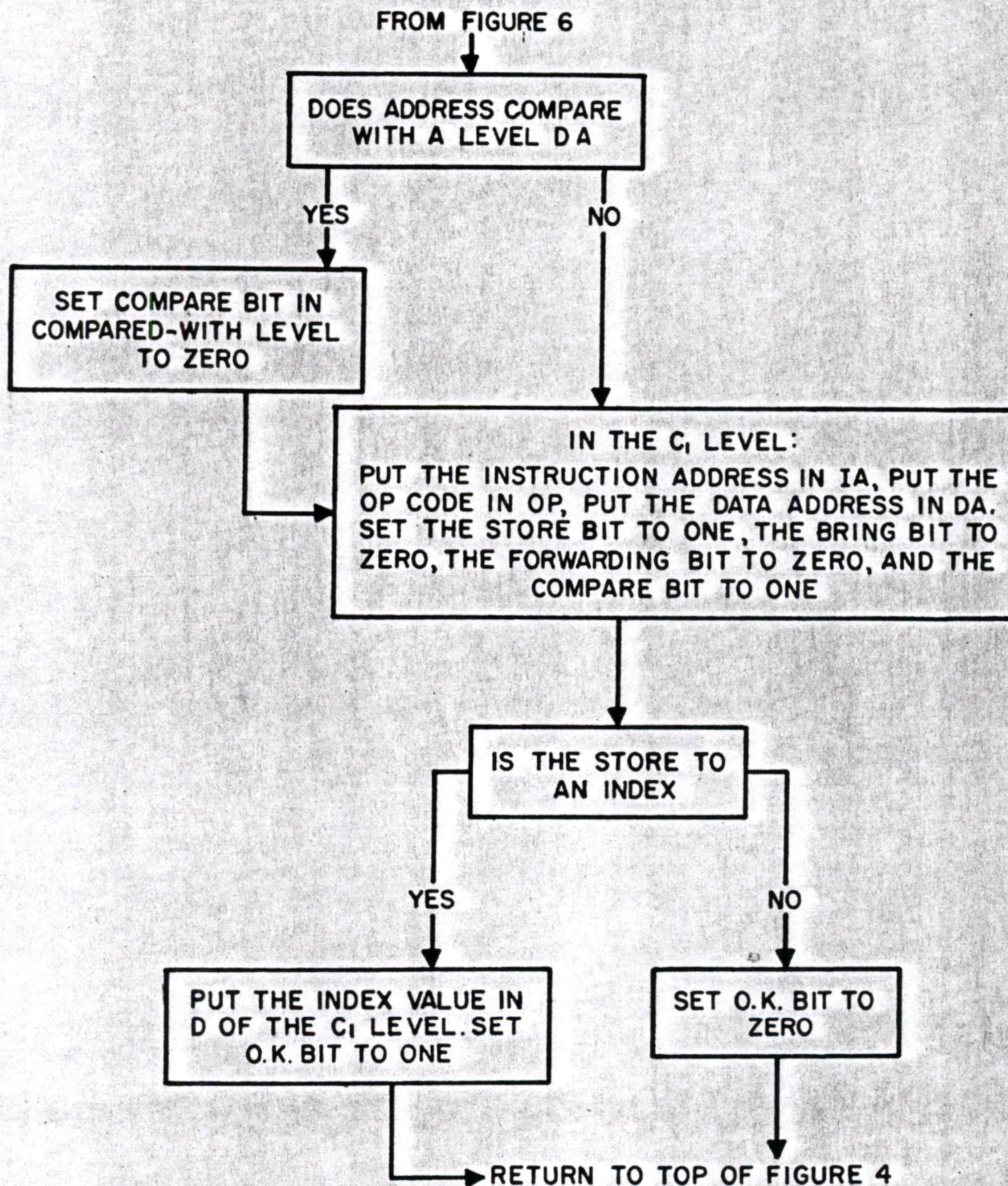


Fig. 6B

Fig 6C

FROM FIGURE 6

IN THE C_1 LEVEL:

PUT THE INSTRUCTION ADDRESS IN IA, PUT THE OP CODE IN OP. PUT THE DATA ADDRESS INTO D (NOTE THIS). SET O.K. BIT TO ONE. SET FORWARDING BIT, THE BRING BIT, AND STORE BIT TO ZERO. SET THE COMPARE BIT TO ZERO (NOTE).

RETURN TO TOP OF FIGURE 6

Fig. 6C

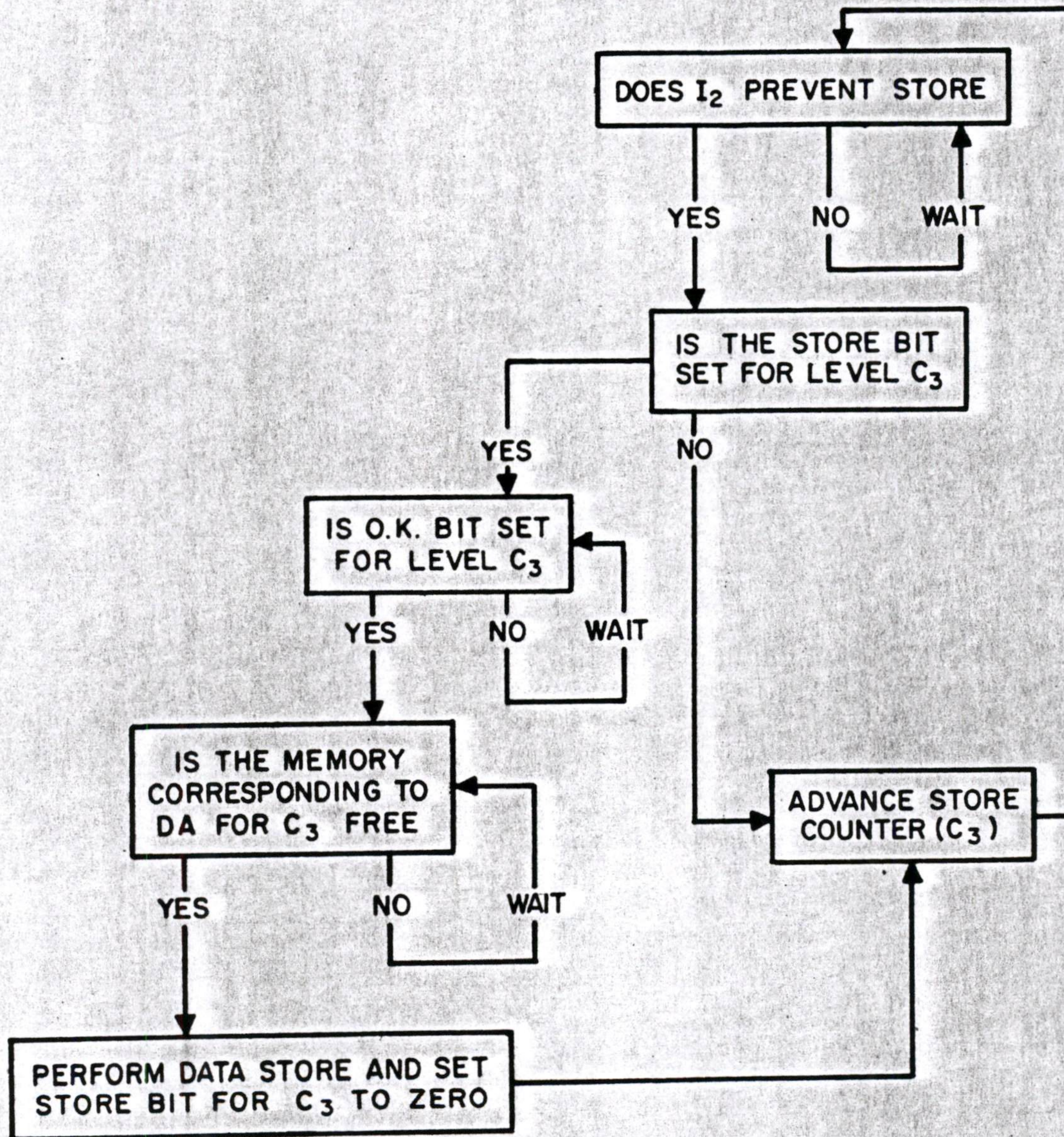


Fig. 8

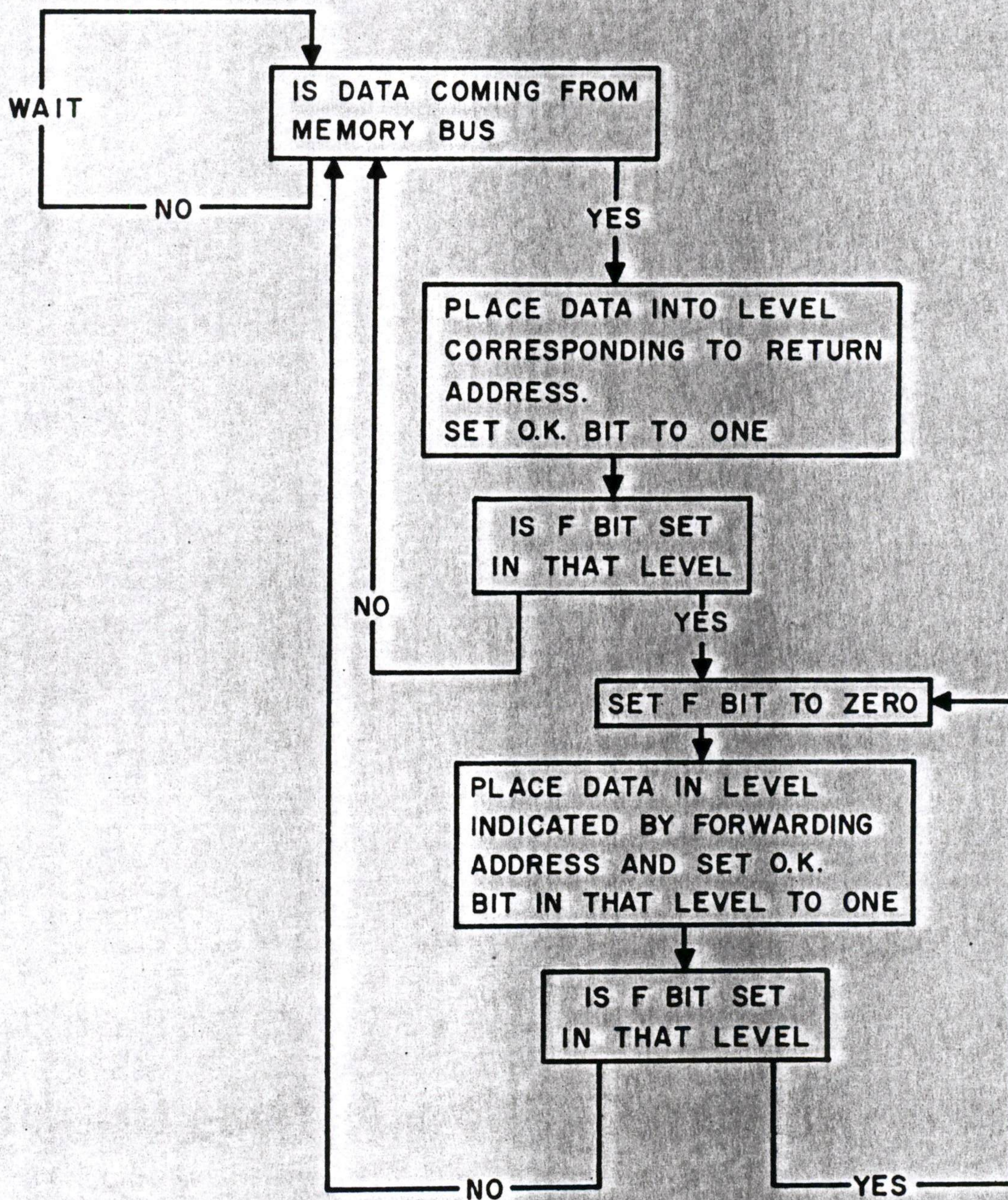


Fig 9

Fig. 9

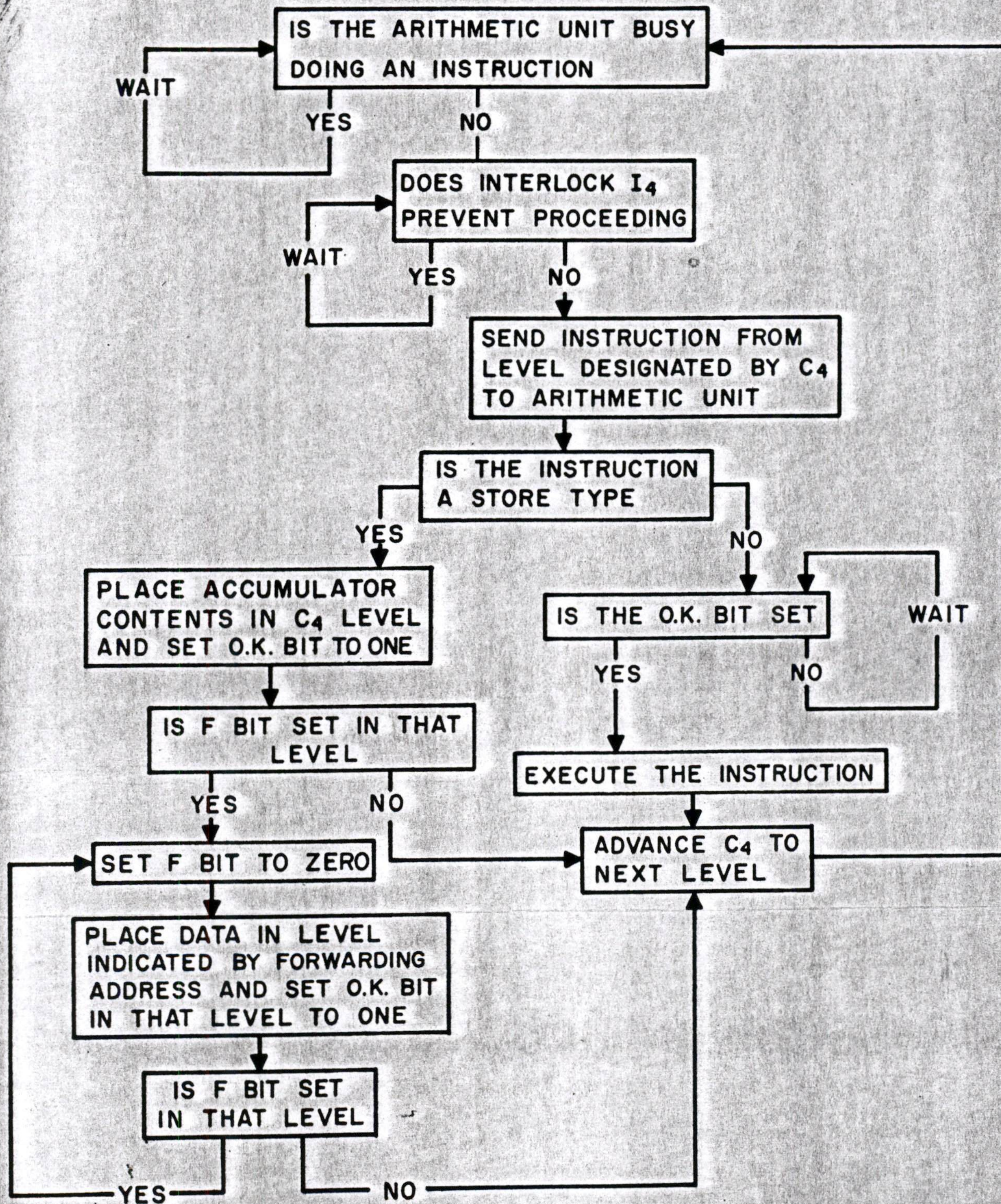


Fig. 10

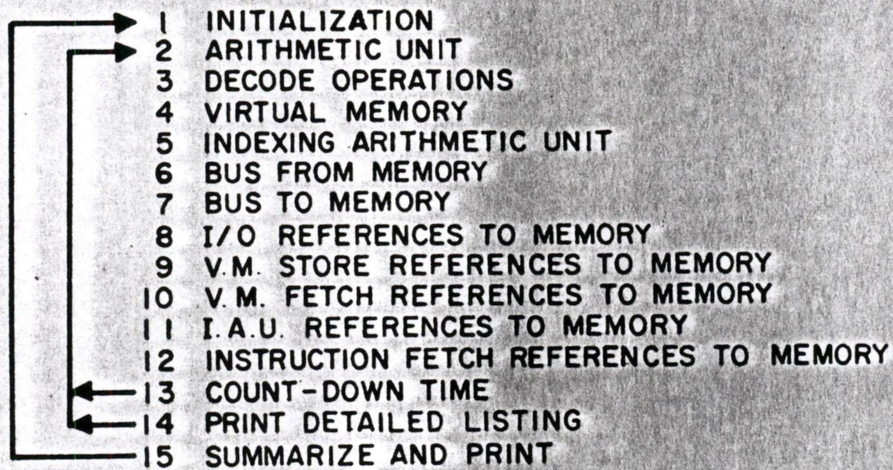


Fig 11

Fig. 11

PROJECT 7000 SIMULATOR 2 COCKE & KOLSKY NOV 57

	II	IS	AU	IF	IM	OF	OM	X1	X2	F1	F2	F3	F4	M1	M2	M3	M4	M5	M6	M7	M8	L1	L2	L3	L4	L5	L6	L7	L8	FD	MD	MC		
1	1																													1		2	1	W
2	1																													1		1	2	CW
3	1																													1			3	CW
4	1					1																								3		2	4	W
5	1					1																								3		1	5	W
6	1										1X																			3			6	W
7	1					3					1X																					2	7	W
8	1					3					1X																					1	8	W
9	1									3X	1X																						9	W
10	1			1						3X	X																					2	10	W
1	1			1						3X	X																					1	1	W
2	1									3X																							2	W
3	1			3						X																						2	3	W
4	1	1		3						X																						1	4	W
5	1	1																															5	W
6	1	2																														2	6	W
7	1	4																														1	7	W
8		1																															8	W
9	2	1																							1							2	9	
10	2	2																							1							1	10	
1	2	4	1					x																	1								1	W
2		1																							1							2	2	W
3	3	1																						2	1					5		1	3	
4	3	2																						2	1					5			4	
5	3	4	2			5		x																2	1							2	5	W
6		1				5																		2	1							1	6	W
7	4	1									5X												3	2	1								7	
8	4	2									5X												3	2	1							2	8	
9	4	2	3					x			5X												3	2	1							1	9	W
10	4	4									5X												3	2	1								10	W
1		1		5							X												3	2	1							2	1	W
2		1		5							X											4	3	2	1					7	4	1	2	
3		1																				4	3	2	1					7	4		3	W
4	5	1				7	4															4	3	2	1							2	4	W
5	5	2				7	4															4	3	2	1							1	5	W
6	5	2								7X				4X								4	3	2	1								6	W
7	5	4								7X				4X								4	3	2	1							2	7	W
8		1								7X				4X								4	3	2	1							1	8	W
9	6	1								7X				4X								4	3	2	5						5		9	W
10	6	2		7						X				4X								4	3	2	5						5	2	10	W
1	6	4		7						X				4X								4	3	2	5						5	1	1	W
2		1												4X								4	3	2	5						5		2	W
3	7	1					5							4X								4	3	6	5					9		2	3	W
4	7	2			4		5							X								4	3	6	5					9		1	4	W
5	7	2			4									X	5X							4	3	6	5					9			5	W
6	7	4				9								X	5X							4	3	6	5							2	6	
7		1	4			9								X	5X							4	3	6	5							1	7	
8	8	1	4								9X			X	5X							4	7	6	5						7		8	
9	8	2									9X			X	5X							4	7	6	5						7	2	9	W
10	8	2						x			9X			X	5X							4	7	6	5						7	1	10	W
1	8	2									9X			X	5X							4	7	6	5						7		1	W
2	8	4		9			7				X			X	5X							4	7	6	5							2	2	W
3		1		9	5		7				X			X	X							4	7	6	5							1	3	W
4		1			5									X	X		7X					8	7	6	5					11			4	CW
5	9	1												X	X		7X					8	7	6	5					11		2	5	C
6	9	2	5												X		7X					8	7	6	5					11		1	6	C
7	9	4	5												X		7X					8	7	6	5					11			7	C
8		1	5			11									X		7X					8	7	6	5							2	8	C
9	10	1	5			11									X		7X					8	7	6	9							1	9	C
10	10	2	5							11X					X		7X					8	7	6	9								10	C
1	10	2	5					x		11X					X		7X					8	7	6	9							2	1	C
2	10	2			7					11X					X		X					8	7	6	9							1	2	CW
3	10	4	6		7			x		11X					X		X					8	7	6	9								3	C
4		1		11						X					X		X					8	7	6	9							2	4	C
5		1	7	11						X							X					8	7	10	9					13		1	5	C
6		1	7														X					8	7	10	9					13			6	C

Fig.

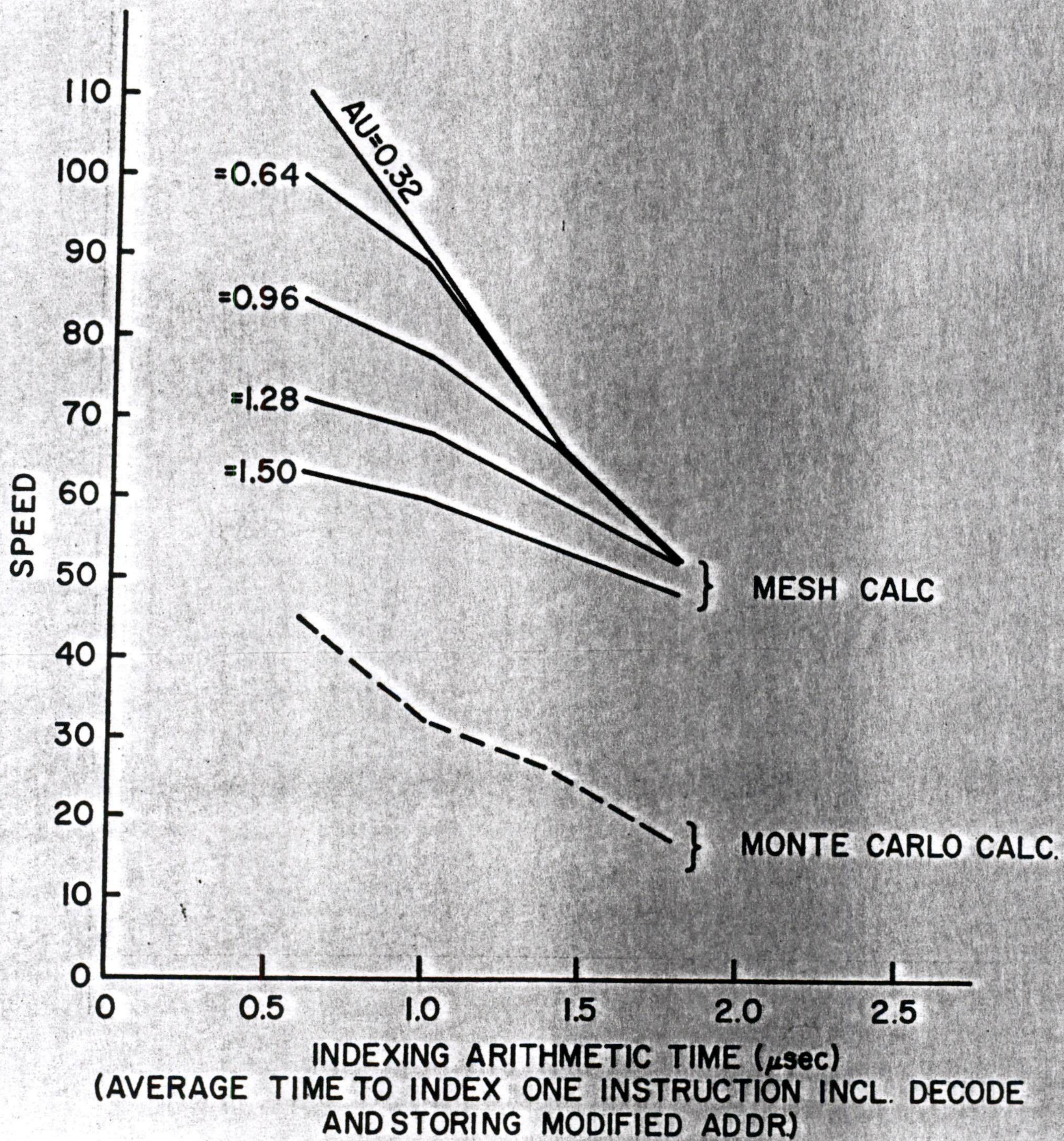


Fig. 15

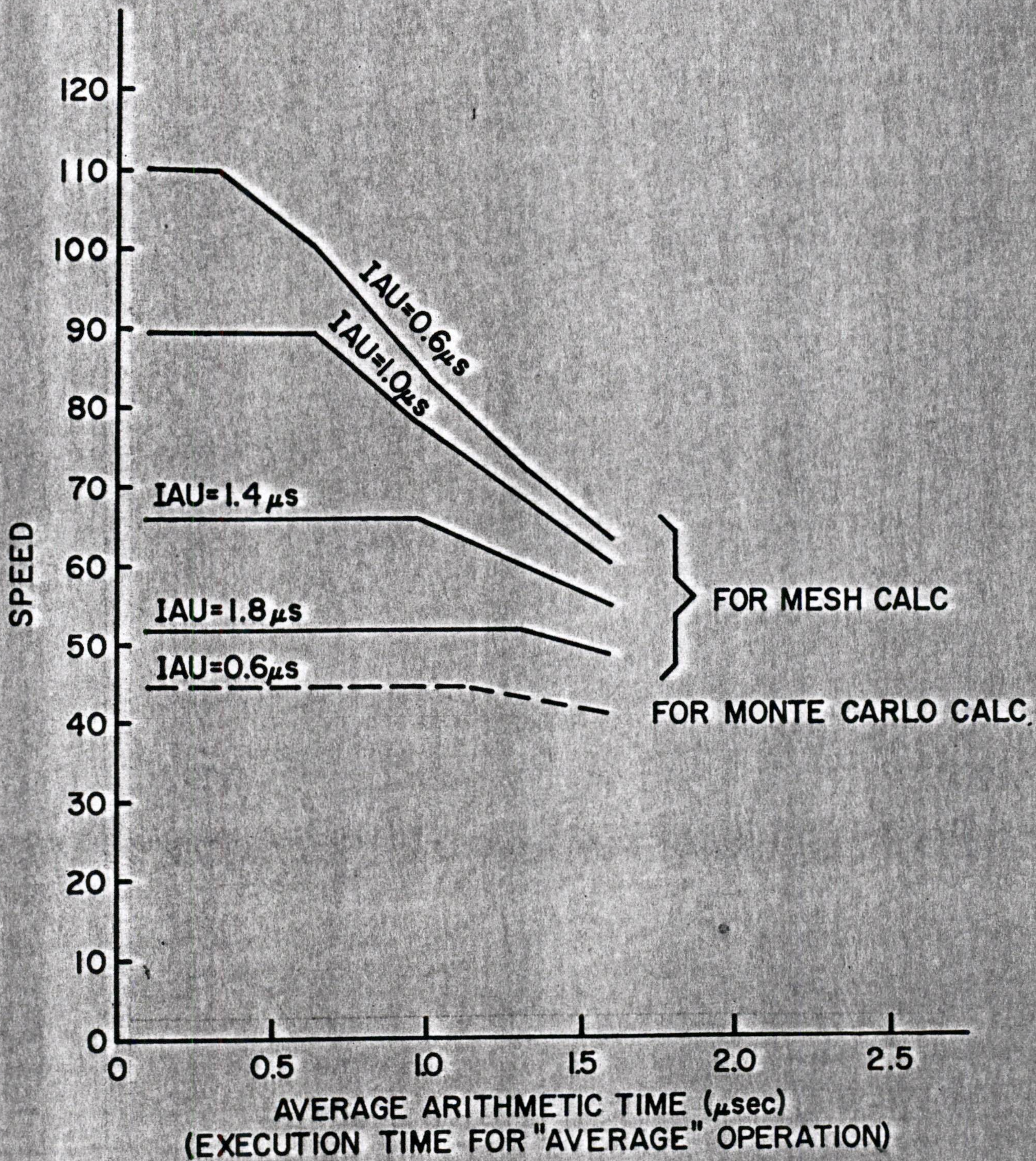


Fig. 16

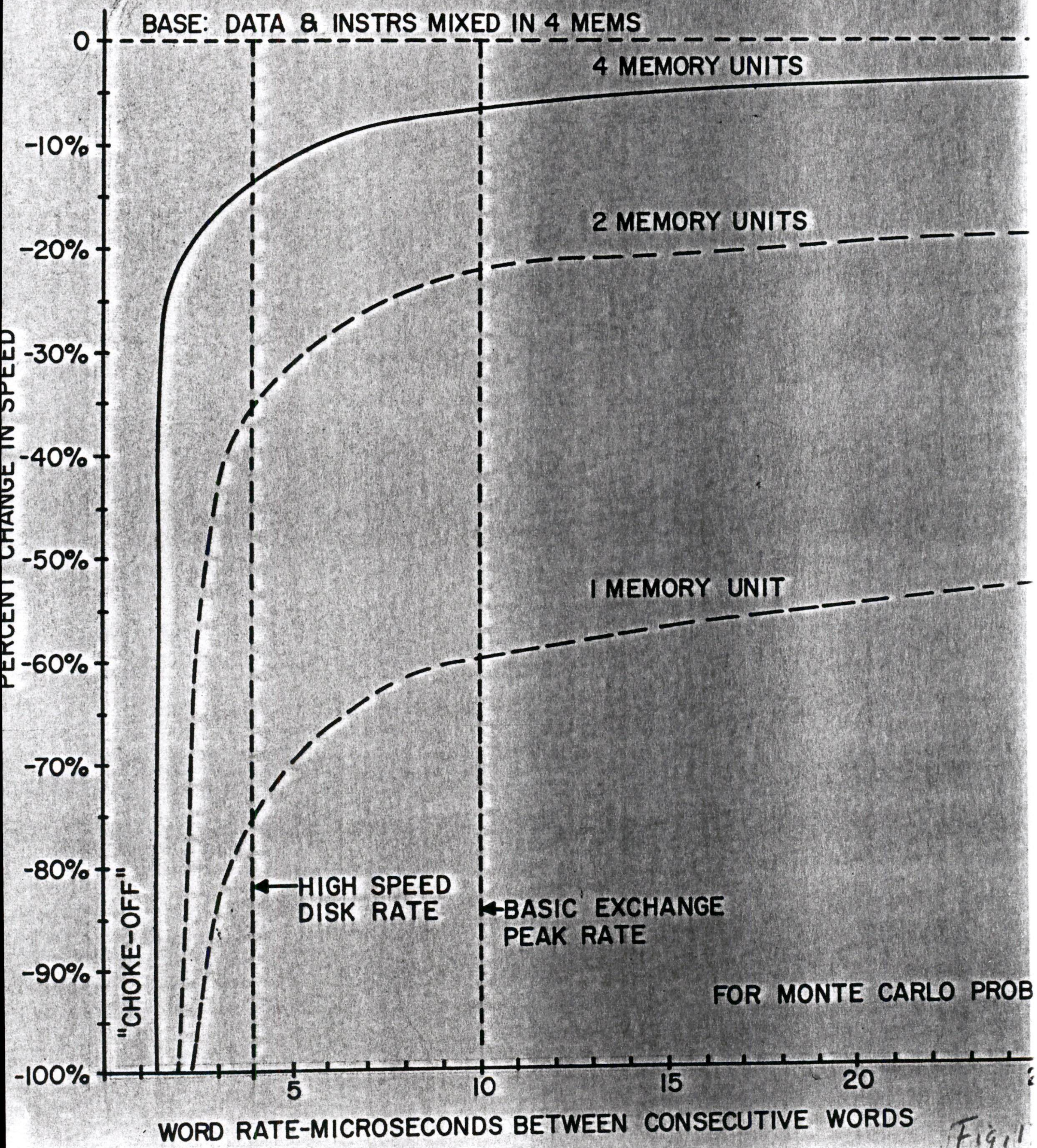


Fig. 1

IBM

Data Processing Division
Product Development Laboratory
Box 390, Poughkeepsie, New York

International Business Machines Corporation

Telephone: Globe 4-1000

November 13, 1959

Dr. Harlan E. Anderson, Chairman
EJCC Publication Committee
Digital Equipment Corporation
Maynard, Massachusetts

Dear Dr. Anderson:

Enclosed are four copies of the manuscript by Mr. Erich Bloch of this Laboratory, entitled "The Engineering Design of the Stretch Computer."

Also included are reproduction copies of the figures and photographs, and an abstract. The figures are identified by number, and a separate sheet contains all the captions.

I trust this material meets your specifications. If we can furnish any additional information, please let us know.

Sincerely yours,

P. J. Nelson, Editor
Publications Department
Laboratory Communications

Encl.
cc: Mr. Erich Bloch

File Copy

THE ENGINEERING DESIGN OF THE STRETCH COMPUTER

by

Erich Bloch

ABSTRACT

The Stretch Computer project was started in 1956 in order to achieve two orders of magnitude of performance improvement over the 704. In order to achieve this goal, all factors that go into the design of the system must contribute towards this goal -- the instruction set, the internal organization, the word length, the circuits and components.

This paper reviews the engineering design of the Stretch System with special emphasis on the central computer. After discussing the input/output, memory, and computer overlap operation, the computer is described in detail. Emphasis is placed on showing the multiplexing of instruction indexing, and instruction and operand fetching, with the execution in the arithmetic units. Examples are given to show where this overlap results in wrong guesses and the resulting difficulties in recovering the information.

The two arithmetic units, namely, a serial eight-bit unit and a parallel 96-bit unit, are described, as well as the algorithm used for the major floating point operations, such as add, multiply and divide.

Transistor counts and performance speeds for the major computer units are given and compared to 704 and 705 speeds. A description of the circuits and packages used in the Stretch System completes the paper.

THE ENGINEERING DESIGN OF THE STRETCH COMPUTER

by

ERICH BLOCH

PAPER TO BE PRESENTED AT

THE EASTERN JOINT COMPUTER CONFERENCE

DECEMBER 1, 1959

BOSTON, MASS.

INTERNATIONAL BUSINESS MACHINES CORP.

DATA SYSTEMS DIVISION

POUGHKEEPSIE, N. Y.

INDEX

I. Introduction

II. The Stretch System

III. The Stretch Computer

- A. The Dataflow
- B. The Arithmetic Units
  - 1. The Serial Arithmetic Unit
  - 2. The Floating Point Arithmetic Unit
- C. Checking
- D. Hardware Count
- E. Performance

IV. Circuits

V. Packaging

VI. Summary

ENGINEERING DESIGN OF THE STRETCH COMPUTER

I. INTRODUCTION

The STRETCH Computer [1] project was started in 1956 in order to achieve two order of magnitudes of improvement in performance over the then existing 704. Although this computer, like the 704, is aimed at scientific problems such as reactor design, hydrodynamics problems, partial differential equations etc., its instruction set and organization are such that it can handle with ease data processing problems normally associated with commercial applications, such as processing of alphanumeric fields, sorting, and decimal arithmetic.

In order to achieve the stated goal of performance, all factors that go into the computer design must contribute towards the performance goal; this includes the instruction set[3], the internal system organization, the data and instruction word length, and auxiliary features such as status monitoring devices, the circuits, packaging, and component technology. No one of them by itself can give this hundred-fold increase in speed; only by the combining and interacting of these contributing factors can this performance be obtained.

This paper reviews the engineering design of the Stretch System with primary concentration on the central computer as the main contributor to performance. In it, these new techniques, devices, and instructions have been pushed to the limit set by the present technology and, therefore, its analysis will convey best the problems encountered and the solutions employed.

II. THE STRETCH SYSTEM

Early in the systems design, it appeared evident that a six times improvement in memory performance and a ten times improvement in basic circuit speed over the 704 was the best one could achieve. To meet the proposed performance criteria, the system had to be organized in such a way that it took advantage of every possible overlap of systems function, multiplexing of the major portion of the system, processing of operations simultaneously, and anticipation of occurrences, wherever possible. The system had to be capable of making assumptions based on the probability that certain events might occur, and means had to be provided to retrace the steps when the assumption proved to be wrong.

This simultaneity and multiplexing of operations reflects itself in the Stretch System at all levels from the overall systems organization to the type cycle of specific instruction. In the following description, this will be discussed in more detail.

If one considers the Stretch System (Fig. 1) from an overall point of view it becomes apparent that the major parts of the system can operate simultaneously:

a. The 2-usec, 16,000-word core memories are self-contained, with their own clocks, addressing circuits, data registers and checking circuits. The memories themselves are interleaved so that the first two memories have their addresses distributed modulo 2 and the other four are interleaved modulo 4. The modulo 2 interleaved memories are used

primarily for instruction storage; since, for high performance instructions, halfword formats are used, the average rate of obtaining instructions is one per 1/2 usec. Similarly, a 0.5 usec data word rate is achieved by the use of the four modulo 4 organized memories. The addressing of the memories and the transfer of information from and to the memories by a memory bus permits new addresses and/or information to pass through the bus every 200 musec.

b. The simultaneously operating Input/Output units are linked with the memories and the computer through the Exchange which, after initial instruction by the computer, coordinates the starting of the I/O equipment, the checking and error correction of the information, the arrangement of the information into memory words, and the fetching and storing of the information from and to memory. All these functions are executed without the use of the computer, so it can in the meantime continue its data processing and computation.

c. The central computer processes and executes the stored program. Here, now, the simultaneity and multiplexing of functions has reached its ultimate.

Before discussing the computer organization, a few general features must be mentioned for completeness:

a. Word length: 64 bits plus eight bits for parity checks and error correction codes.

b. Memory capacity and addressing: A possible 256,000 words can be randomly addressed. These storage positions are all in external memory, except for the 32 first addresses. These positions consist of the internal registers (accumulators, time clocks, index registers).

c. The instructions are single address instructions with the exception of a number of special codes that imply the second address explicitly.

The instruction set (Figure 2) is generalized and contains a full set of single and double precision floating point arithmetic, a full set of variable field length integer arithmetic (binary and decimal). It also has a generalized set of index modification, a branching set as well as a set of I/O instructions. All told, 765 different types of instructions are used in the system.

d. The instruction format (Figure 3) makes use of both half and full words; halfwords accomodate indexing and floating point instructions (for optimum performance these two sets of instructions use a rigid format), and full word formats are used by the variable field length instructions. Notice that the latter specifies the operand field by the address of its leftmost bit, the length of the field, and the byte size, as well as the starting point (offset) of the implied operand (accumulator). Both halves of the word are independently indexable (two I fields).

e. A general monitoring device used for important status triggers is called the Interrupt[2] System. This system monitors the flip-flops, which reflect internal malfunctions, result significance (exponent range, mantissa zero, overflow, underflow), program errors (illegal instruction, protected memory area), and input/output conditions (unit not ready, etc.). The status of these triggers can cause a break in the normal progression of the stored program for fix-up purposes. The status of these triggers is automatically interrogated at all times.

III. THE STRETCH COMPUTER

If one considers the internal organization of the majority of computers that have been produced during the last eight years (and the 704 is a case in point), the organization looks as shown in Fig. 4a. There is a sequential flow of instructions into the computer, and after due processing and execution the next instruction is called from memory. Compare this with Fig. 4b, showing the organization of Stretch, where two instruction words and four operands can be fetched simultaneously. In addition, the execution of the instruction is done in parallel and simultaneously with the described fetching functions.

All the units of the computer are loosely coupled together, each one controlled by its own clock system, which in turn is synchronized by a master oscillator. This multiplexing of the units of the computer results in a large number of registers and adders, since time sharing of the major computer organs is no longer possible. All in all, the computer has 3, 000

register positions and about 450 adder positions.

Despite the multiplexing and simultaneous operation of successive instructions, the result appears as if sequential step-by-step internal operation were utilized. This has made the design of the interlocks quite complex.

A. The Data Flow:

The data flow through the computer is shown in Fig. 5 and is comparable to a pipeline which in a steady state (namely, once filled) has a large output rate no matter what its length. The same is true here; after start-up the execution of the instructions is fast and bears no relation at all to the stages it must progress through.

The Memory Bus is the communication link between the memories on one side and the exchanges and the computer on the other. It monitors the requests for storage to or fetches from memory and sets up a priority scheme. Since I/O units cannot hold up their requests, the exchange will get highest priority, followed by the computer. In the computer the operand fetch mechanism (lookahead) has priority over the instruction fetch mechanism. All told, the memory bus gets requests from and assigns priority to eight different channels.

Since memory can be accessed from multiple sources, and once accessed it is on its own to complete its cycle, a busy condition can exist. Here again, the memory bus tests for the busy conditions and delays the requesting unit until memory is ready to be interrogated on data fetches. The return address is remembered and the requesting unit receives the information when it becomes available. To accomplish this, from the time information is requested the receiving data register is in a reserved status.

Requests for stores and fetches can be processed at a 200 musec rate and the time, if no busy or priority conditions exist, to return the word to the requesting unit is 1.6 usec, a direct function of the memory read-out time.

The Instruction Unit[5] is a computer of its own. It has its own instruction set, its own small memory for index word storage, and its own arithmetic unit. During its operation as many as five instructions can be at various stages of development.

The Instruction Unit fetches the instruction words from memory, it steps the instruction counter, and performs the indexing of instructions and the initiation of data fetches. After a preliminary decoding of the class of instruction, it recognizes its own instructions and executes indexing instructions. On branches, conditional or unconditional, the instruction unit executes these. In the case of unconditional branches, it makes the assumption that the branch will not be successful.

This assumption and the availability of two full-word buffer registers keep the flow of instruction to the computer continuous. Therefore, the rate of instructions entering the instruction unit is for all practical purposes independent of the memory cycle.

Since, for high speed instructions, halfword formats are used, four of these at any one time can be in buffer storage. As soon as the instruction unit starts processing an instruction, the same is removed from the buffer, thus making room for the next memory word access (Fig. 6). Incidentally, half-word instructions and full-word instructions can be intermixed within the same word, and therefore the latter can cross a word boundary. This permits maximum package of instructions in memory and also serves as a

facility for automatic program assemblers and compilers.

The adder path, the index registers, and the transfer bus to lookahead complete the instruction unit system (Fig. 6). It should be noted that the index registers are part of the instruction unit data path, therefore permitting fast access (no long transmission lines) to an index word. There are 16 index words available to the programmer. The index registers, consisting of multiaperture cores, are operated in a nondestructive fashion, since in a representative program, the index word is used nine out of ten times without modifying it. This permits fast operation under these conditions, and additional time is only applied where modification is involved.

Typical operating speeds in the instruction units are as follows:

Cycle Time:		500 musec
Instr. Preparation Rate:		1 usec/half-word instr.
Index Add Time:		500 musec
ND Read Time:	Index Reg.	200 musec
Clear/Write Time:	Index Reg.	400 musec

After processing through the instruction unit, the updated (indexed) instruction enters a level of Lookahead (Fig. 5). Besides the instruction, all necessary information, its associated instruction counter value, and certain tag information are also stored in the same level. The operand, already requested by the instruction unit, will enter this level directly and will be checked and error corrected while awaiting transfer to the arithmetic units for execution.

An interlocked counter mechanism in the lookahead keeps its four levels in step, preventing out of sequence execution of instructions, even if all information for a succeeding one is available, before the previous instruction has been started.

The pre-accessing of operands by the lookahead and of instructions by the instruction unit leads sometimes to embarassing positions, for which a fix-up routine must be provided. Consider the program

(n) STORE Accumulator m

(n + 1) ADD m

and assume instruction (n) is in lookahead, waiting for execution. If (n + 1) now enters the lookahead, a reference to m cannot be made, since the data stored in that position is subject to change by the STORE instruction. The lookahead must recognize this and "forward" the result of instruction (n), when received, to the level where (n + 1) is stored.

Another example is the case where the instruction unit assumed that a conditional branch would not be executed. This instruction is stored in lookahead and, when it is recognized that the branch was successful, all modifications of addressable registers made by the instruction unit in the meantime must be restored. Lookahead in this case acts as a recovery memory for this information. A similar condition exists when interrupts occur due to arithmetic results. The lookahead here again has the data stored pertaining to registers which were modified erroneously in the meantime. The restoring and recovery routines described break into the instruction unit processing, interrupting temporarily the flow of instruction and their indexing.

The arithmetic units described later are slaves to the lookahead, receiving not only operands and instruction codes but also the start execution signal. Conversely, the arithmetic units signal to the lookahead the termination of an operation and, in the case of "To Memory" operations, place into the lookahead the result word for transfer to the proper memory position.

Typical operation times in the lookahead are as follows:

Cycle Time:	250 musec
Transfer Data LA - Arith. Unit:	250 musec

B. The Arithmetic Units

The design of the arithmetic units was established along lines similar to the design of lookahead and the instruction. Every attempt was made to speed up the execution of arithmetic operation by multiplexing techniques and overlapping of the algorithm, where mathematically permissible.

The arithmetic units, consisting of the Serial Unit and the Parallel Unit, use the same arithmetic registers, namely a double-length accumulator (A, B) consisting of 128 bits and a double-length operand register (C, D) consisting of 128 bits. The reason for the use of the same arithmetic registers is the fact that at any time a shift from floating point to variable field length operation or vice versa can be made by the program. Therefore, the result obtained by a floating point operation can serve as the starting operand for a variable field length operation. The chief reason for the double-length registers is the definition of maximum field length to be 64 bits. The field can start with any bit position, and therefore can cross the word boundary.

The execution of the floating point mantissa operations and the variable field length binary multiply and divide operations are performed by the parallel unit, whereas the execution of the floating point exponent operation and the variable field length binary and decimal add-type operations are executed by the serial unit. The square root operation and the binary-to-decimal conversion algorithm are executed in unison by both units. Salient features of the two units will now be described.

1. The Serial Arithmetic Unit. The serial arithmetic unit consists of a switch matrix which can extract 16 consecutive bits from A, B and C, D. These 16 bits then can be aligned in such a way that the low order bit of a field as specified by the instruction is at the right end of the field. This wrap-around circuit then feeds into a carry propagate adder or, in case of logical connect instructions, into the logic unit. At the adder output, a true complement unit and a binary-to-decimal correction unit are used for subtract and decimal operations. The inverse process of extracting is used to insert the processed byte back into the register without disturbing any neighboring positions. Notice that in one clock cycle of 500 musec duration, the information is extracted, the arithmetic is performed and the result inserted back into the registers. In addition, the arithmetic information is checked by parity checks on the switch matrices and by duplication and comparison of the arithmetic procedure in a duplicate unit.

2. Parallel Arithmetic Unit. The parallel arithmetic unit (Fig. 8) is designed to execute floating point operations with a maximum of efficiency. Since both single and double precision arithmetic are performed, the shifter and adder exist in a double length format of 96 bits. This insures almost the same performance for single and double precision arithmetic. The adder is of a carry propagation type with lookahead over 4 bits at a time to reduce the delay that

The four multiple multiplicand groups and the partial product of the previous cycle are now fed into carry save adders of the form,

$$\text{Sum} \quad S = A \vee B \vee C$$

$$\text{Carry} \quad C' = AB + AC + BC.$$

There are four of these adders, two in parallel followed by two more in series (Fig. 8). The output of Carry Save Adder 4 then results in a double rank partial product, the product sum and the product carry. For each cycle this is fed into Carry Save Adder 2, and, during the last cycle, into the carry propagate adder, for accumulation of the carries.

Since no propagation of carries is required in the four cycles, where multiple multiplicands are added, this operation is fast and is the main contributor to the fast multiply time of Stretch.

The Divide scheme[6] has a similarity to the multiply scheme. Multiples of the divisor are used, namely, $\frac{3}{2}$ x Divisor, $\frac{3}{4}$ x Divisor and 1 x Divisor. This plus the shifting over strings of ones and zeros results in the generation of the required 48 quotient bits within thirteen machine cycles. Most machines using a nonrestoring divide method require 48 cycles for 48 quotient bits.

The following example explains this technique. This scheme depends on the use of normalized divisors:

DIVIDEND	(DD)	=	101000000000000
DIVISOR	(DR)	=	1100011
2's COMPDR	($\overline{DR}$)	=	0011101
3/4 DR	(3/4DR)	=	100101001

(a) <u>Using skip over 1/0 only:</u>

While the mantissa operations are performed in multiply and divide are performed by the parallel unit , the serial arithmetic unit executes the exponent arithmetic. Here again is a case where overlap and simultaneity of operation is used to special advantage.

3. Checking. The operation of the computer is checked in its entirety and correction codes are employed where transfers of data from memory input/ output data transfers is involved.

In particular, all information sent to memory has a correction code associated with it which is checked for accuracy on its way from memory. If a single error is indicated, then correction is made and the error is recorded via a maintenance output device. Within the machine all arithmetic operations are checked, either by parity, duplication, or a "casting out three" process. These checks are overlapped with the execution of the next instruction.

4. Hardware Count. Figure 9 shows the percentage of transistors used in the various sections of the machine. It becomes obvious that the floating point unit and the instruction unit use the highest percentage of transistors. In case of the floating point unit this is due to the extensive circuits for multiply and to the additional hardware to achieve speed up the divide scheme.
In the instruction unit, the controls consume the majority of the transistors, because of the high multiplexed operation encountered.

5. Performance. The performance comparisons in Fig. 10 show the increase in speed achieved, especially in the floating point operations, over the 704. It should be noted that for a large number of problems this particular increase in all arithmetic speeds is almost proportional to the performance

increase of the problem as a whole, since the instruction execution times are overlapped to a great extent with the preparation and fetching of instructions. Simulation of Stretch programs on the 704 proved a performance of 100 x 704 in mesh type calculations. Higher performance figures are achieved where double or triple precision calculations are required.

IV. CIRCUITS

Having reviewed the systems organization of Stretch, it is now of interest to discuss briefly the components, circuits, and packaging techniques used to implement the design.

The basic component used in Stretch is the high-speed drift transistor which exists in both an NPN and a PNP version. This transistor has a frequency cut-off of approximately 100 mc and for high-speed operation must be kept out of saturation at all times. This then explains why both the PNP and NPN version are used, mainly to avoid the problem of level translation, which would be required due to the potential difference of the base and the collector. This difference is 6 volts, an optimum point for this device.

Figure 11 shows the basic circuit configuration. It consists of a current source represented by the -30 volt supply and resistor R. The functional operation of the circuits consists of two possible paths represented by transistor A or C. Which path is chosen by the current depends on the condition existing on base A. If point A is positive with respect to ground by 0.4 volts, that particular transistor is cut off, making the emitter of transistor C positive with respect to the base and, therefore, making C conducting. The current supplied by the current source (6 ma) will then flow through transistor C to the load ∅. Output ∅, then, is positive by 0.4 volts with respect to the -6 volt reference. This indicates at ∅ the equivalent function impressed on A. At the same time $\bar{\phi}$ is negative with respect to the -6 volt power supply by 0.4 volt, representing, therefore, the inverse of the function impressed on A. Conversely if A is negative with respect to the ground

reference, transistor A is the conducting one, keeping emitter C negative with respect to its base. The current flows through transistor A, making $\bar{\phi}$ positive with respect to -6 and ϕ negative with respect to -6. Again, the output of ϕ reflects the function impressed on A, whereas $\bar{\phi}$ represents the inverse of the function.

If an additional transistor now is paralleled with A it becomes obvious that only if both bases A and B are positive will output ϕ be positive and $\bar{\phi}$ negative. If any or none of the bases A and B are positive then ϕ will be negative and $\bar{\phi}$ will be positive. In other words, an AND function is obtained on output ϕ.

This principle, which is reflected in all the circuits, is essentially the principle of current switching or current steering.

Logical functions for the PNP circuits are, therefore, a + AND or - OR. Two outputs from each circuit block are available: the AND function and the inverse of the AND function.

A dual circuit exists for NPN transistors with input levels at -6 volts and output levels at ground. This circuit will give the + OR or -AND function.

A thorough investigation of the systems design showed that the circuits described so far are versatile enough to be used throughout the system. However, there are enough special cases (resulting from the many data buses and registers throughout the machine) that could use a distributor function or an overriding function. This caused the design of a circuit which permitted great savings in space and transistors by adding a third voltage level. Figure 12 shows the PNP version of the third-level circuit.

having a gain less than one, after a number of stages require the use of current switching circuits as level setters and gain devices. Both AND and OR circuit are available for both a ground level and a -6 level input. Change from a -6 level circuit to a ground level circuit is obtained by applying the appropriate power supply levels. Due to the variations in inputs and driven loads, the circuits must be designed so that the load can vary over a wide range. This resulted in instability which had to be offset by the feedback capacitor C shown in the circuit.

All functions needed in the computer can be implemented by the use of the aforementioned circuits, including the flip-flop operation, which is obtained by tying a PNP current switch block and an NPN current switch block together with proper feedback.

V. PACKAGING

The circuits described in the last paragraph are packaged in two ways.

(a) A circuit package using the smaller of the two printed circuit boards shown in Figure 14, called a single card, contains AND or OR circuits. It should be mentioned that the wiring is one-sided and that besides the components and transistors, a rail is added which permits the shorting or addition of certain loads depending on the use of the circuits. This rail then has the effect of reducing the different types of circuit boards in the machine. Twenty-four different boards are used and of these, two types reflect approximately 70% of the total single card population of the machine.

REFERENCES

1. S.W. Dunwell - Design objectives for the IBM Stretch Computer EJCC Proceedings pg. 20, December 1956.

2. F.P. Brooks, Jr. - A Program Controlled Program Interuption System, EJCC Proceedings pg. 128, December 1957.

3. W. Buchholz - Selection of an Instruction language, WJCC Proceedings, page 128, May 1958.

4. F.P. Brooks, Jr. - et al, Processing Data in Bits and Pieces, IRE Transactions on Electronic Computers, June 1959.

5. G.A. Blaauw - Indexing and Control - Word Techniques, IBM Journal July 1959.

6. J.E. Robertson - "A New Class of Digital Division Methods," IRE Trans. on Electronic Computers, vol. EC-7, pp. 218-222; September, 1958.

LIST OF ILLUSTRATIONS AND CAPTIONS

Fig. 1: The Stretch System

Fig. 2: The Instruction Set

Fig. 3: Data Word - and Instruction Word Formats

Fig. 4: Comparison of Stretch and 704 Organization

Fig. 5: Stretch Computer - Units and Dataflow

Fig. 6: Instruction Unit

Fig. 7: Serial Arithmetic Unit

Fig. 8: Floating Point Arithmetic Unit

Fig. 9: Component Count

Fig. 10: Comparison of Stretch and 705/704 Operation Times

Fig. 11: Current Switching Circuits (+AND)

Fig. 12: Third Level Circuit

Fig. 13: Emitter Follower Circuit

Fig. 14: The Circuit Package

Fig. 15: The Backpanel

Fig. 16: The Frame (Closed)

Fig. 17: The Frame (Extended)

COMPUTER VOCABULARY

INSTRUCTION CATEGORY	CLASS	MODIFIER	EXAMPLES	NUMBER OF INST
VARIABLE FIELD LENGTH ARITHMETIC	BINARY DECIMAL	SIGNED UNSIGNED SAME SIGN NEGATIVE SIGN	ADD (TO MEMORY) LOAD/STORE MPY DIVIDE CUMULATIVE MPY	280
RADIX CONVERSION	BIN/DEC			32
LOGIC CONNECTS			16 LOGIC STATEMENT	48
FLOATING POINT ARITHMETIC	NORMALIZED UNNORMALIZED	SAME SIGN OPPOSITE SIGN NEGATIVE SIGN NOISY MODE	ADD (SINGLE & DOUBLE) LOAD/STORE MPY/(SINGLE & DOUBLE) DIV (WITH REMAINDER) INTERCHANGE DIVIDE CUMULATIVE MPY SQUARE ROOT	240
INDEXING ARITHMETIC	DIRECT IMMEDIATE PROGRESSIVE			43
BRANCHES STORE INST CTR	UNCONDITIONAL INDEXING INDICATOR BIT	IF {1, 0} (INDICATOR, BIT) SET O LEAVE BIT INVERT BIT		68
TRANSMIT / SWAP I / O INSTRUCTION				24
			TOTAL	735

Fig. 2

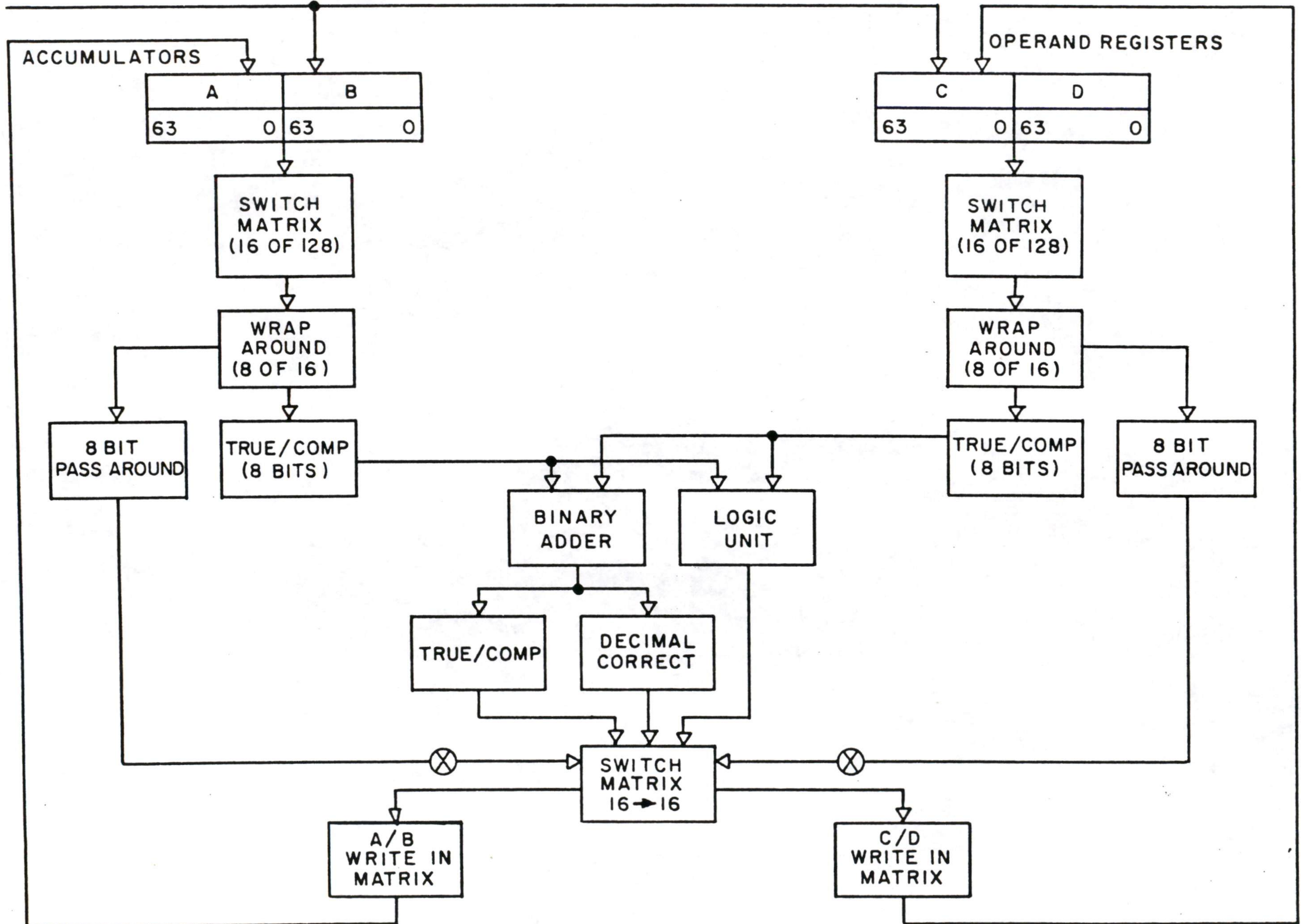


Fig 7. SERIAL ARITHMETIC UNIT

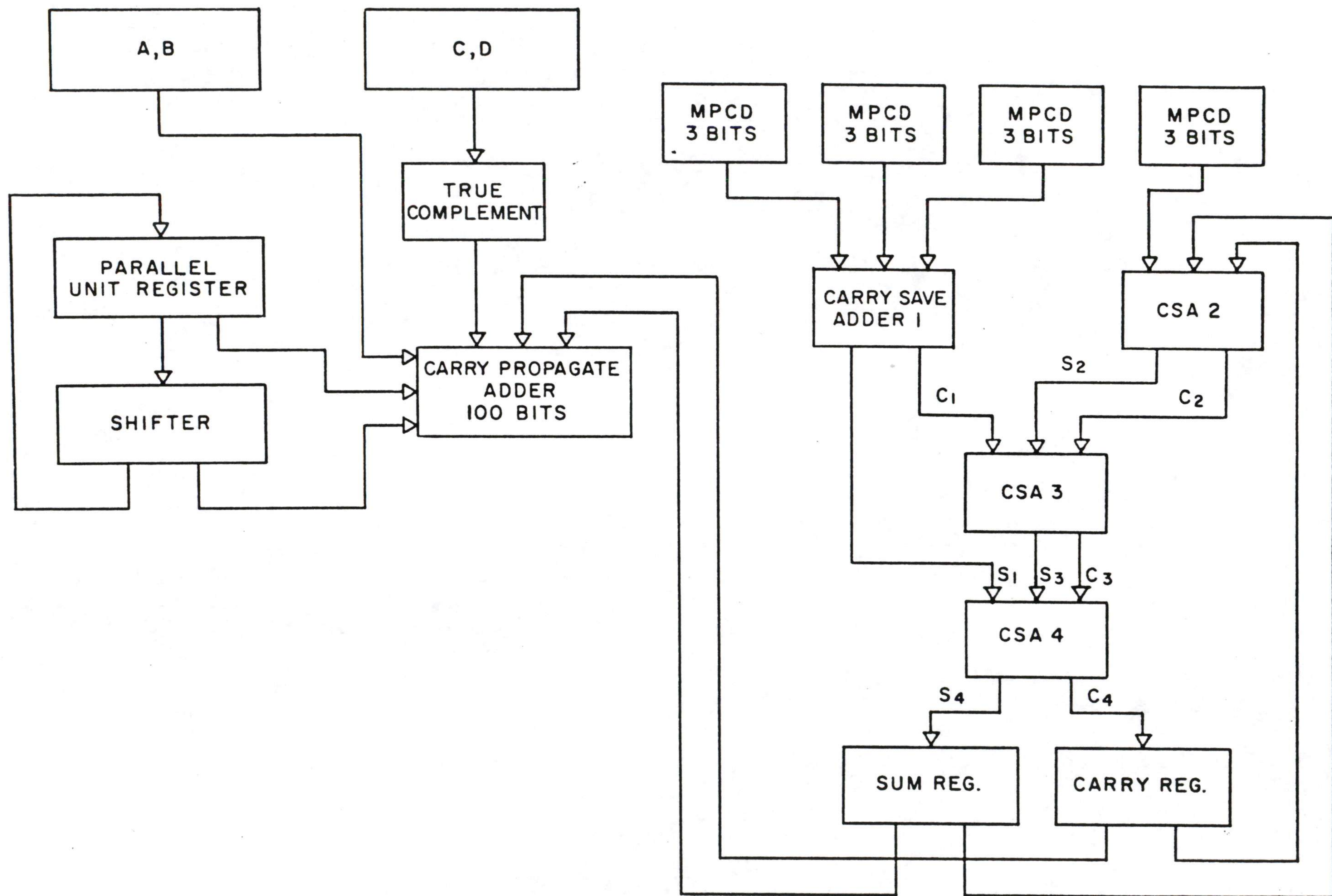


Fig 8: PARALLEL ARITHMETIC UNIT

UNIT	# OF TRANSISTORS	% OF TOTAL	# OF FRAMES
MEMORY CONTROLS	10,500	6.0	2
INSTRUCTION UNIT			
DATAPATH	17,700	22.0	2
CONTROLS	19,500		3 1/2
LOOK-AHEAD			
DATAPATH	17,900	15.6	1
CONTROLS	8,600		1 1/2
ARITH. REGISTERS	10,000	5.9	1
SERIAL ARITH. UNIT			
DATA PATH	10,000	10.5	1 1/2
CONTROLS	8,700		1
FLOATING PT. UNIT			
DATAPATH	32,700	21.0	2 1/2
CONTROLS	3,000		1/2
CHECKING	24,500	14.5	1
INTERRUPT SYSTEM	6,000	3.5	1/2
TOTAL	169,100	100.0	18

DOUBLE CARDS	4,025
SINGLE CARDS	18,747
POWER	21 KW

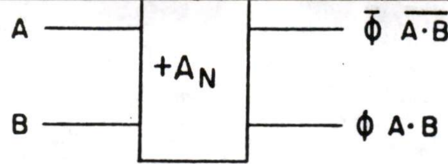
Fig 9 COMPUTER COMPONENT COUNT

COMPARISON OF STRETCH CHARACTERISTICS AND OPERATION TIMES WITH 704/705

OPERATION	IBM 704	IBM 705	STRETCH
1. FLOATING POINT			
EXPONENT RANGE	$\pm 2^{\pm 128}$		$\pm 2^{\pm 2048}$
MANTISSA BITS	27		48
FLOATING ADD	84 USEC		1.0 USEC
FLOATING MPY	204 USEC		1.8 USEC
FLOATING DIV	216 USEC		7.0 USEC
LOAD/STORE	24 USEC		.6 USEC
2. BINARY VARIABLE FIELD LENGTH ARITH.			
BIT RANGE			1 TO 64
16 BIT FIELD: ADD/LOAD/STORE			2.0 USEC
16 BIT FIELD: MPY			10.0 USEC
16 BIT FIELD: DIVIDE			15.0 USEC
3. DECIMAL ARITHMETIC			
DIGIT RANGE		1 → MEM CAPACITY	1 TO 21
FOR 5 DIGITS: ADD		119 USEC	3.5 USEC
FOR 5 DIGITS: MPY		799 USEC	40.0 USEC
FOR 5 DIGITS: DIVIDE		4828 USEC	65.0 USEC
FOR 5 DIGITS: LOAD/STORE		204 USEC	3.2 USEC
4. MISCELLANEOUS			
ERROR CORRECTION	NO	NO	YES
CHECKING	NO	YES	YES
WORDSIZE	36 BITS		64 BITS

Fig. 10

SYMBOL



TRUTH TABLE

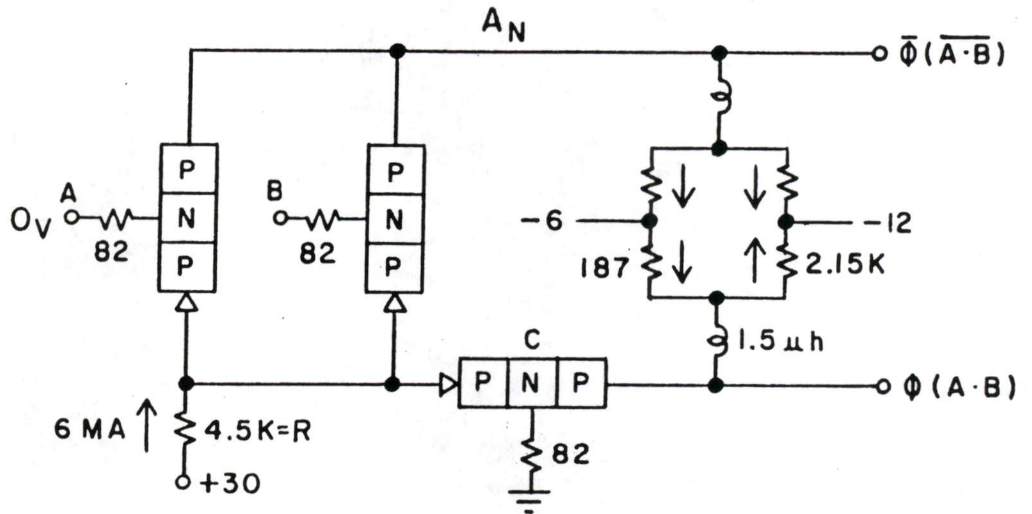
A_N

A	B	Φ	$\overline{\Phi}$
+	+	+	−
+	−	−	+
−	+	−	+
−	−	−	+

$\Phi = A \cdot B$

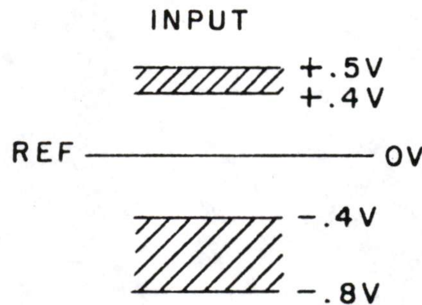
$\overline{\Phi} = \overline{A \cdot B}$
$= \overline{A} + \overline{B}$

CIRCUIT DIAGRAM A_N

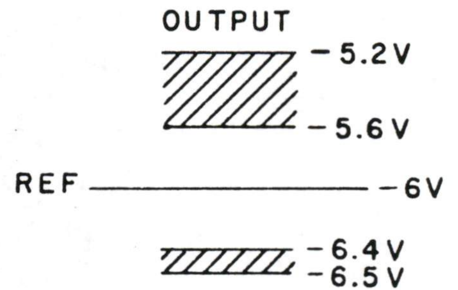


MIN − MAX SIGNAL VOLTAGES

INPUT



OUTPUT



CIRCUIT RESPONSE

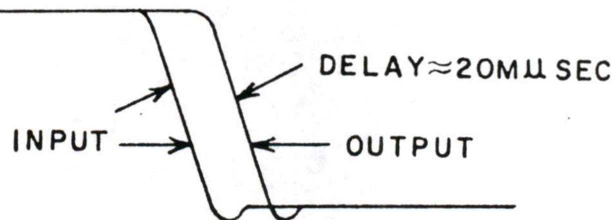
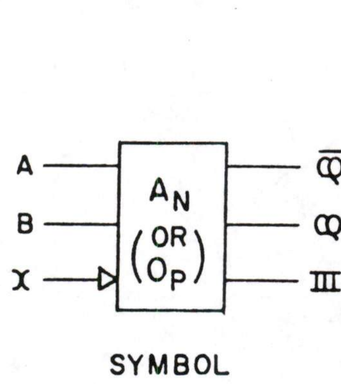


Fig 11: CURRENT SWITCHING CIRCUITS



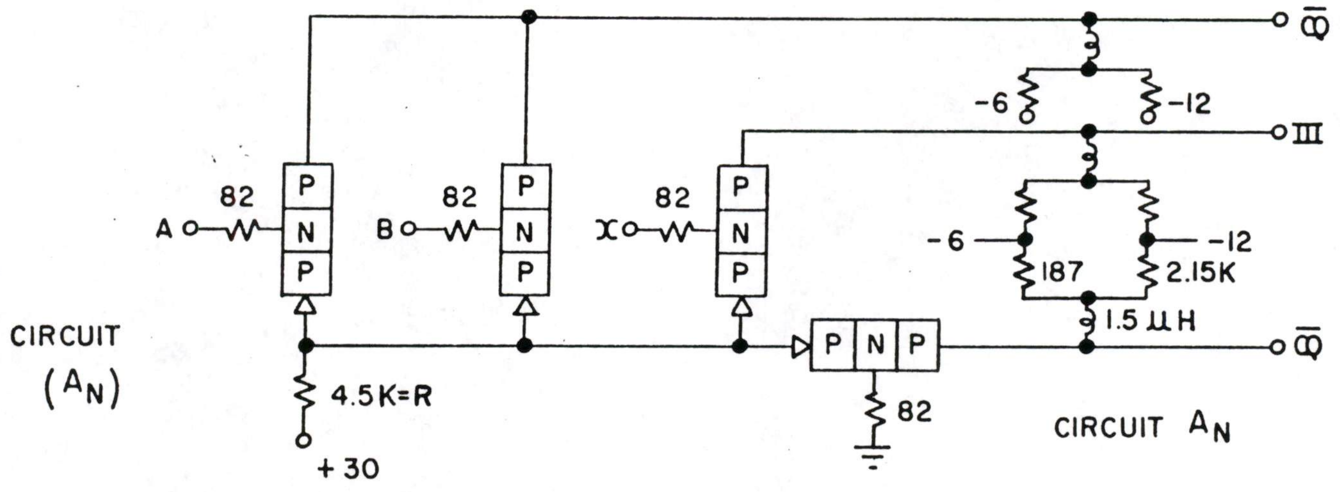
A_N

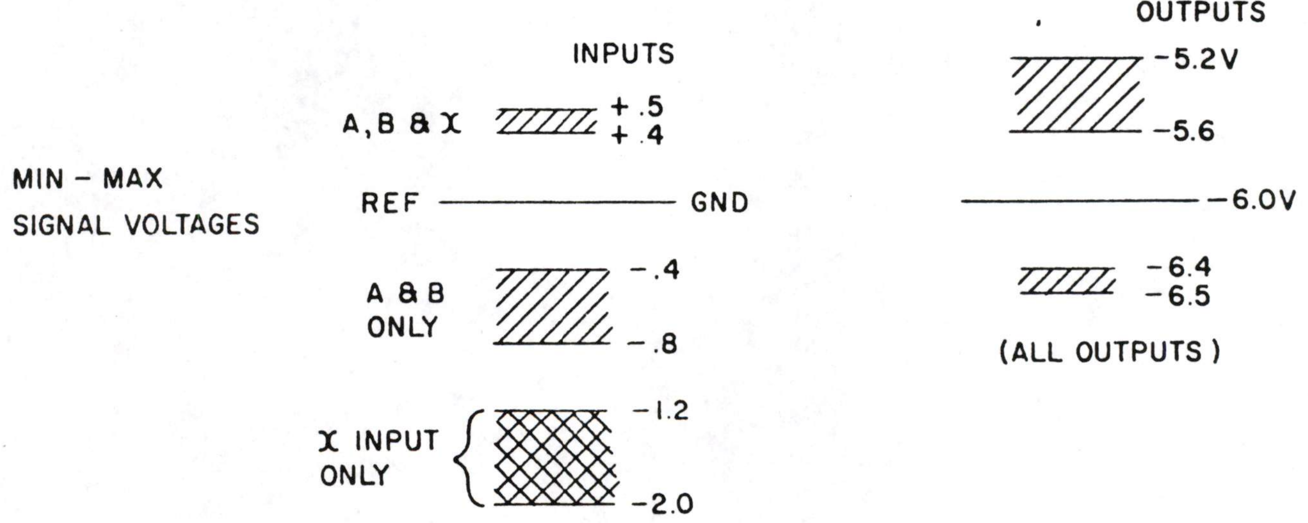
A	B	X	Q	$\overline{Q}$	III
+	+	+	+	−	−
−	+	+	−	+	−
+	−	+	−	+	−
−	−	+	−	+	−
+	+	−	−	−	+
−	+	−	−	−	+
+	−	−	−	−	+
−	−	−	−	−	+

O_P

A	B	X	Q	$\overline{Q}$	III
+	+	+	+	+	−
−	+	+	+	+	−
+	−	+	+	+	−
−	−	+	+	+	−
+	+	−	+	−	+
−	+	−	+	−	+
+	−	−	+	−	+
−	−	−	−	+	+

TRUTH TABLES





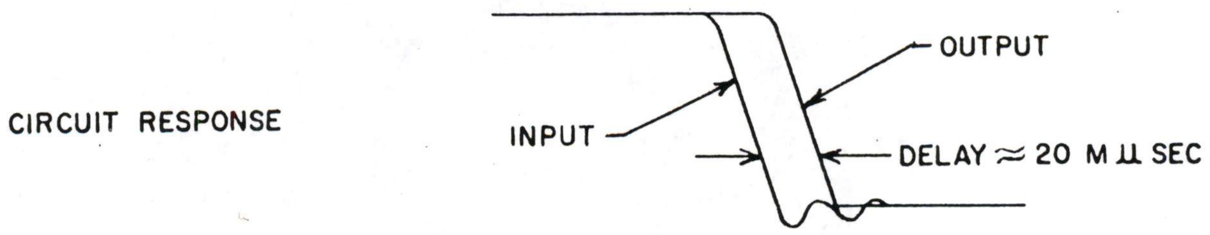
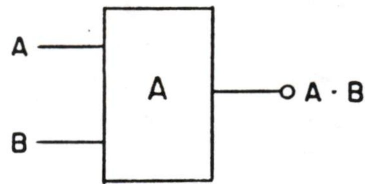
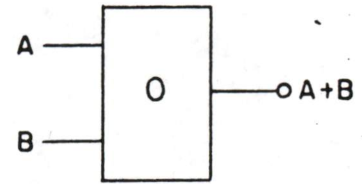


Fig 12: THIRD LEVEL CIRCUIT

CIRCUIT





TRUTH TABLES

AND

A	B	·
+	+	+
−	+	−
+	−	−
−	−	−

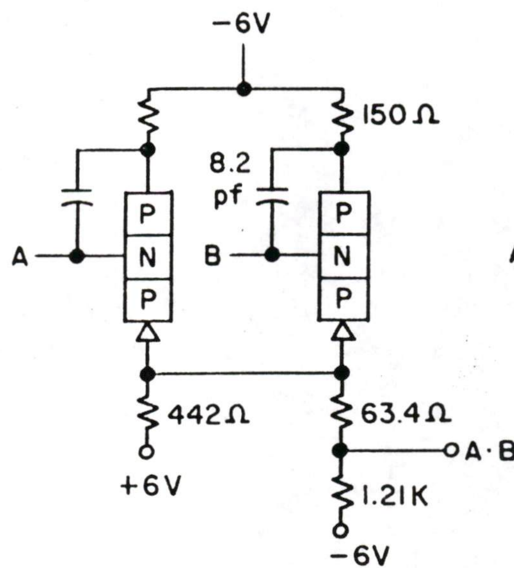
OR

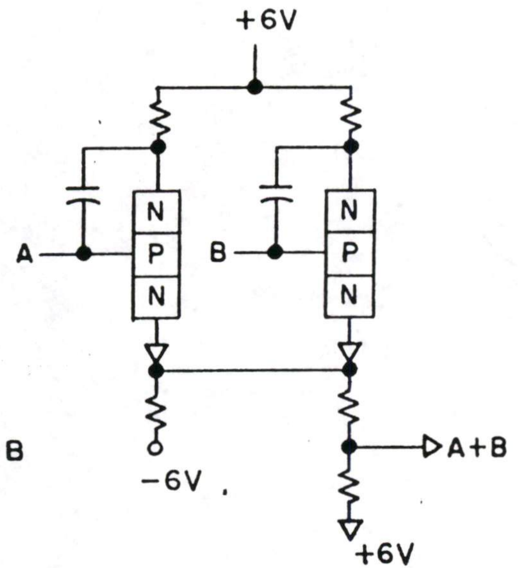
A	B	+
+	+	+
−	+	+
+	−	+
−	−	−

CIRCUIT DIAGRAM

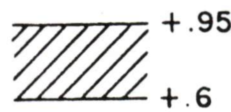
A_N

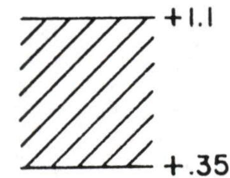
O_N





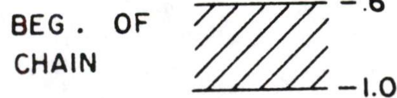
MIN – MAX SIGNAL VOLTAGES

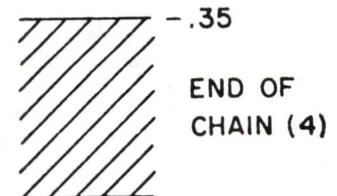




REF —— GND

REF —— GND





CIRCUIT RESPONSE

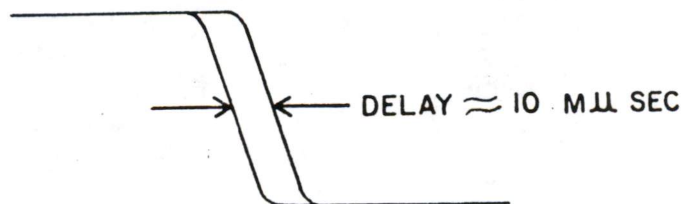


Fig 13: EMITTER FOLLOWER LOGIC

File Copy

EASTERN JOINT COMPUTER CONFERENCE
Boston, December 2, 1959, 2 P.M.

THE MULTI-SEQUENCE COMPUTER AS A COMMUNICATIONS TOOL

BY

J. N. Ackley
Associate Director of Engineering Department

INTERNATIONAL ELECTRIC CORPORATION
Paramus, New Jersey

Formerly a Consultant to
ITT Laboratories
Nutley, New Jersey

THE MULTI-SEQUENCE COMPUTER AS A COMMUNICATIONS TOOL

ABSTRACT

This paper describes possible applications, as a communications tool, of a multi-sequence computer in which more than one sequence or program operates independently, time-sharing the central processing unit. The computer is made to time-share on an on-demand basis between all of the input and output devices. Control and buffering is provided by the central processing unit. A multi-sequence computer, which permits integration of a multiplicity of input and output devices economically, becomes a very rapid and economical message switching center by connecting the communication lines as the input and output devices.

THE MULTI-SEQUENCE COMPUTER AS A COMMUNICATIONS TOOL

This is a report on the merging of two fields: communication switching and computers. Recent advances in the computer art make it possible to satisfy the ever increasing communication switching requirements brought on, in part, by computers themselves employed in centralized data processing systems.

The present record communication systems have significant delays which are not primarily caused by the transmission times but by the time required for the operations in the communication message switching centers.

In the past, most communication has been from human to human. Communication systems have become more complex and automated to meet the ever increasing needs of commercial and military activities. We are faced with a revolution. Increasingly, communication will be between humans and machines and between machines and machines. This transition will place more stringent requirements on accuracy, reliability, and speed.

Present day electromechanical switching centers are limited in their speed of operation due to their electromechanical nature and due to limitations of the transmission means, namely, teletype. Little error detection and correction capability is presently found in these systems.

Centralized data processing requires the error free transmission of large volumes of data to a central point. Usually, the communication with machines or between machines involves little redundancy such as that found in plain English. The computer, although it can make validity checks on the data it receives, cannot fill in missing letters or words as a human can.

Since the present electromechanical systems are special purpose devices, all messages routed through the system must adhere to a very rigid format. This leads to difficulty when trying to integrate data gathering devices and different types of computers with different codes and formats. Military command control systems, especially, require data inputs from varied sources.

Stored program techniques could solve many of these communications problems. A programmed switching center could be programmed to provide the error checking and error correction procedures as required. It could be programmed to translate from one code to another, indeed, perhaps from one language to another. Various speeds and code structures could easily be accommodated.

In addition to improving the features already presently found in some switching centers, stored program techniques could be used to implement features which are not practicable with electromechanical or special purpose switching systems. A programmed switching center could generate, and receive and interpret service messages to and from other machines or human operators. It could test and monitor all of the communication links and reroute messages if necessary to avoid inoperative links. One of the biggest advantages of a programmed switching center is its ability to be reprogrammed to account for changes in operational procedures, in routes and changes of equipment.

One of the most important considerations in employing computer techniques to the communications switching problem is the large number of input and output channels required. Also, all channels must operate simultaneously and independently. Military practice requires that each message be forwarded as far as possible whenever the communication link is available. Thus, the communication switching center must accept a message on each communication line whenever the subscriber wishes to transmit.

For many years the bottleneck on efficient use of computers and on the application of computers to real-time systems has been the problem of integrating input-output devices. Most input-output schemes utilized to present, have involved a large amount of equipment external to the central computer to provide for buffering and control.

Now that the versatility of the high speed random access core memory has been fully appreciated, an almost limitless number of schemes is possible. Indeed, a single computer may use several schemes to integrate various input-output devices.

In order to discuss the various schemes and compare their advantages and disadvantages, it is necessary to establish a classification system. There are four parameters, as shown in Figure 1, which characterize an input-output scheme. These are:

Assembly
Buffer
Transfer
Control

<u>Assembly</u> refers to the process of packaging or unpackaging information into definite size units. In order to characterize the assembly (or the disassembly) process, one must specify how and where the transformation takes place among bits, characters, words, records, and files.

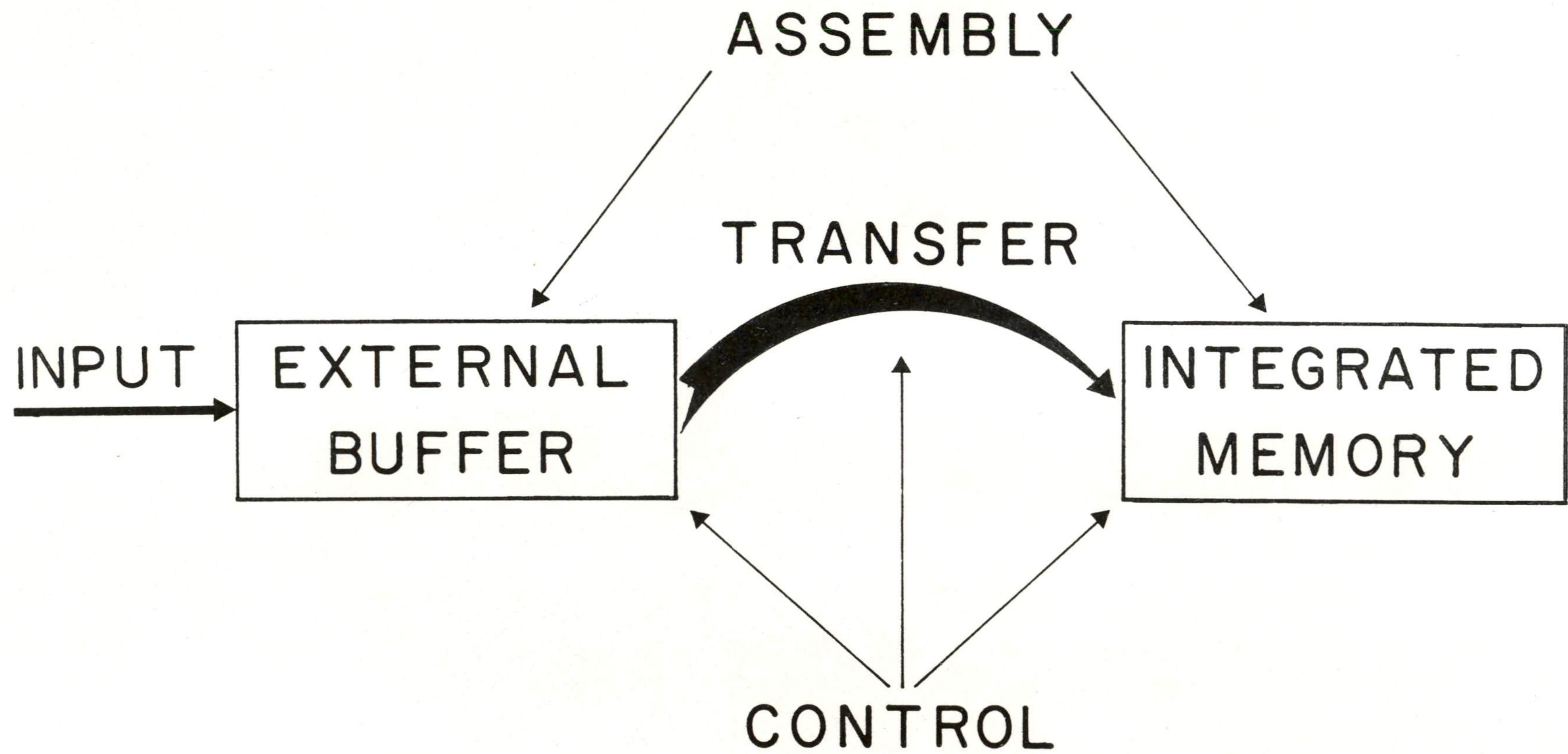


FIG. 1
INPUT- OUTPUT CLASSIFICATION

Buffer refers to the unit external to the central computer which holds information until it can be transferred. The buffer may also be involved in the assembly process. It is characterized by its capacity, access time, and assembly features.

Transfer refers to the process of exchanging information with the integrated (addressable) memory of the central computer. The transfer is characterized by the number of bits transferred in parallel.

Control refers to the process which determines the sequence of operations of an input-output channel. In order to carry out the control function, control words must be supplied to a control device. Typical control words that are usually involved in input-output are:

Selection code
Number of units of information to be transferred
Address in integrated memory at which the transfer is to begin
Address in external device at which the transfer is to begin
Location of next control word

Some or all of these control words are required for any input-output transfer. The device which utilizes these control words can be the central computer or a separate input-output control device. This device provides for the proper sequence of operations.

In designing a system, engineering compromises must be made. Many combinations and permutations of the above factors can be made. Each permutation will have certain advantages and disadvantages which must be weighed against the system application. For example, to take the extremes, a system can be designed like the IBM 709 system which has an external control device (the Data Synchronizer). This device has a register for buffering and additional registers for storing the control words. This system requires only a core memory cycle for a transfer and thus, takes little time away from the central computer during input-output transfers. However, this scheme requires extensive hardware for the external control device.

On the other hand, a system may be designed to store all of the control words in integrated memory and the central computer could supply the control. Such a system would require a minimum of equipment external to the central computer for control of the input-output transfers. However, in a system application which requires a large volume of input-output a large percentage of the capacity of the machine would be tied up in the input-output transfers.

A typical communications switching center may have 50 to 100 two-way communication lines. In order to have a computer perform the functions of a store and forward switching center, it must have a corresponding number of input and a corresponding number of output channels. In addition, it must have a sufficient processing capacity to determine distribution and optimum routing, priority, and perform validity and parity checks if required. Not only must it provide for these large number of input and output channels, but all channels must be capable of operating simultaneously. Neither of the extreme systems described above are satisfactory for this type of operation. However, both of these schemes can be made practical for this type of application by multiplexing as shown in Table I. For example, the external control device can be multiplexed, as it is in the AN/FSQ-7A low speed channel, so that it may service a number of input and output channels simultaneously. However, this also requires that the central computer have a number of independent memory banks so that the transfer of the control words from storage to the external control device and back to integrated memory on each input or output transfer will not saturate the system.

By multiplexing the central computer, it can be used to handle a large number of communication lines. A central computer can be made to time share over a large number of channels by utilizing the multisequence configuration.[1] This configuration provides a number of program counters and a system for switching from one program counter to another, in response to external stimuli.

In order to select the proper scheme, a detailed study must be made of the particular communication switching center requirements. The external control scheme gives a higher capacity under certain conditions, but it requires more equipment. The study of the communication requirements must determine the ratio to the number of core memory cycles required for input and output transfers. If this ratio is sufficiently high, then the central computer control scheme is to be preferred, since relieving the central computer of the burden of input-output transfers would free only a relatively small percentage of the system capacity. On the other hand, if this ratio is low, then the external control scheme is to be preferred since a low ratio implies that the main function is that of input-out transfers.

---

[1] W. A. Clark "The Lincoln TX-2 Computer Development".
J. W. Forgie "The Lincoln TX-2 Input-Output Systems".
Proceedings of the W.J.C.C., 1957.

TABLE I
IN-OUT SYSTEMS FOR COMMUNICATION SWITCHING

ASSEMBLY	*BUFFER*	*TRANSFER*	*CONTROL*
1. BITS TO CHARACTER EXTERNAL	CHARACTER	CHARACTER BREAK-IN	EXTERNAL – MULTIPLEXED INTERRUPT OR MULTISEQUENCE FOR SPECIAL CASES.
2. BITS TO CHARACTER EXTERNAL	CHARACTER	CHARACTER PROGRAMMED	CENTRAL COMPUTER – MULTISEQUENCE

In a system which requires utmost accuracy achieved by the use of redundancy checks, and which must automatically route and reroute messages to account for communications outages and supply other functions such as code translation; this ratio may be in the order of 2 or 3 to 1. Thus, only a 33 to 50 percent increase of capacity is the maximum that could be expected by utilizing an external control device and multiple access integrated memory - - neglecting memory reference conflicts.

Another significant factor which affects total capacity of the system is the assembly process and the capacity of the external buffer. Since most communication is based upon characters of five, six, seven or eight bits, it is desirable to handle the assembly from bit to character externally.

Systems have been proposed utilizing only a single bit external buffer, with the assembly from bit to character to word to message, in integrated memory. However, the capacity is severely reduced since now a transfer must take place on each and every bit of a message. Increasing the size of the external buffer on the other hand increases the capacity, but at the same time increases the amount of equipment. The maximum size buffer to be considered, of course, is equal to that of the word length. Therefore a compromise must be reached between the amount of equipment in the external buffers and the capacity of the system. This compromise is partly influenced by the number of lines to be terminated, by the speed of the circuitry available to the programmed element, and it is also influenced somewhat by the nature of the transmission system.

Those of you who have been through a computer development program will immediately ask, of course, "Is it necessary to design a special computer for the communication switching system"? The question is the most generally phrased, "Can a general purpose computer be utilized?"

This question is not easily answered. The answer must be based on a thorough study of the communication system application. Surely, if the only job of the programmed element is that of communication switching, then there is no need for floating point or even multiply or divide instructions in the instruction vocabulary. Thus, simplifications can be made in the design of a programmed element which is to be used solely for communication. The specialized programmed element can be optimized for communications.

On the other hand, if the computer is to be used for both communications and computations, it may be desirable to design an external control device which can be adapted to a general purpose computer. However, the range of applicability of the general purpose computer

seems rather limited. If the computational requirements are very great, it may be profitable to employ the special purpose programmed element for the communications switching center and also to act as an input-output processor for a much larger data processor. This philosophy is illustrated in the STRETCH and LARC computers where a simple input-output processor is used to relieve the complicated, high speed, data processor of the simple routine tasks associated with input and output editing. The in-out processor can also be used to operate or schedule the operation of the larger data processing system to achieve a much higher utilization of the data processor than would be possible with a human operator.

In selecting a computer or designing a computer for communication system switching center application, consideration must be given to the peak and average traffic rates. Generally, the utilization factor of communication lines is approximately 0.1 to 0.2. This surprisingly low figure is justified because of the queueing problems which would ensue with a very high utilization of any communication link. Assuming a Poisson distribution of message arrivals, a utilization factor approaching unity would result in a queue approaching infinity. Therefore, to minimize the queueing problems and to assure rapid transmission of the message, the utilization factor is desirably kept approximately at 0.2.

The message processing capacity of the switching center must also be designed to exceed the average traffic load in order to eliminate excessive queues of messages awaiting processing.

The use of a multisequence computer for a communication message switching center permits the termination of a number of lines which would saturate the computer with input transfers if all lines were operating at full capacity all of the time. The internal and output processes can be assigned a lower priority in the multi-sequence scheme and these operations suspended until the input peak passes. The probability, of course, of such an occurance is extremely small. Since it also is extremely rare that many of the communication lines would be simultaneously busy with input traffic, additional lines may be terminated with the probability that the switching center would lose an input character. In a completely automated system, this would be caught by the error detection and correction system and cause only a slight delay in transmission.

The amount of excess message processing capacity required to reduce the processing queue to satisfactory proportions depends upon the delays permitted and upon the priority structure of the messages.

A programmed Traffic Control Center is being currently fabricated at ITT Laboratories. This Traffic Control Center which utilizes the multisequence technique was undertaken as a study project in 1957 at the Laboratories and was then proposed by this author as the solution to the communication message switching problem of the SAC Control System, Project 465L, in July 1958.

The system design resulted in a computer with several distinguishing design characteristics which make it peculiarly efficient in handling communications. The central computer is multiplexed by use of the multisequence technique. Separate memory units are provided to store 256 program counters and 256 index registers. One index register is associated with each sequence, thus, in essence, 256 separate sequences may time-share the central computer. Each communication line is terminated by a simple character buffer. Each buffer has associated with it a service request flip-flop. As data becomes available, the service request flip-flop is set and competes for time on the central processing unit. Each instruction has provision for a break-bit or a dismiss-bit as shown in Figure 2 and when set indicates that the present sequence may be interrupted in favor of a higher priority sequence or that the present sequence may be dismissed in which case the service request flip-flop is cleared and the sequence remaining with the highest priority is activated.

To take advantage of statistical averaging of the inputs and outputs, an on-demand scanner shown in Figure 3 does not operate on a fixed cycle, but service is requested immediately if no other channels are busy at the same time. In order to incorporate devices of different speeds and to provide an orderly procedure for servicing requests, if more than one request for service should arrive at a time, the service request scanner operates on a priority basis. A strobe signal proceeds from one service request flip-flop to the next until the first one which is set is encountered, at which point the strobe signal is routed to the coder.

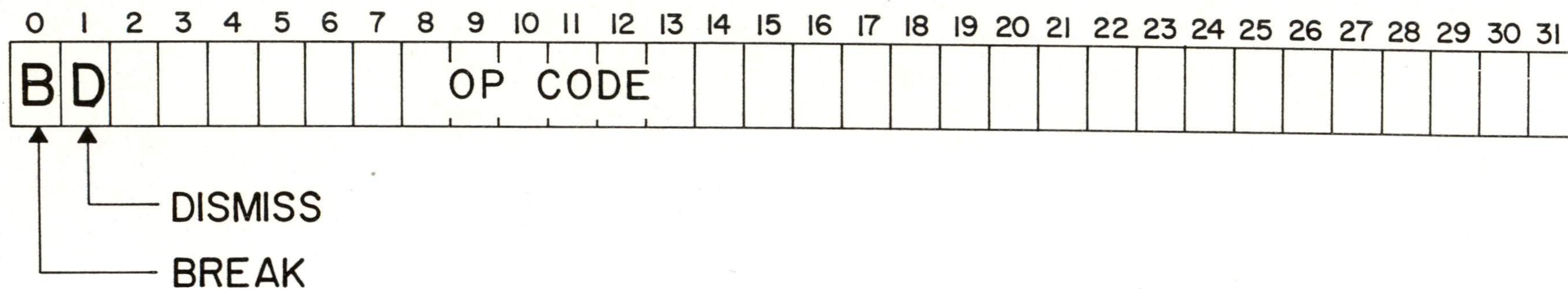


FIG. 2

INSTRUCTION WORD

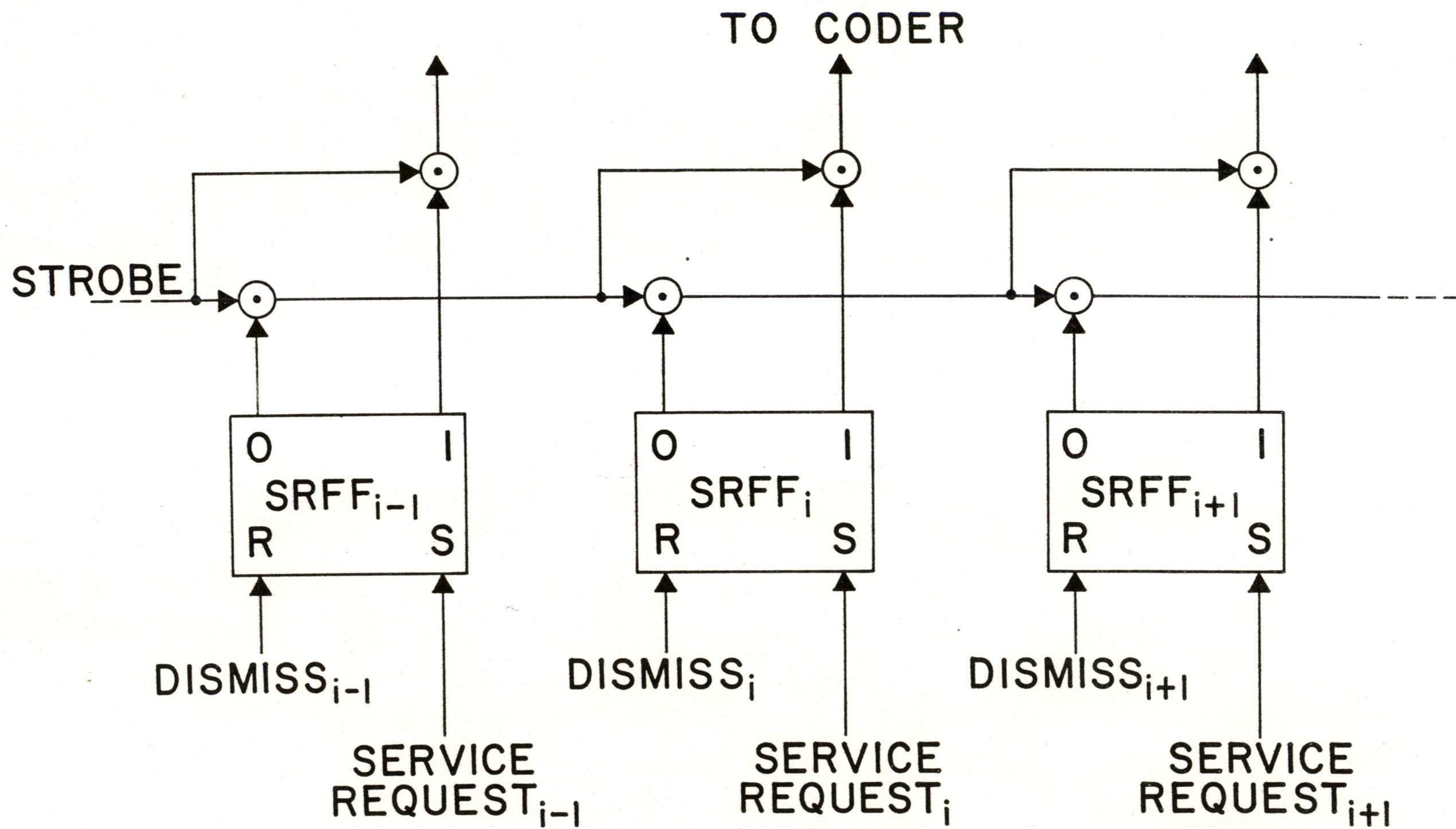


FIG. 3

SERVICE REQUEST SCANNER

Figure 4 shows a typical input channel. When the strobe signal arrives at the coder, it is converted into binary code. If the instruction that the central computer is executing contains a break bit, the output of the coder will be compared with the number in the sequence register. If they are not equal, a sequence of higher priority number has requested service. Notice that the number in the sequence register operates a switch which selects the particular external device to be used at any given time and routes control signals to the external device, such as the dismiss signal. The dismiss signal is issued whenever the present sequence has completed all operations necessary to answer the service request. This signal clears the service request flip-flop and prepares it for receiving the next request for service from the external device.

Figure 5 shows the registers involved in changing sequences. If the number from the coder is found not equal to the number in the sequence register and the current instruction has a break or dismiss bit, a change of sequence is called for. Since the control must return eventually to the old sequence whenever that particular channel requests service again, the program counter must be stored for future reference. Therefore, the first operation is a transfer of the sequence number from the sequence register to the program counter memory address register. The program counter is then stored at the location so specified. After the old program counter has been stored, the new sequence number is read from the coder into the sequence register and then to the program counter memory address register. The stored program counter stored at this location is now placed in the program counter and the computer continues to take instructions specified by the program counter.

In order that each sequence may have its own index register, the contents of the sequence register are transferred into the index memory address register and the contents of the location so specified are used for the indexing operation. Since the program counter memory and the index memory are independent units and can operate concurrently with the main integrated memory of the computer, changes of sequence can take place without any loss of time, that is, an instruction with a break or dismiss in one sequence may be immediately followed by an instruction in some other sequence.

Note that if the contents of the accumulator were stored along with the contents of the program counter, we would truely have a multiplexed computer. However, this feature was not necessary for the communication switching center, but care must be taken during programming in placing break and dismiss bits only at those points in a sequence where the contents of the arithmetic unit are immaterial.

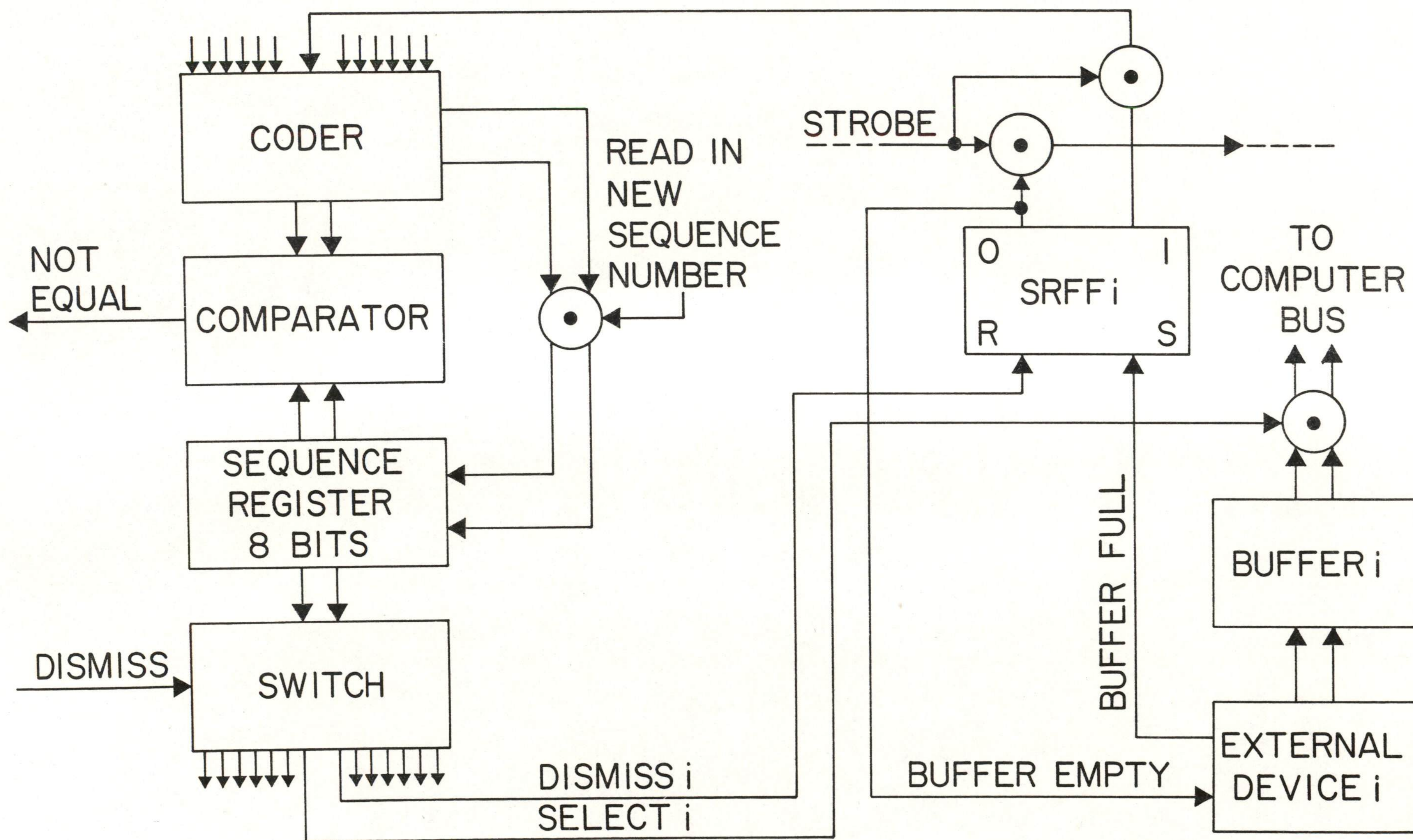


FIG. 4

TYPICAL INPUT CHANNEL

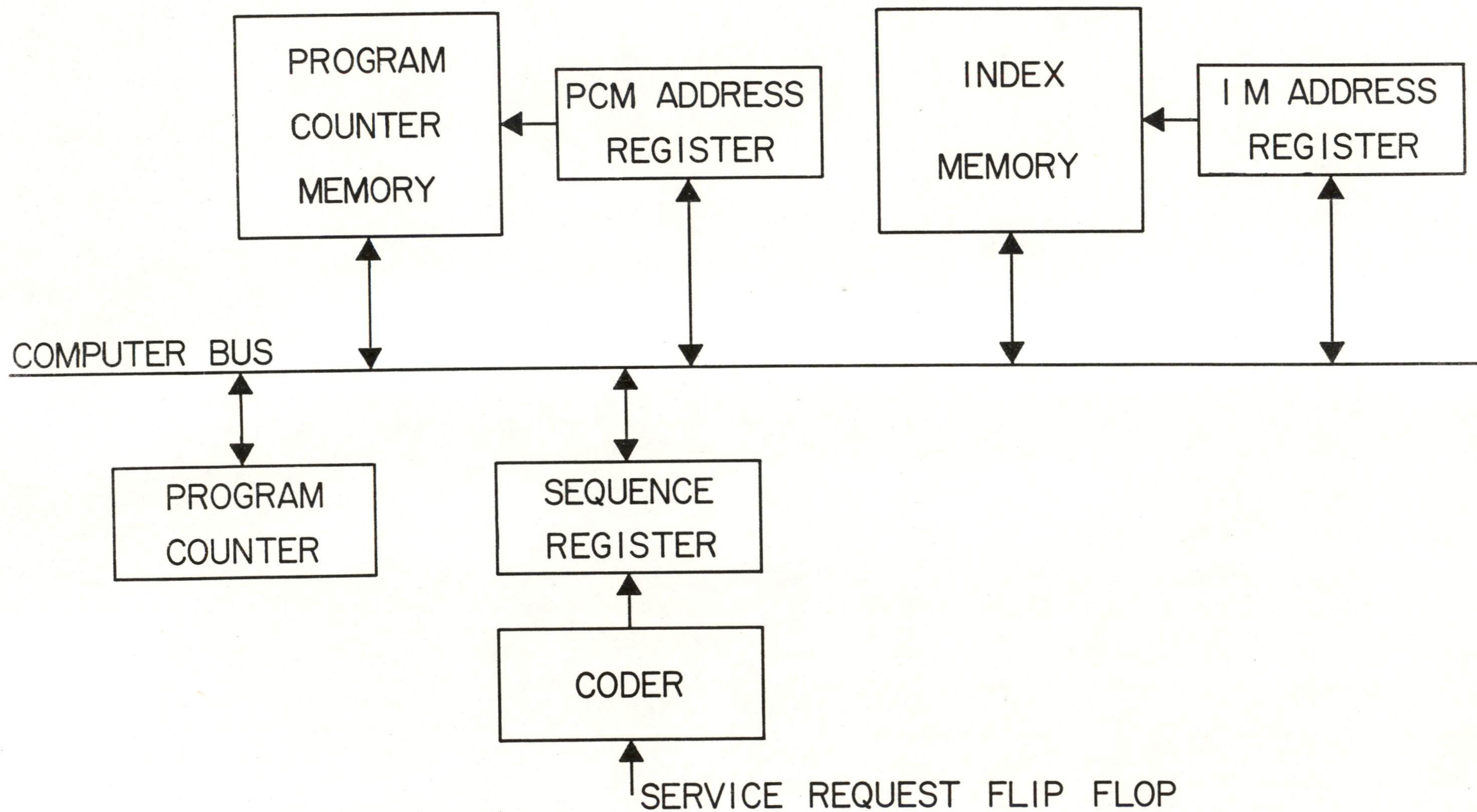


FIG. 5

REGISTERS INVOLVED IN CHANGING SEQUENCES

Since most communication and data processing systems have either five, six, seven or eight bit characters, most instructions are capable of operating in two modes. In the word mode, the operation applies to the entire computer word of 32 bits. In a character mode a single 8 bit character is referenced. Since there are four eight-bit characters in a 32 bit word, an addressing scheme as shown in Figure 6 was devised which permits addressing of consecutive characters in memory convenient. The two least significant bits of the address part of an instruction refer to a character position within a word. In the word mode, these bits are ignored. The instruction vocabulary contains a liberal quantity of logical instructions, but does not contain multiply or divide or any other numerical operations other than add and subtract.

Most instructions have a bit to indicate the repeat mode as shown in Figure 7. This permits most input-output transfers to be accomplished with a single instruction requiring two core memory cycles. As the characters are transferred in from the external buffer, a check on parity and a check for a control character is performed by common circuitry. If either condition prevails, it is indicated to the program by means of a skip, otherwise the input-output transfers are handled by a single instruction in the repeat mode with the dismiss-bit set, which causes the character to be transferred; the index register to be incremented; and the present sequence dismissed in favor of some other sequence. The message processing sequence has the lowest priority and processes messages between transfers.

Figure 8 shows the comparison of the percentage of utilization of the Traffic Control Center versus the average processing delays. This truely represents an advance in the communication message switching art. Present switching systems delay each message by many minutes. Now, the delay is measured in milliseconds.

Most comprehensive communication systems require both circuit and message switching. The programmed switching center can effect digital circuit switching by associating, by program means, a particular input channel with a particular output channel. The delay under these conditions would be only a few microseconds and would be insignificant compared to the delays in long transmission lines. However, since the traffic is just coming in and going out without any processing, the capacity of the programmed element is used unnecessarily. In a communications system which requires a large amount of circuit switching, it will be preferable to terminate all communication lines in an electronic cross-bar switch, as shown in Figure 9 which has many trunks to the programmed element and is under the control of the programmed element. This configuration would permit both message and circuit switching on an intermixed basis and permit changing from one type of service to another by simple indications to the stored program.

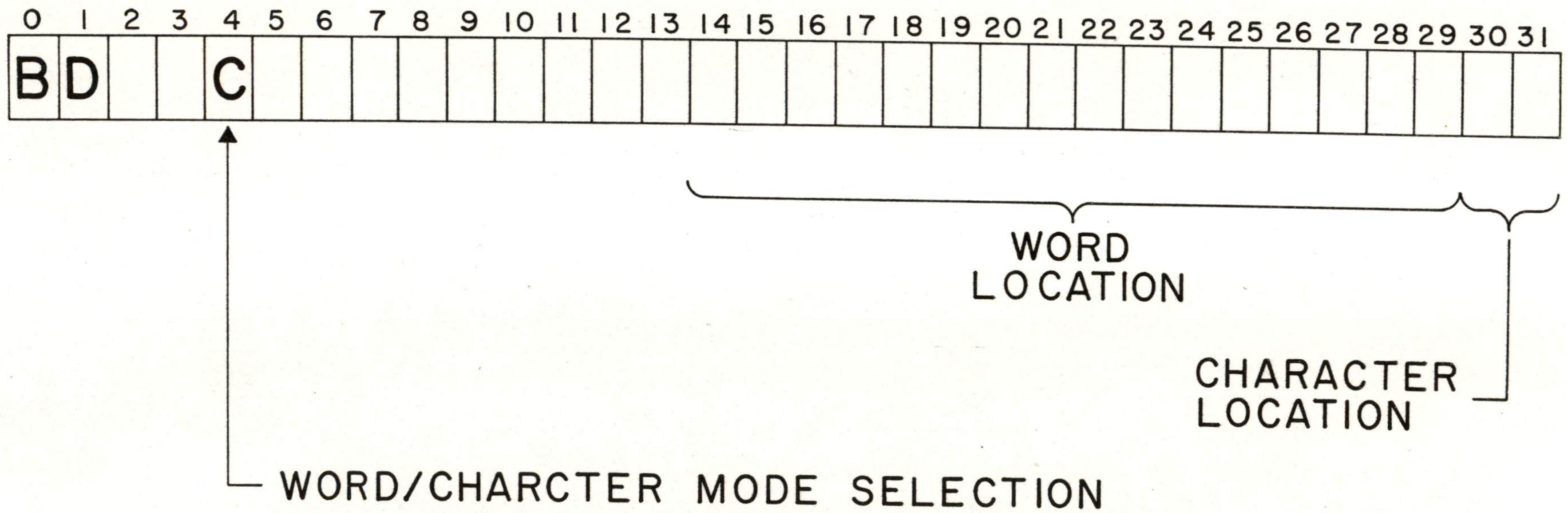


FIG. 6
INSTRUCTION WORD

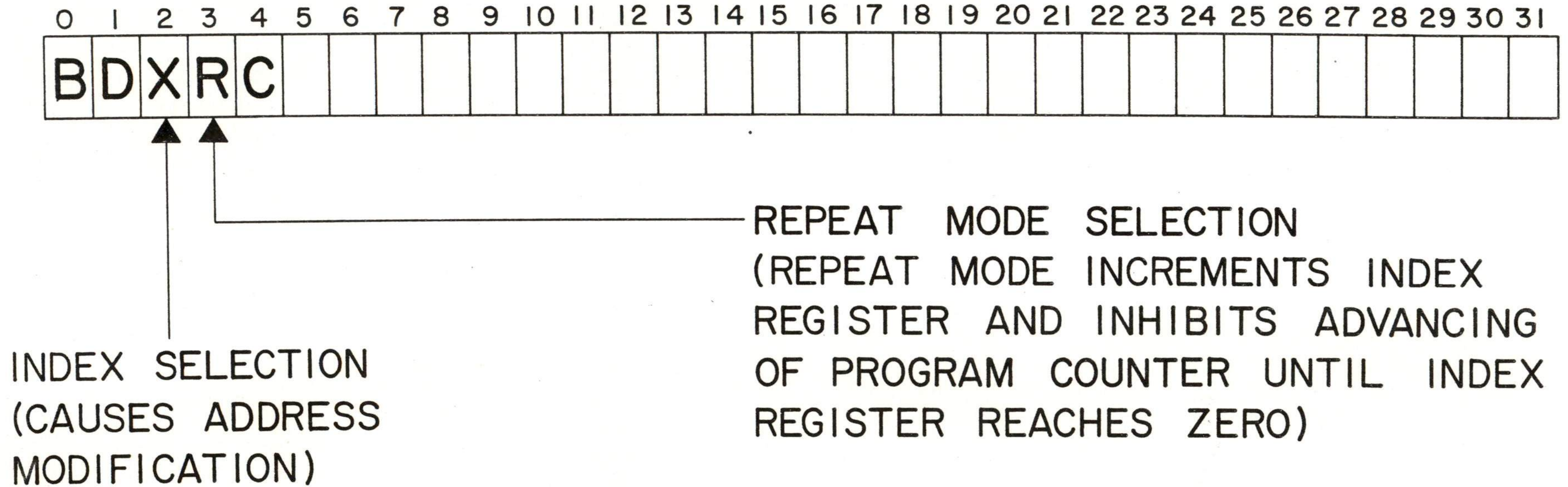


FIG. 7

INSTRUCTION WORD

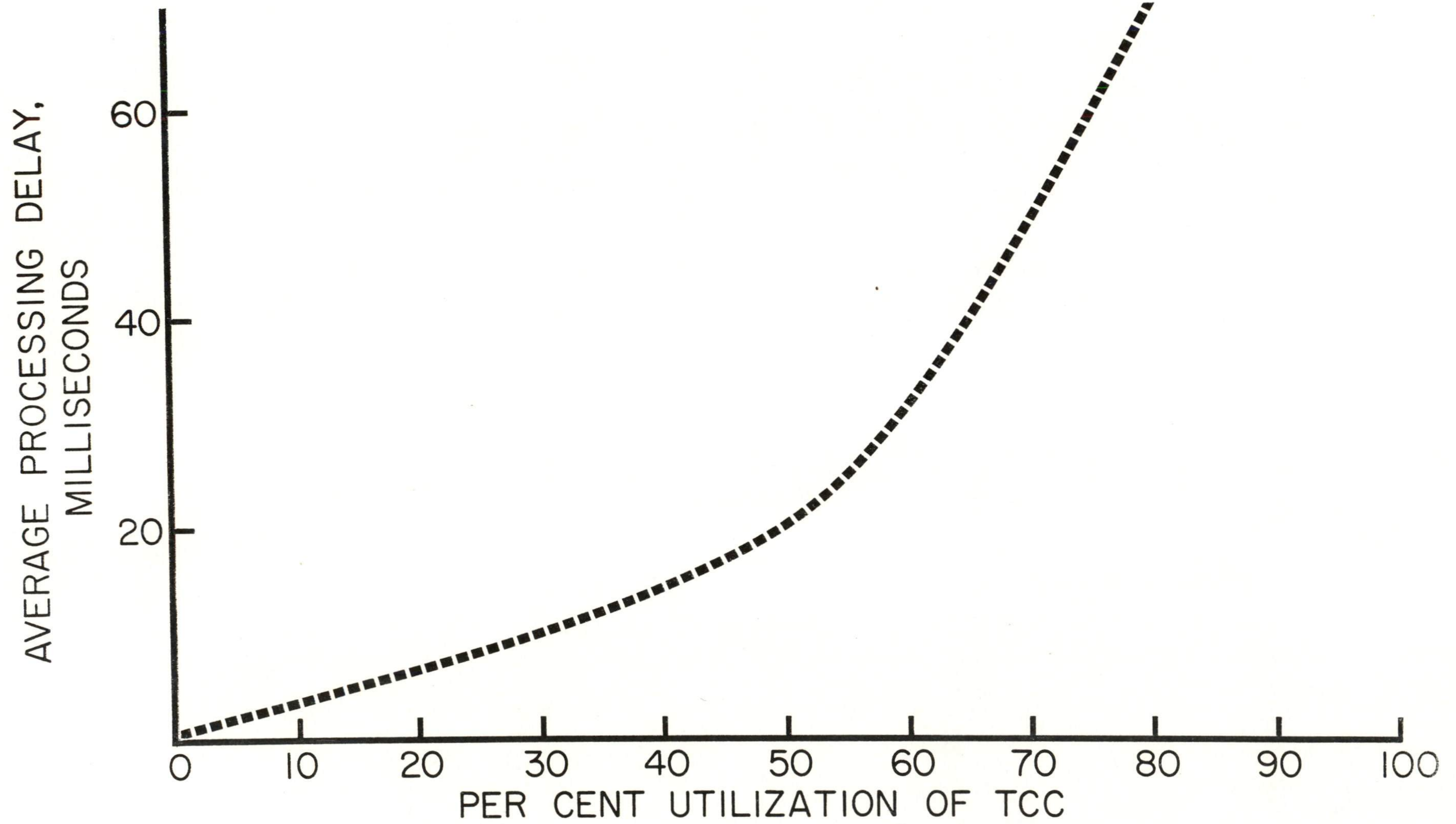


FIG. 8 COMPARISON OF PER CENT UTILIZATION (OF TCC) VS. AVERAGE PROCESSING DELAY

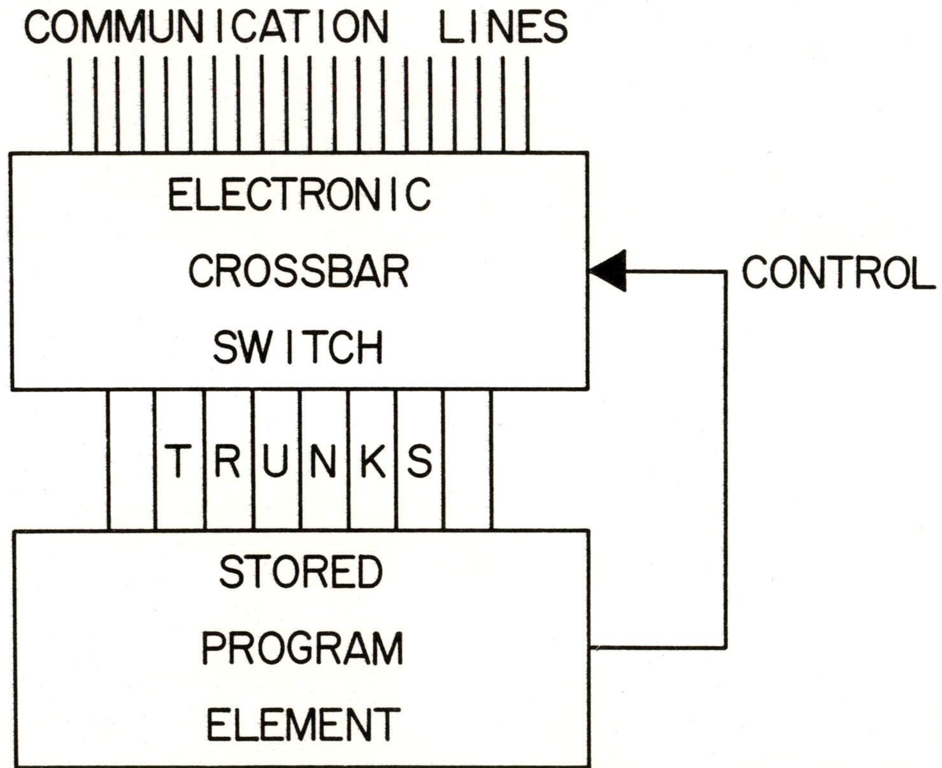


FIG. 9

THE ULTIMATE COMMUNICATIONS SWITCHING CENTER

BIOGRAPHY

JOHN N. ACKLEY, Associate Director, Engineering Department, International Electric Corporation, an ITT associate, responsible for advance planning and research, was a digital systems consultant principally to ITT Laboratories from 1956 to 1958, a staff member of Digital Computer Laboratory (Division 6), Lincoln Laboratory, Massachusetts Institute of Technology from 1952 to 1955, a National Science Foundation Fellow at M.I.T. in 1955, and a Ramo-Wooldridge Fellow at M.I.T. in 1956.

He served in the U. S. Army Signal Corps during the period 1946-1952. He received the BS in EE, MS in EE, and Electrical Engineer degrees from M.I.T.

File Copy

International Electric Corporation

Paramus, New Jersey

COlfax 2-6800

November 11, 1959

Mr. Harlan E. Anderson
Digital Equipment Corporation
Maynard, Mass.

Dear Harlan:

It was a pleasure to run into you at Charlie's cocktail party and congratulations on your new computer.

Enclosed are four copies of my talk for the Eastern Joint Computer Conference. The original drawings are available on request if the multilith copies are not satisfactory for reproduction.

I am looking forward to seeing you and the rest of the crew from the Digital Equipment Corporation at the computer conference.

Sincerely yours,

John N. Ackley

JNA:jb

Enclosures (4)