LA-1725

THE MANIAC

LA-1725

THE MANIAC

PREFACE

The construction program on the MANIAC was started in the summer of 1949 and the computer was completely tested in March, 1952. The group of engineers is under the direction of J. Richardson and consisted, at various times, of W. Orvedahl, E. Klein, H. Demuth, T. Gardiner, H. Parsons, R. Merwin, and J. Breese. In addition, V. Gafke and J. Caulfield provided considerable assistance. Since its completion, solutions to many numerical problems have been computed.

There are several phases to the solution of a problem by an electronic computer. First, there is the formulation of the problem itself by the mathematician or theoretical physicist. Second, this is followed by the detailed preparation of the problem by the programmer for the specific computer. Finally, there is the actual running of the problem on the computer. The present work is primarily an attempt to discuss in some detail the last two stages.

The volume consists of six chapters. Chapter I, Introduction, describes some of the general features of the computer and defines the field of activity associated with it. The treatment is intentionally brief. The remaining chapters are devoted to an elaboration of the salient points.

Chapter II, Coding and Flow Diagrams, is the "raison d'etre" of the opus. Beginning with some elementary problems, it gradually takes the reader through a coding preparation of some complex exercises. The elements of a flow diagram are discussed.

Chapter III, Binary Arithmetic, discusses the various arithmetic operations in terms of the binary system. By the time the reader finishes this part, it is hoped he will regard the binary system as the "natural" one for arithmetic.

Chapter IV, The Computer, is concerned with a simplified discussion of the various components. The objective here is that some knowledge of the engineering side of a computer is very useful to personnel running problems on it. Aiding in the detection of malfunctions and in the localization of them, the programmer helps the engineer in maintaining high performance of the computer.

Chapter V, Descriptive Coding and Subroutines, describes the methods of descriptive coding the the use of the computer itself to aid the programmer in the preparation of problem codes. The discussion of subroutines finds a natural place here.

Chapter VI, Operating Procedures, essentially summarizes some of the material of the earlier sections and describes systematically the steps involved in automatic computational processes, including "which buttons on the computer to press when".

Finally, an Appendix is included. It contains some optional and, we hope, useful material.

John B. Jackson
N. Metropolis

Los Alamos, New Mexico
December 15, 1951.

Acknowledgements: To Mary Boswell, whose patience was excelled only by Job, for typing (and re-typing) the manuscript; to Jean Cornell for converting our sketches into neat drawings and figures; and to all members of the MANIAC group who deluged us with criticisms, especially Mark Wells.

J. B. J.
N. M.

Los Alamos, New Mexico
December 15, 1951.

Revised: July 16, 1954

I. INTRODUCTION

We shall give first a brief description of the general features and characteristics of the computer which has been constructed here.

(i) It is a general purpose computer in contradistinction to a special purpose type. Its design engenders adequate flexibility to handle a wide variety of mathematical problems. The special purpose type may be much simpler in design and more direct in its application to a particular type of problem, but it has its obvious limitations. We do not discuss it further.

(ii) It is a digital, rather than an analogue, computer. Computers have been built which use various analogy devices that correspond to a continuous variable representation. In such analogy computers, numerical information is expressed as measurements of some physical quantity. Among other reasons, it may be mentioned that accuracy requirements argue for the digital type.

(iii) It is electronic (vacuum tubes) in character, as opposed to electro-mechanical (relays). Although both methods are sufficiently reliable, the former is many times faster. For the majority of problems, where the number of operations involved is at least in the hundred thousand range, the difference in speed is quite serious.

The four basic arithmetical operations performed are addition, subtraction, multiplication and division. In principle, one might conceive of a simple computer that does only subtraction, and effects the others by repeated application of that fundamental operation. This is not very practical. On the other hand, one might have argued for including other operations in the basic list; e.g., square rooting, as indeed the ENIAC has included. It appears, however, that the frequency of occurrence of any of these does not warrant the added complication in equipment, especially since these more complicated operations can be effected by rather simple iterative procedures based on the four fundamental operations.

Besides these four arithmetical processes, there are included a few operations which are of a purely logical character, but first,

Some Remarks on Arithmetic

The handling of numerical quantities is done in a digital fashion. The binary system is used for the representation of numbers rather than the conventional decimal system. Everyone knows that in the latter system a number is expressed as a sum of powers of ten with individual co-factors 0 to 9; e.g.,

$$47.23 = \underline{4}\cdot 10^{1} + \underline{7}\cdot 10^{0} + \underline{2}\cdot 10^{-1} + \underline{3}\cdot 10^{-2}$$

In a similar fashion a number may be expressed in the binary system by powers of two with co-factors either $\underline{0}$ or $\underline{1}$; e.g.,

$$101.01 = \underline{1}\cdot 2^{2} + \underline{0}\cdot 2^{1} + \underline{1}\cdot 2^{0} + \underline{0}\cdot 2^{-1} + \underline{1}\cdot 2^{-2}$$

As in the decimal system, the binary point separates the terms with positive exponents from those with negative exponents. The standard capacity for handling numbers in the present computer is 39 numerical bigits preceded by a sign bigit. (The word bigit is defined as binary digit.) There is sufficient flexibility to permit rather easy treatment of those cases requiring higher precision.

For the various arithmetical operations in the computer, it is assumed that the binary point lies immediately to the left of the first numerical bigit, so that all numbers lie in the range

$$-1 < x < 1.$$

It may appear at first that this restriction places a considerable additional burden on the preparation of a problem for the computer. Actually, however, it is quite a simple matter to scale numbers to the appropriate size beforehand, such that the result of any operation does not exceed the allowed range. In those instances where it is not possible to provide appropriate scaling factors in advance, one does have recourse to procedures which adjust the sizes of numbers--the so-called floating point routines.

As mentioned above, the first bigit on the left is used to indicate the sign of a number. One possible convention that might be used would be to say that bigit $\underline{0}$ in that location indicates a positive quantity and that a $\underline{1}$ is to be interpreted as a negative sign. However, it is more convenient to do something different in the case of negative numbers.

In the computer, a negative number x is represented by its complement c with respect to 2, namely

$$c = 2 - |x|$$

Since $|x| < 1$,

c will be in the range $1 < c < 2$

so that the "sign" bigit will be 1 in every case of complementation. For positive numbers it will always be 0. For example, suppose

$$x = -.101110101.....011;$$

then $$c = 1.010001010.....101$$

is its representation in the computer. One observes that a very simple method for obtaining the complement of a number with respect 2, is to "reflect" the number, that is, to replace 0 with 1 and conversely, then to add 1 in the extreme right place. Electronically, interchanging 0 and 1 is easily done. As discussed in detail in later sections, a "flip-flop", or "toggle", is an electronic device which has two stable states; it is essentially a twin triode (a standard type of vacuum tube); either one side is in a conducting state (and its tube elements have one set of definite voltages) with the other side non-conducting (cut-off, and its corresponding elements have another set of voltages) or the opposite situation obtains. It is a symmetrical situation. Normally one examines the voltage level at some particular point of the circuit, say the grid voltage of one of the triodes, and assigns one voltage to the bigit 0 and the other to 1. To obtain the complement of a number in a series of such flip-flops, one would merely examine the opposite symmetrical point of the circuit of each flip-flop; since, if a given flip-flop is in a state corresponding to a 1, the other side of the flip-flop would have a voltage level at the corresponding point identified as a 0. Additional circuitry is required to insert a 1 in the extreme right-hand position.

The notion of complement numbers is a very useful one. Subtraction of two numbers can be replaced by addition. This is convenient since the same electronic circuitry designed to effect addition suffices for the subtraction process. Instead of performing d = (a-b) by direct subtraction techniques, one may add to a the complement of b. That this yields the correct difference can be seen from the following:

Assume a, $b > 0$.

$$a + (2-b) = 2 + (a-b) = 10. + (a-b)$$

in binary form. If $a > b$, and since both a and b have absolute magnitudes less than unity, the difference (a-b) is positive and less than unity. The co-factor 1 of 2^1 does not appear in the computer, the capacity of the computer has been exceeded and that bigit is lost. The 0 co-factor of 2^0 does of course appear, and indicates that the difference (a-b) is positive. In the event $a < b$, our answer would be:

$$1 < d = 2 - (b-a) < 2,$$

which is precisely the desired form for a negative difference, namely the complement with respect to 2. Here the co-factor of 2^0 is appropriately a 1. The cases where a and/or b are negative are left as exercises for the curious students.

Principal Components

Although the computer functions as an entity, it is convenient to speak of its various components. These are:

(i)	arithmetic unit
(ii)	memory
(iii)	input-output
(iv)	control

Arithmetic Unit

The arithmetic unit performs the operations of addition, subtraction, multiplication and division in binary fashion. It is also concerned with such auxiliary operations as shifting of a number to the left or right. Finally, it is associated with certain logical operations.

In appearance the arithmetic unit is similar to the one in Princeton. A parallepiped structure of channel aluminum has six panels on each of its two long sides. The outer panels in each case are reserved for control chassis, the middle four are used for the arithmetic unit proper. Three horizontal rows of arithmetical chassis are located on one of the two principal sides. Each chassis contains two registers. The various registers are designated R1, R2,...R6, starting with the lowest. A register is the residence, or temporary storage, of one of the numerical

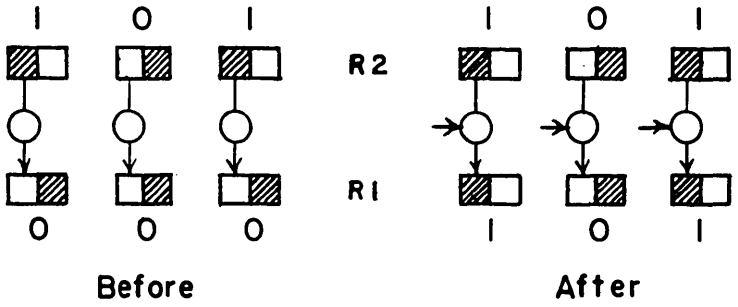
factors in an arithmetical operation. In each such operation three factors occur, so that at first it might be supposed that three registers would suffice. However, the requirement of shifting in multiplication and division necessitates two more. These considerations account for the first five registers; the last, R6, is used exclusively in association with the control and does not participate in any of the basic arithmetical operations, although physically it is located within the arithmetic unit. R1 is the associated register for shifting a number in R2, a principal register. Physically, the pair forms a chassis. Similarly, R3 is associated with the principal register, R4. R5 is a non-shifting register with respect to arithmetic operations.

Before discussing the four basic arithmetical operations, we digress to consider the manner in which a number in one of the two principal registers is shifted. To begin with, a register is an ensemble of 40 "flip-flops", or "toggles", and as mentioned earlier, each flip-flop has two stable states. One of these states represents the binary digit 0 and the other the binary digit 1. The set of flip-flops may then be used to represent a 39-bigit number and its sign.

There exists a variety of methods for electronically transferring information contained in one set of toggles to another. For example, suppose that a given toggle contains a 1 and it is desired to transfer this information to a second toggle. By means of an interconnecting "gate" tube, it is possible (as a result of a voltage change on the gate tube) to set the receiving toggle to a 1, irrespective of its previous state. Another scheme is to have first set the receiving toggle, say to 0, as a separate operation. When the appropriate voltage change is applied to the gate tube, the receiving toggle is set to a 1, otherwise it remains appropriately unchanged. This method is actually the simpler of the two and is the one used. In common parlance we say the receiving flip-flops are "cleared" to 0's and 1's are "gated in". Clearly, 0's and 1's could be interchanged in the preceding statement and provide an alternative scheme.

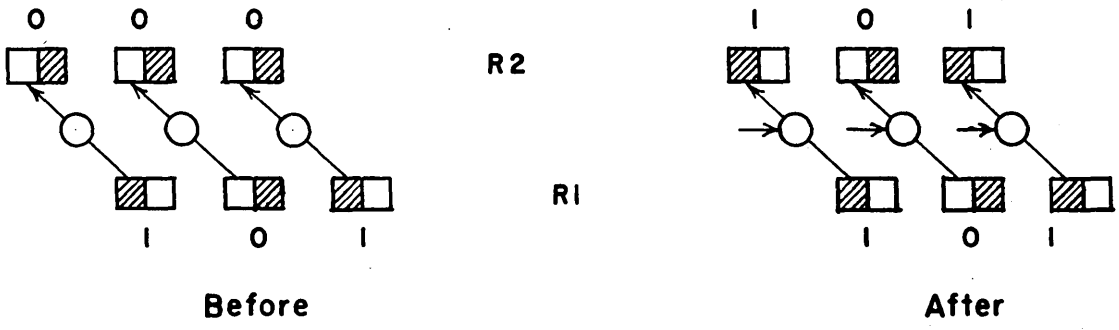
A flip-flop may be symbolically represented as a rectangle in the form of two squares; the shading of one square may be said to correspond to a 0, the shading of the other to a 1. A gate tube is indicated by a circle.

There is a set of gates which connects the flip-flops of R2 to the corresponding ones of R1. These may be shown diagrammatically.



R1 has been previously cleared to 0's. The information in R2 is 101. When an appropriate voltage change is applied to the gate tubes, the first flip-flop of R1 will change its state to represent a 1, the second remains unchanged, and the third behaves like the first. R1 will then have received the information 101.

There is a second set of gates which connects the flip-flops of R1 with the flip-flops of R2 displaced one to the left.

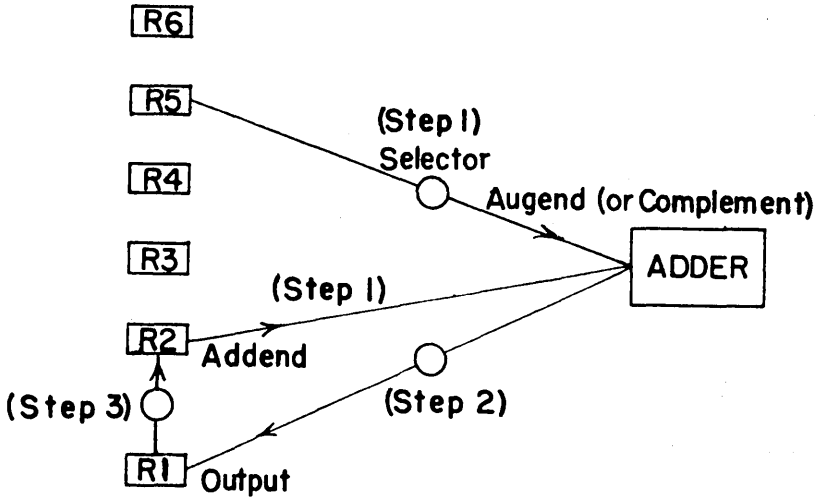


R2 is cleared to 0's. When these gates are opened, the information in R1 is transferred to R2 displaced once to the left. Thus, by these sequences of operations, a number originally in R2 is shifted one place to the left.

Finally, there is a set of "diagonally-right" gates to provide for a shift to the right. Repeated application of the sequence of operations results in a shift by n places. It perhaps should be mentioned that these three sets of gates are unilateral in action and represent all of the interconnections between R1 and R2.

The four basic arithmetical operations are done in terms of simple additions, with shifts where required. Subtraction of a number a is performed by the addition of its complement. Multiplication is done by the detection of the successive bigits of the multiplier, beginning with the rightmost bigit. If the bigit is a 1, an addition of the multiplicand to the partial product is performed followed by a shift of the partial product one place to the right. A 0 multiplier bigit merely shifts the partial product to the right by one, and the next multiplier bigit is examined. For division, the so-called "non-restoring" scheme is used. The complement of the divisor is added to the partial remainder if the signs of the divisor and partial remainder agree; if the signs disagree, the divisor is added directly. A 0 is indicated for the corresponding quotient bigit in the first case, and a 1 for the latter. Strictly speaking, -1 and not 0 is the appropriate bigit. But -1 is indeed very inconvenient to represent in the computer. As von Neumann first pointed out, the pseudo-quotient obtained in this way is very simply related to the true quotient. We shall go into details later.

The adder proper is physically located on the side opposite the registers, and consists of two rows of chassis. One of the two inputs is directly from the register R2. The second input is from R5. Here, however, a choice is made between the number itself or its complement, corresponding to the operation of addition or subtraction. The output of the adder is transferred by means of a set of gates to R1. R2 is then cleared and the sum transferred from R1 to R2. Symbolically,



To recapitulate, the addition process (or subtraction) involves adding to the number in R2 the number (or its complement) in R5. The sum appears finally in R2. The fact that the sum replaces one of the terms is very convenient for the multiplication and division processes, where the sum is the partial product or the partial remainder, respectively. The multiplicand or the divisor resides accordingly in R5.

In the multiplication process the multiplier factor is in R4 and the multiplicand is in R5. R2 is cleared initially. The 39th flip-flop of R4 is examined. If it is a 1, an addition is ordered and the first partial product is formed in R2. (In this first step, the trivial sum of the multiplicand and 0's is done.) The multiplier is now shifted one place to the right, thus placing the next digit to be examined in the end flip-flop of R4. Simultaneously, the partial product in R2 is also shifted one place to the right. In the event that the first digit is a 0, the addition of course is not done but the shifting in both R2 and R4 does take place. It will be noted that the multiplier factor is gradually disappearing in R4. It is convenient, therefore,

to insert the bigits of the partial product that would otherwise be lost as a result of the right shift in R2, into the leftmost flip-flop of R4. Thus the right half of the complete product appears finally in R4 and the significant portion in R2.

For division, the dividend is in R2 and the divisor in R5. A comparison of signs is made and a direct addition is made for unlike signs; for like signs the complement of the divisor is sent to the adder. Accordingly, a 0 or a 1 is introduced into the 39th flip-flop of R4. Both R2 and R4 are shifted one place to the left. The sign of the partial remainder is again compared with that of the divisor and the process repeated 39 times. The quotient appears in R4, and the remainder in R2.

The following short table summarizes the above:

Addition	a + b = Sum
Location	R2 R5 R2
Subtraction	a − b = Difference
	R2 R5 R2
Multiplication	a X b = Product Left + Product Right
	R5 R4 R2 R4
Division	a ÷ b = Quotient + Remainder
	R2 R5 R4 R2

Memory

Thus far we have talked of the various arithmetical operations without indicating how the numbers get to the several registers initially, or where the intermediate results are stored. Nor have we said anything about the location of the sequence of orders associated with a problem. The component of the computer associated with this

activity is described as the memory. Clearly, some of its desired functions are:

(i) to receive and store information from the outside--sequences of instructions as well as initial sets of numbers,

(ii) to transfer numbers upon instruction to the arithmetic unit,

(iii) to receive and retain intermediate results of a calculation until needed at some later stage of the calculation,

(iv) to send instructions as needed to the control,

(v) to transfer the final results to the output mechanism for external consumption.

We distinguish two levels of memory, internal and external. The internal memory is more intimately related to the arithmetic unit and control. It communicates directly with these two units and provides individual numbers and instructions as needed.

Physically, the internal memory is an ensemble of 40 cathode-ray tubes that act in concert, each tube simultaneously providing one bigit of a 40-bigit number upon instruction. The access time, or total time required to transfer a number from the internal memory to the arithmetic unit, is less than ten micro-seconds. The capacity of the internal memory is 1024 forty-bigit numbers; these may be arbitrarily divided between numbers and instructions.

The location or reference in the internal memory of a particular number or instruction is called its address. In our system of instructions there is, associated with each instruction, a single address that refers to a particular number to be called up and operated upon in the arithmetic unit. An instruction consists therefore of a particular operation specified by a group of bigits, together with an address specified by another set of bigits. It turns out that less than 20 bigits are required for each complete instruction, so that it is convenient to place two instructions in one memory location. We shall amplify these remarks in the discussion of the control.

Normally, 40 bigits are used for the representation of a true number. For those cases where sufficient accuracy is obtained from 20 bigits, including sign, there is sufficient flexibility to store

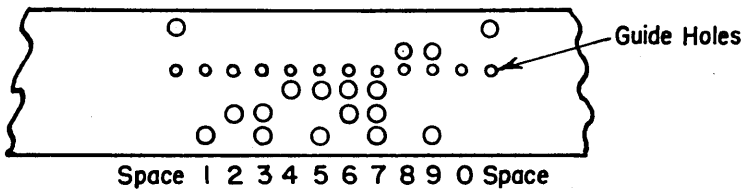
conveniently two 20-bigit numbers in one memory location; separation taking place when needed in the arithmetic unit by shifting.

The external memory is a magnetic drum. It communicates only with the internal memory; therefore, when numbers stored on the magnetic drum are to be used in computation, they are first sent into the electrostatic memory and operated upon from there. The drum has a capacity of 10,000 forty-bigit numbers. Numbers are transferred between the external and internal memory in groups of fifty; hence the addressing of numbers on the drum is by groups of fifty rather than as single numbers. Any group of fifty numbers is stored serially along the circumference of the drum. Such a group of storage is called a drum track, and there are 200 such tracks on the drum. The access time for the drum is 85 milliseconds per block of fifty words.

Input-Output

The set of coded symbols corresponding to the sequence of instructions, together with the set of initial numbers and parameters, is first punched on paper tape with the use of a modified **Flexowriter.** A second tape is then prepared, being punched independently of the first but simultaneously compared with the first; this is merely a checking procedure. The information is then transferred from the verified tape to the internal memory by means of the input device.

The initial set of numbers on the tape is in coded-decimal form; that is, each decimal character is represented by a tetrad of binary digits. For example, the aggregate 1234567890 together with accompanying space symbols would appear on the tape as:



The punched holes correspond to the bigit 1 and unpunched positions to 0. A sequence of such tetrads of binary digits is obviously not

the true binary representation of the corresponding decimal number; e.g.,

decimal number	24
coded decimal	0010 0100
true binary	11000

Consequently, it is first necessary to convert the initial set of coded-decimal numbers into true binaries. But this is a quite simple algorithm which the computer can be directed to perform before entering upon the problem proper. The initial set which must thus be converted is usually quite small compared to the number of numbers the computer handles in the course of the problem, so that the time invested for the conversion is relatively negligible. The same remark applies for the conversion from true binary to coded-decimal representation for the output process; it being still desirable to view answers in decimal notation.

When the desired results are properly converted into coded-decimal notation, they may be directed to the output. The output will simultaneously print the results and punch them on teletype tape. This tape is desirable in the event that the answers are to be reintroduced into the computer.

It should be remarked that beginning with the second problem of any given type it will not again be necessary to manual punch the sequence of instructions. The original tape will be adequate. It is only necessary to punch the new initial numbers and parameters. This portion is usually a small fraction of the total. Finally, it should be noted that the casual observer need never be aware of the fact that internally the computer uses the binary representation for numbers.

Control

The control may be likened to a central nervous system. Its parts spread out physically over the whole computer. It interconnects the various other components and transfers information from one to the other, as well as directs the operations associated with them individually.

Among its various activities, it must:

(i) direct the input component to read information from the teletype tape and transfer it to the internal memory,

(ii) conversely, direct the memory to transfer information to the output tape and printer,

(iii) effect the basic arithmetic operations,

(iv) be able to start at some point in a sequence of orders, extract the first order (from the internal memory), interpret and provide pulses and voltage changes to the components concerned so as to execute the particular order, and when finished proceed to the next order.

These activities are specified by a variety of orders.

In the present control scheme, a one-address system is used; that is, associated with each order is an address referring to some memory location which contains the number upon which the particular order operates. For example, there are eight orders that transfer a number from the memory to R2. The eight possibilities arise from the three choices:

(1) Clear or do not clear R2 before adding number into it.

(2) Complement or do not complement the number being added to R2.

(3) Add the number or its magnitude.

These are the addition and subtraction orders. There are two multiplication orders; one rounds off the product to 39 bigits, the other provides a precise 78 bigit product. There is one division order, one order transferring a number from the memory to R4. There are six orders associated with transfers to the memory, a right and left shift, print, read, and stop orders. Finally, there are a few logical orders that involve an interruption of the present sequence of orders and a transfer of control to some other sequence.

Eight bigits are used to designate an order. Twelve more are conveniently available, of which ten are actually used at present, for the address. Thus each order is 20 bigits, and two orders are equivalent in storage to one true number. Word is used to describe a 40-bigit aggregate; this may be either an order pair or a true number. A coder

is provided with the set of symbols that correspond to the various orders. These code symbols are various pairs of the six letters, A,B,...F.

Let us now attempt a summary by describing the various steps in machine operation. Assume a tape has been prepared with instructions and initial set of numbers. First the tape is fed into the input. The tetrads are read into R5 in serial fashion. Ten tetrads, corresponding to either a true number or to two orders fill R5. A signal is automatically provided that causes the contents of R5 to be transferred to the first memory location; the second set of ten tetrads is read into R5, etc. When the complete tape has been read into the memory, the computer is ready to do business. The operator presses a "start" button. The contents of the first memory location or first word go to R6; these are the first two orders. The first one is examined and executed, then the second. The next word goes to R6 and the sequence continues. Flexibility exists which enables the sequence to be interrupted at some point and the control transferred to some other point in the sequence. For example, it may be desired to repeat a sequence a fixed number of times before proceeding further, as in some iteration scheme. This is conveniently handled by the logical orders. In fact, it is possible to have the number of repetitions be dependent on the fulfillment of some condition in the problem, so that the number of repetitions varies from case to case. Finally, the desired numerical quantities can be reconverted from binary to binary-decimal form, and printed.

Problem Preparation and Operating Techniques

We conclude the present introductory chapter with a brief commentary on the various steps leading up to the execution of a problem by the computer. The first step concerns the formulation of the problem itself. One method would be simply the writing down of the various equations and the various steps to be taken, together with the necessary explanatory remarks. This approach, although feasible, may often become quite complicated and untractable. Instead we follow von Neumann

who proposed the idea of a flow-diagram. This is a very elegant, logical and mathematical description of the problem to be computed. It makes use of a set of conventionalized symbols to describe the course of the control at every stage of the problem. Represented in a very concise way are: (i) the purely mathematical operations, (ii) various logical steps and decisions together with a precise indication of the nature of the corresponding criteria, (iii) the contents of the relevant part of the memory at points where the question might naturally arise.

The flow-diagram of a problem is prepared by the mathematician or physicist. The symbols are few in number, their meanings simple enough so that they are easily mastered. A flow-diagram may be drawn without a specific computer in mind. In practice, however, one usually does plan on the use of a specific computer and takes advantage of this fact in his planning of a problem. A quite superficial knowledge of the particular computer suffices. The important characteristics are: (i) the capacity of the inner memory, (ii) the nature of the external memory, (iii) the extent of the vocabulary, both arithmetical and logical.

The next step in the preparation is the coding. This process consists conveniently of two parts. In the first, the coder prepares a sequence of instructions using a set of readily interpretable symbols that indicate the general nature of the operations. For example, say at some point in the sequence a number is in register R2 and it is intended to add to it another number at the moment residing somewhere in the memory. A possible notation, and the one used here, is:

$$m \rightarrow Ah$$

where m indicates that a number in the memory is to be sent to R2. For historical reasons, the letter A has been used as a symbol for R2; the original intent being that R2 is the accumulator register. h indicates that R2 is not to clear its contents before receiving from the memory but to hold them for a true addition process. It is observed that the specific binary symbols which the computer can interpret are not used yet, nor is the specific location of the number in the memory given.

There is, however, some point to this preliminary step in the coding. In the first place, there are likely to be several improvements or modifications made before one is satisfied with the sequence of instructions finally adopted for a given problem. This form is much easier to follow, both from the point of making a sample hand calculation (for checking purposes) as well as in trouble-shooting (in the event this is necessary) after the problem has reached the computer.

The second step in the coding is a straightforward transliteration from the coder's notation to teletype symbols. This is routine.

A given large problem may often be divided into a set of smaller problems. Some members of this set may occur frequently enough so that it becomes worthwhile to have these portions coded in quite general terms and, in a sense, treated as individual orders but on a somewhat broader basis. For example, integration by Simpson's Rule, or the inversion of an (n x n) matrix. These sub-routines, as they are conventionally called, would form a library of general orders. A problem at hand would then first be decomposed into the sub-routines available from the library, and the remainder coded from the basic individual orders. Obviously some preparations are required for each individual use of a sub-routine; in the case of the inversion of a matrix, the location of the particular elements for the problem at hand must be specified. Nevertheless, there is a great reduction in effort, especially in checking.

II. CODING AND FLOW DIAGRAMS

Introduction

The computer can perform a set of basic operations, both arithmetical and logical. It may be desirable to keep the set small as added electronic equipment (which is roughly proportional to the number of operations) increases the physical complexity of the computer and complicates maintenance. A modest number of thirty-six operations have been chosen to comprise this set. The choice, however, is fluid in that the set may be modified as the need for change is shown.

We say that the computer has a language of its own, for it is able to interpret and execute the given set of orders. We speak of the orders as the vocabulary of the computer. Coding is the translation of the language of the mathematician into the language of the computer.

The four fundamental arithmetic operations (addition, subtraction, multiplication and division) are a part of the vocabulary. All of the arithmetic operations of the vocabulary, of which there are about twenty, involve the four fundamental operations.

The first step in the preparation of any problem for the computer is to arrange the work so that the only arithmetic operations involved are addition, subtraction, multiplication and division. In other words, the problem must be reduced to a form in which it can be solved by numerical procedures.

The usual mathematical formulation of the problems with which we shall be concerned is a differential equation, or a coupled set of such equations, together with a group of boundary (or initial) conditions. There are other types of problems, but they occur less frequently.

The differential equations are of such complexity that analytical methods are not known for obtaining their solutions. The only recourse is to numerical procedures; therefore these problems are ideally suited for the computer.

The first step in the solution of the problem is to replace the differential equations by a set of finite difference equations. We do not discuss here the stability or convergence of such methods, but only mention them as necessary considerations in writing the difference equations. In

such a process of translation, derivatives are replaced by difference quotients, integrals by sums, transcendental functions by algebraic functions, etc. The problem is now tractable in terms of the vocabulary of the computer as it involves only the fundamental operations.

The next step toward a solution is the preparation of the flow diagram. The flow diagram represents the path to be followed by the computer in the solution of the problem. It represents this by sequences of lines oriented with direction arrows. At points of the diagram where computation is to be performed, the lines are interrupted and boxes are inserted that indicate the "local" computation that is to be performed. The diagram represents the purely mathematical operations, the logical steps and decisions, and the relevant memory storage that is required. Five kinds of boxes represent the desired information:

(i) The operation box
(ii) The alternative box
(iii) The substitution box
(iv) The assertion box
(v) The storage box

These are discussed in detail later.

When the flow diagram is completed, the solution is at the coding level; but before discussing the coding we first discuss some background matters. Each of the thirty-six operations of the vocabulary is referred to as an order. Each order has associated with it a number that specifies the location in the memory of the number upon which the order is to operate; e.g., in the multiply order the associated number specifies the location in the memory of the multiplicand factor. This number location is called an address. The memory contains 1024 words. The addresses of these words consist of the decimal numbers 0 through 1023. Binary-wise, it requires ten bigits to express an address as $1023 \equiv 1111111111$. Eight bigits are used for each order; hence eighteen bigits are necessary for each order with its address. It is convenient, however, to allow twenty bigits for their expression as twenty bigits comprise half of a word. Each order with its associated address is called an instruction. Two instructions are stored per word, giving the memory in principle a capacity of 2048 instructions. However, memory storage is also necessary for true numbers, so that in general there will be some combination of instructions and numbers stored.

The computer uses a one-address system. Each instruction may refer to at most one memory location. Some instructions involve only the arithmetic unit and do not refer to the memory. In these instances the address portion has a different function which is described later.

To illustrate the one-address system consider a simple example of summing two numbers, a and b, which are residing in the memory: The sum s = (a+b) is to be stored in the memory. Three instructions are required:

(i) An instruction to bring a into the arithmetic unit

(ii) An instruction to bring b into the arithmetic unit and to form the sum s = (a+b)

(iii) An instruction to store s in the memory

If a is in the arithmetic unit as a result of some previous operation, only the latter two instructions are needed. If a three-address system were used, the above sequence could be expressed with one order which specified all three addresses: the location of a, the location of b, and the location at which s is to be stored. We defer any discussion of the merits of the one-address system versus those of the multiple address type.

The process of coding involves writing down a sequence of instructions to perform the operations indicated on the flow diagram with the desired set of numbers.

The coding in all but the simplest of problems is not a linear sequence. (That is, the control does not follow a unique path; at various points in a problem several courses may be available.) Certain portions of the coded sequence may be performed several times, whereas other sections are omitted temporarily. The logical orders that have been included in the vocabulary provide for such procedures. Furthermore, the coding is not a static sequence in that it usually does not remain fixed throughout the course of the problem. There are certain orders that allow portions of the coding to be altered so that subsequent traversals through the sequence give rise to a variety of patterns.

It is these dynamic and non-linear characteristics of the coding which provide the desired flexibility for scientific computation but which, on the other hand, give rise to complications in coding.

The remainder of this chapter presents a step-by-step approach to coding, beginning with very simple examples and systematically progressing to examples of increasing complexity.

Before coding any actual examples we first discuss the vocabulary as shown in Table I. It contains a list of the explicit orders with a description of each. It will be noted that there are two types of symbols. The first column gives the abbreviated logical symbol for each order, while the second column gives the actual code for the computer.

Orders 1 through 8 are the addition and subtraction orders. All of these involve R2 (the accumulator register) and a memory location that is specified in the instruction. The first four of these orders clear R2 (set it to 0's) and then add (subtract) the specified word to the 0's in R2. The remaining four orders actually add (subtract) the contents of the specified memory location to the number residing in R2. In a sense, the first four orders are communication orders (they do, however, also allow the magnitude or complement of a number to be inserted) while the latter four are true add or subtract orders.

Consider the example of forming the sum (difference) of two numbers, a and b, and storing the sum $s = (a + b)$, (difference $s = a - b$) in the memory. Assume that a and b are residing in the memory, say at addresses 1 and 2, respectively; and the sum (difference) is to be stored in 3. The instructions are:

1. m⟶Ac 1 a to R2
2. m⟶Ah (s = a + b) / (m⟶Ah-) 2 (s = a - b) to R2
3. A⟶m 3 s to 3

Each order has immediately following it the memory address to which the instruction refers. In a column to the right of the instruction is shown the action that takes place due to each instruction.

If the sum of more than two numbers is formed it is not necessary to send each sum of two numbers into the memory and repeat the three orders. A sum of several numbers may be formed in R2 which requires one additional order for each new number added to the sum; only the final sum is sent to the memory.

In orders 2, 4, 6, and 8 where subtraction is desired this is done by taking the complement of the number with respect to 2 and then performing

TABLE I

(m is the word at address m in the memory)
(The word at its original position is never cleared)

	Abbreviation	Code	
1.	m⟶Ac	AA	Replace the number in R2 by m.
2.	m⟶Ac-	AB	Replace the number in R2 by the complement (the negative) of m.
3.	m⟶AcM	AE	Replace the number in R2 by the absolute value of m.
4.	m⟶Ac-M	AF	Replace the number in R2 by the complement of the absolute value of m.
5.	m⟶Ah	BA	Add m to the number in R2.
6.	m⟶Ah-	BB	Add to the number in R2 the complement of m.
7.	m⟶AhM	BE	Add to the number in R2 the absolute value of m.
8.	m⟶Ah-M	BF	Add to the number in R2 the complement of the absolute value of m.
9.	m⟶Q	EB	Replace the number in R4 by m.
10.	X	DA	Clear R2 and multiply m by the number in R4. The 39 most significant bigits of the product appear in R2. The 2^{-39} bigit position of R2 is set to 1. R4 is set to 0's.
11.	X'	DB	Clear R2 and multiply m by the number in R4. The left-hand 39 bigits appear in R2, the right-hand 39 bigits in R4. The sign bigit of R4 is set to 0.
12.	÷	DD	Divide the number in R2 by m. The quotient appears in R4, two times the remainder appears in R2.
13.	T	CA	Transfer the control to the left-hand order of m.
14.	T'	CB	Transfer the control to the right-hand order of m.
15.	C	CC	If the number in R2 is ≥ 0, transfer the control as in T, otherwise continue to next order in sequence.
16.	C'	CD	If the number in R2 is ≥ 0, transfer the control as in T', otherwise continue to next order in sequence.
17.	Q⟶m	EC	Replace m by the number in R4.
18.	A⟶m	DC	Replace m by the number in R2.

TABLE I (Cont.)

19.	S⟶m	FA	Replace the address (bigits 8-19) of the left-hand order of $\underline{m}$ by the 12 bigits 8-19 in R2.
20.	S⟶m'	FB	Replace the address (bigits 28-39) of the right-hand order of $\underline{m}$ by the 12 bigits 28-39 in R2.
21.	HS⟶m	FC	Replace the left-hand 20 bigits (bigits 0-19) of $\underline{m}$ by the 20 bigits 0-19 in R2.
22.	HS⟶m'	FD	Replace the right-hand 20 bigits (bigits 20-39) of $\underline{m}$ by the 20 bigits 20-39 in R2.
23.	Rn	EE	Right shift R2 and R4 n places where n is specified in the address bigits of the order. This replaces the contents $\lambda_0, \lambda_1 \cdots \lambda_{39}$ of R2 and $\sigma_0, \sigma_1 \cdots \sigma_{39}$ of R4 by $\lambda_0 \cdots \lambda_0, \lambda_1 \cdots \cdots \lambda_{38-n}, \lambda_{39-n}$, and $\lambda_{39-n+1}, \lambda_{39-n+2} \cdots \lambda_{39}, \sigma_0, \sigma_1 \cdots \sigma_{39-n}$.
24.	Ln	DE	Left shift R2 and R4 n places where n is specified in the address bigits of the order. This replaces the contents $\lambda_0, \lambda_1 \cdots \lambda_{39}$ of R2 and $\sigma_0, \sigma_1 \cdots \sigma_{39}$ of R4 by $\lambda_n, \lambda_{n+1} \cdots \lambda_{39}$, 0 ⋯ 0 and $\sigma_n, \sigma_{n+1} \cdots \sigma_{39}, \lambda_0, \lambda_1 \cdots \lambda_{n-2}, \lambda_{n-1}$.
25.	a⟶Ac	EF	Replace the number in R2 by the 12 address bigits of this order (into positions 0-11 of R2).
26.	a⟶Ah	DF	Add to the number in R2 the 12 address bigits of this order (into positions 0-11 of R2).
27.	DS	ED	Set the sign bigit of the number in R2 to 0.
28.	Flexo Print	EA	Print $\underline{m}$ on the page printer (slow speed).
29.	Read	FF	Replace $\underline{m}$ by the next word to come under the reading head of the paper tape reader.
30.		FE	(NOT PRESENTLY USED)
31.	Punch	CF	Punch $\underline{m}$ on paper tape.
32.	Sync Print	CE	To be used in a subroutine which simultaneously prints $\underline{m}_i, \underline{m}_{i+1}, \underline{m}_{i+2}$ and $\underline{m}_{i+3}$; $\underline{i}$ is to be communicated to the routine (high speed).

TABLE I (Concl.)

33.	m⟶D	BD	Read 50 successive words from the memory starting with the word at the address specified by bigits 8-19 of the instruction. Write these 50 words into the drum on the track specified by bigits 20-27. Then transfer the control to the left-hand instruction of the word at the address specified by the bigits 28-39.
34.	D⟶m	BC	Read the 50 words from the track of the drum specified by bigits 20-27 of the instruction. Write these words into 50 successive memory locations starting with the address specified by bigits 8-19. Then transfer the control to the left-hand instruction of the word at the address specified by bigits 28-39.
35.	Q⟶t	AD	Write the number in R4 onto the magnetic tape.
36.	t⟶Q	AC	Replace the number in R4 by the first word to come under the reading head of the magnetic tape reader.
37.	Stop	OFF	Stop computation. (Pressing start next order button will allow machine to continue in normal sequence.)

NOTE: An address of 800 refers to the quotient register (R4) when using orders 1 through 8; i.e., AA800 says replace the number in R2 by the number in R4.

a normal addition. The complement scheme is described in detail in the chapter on binary arithmetic. When an address 100000000000 which corresponds to 2048 decimally is used with any of the orders 1 through 8, it has the effect of treating R4 (the quotient register) as a memory position with the address 2048. The number residing in R4 can then be added into R2 as described by any one of the orders 1 through 8.

Order 9 transmits a number from the memory to R4 (the quotient register). R4 does not have add facilities; hence a number being transmitted to R4 replaces the number that is in R4.

Orders 10 and 11 are the two multiplication orders. Before either of these orders may be given, the multiplier must be in R4 (either as the result of some previous operation or by a preceding $m \rightarrow Q$ order). The 39 most significant bigits of the product appear in R2. Order 10 gives only the 39 most significant bigits of the product rounded off. Order 11 gives a full 78 bigit product; the rightmost 39 bigits appear in R4. The multiply order supplies the multiplicand.

Order 12 is the divide order. It is assumed that the dividend is in place in R2; the divide order itself provides the divisor. The quotient is located in R4, and two times the remainder appears in R2.

Order 13 is a transfer order. This interrupts a sequence and causes the computer to continue with another sequence beginning with the instruction specified by the address part of the transfer instruction. As an example of a transfer instruction, suppose that a sequence of instructions is being performed and in the 25th step of the sequence a transfer is encountered:

```
.
.
.
25   T   125
.
.
.
124
125
.
.
.
```

The transfer instruction has the address 125, so that the sequence of code from 26 to 124 is omitted. The computer would execute Instruction 125 and continue sequentially from there.

Since an instruction word consists of two instructions and the flexibility of being able to transfer into either instruction of a word is desired, it is necessary to have two transfer orders to accomplish this. This accounts for Order 14, the T' order, as well as Order 13. Hence, in the above example, 25 may have read T 125 or T' 125, depending on whether the transfer was desired to the left or right instruction of Instruction Word 125.

The two conditional transfer orders, 15 and 16, either execute the transfer as in the T orders discussed immediately above, or the orders require no action, in which case the computer continues along the original sequence. The conditional transfer is effective or not, depending on the sign of the number, N, in R2 at the time the order is to be performed: if $N \geq 0$, the transfer does occur, and a new sequence of instructions is started at the location specified by the address part of the instruction; if $N < 0$, the computer continues with the original sequence of instructions.

Orders 17 and 18 are the two orders that send information from the arithmetic unit to the memory. Order 17 transmits from R4 to the memory, and 18 transmits from R2 to the memory. When any register or memory location sends information to any other register or memory location, the information is still available at its original position.

Orders 19 through 22 are the substitution orders. These orders make alterations in instructions. By means of 19 and 20, any instruction may have its address changed. The new address is first formed in R2 and then inserted into the desired instruction by means of a substitution order. The use of the substitution orders is explained in detail in Problem 2. The two half word substitution orders (Numbers 21 and 22) may alter whole instructions rather than just the address. These two orders may also be used in storing half precision numbers. The details of their use will be covered by later examples.

Orders 23 and 24 are the right and left shift orders. They give a means of dividing or multiplying by powers of 2 by shifting a number right or left in R2; e.g., if a number a = 0.00001111 is residing in R2

and it is desired to multiply this number by 2^4, this can be effected by a left shift of 4 places, which displaces the number 4 units to the left.

$$a = 0.00001111$$

$$a \times 2^4 = 0.11110000$$

A right shift effects division by powers of 2 by displacing the number to the right. In a left shift R4 may be considered an extension of R2 to the left; hence a number shifting left out of R2 fills into R4 beginning in the least significant end of R4. In a right shift R4 may be considered an extension of R2 to the right and a number shifting right out of R2 fills into R4 beginning in the most significant end of R4. Since R2 and R4 are so interconnected for shifting operations, these operations may be used for separating a multiplex of numbers occupying one word. Either a left or right shift of 40 places will transfer completely a number from R2 to R4.

Orders 25 and 26, a⟶Ac and a⟶Ah, treat their associated addresses as true numbers. The addresses of these instructions are sent into R2 (either a clearing or an adding action) into bigit positions 0 through 11. Many times in the type of problem in which we will be interested there are small numerical constants of three significant decimal digits or less. Rather than use an entire memory location to store such constants, they can often be expressed in the address position of an a⟶A instruction. As an example consider that a quantity

$$ax^2 + bx$$

has been formed and is in R2. It is desired to add a constant term $\underline{k}$ where $\underline{k} = .583$. This may be expressed in the a⟶Ah order as

	.	.
	.	.
	.	.
	.	$ax^2 + bx$ in R2
(iii)	a⟶Ah 583	$ax^2 + bx + (.583 = k)$ to R2

where .583 is expressed by its binary equivalent. Eleven bigits give the same precision as 3.3 decimal digits, so any three-decimal digit fraction may be expressed in the address position of an a⟶A order.

The explanation of the remainder of the orders as given in Table I is adequate; hence we return to the task at hand, the coding of typical problem-examples.

The coding of a problem may be divided into two parts:

(i) The logical coding

(ii) The computer (numerical) coding

Each of these parts involves several steps. At the present level of our knowledge and skill, it seems convenient to have both a logical and a numerical symbol for each order. The logical symbols are used in part (i), while the numerical symbols are used in part (ii).

The logical symbol attempts to be a descriptive abbreviation of the action of that instruction; the associated memory location is preliminarily specified by a combination of a letter and a number; the letter identifies some group storage and the number identifies a member of that group; e.g., m⟶Ac B.4 is interpreted as: Bring from the memory to the Accumulator (R2), clearing the accumulator first, the number at memory location B.4. One reason for not assigning specific numerical memory locations at the outset of a problem is that the extent and disposition of the memory requirements are not immediately obvious. A set of logical symbols is more meaningful to the coder than an abstract code; it expedites the actual coding and facilitates checking.

The abstract coding is merely a transliteration from the logical code to the numerical code. The numerical code is shown in the second column from the left in Table I. Each order is represented by a combination of two of the letters, A,B,C,D,E,F, where each letter expresses a tetrad (4) of bigits. These are:

A	1010	D	1101
B	1011	E	1110
C	1100	F	1111

When the coding has been written in numerical form, the teletype tape (which is the present means of putting the coded sequence into the memory unit) is prepared. The actual coding examples are treated in the following pages.

Problem 1

We propose to form the rational function y with constant coefficients where

$$y = \frac{ax^2 + bx + c}{ex + f}$$

Assume that x, a, b, c, e, and f are in the memory at known addresses. As previously mentioned, the memory locations are denoted by capital letters rather than using true number addresses; e.g., the notation A.1: a implies that the quantity a is stored in the memory at address A.1. The storage of the problem is:

A.1:a	A.4:e
A.2:b	A.5:f
A.3:c	A.6:x

and when y is formed it is to be stored in A.7.

As a preparatory step in coding the problem, we form y by a sequence of arithmetic operations in which each step involves only one operation. Such a sequence is:

1. $e \cdot x$
2. $ex + f$
3. $a \cdot x$
4. $ax + b$
5. $(ax + b)x$
6. $ax^2 + bx + c$
7. $y = \frac{ax^2 + bx + c}{ex + f}$

Since the computer can accomplish only one arithmetic operation at a time, the above sequence is precisely the procedure that one must go through in coding the problem, insofar as the arithmetic is concerned.

We now proceed with the coding. In the preliminary logical code, each instruction is treated as a word rather than the actual case of two instructions per word. The left-hand column is the code abbreviation, and the next column indicates the operations that have taken place in the arithmetic unit, while the last column is conveniently used for memory storage. During the course of the problem, a storage location in the memory is needed to store an intermediate value of the computation. This position is denoted as B.1.

The sequence is:

1.	m⟶Q	A.6	x to R4	
2.	X	A.4	e•x in R2	
3.	m⟶Ah	A.5	ex + f in R2	
4.	A⟶m	B.1		ex + f to B.1
5.	m⟶Q	A.1	a to R4	
6.	X	A.6	a•x in R2	
7.	m⟶Ah	A.2	ax + b in R2	
8.	L40		ax + b to R4	
9.	X	A.6	(ax + b)x in R2	
10.	m⟶Ah	A.3	$ax^2 + bx + c$ in R2	
11.	÷	B.1	$y = \frac{ax^2 + bx + c}{ex + f}$ in R4	
12.	Q⟶m	A.7		y to A.7

Note that the denominator was formed before the numerator. If the reverse had been the case, the numerator when formed would have been stored in, say, B.1. When the denominator was formed it, too, would have been stored in, say, B.2. The numerator would then be brought in and the division performed. Coding in this fashion, however, would have required two additional instructions and one word more storage in all making the coding two words longer than it is at present.

Instruction 8 in the above sequence, which is L40, is a means of communication from R2 to R4. L40 shifts the entire word including the sign from R2 to R4. If this were not available, it would be necessary to send the word from R2 to the memory and then from the memory to R4, thus requiring one additional instruction.

Recall that each instruction word in the memory actually contains two instructions. The next step of the coding is to arrange the sequence of instructions into words. If we assume that the routine starts at address 1 in the memory, the sequence then occupies memory locations 1 through 6 (since it contains 12 instructions, 6 words are required). At this time, the constants of the problem are given true memory addresses. Since there are six such quantities (where each quantity comprises one word), memory locations 7 through 12 are allotted for these. When y is formed it will be stored at address 13. One temporary location is needed which is designated as 14.

The sequence becomes:

1.	m⟶Q	12	X	10
2.	m⟶Ah	11	A⟶m	14
3.	m⟶Q	7	X	12
4.	m⟶Ah	8	L	40
5.	X	12	m⟶Ah	9
6.	+	14	Q⟶m	13
7.	a			
8.	b			
9.	c			
10.	e			
11.	f			
12.	x			
13.	-			
14.	-			

Memory locations 13 and 14 are used for quantities formed within the routine; hence they must be empty or their contents must be irrelevant at the time the sequence is to be executed by the computer.

When the coding is in final form such that the input teletype tape is to be prepared, one has the instructions reduced to numerical form and has available the true numerics for all of the involved quantities. Assume, for example, that

a	= .075329	e	= .83291
b	= .12391	f	= .69736
c	= .017326	x	= .32915

The final coding is:

1.	EB012DA010	8.	0.123910000
2.	BA011DC014	9.	0.017326000
3.	EB007DA012	10.	0.832910000
4.	BA008DE040	11.	0.697360000
5.	DA012BA009	12.	0.329150000
6.	DD014EC013	13.	0.000000000
7.	0.07532900	14.	0.000000000

Problem 2

We modify the preceding problem with a slight logical twist. Assume that the calculation of the rational function y is a part of some larger problem and that x has been computed as part of a previous routine and stored in some memory location other than the one assigned to it (A.6 in the preceding example). Indeed, there may be a series of such x values. Further, when y is computed it is to be stored, not in A.7, but at some other memory location where it will be used in subsequent parts of the calculation. In other words, we ask what modifications must be made to the sequence of instructions in Problem 1 in order to render it more flexible and assimilable in a larger problem.

One possibility is to reserve memory location A.6, not for storing x itself as was done earlier, but instead to store the address at which x may be found. A.6 does not contain x, but it does tell us where in the memory x is located. Similarly, we may use A.7, not to store y itself, but to contain the address at which y is to be stored when formed.

Suppose then, as a preceding part of some problem, x has been computed and stored in, say, memory location M.1; and we wish to use the routine outlined in Problem 1 to calculate the rational function given there with the stipulation that y should be stored in N.1.

It is necessary to place the address M.1 in location A.6 and address N.1 in location A.7. Thus, in the course of the calculation, when x is required, A.6 is consulted, giving the information where x is actually located. Finally, A.7 provides the information where y is to be stored, namely in N.1. Thus, this rational function routine may be used several times in the course of a large problem; each time, however, it is necessary to provide the corresponding address for the locations x and y.

Making these changes in this routine leads to the simplest illustration of using the substitution order. Without attempting to justify the utility of it at this point, we proceed with the simple example.

Instructions 1 through 6 of the following code sequence are the additional instructions required for the substitutions. The function of these first instructions is to provide appropriate addresses to subsequent instructions that involve x and y. Recall that x resides at

location M.1, and the numerical value of M.1 is at A.6. The preliminary instructions thus involve taking the numerical quantity M.1 from location A.6 in the memory to the arithmetic unit. From there it may be inserted into the address part of the instruction that first involves x. This is accomplished by the substitution order. Repeated application of this order introduces this same address into all the other instructions that require it. In the example observe that Instruction 8 of the code is the first instruction referring to x and requiring the particular address where x resides. Two instructions, here taken to be 1 and 2, are required to provide Instruction 8 with the appropriate address. These are:

(i) An instruction to transfer the contents of A.6, namely the address of x, to the arithmetic unit;

(ii) A substitution order which has the effect of transferring this address of x into Instruction 8.

Inasmuch as this address is also required for Instructions 11 and 15, two more substitution orders, Instructions 3 and 4, are needed for them. Finally, the address referring to the location of y is needed for Instruction 18; two more instructions, 5 and 6, accomplish this, thus accounting for the six preparatory instructions.

At the start of the problem, Instructions 8, 11, 15, and 18 have blank addresses. After the control has proceeded through Instruction 6, all of the instructions have the proper addresses.

The storage is as before, with the changes as noted above,

A.1: a
A.2: b
A.3: c
A.4: e
A.5: f
A.6: M.1
A.7: N.1
B.1:
M.1: x
N.1:

The coding is:

1.	m⟶Ac	A.6	M.1 to R2	
2.	S⟶m	8		M.1 to (8-19)8
3.	S⟶m	11		M.1 to (8-19)11
4.	S⟶m	15		M.1 to (8-19)15
5.	m⟶Ac	A.7	N.1 to R2	
6.	S⟶m	18		N.1 to (8-19)18
7.	m⟶Q	A.4	e to R4	
8.	X	[ ]	e•x in R2	
9.	m⟶Ah	A.5	ex + f in R2	
10.	A⟶m	B.1		ex + f to B.1
11.	m⟶Q	[ ]	x to R4	
12.	X	A.1	a•x in R2	
13.	m⟶Ah	A.2	ax + b in R2	
14.	L40		ax + b to R4	
15.	X	[ ]	$ax^2 + bx$ in R2	
16.	m⟶Ah	A.3	$ax^2 + bx + c$ in R2	
17.	÷	B.1	$y = \frac{ax^2 + bx + c}{ex + f}$ in R4	
18.	Q⟶m	[ ]		$\underline{y}$ to N.1

In coding the problem into word form, the instructions into which addresses are being substituted may be either the left-hand or the right-hand instruction of a word. In Table I, Orders 19 and 20 account for this. They read:

"19. S⟶m Replace the address (bigits 8-19) of the left-hand order of $\underline{m}$ by the 12 bigits 8-19 in R2.

20. S⟶m' Replace the address (bigits 28-39) of the right-hand order of $\underline{m}$ by the 12 bigits 28-39 in R2."

Since it is desirable to substitute into either a left-hand or right-hand instruction from an address which has been brought into R2, the following custom in storing addresses is adopted: Consider an address $\underline{x}$ as an integer which may assume values from $\underline{0}$ to 1023. Rather than storing $\underline{x}$, store

$$(x)_o \equiv 2^{-19}\underline{x} + 2^{-39}\underline{x},$$

where $(x)_o$ is called the memory position mark $\underline{x}$. Since $\underline{x}$ is an integer, when $(x)_o$ is brought into R2 the addresses are so positioned that either S⟶m or S⟶m' may be used as required.

The instructions are now paired into words. There are 18 instructions or 9 words which, if the coding starts at word 1, give instruction-words from address 1 through 9. The numerics then start with address 10 and go through address 17.

1.	m $\longrightarrow$Ac	15	S$\longrightarrow$m'	4
2.	S $\longrightarrow$m	6	S$\longrightarrow$m	8
3.	m$\longrightarrow$Ac	16	S$\longrightarrow$m'	9
4.	m$\longrightarrow$Q	13	X	[ ]
5.	m$\longrightarrow$Ah	14	A$\longrightarrow$m	17
6.	m$\longrightarrow$Q	[ ]	X	10
7.	m$\longrightarrow$Ah	11	L40	
8.	X	[ ]	m$\longrightarrow$Ah	12
9.	$\bullet$	17	Q$\longrightarrow$m	[ ]
10.	a			
11.	b			
12.	c			
13.	e			
14.	f			
15.	$(M.1)_o$			
16.	$(N.1)_o$			
17.				

The storage has been changed to include the appropriate values $(M.1)_o$ and $(N.1)_o$.

In the final coding, Instructions 4', 6, 8, and 9' may initially be given any address as this address is irrelevant (the correct addresses are supplied during the course of the computation). For uniformity, the plan of initially setting these addresses to 0 is adopted.

Problem 3

The numbers $a_1, a_2, a_3 \cdots a_n$ and the numbers $b_1, b_2, b_3 \cdots b_n$ are stored in the memory. It is desired to form the following product sum

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

The storage of the a's and b's is arranged so that

$$A.1{:}a_1,\ A.2{:}a_2 \cdots A.i{:}a_i \cdots A.n{:}a_n$$

and

$$B.1{:}b_1,\ B.2{:}b_2 \cdots B.i{:}b_i \cdots B.n{:}b_n$$

That is, the a's are stored consecutively in one section of the memory and the b's are stored consecutively in another section. The sum, when it is formed, is to be stored in the memory at address C.1.

If n = 1, the coding is trivial; it is:

1.	m⟶Q	A.1	a_1 to R4	
2.	X	B.1	$a_1 b_1$ in R2	
3.	A⟶m	C.1		$a_1 b_1$ to C.1

The problem may be extended to n = 2 by adding the following instructions:

4.	m⟶Q	A.2	a_2 to R4	
5.	X	B.2	$a_2 b_2$ in R2	
6.	m⟶Ah	C.1	$a_1 b_1 + a_2 b_2$ in R2	
7.	A⟶m	C.1		$a_1 b_1 + a_2 b_2$ to C.1

One method of extending the coding to the general case of n elements in the sum is to have the first three instructions followed by (n - 1) repetitions of Instructions 4 through 7 with the appropriate A.i and B.i being used in place of the A.2 and B.2. This method becomes very costly with respect to available memory space as n becomes large, since each increase of n by 1 increases the code by four instructions.

The coding for the general case n is:

1.	m⟶Q	A.1	a_1 to R4	
2.	X	B.1	$a_1 b_1$ in R2	
3.	A⟶m	C.1		$a_1 b_1$ to C.1
4.	m⟶Q	A.2	a_2 to R4	
5.	X	B.2	$a_2 b_2$ in R2	
6.	m⟶Ah	C.1	$a_1 b_1 + a_2 b_2$ in R2	
7.	A⟶m	C.1		$a_1 b_1 + a_2 b_2$ to C.1

8.	m $\longrightarrow$ Q	A.3	a_3 to R4
9.	X	B.3	$a_3 b_3$ in R2
10.	m $\longrightarrow$ Ah	C.1	$a_1 b_1 + a_2 b_2 + a_3 b_3$ in R2
11.	A $\longrightarrow$ m	C.1	$a_1 b_1 + a_2 b_2 + a_3 b_3$ to C.1
•			
•			
•			
4i-4.	m $\longrightarrow$ Q	A.i	a_i to R4
4i-3.	X	B.i	$a_i b_i$ in R2
4i-2.	m $\longrightarrow$ Ah	C.1	$a_1 b_1 + a_2 b_2 + \cdots + a_i b_i$ in R2
4i-1.	A $\longrightarrow$ m	C.1	$a_1 b_1 + a_2 b_2 + \cdots + a_i b_i$ to C.1
•			
•			
•			
4n-4.	m $\longrightarrow$ Q	A.n	a_n to R4
4n-3.	X	B.n	$a_n b_n$ in R2
4n-2.	m $\longrightarrow$ Ah	C.1	$a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=0}^{n} a_i b_i$ in R2
4n-1.	A $\longrightarrow$ m	C.1	$\sum_{i=1}^{n} a_i b_i$ to C.1
4n.	STOP		

By using this method, 4n instructions are needed. If n is large, say 50-100, then 200 to 400 instructions or 100 to 200 words of coding are needed.

Note, however, that the only changes in the coding for each i are the changes in the addresses of the instructions (m$\longrightarrow$Q A.i) and (X B.i), and as i is increased by 1 the addresses of these two instructions are also increased by 1.

If by some means the computer can be directed to go repeatedly through the coding and at each traversal to increase by 1 the addresses of the instructions (m$\longrightarrow$Q A.i) and (X B.i), the length of the total coding can be shortened greatly. By means of the transfer orders a section of the coding can be traversed as many times as is desired; and at each passage through the coding the instructions (m$\longrightarrow$Q A.i) and (X B.i) are brought into the arithmetic unit and 1 is added (in the correct address position) to each of them. It is, of course, necessary to have available in the memory the

appropriate $\underline{1}$ to increase the addresses. It may be either 1×2^{-19}, 1×2^{-39} or, in fact, both may be needed. At present we store 1×2^{-m} in C.2, and fix upon $\underline{m}$ later in the coding. The sequence is:

1.	m⟶Q	A.1	a_1 to R4	
2.	X	B.1	a_1b_1 in R2	
3.	A⟶m	C.1		a_1b_1 to C.1
4.	m⟶Q	A.2	a_2 to R4	
5.	X	B.2	a_2b_2 in R2	
6.	m⟶Ah	C.1	$a_1b_1 + a_2b_2$ in R2	
7.	A⟶m	C.1		$a_1b_1 + a_2b_2$ to C.1
8.	m⟶Ac	4	(m⟶Q A.2) to R2	
9.	m⟶Ah	C.2	(m⟶Q A.2 + 1) in R2	
10.	A⟶m	4		(m⟶Q A.3) to 4
11.	m⟶Ac	5	(X B.2) to R2	
12.	m⟶Ah	C.2	(X B.2 + 1) in R2	
13.	A⟶m	5		(X B.3) to 5
14.	T	4		

The first seven instructions are the same as before. Instructions 8, 9, and 10 bring Instruction 4 into the arithmetic unit, add $\underline{1}$ to its address, and again store the instruction in 4, its correct location. Instructions 11, 12, and 13 do the same to Instruction 5. Instruction 14 transfers the control back to Instruction 4 to traverse that section of coding again (the necessary addresses have been increased by $\underline{1}$).

The above sequence is not yet complete as it does not provide a means of stopping the cyclic process when $\underline{n}$ is reached. By changing the transfer order to a conditional transfer order and adding the following instructions, we introduce a means of knowing when the cyclic process is finished. The number of traversals through the cyclic process is kept track of by keeping a count in, say, location C.3, and for each passage the count is increased by one and also examined to determine whether the desired value has been reached. It is this examination which is performed by the conditional transfer order. To initiate the count we store 2×2^{-m}. Since the first two terms of the product sum $a_1b_1 + a_2b_2$ are formed before the counting process is initiated, these two terms are included in the count by starting the count at 2. When the count reaches $\underline{n}$, instead of transferring back to Instruction 4 the control goes along the other branch of the conditional transfer instruction, and in this case terminates with a stop order.

The additional coding is added, starting at Instruction 14.

14.	m ⟶ Ac	C.3	2×2^{-m} to R2
15.	m ⟶ Ah	C.2	$(2+1) \times 2^{-m} = 3 \times 2^{-m}$ in R2
16.	A ⟶ m	C.3	3×2^{-m} to C.3

$n \times 2^{-m}$ is needed; it is stored in C.4.

17.	m ⟶ Ac	C.4	$n \times 2^{-m}$ to R2
18.	m ⟶ Ah-	C.3	$n \times 2^{-m} - 3 \times 2^{-m}$ in R2
19.	C	4	
20.	STOP		

Note that the count in C.3 is increased just before it is subtracted from $n \times 2^{-m}$. When the count becomes equal to $n \times 2^{-m}$, the subtraction gives 0 (which is interpreted as positive) and the conditional transfer sends the control back to Instruction 4 to finish the n^{th} term of the product sum. The next time through the sequence the count is increased to $n + 1$; the subtraction now gives a negative difference; and the conditional transfer is not effective. The control then proceeds to Instruction 20 and stops as is desired.

The coding is 20 instructions, which is 10 words. We start the sequence at address 1; hence it occupies words 1 through 10. Four words of storage are needed during the course of the problem; for these addresses 11 through 14 are assigned. Let us set n to 100 and store the a_i's in 16 through 115 and the b_i's in 116 through 215.

The sequence is:

1.	m ⟶ Q	16	X	116
2.	A ⟶ m	11	m ⟶ Q	17
3.	X	117	m ⟶ Ah	11
4.	A ⟶ m	11	m ⟶ Ac	2
5.	m ⟶ Ah	12	A ⟶ m	2
6.	m ⟶ Ac	3	m ⟶ Ah	15
7.	A ⟶ m	3	m ⟶ Ac	13
8.	m ⟶ Ah	12	A ⟶ m	13
9.	m ⟶ Ac	14	m ⟶ Ah-	13
10.	C'	2	STOP	
11.	-			
12.	1×2^{-39}			
13.	2×2^{-39}			
14.	$n \times 2^{-39} = 100 \times 2^{-39}$			
15.	1×2^{-19}			

16. a_1
17. a_2
. . .
115. a_{100}
116. b_1
117. b_2
. . .
215. b_{100}

In words 12 and 15, 1×2^{-39} and 1×2^{-19} are stored. These are both needed as the two instructions that have their addresses increased are in opposite sides of their respective words.

The code sequence is reduced from 200 words to 15 words by being able to use the same section of code repeatedly and altering addresses of the instructions as the control proceeds through the code.

The use of substitution orders in this problem was purposely avoided. As we shall presently see, the change in addresses could have been accomplished more efficiently by their use. However, our purpose is not necessarily to illustrate the shortest method for coding a sequence but to illustrate many methods so that a broad foundation may be laid for subsequent work.

We adopt the nomenclature set forth by von Neumann and call any such repetitive process (whether it be the above, or a solution of a partial differential equation by successive approximations, or numerical integration of a function by some stepwise method, or other iterative procedures) a simple induction.

We have now reached the point where any further examples have a great enough complexity to demand a systematic approach. This leads to the discussion of the flow diagram.

Flow Diagram

The flow diagram, as the name implies, indicates the course of the control through a coded sequence of instructions. As previously mentioned, the flow diagram represents in a concise way

(i) The purely mathematical operations

(ii) The various logical steps and decisions together with a precise indication of the corresponding criteria

(iii) The contents of the relevant parts of the memory where the question might naturally arise

To facilitate the interpretation of such diagrams and to avoid ambiguities, it is convenient to have a set of conventionalized symbols associated with these flow diagrams.

The direction of motion of the control through the flow diagram is indicated by lines oriented with arrows as in Figure 1. A simple induction is denoted by a closed loop as in Figure 2 and is called an induction loop.

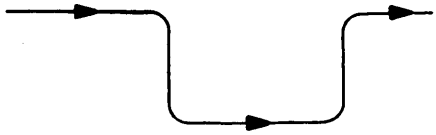


Figure 1.

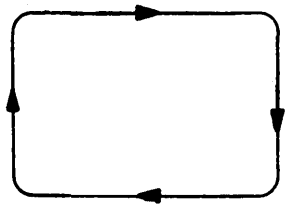


Figure 2.

Any non-looped segment of the flow diagram is described as a linear section, while a looped segment is said to be non-linear.

By themselves the above lines are incomplete as they do not show the arithmetic or logical processes that are involved. The arithmetical operations are described in the operation boxes. Figure 3 shows the symbolization of the operation box.

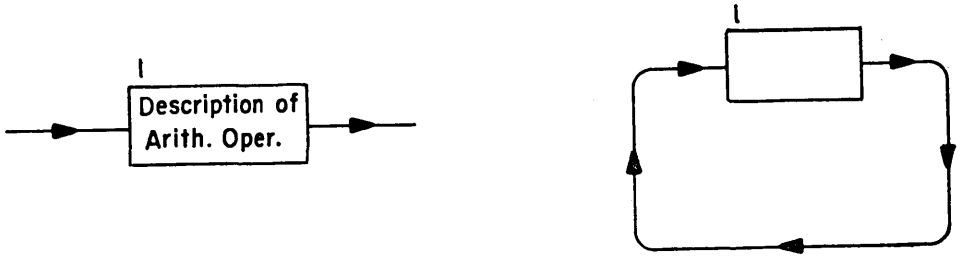


Figure 3.

The operation box has one entrance and one exit for the control. The contents of the box indicate the arithmetic operations and transfers of information among the various storage locations that are to take place when the control reaches that stage. Individually, an operation box may be treated as a linear portion of the flow diagram, although it may be an element of an induction loop. Each operation box of a flow diagram is identified by an Arabic numeral.

The induction loop as shown in Figure 3 is not complete, as it shows neither a point of entrance nor a point of exit.

To show the former, two or more paths of a flow diagram merge into a common continuation as shown by the heavy lines of Figure 4. These mergers are not unique to an induction loop for they are also useful where several linear sequences have a common continuation.

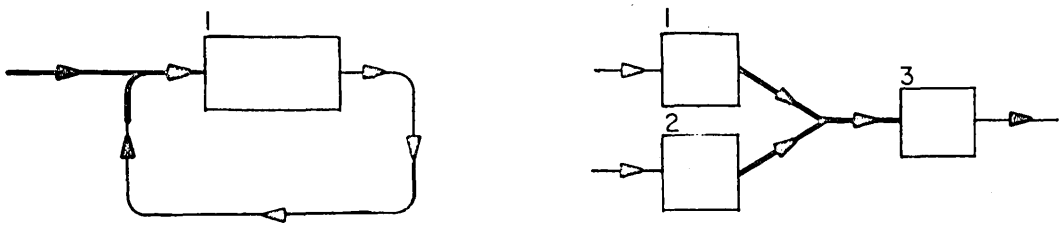


Figure 4.

In order to effect an exit from an induction loop, use is made of a second type of box called the alternative box (conditional transfer box). The alternative box has one entrance, but two exits which are labeled the positive (non-negative) and the negative exits. This box specifies the criterion by which the control follows either one exit or the other. The decision is usually based upon some mathematical expression that is first formed in the Accumulator. In the coding, the conditional transfer instruction is given immediately after the discriminating quantity has been formed in the Accumulator. If the quantity is positive or 0, the control proceeds along the so-called positive branch, whereas if the quantity is negative, the negative branch is followed. By convention, the positive branch corresponds to an interruption of the sequence and a transfer of the control to the instruction pair specified by the address part of the conditional transfer. On the negative branch the control proceeds sequentially without interruption. The alternative box may be associated with a linear sequence

of a flow diagram as well as with an induction loop; i.e., a linear sequence may divide into two sequences, the choice of which may be made by an alternative box. Figure 5 illustrates an alternative box (emphasized by heavy lines) used in a linear sequence, and also in association with an induction loop. The alternative box is identified by an Arabic numeral, as is the operation box.

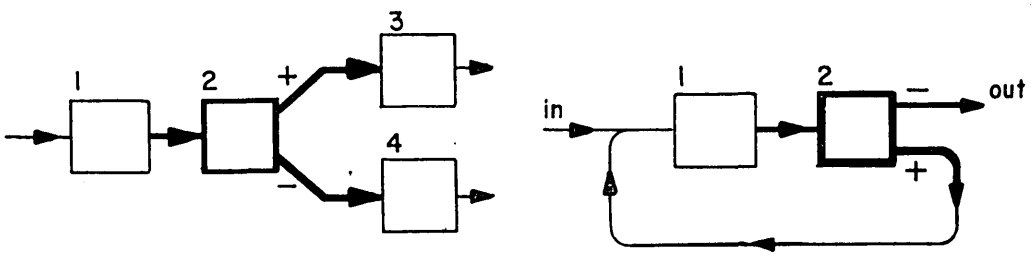


Figure 5

Since an alternative box is the means of exit from an induction loop, it is the alternative box that indicates when the loop has been traversed the appropriate number of times. The quantity upon which the conditional transfer instruction is to act should then remain positive until the loop has been traversed the correct number of times and then this quantity is to become negative. (It may happen, at times, that it is more advantageous for the negative branch to return through the loop, with the positive branch providing the exit.) As an example:

If we are doing an iterative process to approximate some function--say a trigonometric function, square root of a number, or some other such scheme--then we know that the error in the approximation to the function is less than the difference between any two successive approximations. We then decide upon the accuracy, say ∂, for the approximation to the function. If we denote an approximate by S_i, then the desired accuracy is obtained when $|S_i - S_{i+1}| < \partial$. Therefore, in such a process, if the conditional transfer acts upon the quantity $|S_i - S_{i+1}| - \partial$, this quantity will be positive until the desired condition obtains.

An induction loop may involve a process in which the loop is to be traversed a fixed number of times. For these processes a simple counting procedure is used to determine the termination of the induction.

In the initial step of the induction the count is set to some starting value (usually 0 or 1). At each traversal of the loop the count, which may be called i, is increased by 1. An upper limit to the count, which is called I, is chosen, such that the quantity I-i first becomes negative when the loop has been traversed the correct number of times, hence satisfying the required conditions.

In a linear sequence the alternative box often indicates a single quantity which is the result of previous computation where the course to be followed depends upon this quantity being positive or negative. Figure 6 indicates several alternative boxes with their contents.

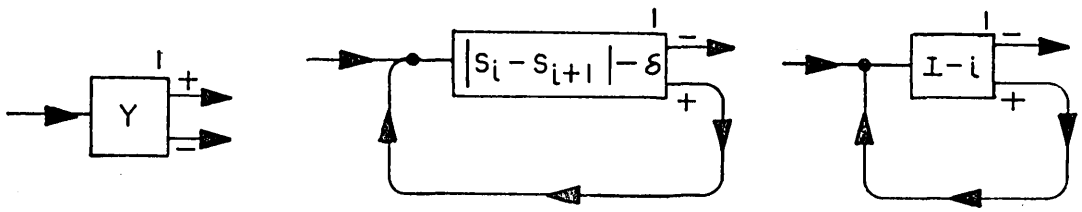


Figure 6.

By means of an alternative box an induction loop may be traversed as many times as desired and then the control is advanced to the next stage of a calculation. Each time the induction loop is traversed the control essentially repeats a fixed sequence of orders. At each traversal, though, the control operates on a different set of numbers and either sends the results to fixed memory positions each time, or else sends the results to locations dependent upon the set of numbers being operated upon. The operation boxes in an induction loop should contain relationships that are valid in general for any traversal through the loop; e.g., consider the iterative process for the square root of a number u where $u < 1$ (we defer any mathematical discussion until later). The first approximation Z_o is chosen equal to 1, and the successive ones given by

$$Z_o = 1$$

$$Z_1 = 2^{-1}(Z_o + u/Z_o)$$

$$Z_2 = 2^{-1}(Z_1 + u/Z_1)$$

$$\vdots$$

$$Z_{i+1} = 2^{-1}(Z_i + u/Z_i)$$

$$\vdots$$

The successive iterates are to be done in an induction loop where Z_o is an initial step apart from the loop. In the first traversal of the loop

$$Z_1 = 2^{-1}(Z_o + u/Z_o)$$

is computed. The next traversal computes

$$Z_2 = 2^{-1}(Z_1 + u/Z_1)$$

the third traversal Z_3, and so on. How, then, with one set of equations in an operation box is the desired notation indicated for each traversal? This is done in the following manner:

The contents of the operation box do not represent any specific traversal of the loop; hence an index is adopted that represents the general traversal; e.g., for the square root the operation box would contain

$$Z_{i+1} = 2^{-1}(Z_i + u/Z_i).$$

This index is the variable of induction that describes the inductive process, for if

$$Z_o = 1$$

$$Z_{i+1} = 2^{-1}(Z_i + u/Z_i) \qquad (i = 0,1,2\cdots)$$

$$\lim_{i\to\infty} Z_{i+1} = \sqrt{u}$$

then the process in question is completely described. Although the operation box does give the general expression, a means is needed for ascribing the appropriate value to the variable of induction for each traversal. This is done by the substitution box. Its function is to bring into agreement the notation of all quantities in which the variable of induction occurs with the notation that corresponds to a specific traversal of the loop. In other words, the substitution box makes the notation agree with the set of numbers upon which the succeeding boxes act during the forthcoming traversal of the loop.

A substitution is indicated as $a \to i$. It is interpreted as meaning that during the forthcoming interval and until a new substitution is made, everywhere that i occurs it is to be replaced by a. This first case is obvious enough. However, the substitutions are not restricted to constants replacing the variable of induction. In fact, the substitution often contains some function of i; e.g., the substitution of $i+1 \to i$ is used frequently. In the instance where the variable i occurs in both members of the substitution, it may conveniently be

interpreted in the following way: For the i's that occur to the left of the arrow the substitution from the preceding interval remains valid. The quantities on the left of the arrow will then not contain i anywhere in their expression and the substitution is made as described above; e.g., suppose that a substitution $a \rightarrow i$ has been indicated. After a sequence of boxes a new substitution $i+1 \rightarrow i$ is then indicated. First substitute a (the value of the immediately preceding substitution) for the i that occurs to the left of the arrow. The substitution now reads $a+1 \rightarrow i$ and we then proceed as in the above simple case. The next time the control returns to this substitution box it would be interpreted as $(a+1) + 1 = a + 2 \rightarrow i$.

Note that substitution boxes do not involve any arithmetic operations or transfers of numbers. They merely make changes in notation (transformations) such that the flow diagram indicates each stage of the computation in a precise manner. The substitution box is identified by a lower case Latin letter.

We continue with the square root example and illustrate the use of substitution boxes. The flow diagram for the process is:

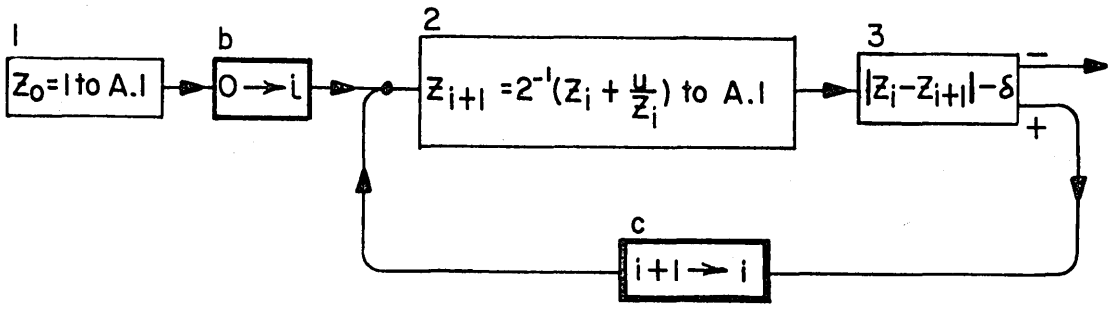


Figure 7.

(i) Operation box 1 initiates the induction by setting $Z_o = 1$ and storing it in A.1

(ii) Substitution box b indicates that everywhere in the following boxes up to the next substitution box wherever the variable of induction i occurs it is to be replaced by 0.

(iii) As a result of box b, operation box 2 indicates that

$$Z_1 = 2^{-1}(Z_o + u/Z_o)$$

and Z_1 is stored in A.1. The alternative box, box 3, indicates that the conditional transfer is to act upon $|Z_o - Z_1| - \delta$.

(iv) Box c is a substitution box of the second type discussed in the preceding paragraphs, namely the substitution $i+1 \longrightarrow i$. In the interval leading into this box the substitution $0 \longrightarrow i$ was valid. We replace the i to the left of the arrow by 0. The substitution is then $1 \longrightarrow i$. Operation box 2 now indicates

$$Z_2 = 2^{-1}(Z_1 + u/Z_1)$$

and alternative box 3 indicates $|Z_1 - Z_2| - \partial$. When substitution box c is again traversed, it will indicate $2 \longrightarrow i$, and the iterative process is advanced another step.

With the aid of the substitution box we have been able to describe completely and precisely the desired inductive process.

Throughout the flow diagram many symbols and notations are introduced (such as the variable of induction) that are relevant only in the flow diagram and often for only isolated parts of the flow diagram. These quantities are usually without any physical meaning apart from the process that they are describing in the flow diagram. These quantities are called bound variables. The Z's of the square root routine are such a variable. In passing from one section of the flow diagram to another these bound variables may take on new significance in describing some other process (such as the variable of induction i in the induction loop). The concept of the substitution box is extended to cover substitutions involving any bound variables.

There is one other box that is an integral part of the flow diagram; it is the assertion box. Its usefulness stems from the fact that at certain points of the flow diagram, bound variables may acquire a fixed value with a fixed meaning; e.g., in the square root diagram when $Z_{i+1} - Z_i < \partial$, then to sufficient accuracy $Z_{i+1} = \sqrt{u}$, where u is the number for which the square root is being extracted. Whenever such conditions are attained one may state this relationship by means of an assertion box. Hence, if we again consider the flow diagram of the square root routine and consider the negative branch which terminates the process, we have:

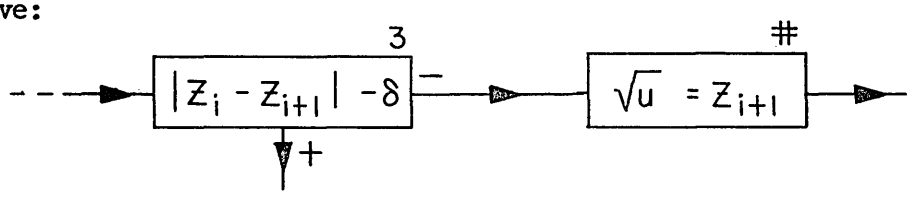


Figure 8.

When the control completes the process and proceeds along the negative branch, then Z_{i+1} is the desired $\sqrt{u}$. This fact is stated in the assertion box. The assertion box is identified by a crosshatch (#).

The discussion of the various boxes is completed by discussing the storage boxes. There are two kinds of storage with which we are concerned. In the first place, there will be a set of numbers that originate with the problem and will remain unchanged throughout the course of the problem. The storage necessary for this type of quantity is called static storage. The storage requirement that originates from computation within the problem is called dynamic storage. We are not concerned here with the static storage as it is unchanged throughout a problem. However, at certain points along the flow diagram it is convenient to indicate the contents of the dynamic storage concerned with the local computation about to be performed. The storage boxes are connected to the flow lines of the diagram by dotted lines. (These boxes are not an integral part of the flow diagram.) In Figure 9 the flow diagram for the square root routine is shown complete with storage boxes.

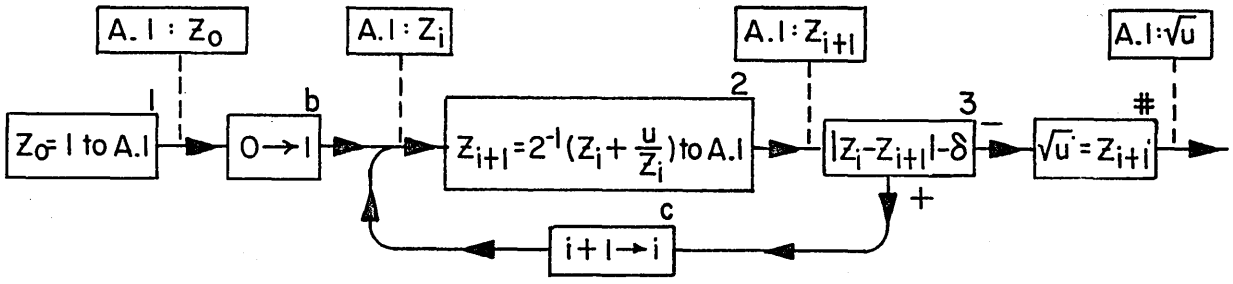


Figure 9.

The examples indicate a complete set of storage boxes indicating all relevant changes. In actual practice, however, the procedure will be to indicate storage boxes only when they are useful and needed for clarity.

The substitutions indicated by the substitution boxes are also valid for the storage boxes. Consider Figure 9: on either side of the substitution box b, a storage box is indicated. The storage box to the left of Box b shows that $A.1:Z_0$, while the box to the right of Box b shows that $A.1:Z_i$. If, however, 0 is substituted for the i as is indicated by Box b, the two storage boxes agree, as they should at this time. Similarly, the storage box immediately to the left of Box 2 is brought into agreement with the storage box to the right of Box 2 each time substitution box c is traversed.

Let us recapitulate at this time:

(i) The operation box indicates the arithmetic operations and the transfers of numbers that are to take place. In the arithmetic operations the relationships are expressed by equality signs; i.e., $y = ax^2 + bx + c$, $y = f(x,t)$, or some other such expression. The quantity that is being formed is always written as the left member of the equation while all of the known values are included in the right member of the equation. The operation box has an accompanying identifying letter or number. Arabic numerals are used to identify such boxes.

(ii) The alternative box is associated with the conditional transfer. The conditional transfer acts upon the quantity or quantities indicated in the box; and the control follows the positive exit or negative exit, according as the transfer is effective or not. The address of the conditional transfer instruction must be the address corresponding to the positive exit of the box; and immediately after the conditional transfer instruction is the sequence that the negative branch will follow.

(iii) The substitution box indicates changes that occur in bound variables. These are changes in notation (or transformations, if you like) and they do not involve any arithmetic operations or transfers of numbers. The substitution box is usually concerned with the variable of induction in an inductive process; and by attributing successive values to the variable of induction wherever it occurs in the general expression of the process, the induction is completely described. The contents of the substitution box are indicated with an arrow, such as $a \longrightarrow i$ where this is read as substitute a for i.

(iv) The assertion box states an existing condition. At certain points of the diagram a bound variable may acquire a fixed value. The assertion box merely states this fact.

(v) The storage box indicates the relevant storage locations of the quantities needed for computation in a sequence of operation boxes.

We have now completed the discussion of the important components of the flow diagram. There are certain refinements to the flow diagram that will be introduced as the need for them arises in the forthcoming examples.

Problem 4

We propose to extract the square root of a number u by means of the iterative process

$$Z_{i+1} = 2^{-1}(Z_i + u/Z_i) \qquad (i = 0,1,2 \cdots)$$

$$\lim_{i \to \infty} Z_{i+1} = \sqrt{u}$$

Since the computer requires that all numbers be in the range $-1 \leq x < 1$, u is restricted so that $0 \leq u < 1$. At each step of the iterative process the division u/Z_i must be performed. Since $u < Z_i$ must hold for this to be a legal operation, it must either be shown that this condition does hold or else the necessary adjustments must be made (by coding) such that the condition is true.

We propose to show the former as follows:

$$Z_{i+1} = 2^{-1}(Z_i + u/Z_i) \qquad \text{(Eq. 1)}$$

$$Z_{i+1} - \sqrt{u} = Z_i/2 + u/2Z_i - \sqrt{u}$$

$$= (1/2Z_i)(Z_i^2 - 2Z_i\sqrt{u} + u)$$

$$Z_{i+1} - \sqrt{u} = (1/2Z_i)(Z_i - \sqrt{u})^2 \qquad \text{(Eq. 2)}$$

Assume $Z_o > 0$, then from (Eq. 1) all $Z_i > 0$. Since all $Z_i > 0$, the right member of (Eq. 2) is positive; hence the left member is positive and

$$Z_{i+1} > \sqrt{u} > u.$$

If $Z_o > u$, which is done by setting $Z_o = 1$, then all

$$Z_i > u$$

and the quotient u/Z_i will not exceed the allowed limits of the computer. In choosing $Z_o = 1$, Z_1 is formed as

$$Z_1 = 2^{-1} + 2^{-1}u$$

which is used as the first step of the inductive scheme.

We must ascertain which Z_{i+1} is to terminate the induction. This could be done by determining the number of iterations necessary to compute the worst case, namely $u = 2^{-39}$, and then traverse the induction loop that fixed number of times, irrespective of the size of u. Let us, however, do something slightly different.

We know $\underline{u}$ to within an error Δu where

$$\Delta u = 2^{-1} \cdot 2^{-39} = 2^{-40}$$

as this is an error introduced by the physical size of the computer
The error $\Delta\sqrt{u}$ in determining $\sqrt{u}$ is found as

$$u + \Delta u = (\sqrt{u} + \Delta\sqrt{u})^2$$

$$u + \Delta u = u + 2\sqrt{u}\Delta\sqrt{u}$$

neglecting second order terms. Hence,

$$\Delta\sqrt{u} = \frac{\Delta u}{2\sqrt{u}}$$

For our case

$$\epsilon_u \equiv \Delta\sqrt{u} = 2^{-41}/\sqrt{u}, \quad u \neq 0$$

For $u = 0$ we have

$$\Delta u = (\Delta\sqrt{u})^2$$

$$2^{-40} = (\Delta\sqrt{u})^2$$

$$\epsilon_o \equiv \Delta\sqrt{u} = 2^{-20}$$

The error ϵ_u varies from 2^{-20} when $u = 0$ to 2^{-41} when $u = 1$.
The iterative process should certainly stop whenever

$$Z_{i+1} - \sqrt{u} \leq \epsilon_u.$$

We propose to show that whenever

$$Z_i - Z_{i+1} \leq 2^{-21}$$

then

$$Z_i - \sqrt{u} \leq \epsilon_u$$

and the iterative process is complete.

First let us show that

$$Z_{i+1} - \sqrt{u} \leq 1/2(Z_i - \sqrt{u}).$$

Since all

$$Z_i \geq \sqrt{u} > u$$

then

$$u/Z_i \leq \sqrt{u}$$

$$Z_i + u/Z_i \leq Z_i + \sqrt{u}$$

$$1/2(Z_i + u/Z_i) \leq 1/2(Z_i + \sqrt{u})$$

The left-hand side is by the definition of the iterative process equal to Z_{i+1}; hence

$$Z_{i+1} \leq 1/2(Z_i + \sqrt{u})$$

$$Z_{i+1} - \sqrt{u} \leq 1/2(Z_i - \sqrt{u})$$

From this it follows that

$$Z_{i+1} - \sqrt{u} \leq Z_i - Z_{i+1}.$$

If the iterative process is terminated when

$$Z_i - Z_{i+1} \leq 2^{-21}$$

then

$$Z_{i+1} - \sqrt{u} \leq 2^{-21}$$

and adding these two inequalities gives

$$Z_i - \sqrt{u} \leq 2^{-20}.$$

Hence, from (Eq. 2)

$$(Z_i - \sqrt{u})^2 = 2Z_i(Z_{i+1} - \sqrt{u}) \leq 2^{-40}$$

we define

$$\epsilon_i \equiv Z_{i+1} - \sqrt{u}$$

then

$$2Z_i\epsilon_i \leq 2^{-40}.$$

Since

$$\sqrt{u}/Z_i < 1$$

$$2Z_i(\sqrt{u}/Z_i)\epsilon_i \leq 2^{-40}$$

and

$$\epsilon_i \leq 2^{-41}/\sqrt{u}.$$

This completes the proof, for if the induction is stopped when

$$Z_i - Z_{i+1} \leq 2^{-21}$$

then

$$Z_{i+1} - \sqrt{u} = \epsilon_i \leq \epsilon_u = 2^{-41}/\sqrt{u}$$

as is desired.

Since the flow diagram has previously been discussed in detail, we turn directly to the coding which is done with the aid of the flow diagram.

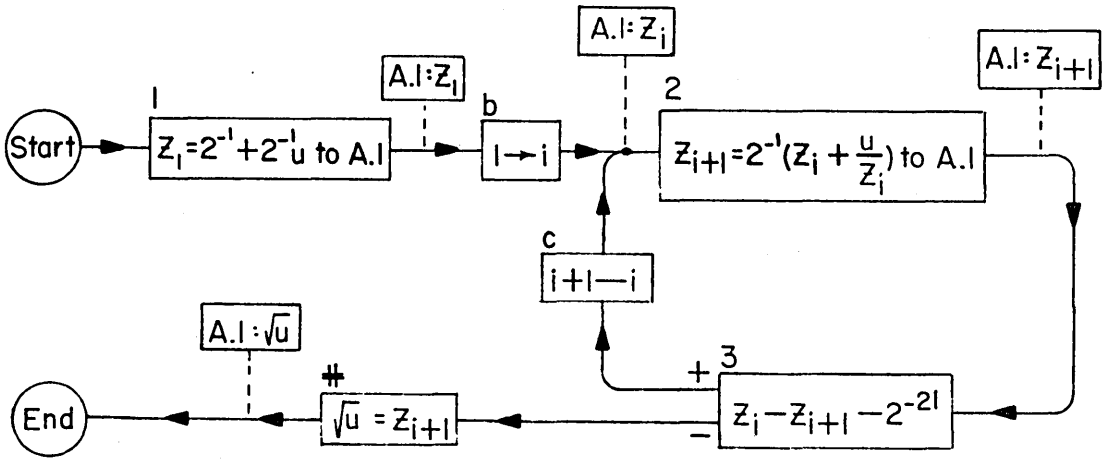


Figure 10.

Storage locations are needed for u, for the number 2^{-1}, for the number 2^{-21}, and a temporary location for intermediate results. These are designated as

B.1: u B.3: 2^{-21}

B.2: 2^{-1} B.4:

and the Z's are stored in A.1; hence utilizing the same location for the successive iterates.

In the initial coding each box is treated independently. The coding is:

Box 1.

1.	m → Ac	B.1	u to R2	
2.	R 1		$2^{-1}(u)$ in R2	
3.	m → Ah	B.2	$Z_I = 2^{-1}(1+u)$ in R2	
4.	A → m	A.1		Z_1 to A.1

Box 2.

1.	m → Ac	B.1	u to R2	
2.	÷	A.1	u/Z_i in R4	
3.	Q → m	B.4		u/Z_i to B.4
4.	m → Ac	B.4	u/Z_i to R2	
5.	m → Ah-	A.1	$u/Z_i - Z_i$ in R2	

6.	R 1		$(Z_{i+1} - Z_i) = 2^{-1}(u/Z_i - Z_i)$ in R2
7.	A⟶m	B.4	$Z_{i+1} - Z_i$ to B.4
8.	m⟶Ah	A.1	$Z_{i+1} = 2^{-1}(u/Z_i - Z_i) + Z_i = 2^{-1}(u/Z_i + Z_i)$ in R2
9.	A⟶m	A.1	Z_{i+1} to A.1

Box 3.

1.	m⟶Ac-	B.4	$Z_i - Z_{i+1}$ to R2
2.	m⟶Ah-	B.3	$Z_i - Z_{i+1} - 2^{-21}$ in R2
3.	C	Box 2,1	
4.	Stop		

In Box 2, observe how Z_{i+1} is formed. It is known that $Z_i < 1$ and $u/Z_i < 1$, but it does not follow that $Z_i + u/Z_i < 1$. Z_{i+1} could be formed by first obtaining $2^{-1}(u/Z_i)$ and then adding $2^{-1}Z_i$ to it. This, however, would require additional orders as $2^{-1}Z_i$ would have to be formed and stored before proceeding to $2^{-1}(u/Z_i)$, in order that the addition of the two terms could take place at this time. It is more efficient to form Z_{i+1} in the following way: since Z_i and u/Z_i are both positive, the difference

$$u/Z_i - Z_i < 1.$$

Therefore, the difference is formed and shifted right 1 to obtain

$$2^{-1}(u/Z_i - Z_i).$$

Observe that

$$Z_{i+1} - Z_i = 2^{-1}(u/Z_i - Z_i). \qquad \text{(Eq. 3)}$$

If Z_i is now added to both members, then

$$Z_{i+1} = 2^{-1}(u/Z_i - Z_i) + Z_i = 2^{-1}(u/Z_i + Z_i).$$

Equation 3 above expresses the negative of the quantity $Z_i - Z_{i+1}$ desired for the discrimination in Box 3. $Z_{i+1} - Z_1$ is stored in B.4 so that it will be directly available for Box 3. In fact, if Z_{i+1} had not been formed by first forming and saving the quantity $Z_{i+1} - Z_i$, Z_{i+1} could not have been stored in A.1, as Z_i would then still be needed for Box 3. This would mean that Z_{i+1} would be sent to B.4 until the completion of Box 3 at which time it could be sent to A.1. Again, this would have required additional coding.

In pairing the instructions into words, we start the coding at Word 1. No connecting instructions are needed between the boxes. The total number of instructions is:

Box 1:	4 instructions
Box 2:	9
Box 3:	3
and a "stop" instruction :	1
total :	17 instructions

which require 9 words. Five words of storage are needed which account for Words 10 through 14. The sequence is:

1.	m → Ac	10		R 1	
2.	m → Ah	11		A → m	13
3.	m → Ac	10		+	13
4.	Q → m	14		m → Ac	14
5.	m → Ah-	13		R 1	
6.	A → m	14		m → Ah	13
7.	A → m	13		m → Ac-	14
8.	m → Ah-	12		C	3
9.	Stop				
10.	u				
11.	2^{-1}				
12.	2^{-21}				
13.					
14.					

The conditional transfer instruction in the right half of Word 8 transfers to the first instruction of Box 2. When the instructions are paired, the first word of Box 2 becomes the left-hand instruction of Word 3; hence the conditional transfer instruction is the transfer to the left-hand instruction of Instruction-pair 3.

Before discussion of Problems 5 and 6, on the conversion of numbers from one base system to another, some remarks should be made on the form of input and output data. Although the computer operates with numbers expressed in the binary base, the human operator is apt to find that he has, through years of exposure, become firmly bound to the decimal number system. It is then certainly to the advantage of the operator to find some means of communication to and from the machine that can be expressed in decimal numbers. Before discussing the problems related to such a scheme, we first make a few remarks on the input-output problem in general.

Even though we are at present mainly interested in input and output data in the decimal number system we do not wish to exclude input and output as true binary numbers. In fact, whenever any data is printed for subsequent consumption by the computer it should obviously remain in the binary base; furthermore, it is both convenient and simple to have instruction words coded in their true binary form. As we have more experience with the computer and with binary numbers, our dependence upon the decimal system may wane, and we may find ourselves operating solely with binary numbers. We first consider the input-output in the binary system and from that develop the scheme for handling decimal numbers.

It is not practical to have the keyboard of the tape punch or the type bars of the printer operate in true binary notation, for this would mean that forty characters would have to be printed or punched per word; and even though one needs to recognize only 0's and 1's, it is difficult to examine words forty characters long. Let us arrange the bigits into groups of, say, three or four bigits and specify a character to identify each unique combination. We choose groups of three or four since these correspond to eight and sixteen unique characters, respectively, which are each fairly close in number to the usual ten characters in the decimal system. Such choices shorten the word length from forty bigits to either thirteen or ten characters, accordingly. For the present discussion, we fix upon groups of fours (tetrads) and identify each tetrad by a single character. Since sixteen characters are needed, we are really operating in the hexadecimal (16) number base. For those tetrads that have single decimal digit equivalents, the corresponding decimal characters are used to identify them. The remaining six tetrads are identified by the letters A,B ••• F. Table II shows the hexadecimal characters with their binary tetrad equivalents.

TABLE II

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

The keyboard of the tape punch and the type bars of the page printer have sixteen characters. In tape preparation, the conversion from hexadecimal to binary is effected directly by the punching equipment. When one of the sixteen keys of the keyboard is depressed the punch is set up

so that it punches the binary equivalent on the tape (in a tetrad of bigits). Similarly, when printing is desired a tetrad of bigits actuates the type bars and the hexadecimal equivalent is printed.

To return to the decimal input-output problem, we have at our disposal the first ten ordinal characters of the hexadecimal notation which are identical to the ten decimal characters 0,1 ··· 9. To prepare a tape of decimal information, we depress the keys corresponding to the individual decimal characters of the desired number. The punch converts the decimal characters into tetrads of bigits which give a "coded-decimal" representation of the number. The coded-decimal form of a number is not identical to the number's true binary equivalent. For example, consider the decimal number 512. Its coded-decimal representation is 0101 0001 0010 while its true binary representation is 1000000000. There is a very simple algorithm by which we can convert the coded-decimal number into its true binary equivalent. The output problem involves the converse. We need an algorithm by which a true binary number can be converted into its coded-decimal equivalent so that the printer may produce the number in its decimal form. We consider first the input problem--the conversion of a coded-decimal number into its true binary equivalent.

Problem 5

Since a tetrad of bigits is used to represent a single decimal digit, and since the standard word length is forty bigits, each word is comprised of ten tetrads. The first tetrad on the left is used to indicate the sign of a number. This means that the computer is able to store a nine digit coded-decimal number with its sign. In following the present sign representation for binary numbers, the tetrad 0000 designates a positive number and the tetrad 1111 designates a negative number. Negative coded-decimal numbers are represented as signed numbers rather than as complement numbers as used for negative binary numbers. As examples, a positive and a negative coded-decimal number are shown.

\+ .765432109: 0000 0111 0110 0101 0100 0011 0010 0001 0000 1001

\- .543010678: 1111 0101 0100 0011 0000 0001 0000 0110 0111 1000

The conversion of coded-decimal number a' into the true binary number a may be performed as follows: The absolute value of a' is converted and then the sign is determined. The absolute value is obtained by neglecting the sign tetrad of a'. The sign tetrad comprises bigits (0—3); hence

$$|a'| = \text{bigits } (4—39) \text{ of } a' \qquad 0 \leq |a'| < 1 \qquad \text{(Eq. 4)}$$

Recall that each decimal digit treated as an integer is represented by its true binary equivalent in the coded-decimal notation. The tetrad represented by the bigits

$$(4i \text{---} 4i+3) \qquad (i = 1,2 \cdots 9)$$

beginning at the left of $\underline{a}'$ represents the decimal digit $\underline{w}_i$. The first tetrad from the left corresponds to the 10^{-1} position, the second tetrad to the 10^{-2}, and so on. Therefore,

$$(4i \text{---} 4i+3) = 10^{-i} w_i \quad (i = 1,2 \cdots 9) \qquad \text{(Eq. 5)}$$

and furthermore,

$$|a'| = \sum_{i=1}^{9} 10^{-i} w_i; \qquad \text{(Eq. 6)}$$

e.g., $|a'| = .0111\ 0101\ 0110\ 1001\ 0001\ 0000\ 0100\ 0011\ 1000 = .756910438 =$

$$= \sum_{i=1}^{9} 10^{-i} w_i = 7/10^1 + 5/10^2 + 6/10^3 + 9/10^4 + 1/10^5 + 0/10^6 + 4/10^7 + 3/10^8 + 8/10^9.$$

Since each tetrad is, by itself, in true binary form if considered as an integer, one method of converting the number is to divide each tetrad by its appropriate power of 10 (expressed, of course, as a binary number) and add the results of all such divisions; e.g., .25 is .0010 0101 in coded-decimal form and to convert this to a true binary we perform the steps

$$\frac{0010}{1010} + \frac{0101}{(1010)(1010)} = \frac{0010}{1010} + \frac{0101}{1100100} = 0.01,$$

and 0.01 is the true binary form of the decimal number .25. However, let us do something slightly different. Multiply and divide the right member of (Eq. 6) by $10^9 \cdot 2^{-39}$. This gives

$$|a'| = \frac{\sum_{i=1}^{9} 10^{9-i} w_i \cdot 2^{-39}}{10^9 \cdot 2^{-39}} \qquad \text{(Eq. 7)}$$

The conversion may now be effected by multiplying each tetrad w_i by $10^{9-i} \cdot 2^{-39}$, adding the products of all such multiplications, and then dividing the resultant sum by $10^9 \cdot 2^{-39}$. Each tetrad w_i has a co-factor, 10^{9-i}, which is ten greater than the cofactor of the immediately succeeding tetrad. The conversion from the coded decimal number $\underline{a}'$ to the binary number $\underline{a}$ is then described by the following inductive process.

$$
\begin{aligned}
a_o &= 0 \\
a_1 &= 10a_o + 2^{-39}w_1 \\
a_2 &= 10a_1 + 2^{-39}w_2 \\
&\vdots \\
a_{i+1} &= 10a_i + 2^{-39}w_{i+1} \\
&\vdots \\
a_9 &= 10a_8 + 2^{-39}w_9 \\
a &= \frac{a_9}{10^9 \cdot 2^{-39}}
\end{aligned}
$$

The tetrads are isolated with the aid of the left shift order. First the magnitude of $\underline{a}'$ is formed by bringing $\underline{a}'$ into R2 (the Accumulator) and effecting a left shift of 4. The portion of $\underline{a}'$ left in R2 is $|\underline{a}'|$. R4 (the quotient register) is then set to $\underline{0}$. A subsequent left shift by 4 now has the effect that $\underline{w}_1$ appears in the extreme right of R4. The first tetrad $\underline{w}_1$ has thus been separated from the remaining tetrads, and since $\underline{w}_1$ appears in the extreme right of R4 it is $2^{-39}w_1$, as desired. $\underline{a}_1$ is now formed as:

$$a_1 = 10a_o + 2^{-39}w_1.$$

$\underline{w}_2$ is isolated in the same manner as was $\underline{w}_1$ and then $\underline{a}_2$ is formed, and so on, until $\underline{a}_9$ is formed. A multiplication by ten at each step cannot directly be done as this is an illegal operation, since allowed multiplication factors must be in the range $|x| < 1$. However, a multiplication by ten may be simulated by doing a series of left shifts and an addition for

$$10a_i = 2^3a_i + 2a_i.$$

The inductive process may be written as:

$$
\begin{aligned}
a_o &= 0 \\
a_{i+1} &= 2^3a_i + 2a_i + 2^{-39}w_{i+1} \quad (i = 0,1 \cdots 8) \\
a &= \frac{a_9}{10^9 \cdot 2^{-39}}
\end{aligned}
$$

The w_i's are also formed by an inductive scheme where

$$a'_o = |a'|$$

$$w_1 = 2^4 a'_o \text{ (integer part)} \qquad a'_1 = 2^4 a'_o \text{ (fractional part)}$$

$$w_2 = 2^4 a'_1 \text{ (integer part)} \qquad a'_2 = 2^4 a'_1 \text{ (fractional part)}$$

$$\vdots \qquad\qquad\qquad \vdots$$

$$w_{i+1} = 2^4 a'_i \text{ (integer part)} \qquad a'_{i+1} = 2^4 a'_i \text{ (fractional part)}$$

$$\vdots$$

$$w_9 = 2^4 a'_8 \text{ (integer part)} \qquad a'_9 = 0 = 2^4 a'_8 \text{ (fractional part)}$$

There remains finally the determination of the appropriate sign to affix to the true binary number $\underline{a}$. It is recalled that the extreme left tetrad is reserved to denote the sign of $\underline{a}'$. A sufficient method is to examine the leftmost bigit of $\underline{a}'$. If this is $\underline{0}$, $a' \geq 0$ and $\underline{a}$ is to be positive. If the leftmost bigit of $\underline{a}'$ is a $\underline{1}$, then $a' < 0$ and $\underline{a}$ is to be formed as a complement.

The only operations that are performed on $\underline{a}'$, the coded-decimal number, are a series of left shifts by 4. To simplify the coding and flow diagram, the number $\underline{a}'$ is treated as though the binary point is immediately left of the first bigit position. In other words, the normal sign bigit (the 2^0 position) is treated as a numerical bigit, in fact the 2^{-1} bigit position. After the first left shift of $\underline{a}'$ by 4, the first significant bigit of w_1 is in the leftmost bigit position. After w_1 is isolated by a left shift of 4 places, the first bigit of w_2 is in the leftmost bigit position, and so on with the remaining w's. The consequence of treating $\underline{a}'$ in this fashion is discussed in the coding of the problem.

Since nine tetrads must be operated upon, the induction loop must be traversed nine times. The method used for determining when to stop in the induction is essentially to discriminate upon the quantity

$$I - i \qquad (I = 8; \text{ and } i = 1, 2 \cdots 9, \text{ successively})$$

When $i = 9$ (which corresponds to the completion of the 9^{th} traversal of the induction loop), the discrimination on $(I - i)$ becomes negative for the first time and the induction process is stopped as desired.

The storage requirements are as follows: $\underline{a}'$ (the coded-decimal number) is initially in the memory at address A.1. When $\underline{a}$ (the true binary number) is formed it is to be stored at A.2. Storage is needed for the numbers $\underline{0}$ and $10^9 \cdot 2^{-39}$. These are stored in C.1 and C.2, respectively. Four intermediate storage locations are needed during the course of the conversion. These are designated as B.1, B.2, B.3, and B.4.

We are now ready to draw the flow diagram and do the coding. The flow diagram is shown in Figure 11.

In the flow diagram, Box 1 sets up the initial steps of the inductions over $\underline{a}'$ and $\underline{a}$. It sets

$$a'_o = |a'| = 2^4 a' \text{ (fractional part)}$$
$$a_o = 0$$

as is indicated in the description of the induction on the preceding page. This box also sets the upper limit $I=8$ of the induction. Box $\underline{b}$, Box 2, and Box $\underline{c}$ complete the description of the induction. Box 2 forms

$$a'_{i+1} = 2^4 a'_i \text{ (fractional part)}$$
$$2^{-39} w_{i+1} = 2^4 a'_i \text{ (integer part)}$$
$$a_{i+1} = 10 a_i + 2^{-39} w_{i+1}$$

with Boxes $\underline{b}$ and $\underline{c}$ ascribing the appropriate values to the variable of induction $\underline{i}$. In Box 3, the conditional transfer box, the quantity upon which the discrimination is made is more conveniently $I-(i+1)$ rather than $I-i$ as previously discussed. In discriminating upon $I-(i+1)$, $\underline{i}$ assumes the values $0,1 \cdots 8$. This is then equivalent to the discrimination $I-i$ where $i = 1,2 \cdots 9$. Box 4 forms $|\underline{a}|$ by dividing $\underline{a}_9$ by $10^9 \cdot 2^{-39}$. Finally, Boxes 5, 6, and 7 are concerned with determining the correct sign for the true binary number $\underline{a}$.

The coding is:

Box 1.

1.	m⟶Ac	A.1	a' to R2	
2.	L	4	$a'_o = \|a'\| = 2^4 a'$	in R2
3.	A⟶m	B.1		a'_o to B.1
4.	a⟶Ac	0	$a_o = 0$ to R2	
5.	A⟶m	B.2		a_o to B.2
6.	a⟶Ac	$8 \cdot 2^{-11}$	$I = 8 \cdot 2^{-11}$ to R2	
7.	A⟶m	B.3		I to B.3

CODED DECIMAL to BINARY CONVERSION

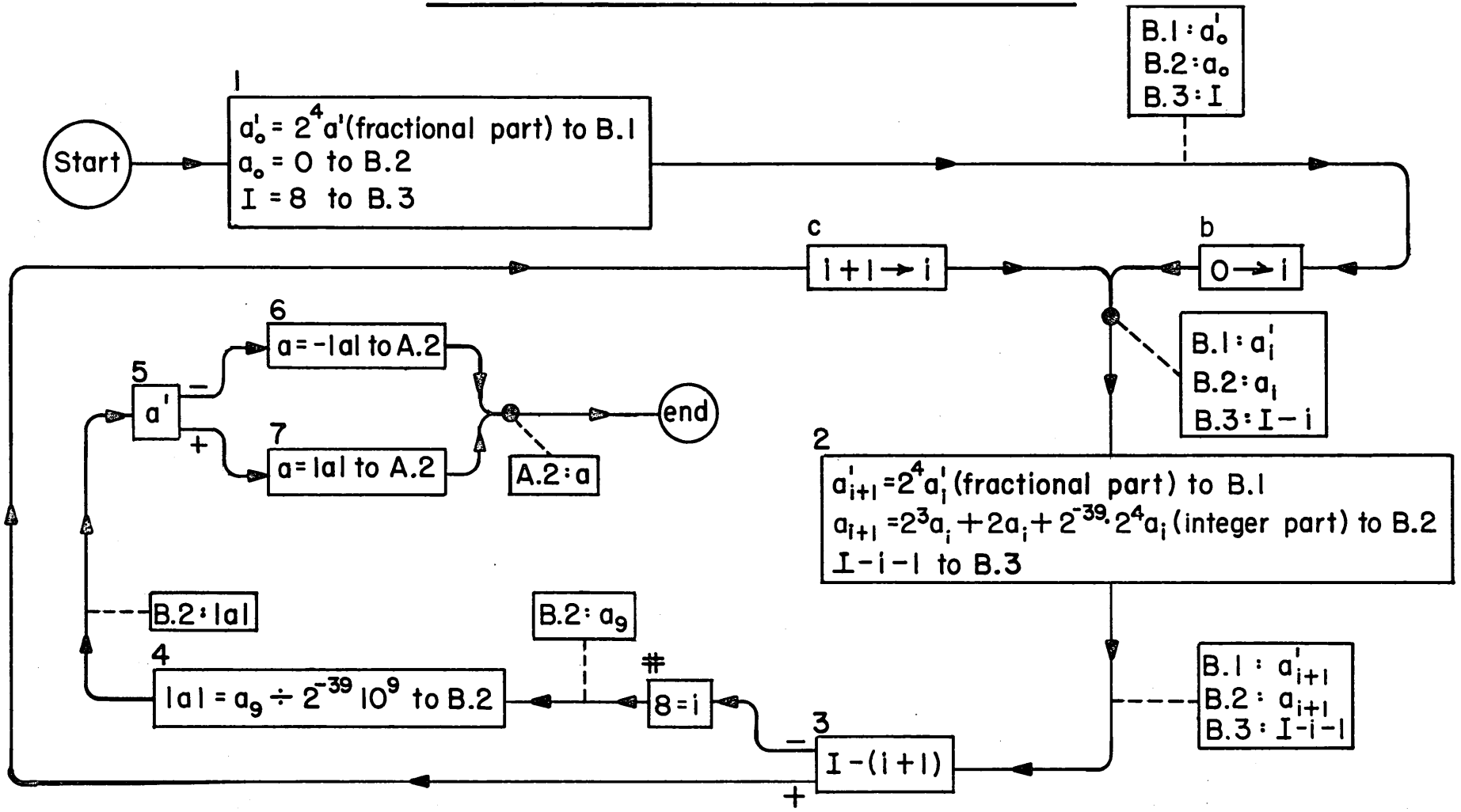


FIG. 11

STORAGE

A.1 : a'
A.2 : a (when formed)

C.1 : 0
C.2 : $10^9 \cdot 2^{-39}$

B.1 : —
B.2 : —
B.3 : —
B.4 : —

Box 2.

1.	m⟶Ac	B.1	a'_i to R2	
2.	m⟶Q	C.1	0 to R4	
3.	L	4	$2^{-39}w_{i+1} = 2^4 a'_i$(int.pt.) to R4	
			$a'_{i+1} = 2^4 a'_i$(fract.pt.) in R2	
4.	A⟶m	B.1		a'_{i+1} to B.1
5.	Q⟶m	B.4		$2^{-39}w_{i+1}$ to B.4
6.	m⟶Ac	B.2	a_i to R2	
7.	L	2	$2^2 a_i$ in R2	
8.	m⟶Ah	B.2	$2^2 a_i + a_i$ in R2	
9.	L	1	$2^3 a_i + 2a_i$ in R2	
10.	m⟶Ah	B.4	$a_{i+1} = 2^3 a_i + 2^{-39}w_{i+1}$ in R2	
11.	A⟶m	B.2		a_{i+1} to B.2
12.	m⟶Ac	B.3	I-i to R2	
13.	a⟶Ah	$-1 \cdot 2^{-11}$	I-i-1 in R2	
14.	A⟶m	B.3		I - (i+1) to B.3

Box 3.

1.	m⟶Ac	B.3	I - (i+1) to R2
2.	C	2,1	

Box 4.

1.	m⟶Ac	B.2	a_9 to R2	
2.	÷	C.2	$\|a\| = a_9/10^9 \cdot 2^{-39}$ in R4	
3.	Q⟶m	B.2		|a| to B.2

Box 5.

1.	m⟶Ac	A.1	a' to R2
2.	C	7,1	

Box 6.

1.	m⟶Ac-	B.2	$a = -\|a\|$ to R2	
2.	A⟶m	A.2		a to A.2
3.	Stop			

Box 7.

1.	m⟶Ac	B.2	a = |a| to R2	
2.	A⟶m	A.2		a to A.2
3.	T	6,3		

In the coding in Box 1 the a⟶Ac order has been used in Instructions 4 and 6. Recall that this order replaces the number in R2 by the twelve address bigits of the instruction; i.e., R2 is cleared to 0's and the twelve address bigits of the instruction a⟶Ac are added into R2 into positions 0 through 11. In Instruction 4, the number 0 is desired in R2; hence the instruction a⟶Ac has 0 as its address. Instruction 6 forms I=8. Since the integer 8 cannot be stored, we store $8\cdot 2^{-m}$ where m is at least 4 so that $8\cdot 2^{-m} < 1$. The a⟶Ac may be utilized to form I and save the word of storage that would be needed initially to store the $8\cdot 2^{-m}$. Since I is formed in this manner we have the freedom of choosing $I=8\cdot 2^{-4}$, $8\cdot 2^{-5}$ ... $8\cdot 2^{-11}$. I is chosen as $8\cdot 2^{-11}$ for this case. In Box 2 where (I-i-1) is formed the 1 that is subtracted must have the same cofactor 2^{-m} as does the I; hence to do this the instruction a⟶Ah is used with the associated address being $-1\cdot 2^{-11} \equiv$ **FFF in hexadecimal notation.**

In Box 2, the first five instructions are concerned with forming $2^{-39}w_{i+1}$ and a'_{i+1}. Before the left shift of 4 is executed (Instruction 3), R4 must be set to 0. This is done because $2^{-39}w_{i+1}$ is needed by itself and if R4 were not 0 the left shift of 4 would place $2^{-39}w_{i+1}$ into R4, but whatever number y that had been in R4 at the time of the shift would merely be shifted left 4 places and R4 would contain $2^4 y + 2^{-39}w_{i+1}$ rather than the desired $2^{-39}w_{i+1}$. For clarity, we show in the following example how a left shift of 4 isolates each tetrad. Suppose the number 0.98 is to be converted into true binary form. In coded-decimal form it first appears in R2 as the following sequence of tetrads:

0.98...	0000 1001 1000...
	(+) (9) (8)

Normally, the leftmost bigit is reserved for the sign bigit. Inasmuch as no arithmetic operations are to be performed on a'_i except for shifting to the left, it is convenient to disregard the usual function of the leftmost position as corresponding to the sign bigit. The aim at this point is merely to separate successively the various tetrads. The first

left shift of 4 produces in R2

$$a'_0 = |a'| = 1001\ 1000\cdots$$

The next time a left shift of 4 occurs, R2 contains

$$a'_1 = 1000\ \cdots\cdot\ \cdots\cdot$$

and R4 has $00\cdots0\cdots\cdots1001$. Since $2^{-39}w_i$ is desired, one sees that in R4 the usual binary point convention is restored; namely, after the first bigit position. Hence the tetrad in R4 can participate in normal arithmetic operations.

If one had adhered strictly to the sign convention for R2, some needless complications in the coding would have resulted.

Also in Box 2 we see that $2^{-39}w_{i+1}$ in R4 must be sent to temporary storage (Instruction 5) before

$$a_{i+1} = 2^3 a_i + 2a_i$$

is formed in R2 (Instructions 6 through 9). This is necessary as R4 shifts in concert with R2, hence altering its contents.

The final coding is left as an exercise for the student, and the conversion of a true binary number into its coded-decimal equivalent is considered.

Problem 6

When the formal calculation of a problem on the computer is finished the desired answers are to be converted from true binary form into coded-decimal notation so that the teletype page printer produces the true decimal representation of the desired numbers.

We develop this conversion scheme in the following way: The true binary number $\underline{a}$ is to be converted into its coded-decimal equivalent $\underline{a}'$. Since coded-decimal numbers are stored as signed numbers rather than complement numbers, $|a|$ is first converted to $|a'|$, and then the appropriate sign is prefixed. Since $|a| < 1$, it has a decimal equivalent which may be written as

$$|a| = 10^{-1}w_1 + 10^{-2}w_2 + \cdots + 10^{-9}w_9. \qquad \text{(Eq. 8)}$$

The problem is to determine the $\underline{w}$'s. If $10|a|$ (multiplication by ten in binary form) is formed, there is an integer part and a fractional part to the number. We see from (Eq. 9) that the integer part corresponds to the decimal digit $\underline{w}_1$.

$$10|a| = w_1 + 10^{-1}w_2 + \cdots + 10^{-8}w_9. \qquad \text{(Eq. 9)}$$

If the fractional part of $10|a|$ is now multiplied by ten, the integer part is just $\underline{w}_2$, etc. The following inductive process to produce each of the decimal digits is used:

$$\begin{aligned} a_o &= |a| \\ 10a_o &= w_1 + a_1 \\ 10a_1 &= w_2 + a_2 \\ &\vdots \\ 10a_i &= w_{i+1} + a_{i+1} \\ &\vdots \\ 10a_8 &= w_9 + a_9 \end{aligned}$$

where the $\underline{w}_i$'s are the binary equivalents of the decimal digits. In the coded-decimal representation, each decimal digit is represented as a tetrad of bigits; hence each $\underline{w}_i$ is separated as a tetrad of bigits. This is done by multiplying by ten in the following way:

$$10a_i = 2^4(2^{-1}a_i + 2^{-3}a_i).$$

The left shift of 4 separates the integer part (w_{i+1}) from the fractional part (a_{i+1}) by shifting w_{i+1} into the quotient register (R4) as a tetrad and leaving the fractional part in the accumulator (R2).

The coded-decimal number $\underline{a}'$ is formed by the following inductive process:

$$a'_o = \begin{cases} 0 \text{ if } \underline{a} \geq 0 \\ F \cdot 2^{-39} \text{ if } \underline{a} < 0 \end{cases}$$

$$a'_1 = 2^4 a'_o + 2^{-39} w_1$$

$$a'_2 = 2^4 a'_1 + 2^{-39} w_2$$

$$\vdots$$

$$a'_{i+1} = 2^4 a'_i + 2^{-39} w_{i+1}$$

$$\vdots$$

$$a'_9 = 2^4 a'_8 + 2^{-39} w_9$$

$$a' = a'_9$$

Note that each $\underline{w}_i$ is desired as $2^{-39} w_i$, which is precisely the quantity that appears on the right in R4 as a result of the left shift of 4 places.

As in the previous problem the induction has nine steps; hence the same index representation is used. The flow diagram is shown in Figure 12. The required storage is indicated on the flow diagram. The coding is:

Box 1.

1.	m⟶Ac	A.1	$\underline{a}$ to R2
2.	C	3,1	

Box 2.

1.	m⟶Ac	C.1	$a'_o = F \times 2^{-39}$ to R2
2.	A⟶m	B.1	a'_o to B.1

Box 4.

1.	m⟶AcM	A.1	$2^{-1} a_o = \lvert a \rvert$ to R2
2.	A⟶m	B.2	$2^{-1} a_o$ to B.2
3.	a⟶Ac	$8 \cdot 2^{-11}$	I = 8 to R2
4.	A⟶m	B.3	$\underline{I}$ to B.3

BINARY to CODED DECIMAL CONVERSION

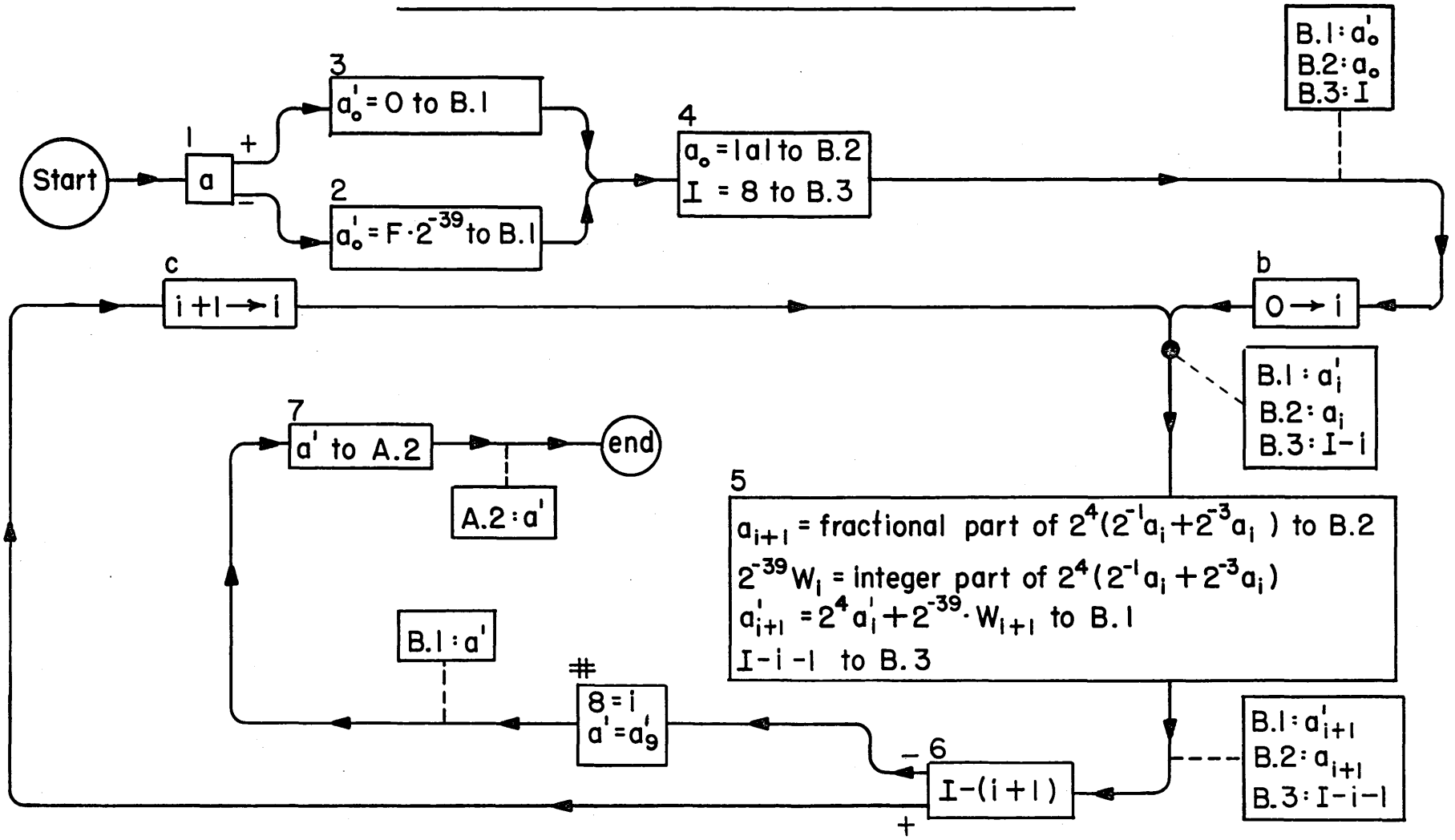


FIG. 12

STORAGE

A.1 : a	C.1 : $F \times 2^{-39}$	B.1 : —	B.3 : —
A.2 : a' (when formed)	C.2 : 0	B.2 : —	B.4 : —

Box 5.

1.	m⟶Ac	B.2	$2^{-1}a_i$ to R2	
2.	R2		$2^{-3}a_i$ in R2	
3.	m⟶Ah	B.2	$2^{-1}a_i + 2^{-3}a_i$ in R2	
4.	m⟶Q	B.1	a'_i to R4	
5.	L4		$a_{i+1} = 2^4(2^{-1}a_i + 2^{-3}a_i)$ fract. pt. in R2	
			$a'_{i+1} = 2^4 a'_i + 2^{-39}w_{i+1}$ in R4	
6.	Q⟶m	B.1		a'_{i+1} to B.1
7.	R1		$2^{-1}a_{i+1}$ in R2	
8.	DS			
9.	A⟶m	B.2		$2^{-1}a_{i+1}$ to B.2
10.	m⟶Ac	B.3	I-i to R2	
11.	a⟶Ah	$-1\cdot 2^{-11}$	I-i-1 in R2	

Box 6.

1.	C	4,4	

Box 7.

1.	m⟶Ac	B.1	a' to R2	
2.	A⟶m	A.2		a' to A.2
3.	Stop			

Box 3.

1.	m⟶Ac	C.2	$\underline{0}$ to R2	
2.	A⟶m	B.1		$a'_0 = 0$ to B.1
3.	T	4,1		

In Box 5 a_{i+1} and a'_{i+1} are formed simultaneously. R4 is utilized for a'_{i+1} and R2 for a_{i+1}. This can be done since

$$a'_{i+1} = 2^4 a'_i + 2^{-39}w_{i+1} \quad \text{and}$$

$$a_{i+1} = 2^4(2^{-1}a_i + 2^{-3}a_{i+1}) \text{ fractional part}$$

are formed by a left shift of 4 and R2 and R4 shift in concert. As in

the previous problem (the conversion from coded-decimal form to binary form) the binary point in R2 is treated as though it were immediately left of the sign bigit. The reason for this is the same as in the previous example--the sign of R2 shifts with the number; hence, when the left shift of 4 is performed, the sign position should contain the first bigit of the w_{i+1} that is being isolated. There is, however, the complication introduced of having to shift the number $\underline{a}_i$ to the right in forming the quantity

$$2^{-1}a_i + 2^{-3}a_i.$$

Recall that in a right shift the sign bigit fills into the bigit positions vacated by the shift. The quantity $\underline{a}_i$ is a positive fraction; hence, in shifting right, $\underline{0}$'s should fill into the vacated positions. However, in using the sign position as the first significant bigit of $\underline{a}_i$, whatever this first bigit is, either a $\underline{1}$ or a $\underline{0}$, it will fill into the vacated positions. This, then, would give an incorrect result if the first bigit were a $\underline{1}$. To avoid this difficulty first form $2^{-1}a$ which means that the sign position no longer contains a significant bigit of $\underline{a}_i$. Then set the sign to $\underline{0}$ and proceed in a normal fashion. In Box 4 where we first set

$$a_o = |a|$$

we have really formed

$$2^{-1}a_o = |a|$$

since $\underline{a}$ has the normal binary point convention. In all subsequent steps $2^{-1}a_{i+1}$ is formed by a right shift of $\underline{1}$ followed by a drop sign order (see Box 5, Instructions 7 and 8). Instruction 1 of Box 5 brings $2^{-1}a_i$ into the accumulator and the quantity $2^{-1}a_i + 2^{-3}a_i$ is subsequently formed. Instruction 4 places a'_i into R4; and Instruction 5, the left shift of 4, then forms a'_{i+1} in R4 and a_{i+1} in R2. Instructions 7 and 8 then form $2^{-1}a_{i+1}$ and prefix the correct positive sign.

Instructions 10 and 11 of Box 5 form (I-i-1) but note that the quantity is not immediately stored. Since (I-i-1) is in R2, Box 6 consists only of the conditional transfer instruction. Instead of the conditional transfer instruction transferring to the first instruction of Box 5, it transfers to the last instruction of Box 4. The last instruction of Box 4 is the instruction that initially sent $\underline{I}$ to storage; hence that same instruction is now used to store (I-i-1). This saves a needless duplication of a storage order. In the previous conversion problem, the same scheme could have been

used. Compare the last two instructions of Box 5 and all of Box 6 of this problem with Instructions 12, 13, and 14 of Box 2 and all instructions of Box 3 of the previous problem.

In coding the various boxes, they have been coded in the sequence that corresponds to their correct position in the final coding. This sequence is Boxes 1, 2, 4, 5, 6, 7, and finally, 3. Box 2 must immediately follow Box 1 as it corresponds to the negative branch of the transfer. Then continuing from Box 2, the flow lines go to Boxes 4, 5, 6, and 7. We may insert Box 3 after Box 7 since Box 3 is reached from Box 1 by the satisfied conditional transfer, and then Instruction 3 of Box 3 sends the control to Box 4 as is desired.

Problem 7

We propose to evaluate the integral $\int_o^a f(x)dx$ where $a < 1$. We assume that $f(x)$ is continuous in the interval $0 \leq x \leq a$ and that the value of the integral as well as the value of any intermediate steps of the integration lies in the range of the computer. The value of the integral is approximated by Simpson's method for stepwise integration. The function $f(x)$ is given at the equidistant values $x_o(=0)$, x_1, x_2 $\cdots$ $x_I(=a)$. The values $f(x_o)$ $f(x_1)$ $f(x_2)$ $\cdots$ $f(x_I)$ are stored in the memory at $I+1$ consecutive storage locations. If x is taken as the interval between the various x_i's, then Simpson's Rule may be stated as

$$\int_o^a f(x)dx + \epsilon_r = \frac{\Delta x}{3}[f(x_o)+4f(x_1)+2f(x_2)+4f(x_3)+ \cdots +4f(x_{I-1})+f(x_I)],$$

where ϵ_r is the error term. To evaluate an integral by Simpson's Rule $f(x)$ must be determined at an odd number of x values (an even number of Δx intervals).

The integral is evaluated by using the following inductive process:

$$\begin{aligned}
\textstyle\sum_{-1} &= 0 \\
\textstyle\sum_{o} &= \textstyle\sum_{-1} + \frac{\Delta x}{3} f(x_o) \\
\textstyle\sum_{1} &= \textstyle\sum_{o} + \frac{4\Delta x}{3} f(x_1) \\
\textstyle\sum_{2} &= \textstyle\sum_{1} + \frac{2\Delta x}{3} f(x_2) \\
&\vdots \\
\textstyle\sum_{i} &= \textstyle\sum_{i-1} + \frac{\xi\Delta x}{3} f(x_i) \quad \text{where} \quad \xi \begin{cases} = 4 \text{ when } i \text{ is odd} \\ = 2 \text{ when } i \text{ is even} \end{cases} \\
&\vdots \\
\textstyle\sum_{I} &= \textstyle\sum_{I-1} + \frac{\Delta x}{3} f(x_I)
\end{aligned}$$

where $\sum_I \simeq \int_o^a f(x)$ $(x_o = 0, x_I = a)$ to the desired accuracy.

The inductive scheme that is chosen to describe the integration is perhaps neither the simplest in coding nor the shortest in computing time. It is used principally because an innovation is introduced into the flow diagram.

Three decisions must be made in traversing the induction:

(i) If $\underline{i} = 0$ or I, then $\frac{\Delta x}{3} f(x_i)$ is added to the partial summation.

(ii) If $\underline{i}$ is odd, then $\frac{4\Delta x}{3} f(x_i)$ is added to the partial summation.

(iii) If $\underline{i}$ is even, then $\frac{2\Delta x}{3} f(x_i)$ is added to the partial summation.

As previously discussed, the conditional transfer instruction allows the control to make a decision and follow one of two paths, dependent upon the decision. To make three decisions as outlined above, two alternative boxes could be used in sequence. However, let us approach the problem in a slightly different manner.

The first time through the induction it is desired to form $\frac{\Delta x}{3} f(x_o)$. As $\frac{\Delta x}{3} f(x_o)$ is formed, the next step of the induction is to form $\frac{4\Delta x}{3} f(x_1)$ and as $f(x_1)$ is operated on it is known that next $\frac{2\Delta x}{3} f(x_2)$ is to be formed. In fact, at each passage through the induction it is known what the forthcoming traversal should form. Let us, then, represent three operation boxes which for convenience we call Boxes 2, 3, and 4. Box 2 forms $\frac{\Delta x}{3} f(x_i)$; Box 3 forms $\frac{4\Delta x}{3} f(x_i)$; and Box 4 forms $\frac{2\Delta x}{3} f(x_i)$. Rather than use a sequence of alternative boxes to direct the control to the correct operation box (Box 2, 3, or 4, according as $\underline{i} = 0$ or I, $\underline{i}$ = odd, $\underline{i}$ = even $\neq 0$ nor I), a transfer instruction is used to which is supplied, at the appropriate time, the various addresses corresponding to the entrance points of Boxes 2, 3, or 4. To simplify the discussion this transfer instruction is called λ. In the initial traversal of the induction, $\underline{\lambda}$ is to have an address that sends the control to Box 2 where it forms $\frac{\Delta x}{3} f(x_o)$; hence in setting up the initial step of the induction the address corresponding to Box 2 is supplied to $\underline{\lambda}$. At the time that the control is operating in Box 2, it is known that the next step of the induction should form $\frac{4\Delta x}{3} f(x_1)$ which corresponds to Box 3; hence as part of the operations performed in Box 2 the address for Box 3 is supplied to $\underline{\lambda}$. Similarly, when the control traverses Box 3 it is known that the next traversal of the induction should involve Box 4 which forms $\frac{2\Delta x}{3} f(x_2)$; hence Box 3 supplies, among other things, the address of Box 4 to $\underline{\lambda}$. And when in Box 4, the control should return to Box 2 on the next traversal so $\underline{\lambda}$ is supplied with the address corresponding to Box 3. The final step of the induction is to form $\frac{\Delta x}{3} f(x_I)$. This is done by a discrimination on i-I, which is negative until i=I, at which time the last term is formed.

The position of the flow diagram at which the transfer instruction $\underline{\lambda}$ occurs is represented by an interruption in the flow line with a circle containing the Greek letter $\underline{\lambda}$. The circle has one point of entrance but no point of exit. See Figure 13. In general, the Greek letter is not restricted to $\underline{\lambda}$ and any letter could be used. The various points to which the transfer is to send the control are also represented by circles which contain the same Greek letter as the transfer circle. These Greek letters are, however, indexed for identification. These circles have no point of entrance but one point of exit as shown in Figure 14.

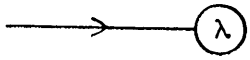


Figure 13.

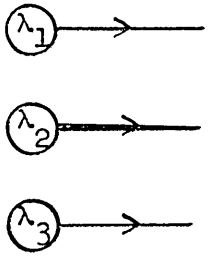


Figure 14.

Such a set of symbols is said to represent a set of variable remote connections.

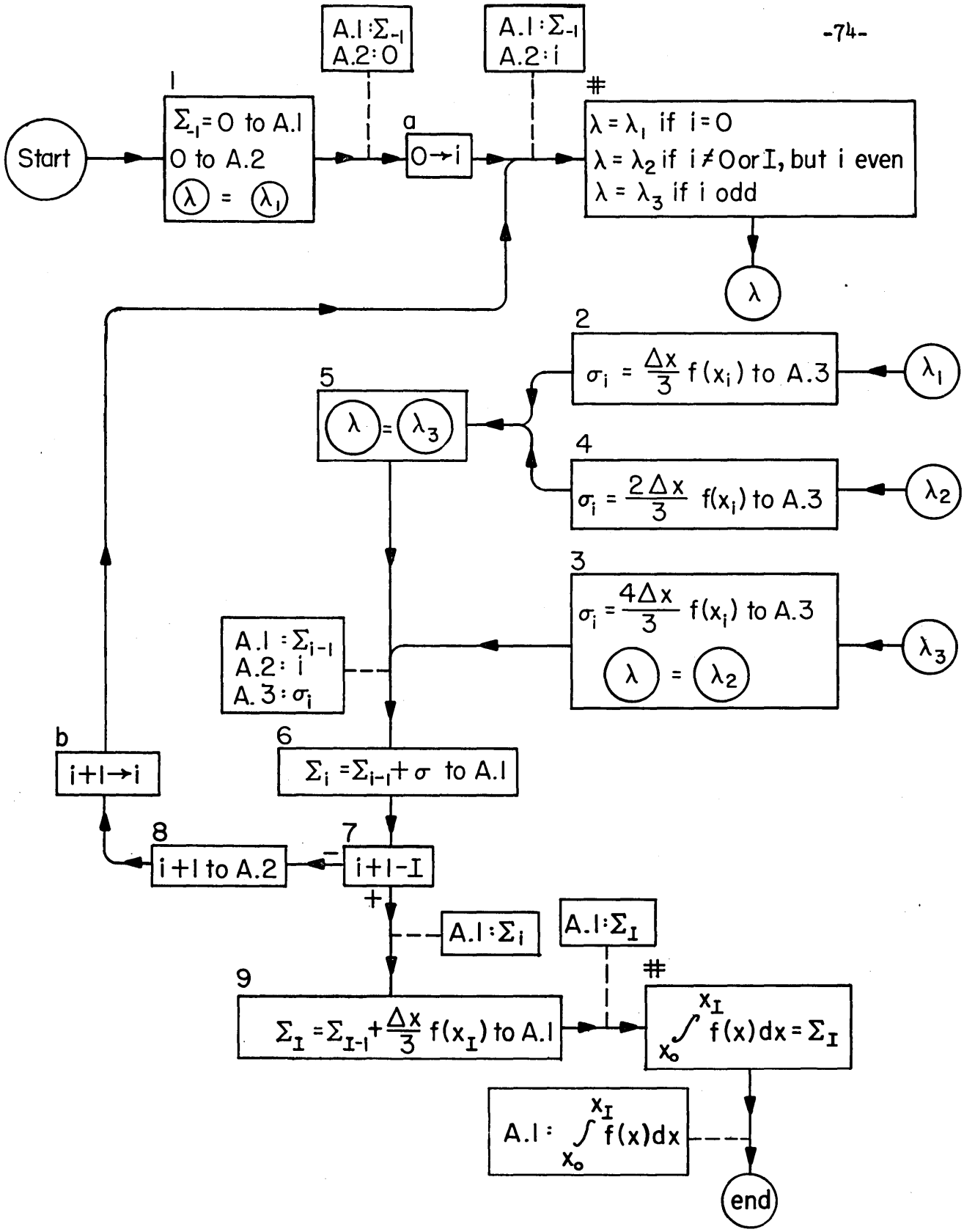
The appropriate addresses are supplied to the transfer $\underline{\lambda}$ in various operation boxes by making use of the substitution instructions. The operation is denoted as $(\lambda) = (\lambda_i)$ where we enclose the λ_i's in circles to show that they are addresses which are concerned with variable remote connections. $(\lambda) = (\lambda_i)$ is interpreted as meaning that the address represented by λ_i is to be supplied (substituted) into the transfer instruction λ.

The flow diagram includes the use of the variable remote connections. The flow diagram is shown in Figure 15.

At any step of the integration $\sum_i$ is used to represent the sum of the terms in Simpson's Rule up to that point. When the integration is completed $\sum_I$ represents the value of the integral to the desired accuracy.

Box 1 of the flow diagram sets $\sum_{-1}$ to $\underline{0}$ as an initial step for the induction. The variable of induction $\underline{i}$ is set to $\underline{0}$. λ is set to λ_1 so that the first traversal of the induction will be through Box 2.

Immediately following Box 2, λ is set to λ_3 so that after going through Box 2 the next traversal will correctly include Box 3. In Box 3, λ is set to λ_2 so that the following traversal includes Box 4, and so on until the induction is completed. At each traversal of the induction only one of the boxes, 2, 3, or 4, is included.



INTEGRATION by SIMPSON'S RULE

FIG. 15

Box 7 discriminates on the quantity i+1-I. This means that the conditional transfer is effective when i = I-1. At this time $\sum_{I-1}$ has just been formed. The final step of the induction, the formation of $\sum_{I}$ is done in Box 9.

Storage is needed to store the numbers corresponding to the addresses λ_1, λ_2, and λ_3. These addresses are stored as position marks and

B.1: $(\lambda_1)_o$
B.2: $(\lambda_2)_o$
B.3: $(\lambda_3)_o$

The following storage is also needed:

B.4: $(1)_o$
B.5: $(I)_o$
B.6: $\frac{\Delta x}{3}$

The values of $f(x_i)$ are stored in I+1 successive locations where C.0 stores $f(x_0)$, C.1: $f(x_1)$ ··· C.i: $f(x_i)$ ··· C.I: $f(x_I)$. The value of the initial address C.0 is needed and it is stored in

B.7: $(C.0)_o$

as a position mark. Any particular value $f(x_i)$ is brought into the arithmetic unit by forming its address as

$$(C.i)_o = (C.0)_o + (i)_o \text{ in R2}$$

The address C.i is then substituted into the instruction which is to operate upon the corresponding $f(x_i)$.

The coding is:

Box 1.

1.	a⟶Ac	0	0 to R2	
2.	A⟶m	A.1		$\sum_{-1} = 0$ to A.1
3.	A⟶m	A.2		0 to A.2
4.	m⟶Ac	B.1	$(\lambda_1)_o$ to R2	
5.	S⟶m	1,11		λ_1 to (8-19)11
6.	m⟶Ac	B.7	$(C.0)_o$ to R2	
7.	m⟶Ah	A.2	$(C.i)_o = (C.0)_o + (i)_o$ in R2	
8.	S⟶m	1,10		C.i to (8-19)10
9.	m⟶Q	B.6	$\frac{\Delta x}{3}$ to R4	
10.	X	[C.i]	$\frac{\Delta x}{3} f(x_i)$ in R2	
11.	T	[λ]		

Box 2.

1.	A⟶m	A.3		$\sigma_i = \frac{\Delta x}{3} f(x_i)$ ro A.3

Box 5.

1.	m⟶Ac	B.3	$(\lambda_3)_o$ to R2	
2.	S⟶m	1,11		λ_3 to(8-19)11

Box 6.

1.	m⟶Ac	A.3	σ_i to R2	
2.	m⟶Ah	A.1	$\sum_i = \sum_{i-1} + \sigma_i$ in R2	
3.	A⟶m	A.1		$\sum_i$ to A.1

Box 7.

1.	m⟶Ac	A.2	$(i)_o$ to R2	
2.	m⟶Ah	B.4	$(i+1)_o$ in R2	
3.	m⟶Ah-	B.5	$(i+1-I)_o$ in R2	
4.	C	9,1		

Box 8.

1.	m⟶Ac	A.2	$(i)_o$ to R2	
2.	m⟶Ah	B.4	$(i+1)_o$ in R2	
3.	A⟶m	A.2		$(i+1)_o$ to A.2
4.	T	1,6		

Box 3.

1.	L(2)		$\sigma_i = \frac{4\Delta x}{3} f(x_i)$ in R2	
2.	A⟶m	A.3		σ_i to A.3
3.	m⟶Ac	B.2	$(\lambda_2)_o$ to R2	λ_2 to (8-19)11
4.	S⟶m	1,11		
5.	T	6,1		

Box 4.

1.	L(1)		$\sigma_i = \frac{2\Delta x}{3} f(x_i)$ in R2	
2.	A⟶m	A.3		σ_i to A.3
3.	T	5,1		

Box 9.

1. $m \longrightarrow Q$ B.6 $\frac{\Delta x}{3}$ to R4
2. X C.I $\frac{\Delta x}{3} f(x_I)$ in R2
3. $m \longrightarrow Ah$ A.1 $\sum_I = \sum_{I-1} + \frac{\Delta x}{3} f(X_I)$ in R2

In Boxes 2, 3, and 4 the quantity $\frac{\Delta x}{3} f(x_i)$ is needed. Rather than code this separately in each box, it is coded immediately preceding the variable transfer λ. This is coded in Instructions 6 through 10 of Box 1. The transfer instruction at the end of Box 8 transfers the control into Instruction 6 of Box 1 for this computation is to be done for all traversals in the induction. The coding of Boxes 2, 3, and 4 starts with the quantity $\frac{\Delta x}{3} f(x_i)$ in R2.

There are 38 instructions in all. Pairing these into words gives 19 words of instructions.

The word coding is:

1.	a⟶Ac	000	A⟶m	028
2.	A⟶m	029	m⟶Ac	021
3.	S⟶m	006	m⟶Ac	027
4.	m⟶Ah	029	S⟶m'	005
5.	m⟶Q	026	X	[ ]
6.	T'	[ ]	A⟶m	030
7.	m⟶Ac	023	S⟶m	006
8.	m⟶Ac	030	m⟶Ah	028
9.	A⟶m	028	m⟶Ac	029
10.	m⟶Ah	024	m⟶Ah-	025
11.	C	018	m⟶Ac	029
12.	m⟶Ah	024	A⟶m	029
13.	T'	003	L(2)	002
14.	A⟶m	030	m⟶Ac	022
15.	S⟶m	006	T	008
16.	[ ]		L(1)	001
17.	A⟶m	030	T	007
18.	m⟶Q	026	X	(C.I)
19.	m⟶Ah	028	A⟶m	028
20.	Stop			

21. $(\lambda_1)_o = (6)_o$
22. $(\lambda_2)_o = (16)_o$
23. $(\lambda_3)_o = (13)_o$
24. $(1)_o$
25. $(I)_o$
26. $\frac{\Delta x}{3}$
27. $(C.0)_o$
28. A.1
29. A.2
30. A.3

The transfer instruction $\underline{\lambda}$ must transfer the control at various phases of the problem into Box 2, Instruction 1; into Box 3, Instruction 1; and into Box 4, Instruction 1. As the coding was done the transfer was fixed as a prime transfer since Box 2, Instruction 1, and Box 3, Instruction 1 each were on the right side of their respective words. The first instruction of Box 4, however, naturally falls as the left side of an instruction word. This meant that the left half of Word 16, the start of Box 4, was left blank and Box 4 was started as the right-hand instruction. Perhaps by shifting the arrangement of Boxes 3, 4, and 9 this could have been avoided.

A better method of avoiding this would be to use the half word substitution instruction. In Words 21, 22, and 23, where the numerical values of λ_i's are stored, rather than storing just the addresses the following should be stored:

21. $(\lambda_1)_o$ = CB006CB006
22. $(\lambda_2)_o$ = CA016CA016
23. $(\lambda_3)_o$ = CB013CB013

Then by a half word substitution the order as well as the address of the transfer instruction may be altered. Box 4 would now start with the left-hand instruction of Word 16 which saves the previously wasted half word.

In the right-hand instruction of Word 18, the address C.I is inserted in parentheses. C.I is a known address, but for the example no numerical values were assigned for the C.i storage, nor was the number of intervals $\underline{I}$ determined. For this reason the C.I is indicated in parentheses rather than as a numerical address.

The addresses of the instructions in the word code are written as three characters. Writing numerical addresses in this fashion tends to avoid errors in transcribing the word code into the numerical code as addresses are represented in the numerical code by three characters.

Problem 8

Although the computer operates with a fixed binary point, at times it is advantageous to use a floating binary point. The floating point method (hereinafter referred to as FPM) allows each number to be expressed as a fraction and a characteristic (an exponent). For example the decimal number

$$7798.543210$$

or its equivalent

$$.7798543210 \times 10^{4}$$

expressed in floating point notation would be

$$.7798543210, + 4$$

where the +4 is the positive exponent of 10 associated with the number. Similarly, a binary number

$$1011.1001$$

expressed in floating point notation would be

$$.10111001, + 100$$

where the +100 is the positive exponent of 2 associated with the number.

The discussion here will pertain only to floating binary point operation. Although the computer operates with binary numbers, there are floating point schemes where the characteristic (exponent) may be expressed to a base other than the base two, such as the more familiar decimal base. Since the computer operates with binary numbers, it is inherently easier to use the floating binary point scheme, or at least a scheme where the base of the characteristic is a power of two, such as the octal or hexadecimal base. For much of the floating point operation a choice of expressing the characteristic to a base 16, 32, or even 128 might simplify floating point procedures.

The need for FPM may arise where the ranges (the maximum and minimum) of the quantities entering into the computation are not known within reasonable limits; or where the range of the quantities is so great that the scaling of numbers for fixed binary point operation causes undue loss of the significant figures of the numbers. When a problem is scaled for fixed point operation, the loss of significant figures caused by the numbers becoming too small is as important a consideration as is numbers becoming too large.

The use of the FPM is, in general, discouraged for must computation as it greatly slows down the effective computer speed. In most problems, scaling may be accomplished without undue loss of significant figures. In cases where the scaling is difficult to accomplish, a scheme of self-adjusting scaling or the use of scaling checks may be employed as an aid to scaling.

Addition is chosen as the example for FPM. The other operations are accomplished by a somewhat similar scheme.

To add two numbers that are represented in floating point notation, the exponents must first be made the same. This may be shown by the following decimal example:

$$\begin{array}{rl} & .753,\ 3 \\ + & .325,\ 2 \end{array}$$

These numbers are

$$\begin{array}{l} .753 \times 10^{3} \\ .325 \times 10^{2} \end{array}$$

and for the numbers to be summed, the powers of 10 must be the same; therefore,

$$\begin{array}{l} .753 \times 10^{3} \\ .0325 \times 10^{3} \\ \hline .7855 \times 10^{3} \end{array}$$

To do the operations in the computer, all numbers must be less than 1. The smaller exponent must always be made equal to the larger as this has the effect of making the number whose exponent is increased, smaller, which keeps it less than 1.

The addition operation is accomplished by the computer as follows:

(i) The exponents are compared. If they are not the same, the smaller exponent is increased. The difference between the exponents is the amount by which the smaller is increased.

(ii) For each increase of the exponent by 1, the number should be multiplied by 2^{-1}. A multiplication by 2^{-1} corresponds to the number being shifted right by 1.

(iii) After the smaller number has been adjusted, the addition is done. The exponent of the sum is the same exponent as the numbers, unless the sum is greater than 1. In this case the sum is shifted right 1 and the exponent is increased by 1.

For example:

$$\begin{array}{rl} & a = .11111101,\ 100 \\ + & b = .10110010,\ 011 \\ \hline & s \end{array}$$

The exponent of b is 1 less than the exponent of a; therefore b is shifted right 1, and 1 is added to its exponent. Now

$$\begin{array}{rl} & a = 0.11111101,\ 100 \\ + & b = 0.01011001,\ 100 \\ \hline & s = 1.01010110,\ 100 \end{array}$$

The sum of the two numbers is greater than 1 so s is shifted right 1 and the exponent is increased by 1.

$$s = 0.10101011,\ 101$$

In the computer, if the sum of the numbers is greater than 1, it cannot be adjusted simply by a right shift of 1 as indicated above since the sign bigit propagates in a right shift. To avoid this difficulty, the addend and augend are each shifted right by 1 and their exponents increased by 1 before the addition is done. Then no spillage can occur in the addition.

Numbers to be operated on by FPM are adjusted into a standard form where the first significant bigit of the number is in the 2^{-1} bigit position. All fractions F are therefore in the range

$$1/2 \leq F < 1$$

Floating point numbers have 27 significant bigits which, with the sign bigit, occupy bigit positions 0 through 27. Positions 28 through 39 of the word are used for storing the exponent, and a number and its associated exponent are stored in one word. The 27 bigits of the number correspond to about 8 decimal digits. The 12 bigits allowed for the exponent are more than ample; however, 12 are used since the bigits (28-39) may be conveniently manipulated by the s→m' instruction.

Positive and negative exponents are allowed, and the 12 bigits (28-39) for expressing the exponent n give a range

$$-2048 \leq n < 2048$$

Negative exponents are represented as complement numbers. The first bigit of the exponent is considered its sign bigit. The exponent n is an integer and it is represented as $n \cdot 2^{-11}$.

We propose to form the sum of two numbers a and b with exponents α and β, respectively. The fractions a and b are in standard notation, that is

$$1/2 \leq a,\ b < 1$$

After the addition, the sum s is adjusted to standard form.

As a first step of the procedure, a and b are each shifted right by 1 and their respective exponents are increased by 1. This insures that the sum $s = a + b < 1$.

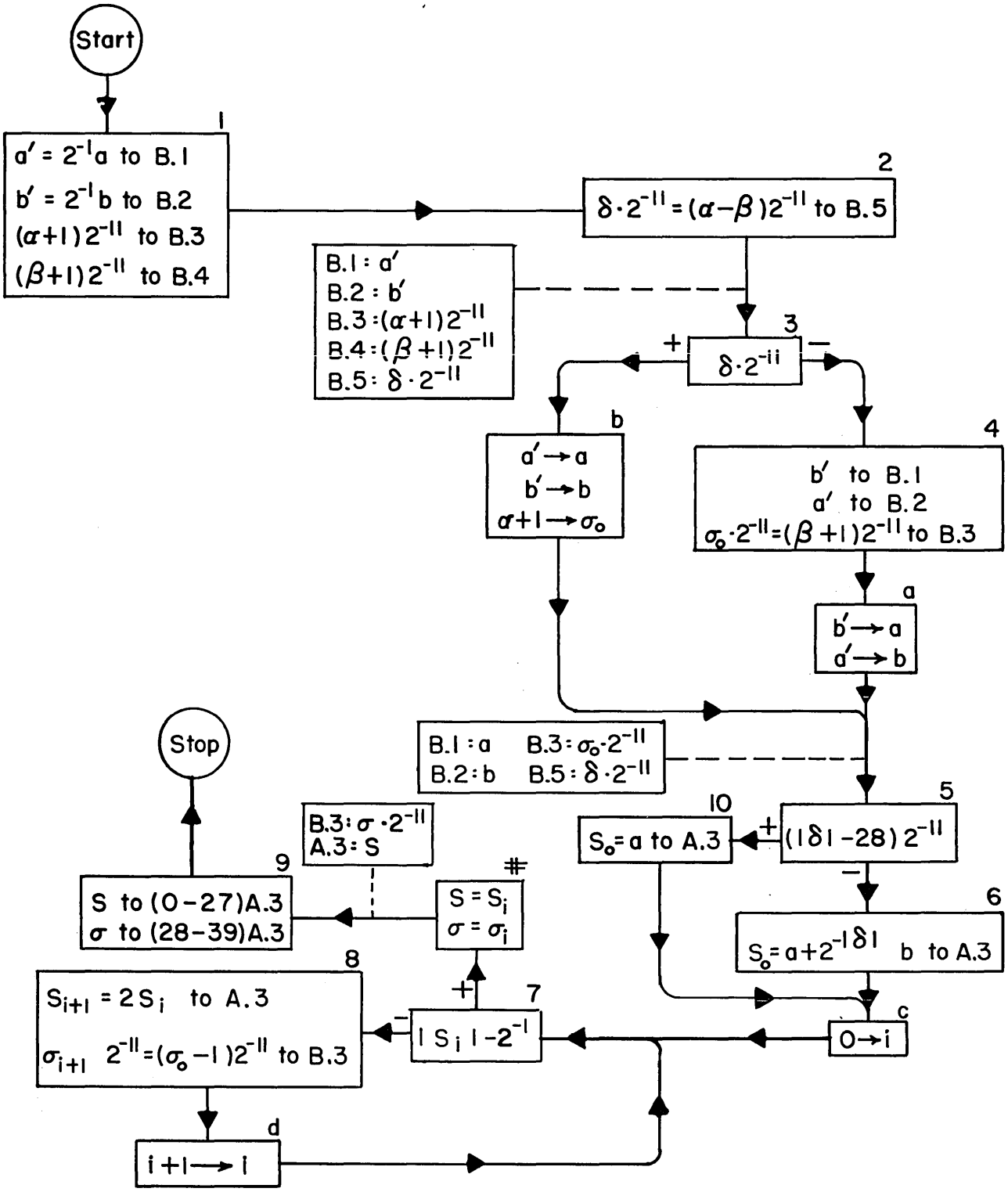
The difference in the exponents is determined. If the difference is greater than 27, the sum is set to the value of the number with the larger exponent. A difference of more than 27 means that the smaller exponent must be increased by at least 28, and the number associated with the exponent must be shifted right the corresponding number of places. Since the numbers are represented as a sign bigit and 27 significant bigits, a number shifted right by 28 places can make no contribution to the sum. If the difference in the exponents is less than 28, the smaller is adjusted to be equal to the larger. The sum of the numbers is then formed and put in standard notation. We now examine the flow diagram shown in Figure 16. The storage of the problem is:

A.1: a (0-27) α (28-39)
A.2: b (0-27) β (28-39)
A.3: s (0-27) σ (28-39)

Box 1 shifts a and b right 1 and increases each of the exponents. Box 3 discriminates on the difference of the exponents to determine which exponent is the greater. The problem is arranged so that the number with the larger exponent must be in location B.1 and its exponent must be in B.3. If $\alpha \geq \beta$ no changes of storage need be made. If $\alpha < \beta$ then the positions of a and b are interchanged and $\beta+1$ is put into B.3. This is done in Box 4. Box 5 discriminates on the difference of the exponents to see if this difference is greater than 27. If the difference is greater than 27, the sum is set to a, the number with the larger exponent. If the difference is less than 28, the sum is

$$s = a + 2^{-|\alpha-\beta|}{}_b$$

and the exponent is the exponent of a. A discrimination is made on the sum s to see if it is in standard form. If it is not, the sum is shifted



FLOATING POINT ADDITION

FIG. 16

left until the first significant bigit is in the 2^{-1} bigit position. This is done in Boxes 7 and 8. Box 9 combines the sum and its exponent and stores them in A.3.

The storage locations B.1, B.2, B.3, B.4, and B.5 are needed to store intermediate values during the computation.

The coding is:

Box 1.

1.	m⟶Ac	A.1	a(0-27) ,α(28-39)	to R2
2.	R(1)		$a' = 2^{-1}a$ in R2	
3.	A⟶m	B.1		a' to B.1
4.	m⟶Ac	A.2	b(0-27) ,β(28-39)	to R2
5.	R(1)		$b' = 2^{-1}b$ in R2	
6.	A⟶m	B.2		b' to B.2
7.	m⟶Ac	A.1	a(0-27) ,α(28-39)	to R2
8.	L(28)		$\alpha \cdot 2^{-11}$ in R2	
9.	a⟶Ah	2^{-11}	$(\alpha+1)2^{-11}$ in R2	
A.	A⟶m	B.3		$(\alpha+1)2^{-11}$ to B.3
B.	m⟶Ac	A.2	b(0-27) ,β(28-39)	to R2
C.	L(28)		$\beta \cdot 2^{-11}$ in R2	
D.	a⟶Ah	2^{-11}	$(\beta+1)2^{-11}$ in R2	
E.	A⟶m	B.4		$(\beta+1)2^{-11}$ to B.4

Box 2.

1.	m⟶Ac	B.3	$(\alpha+1)\cdot 2^{-11}$ to R2	
2.	m⟶Ah-	B.4	$\partial = (\alpha - \beta)2^{-11}$ in R2	
3.	A⟶m	B.5		∂ to B.5

Box 3.

1.	C	5,1	∂ in R2

Box 4.

1.	m⟶Q	B.2	b' to R4	
2.	m⟶Ac	B.1	a' to R2	
3.	Q⟶m	B.1		a = b' to B.1
4.	A⟶m	B.2		b = a' to B.2
5.	m⟶Ac	B.4	$\sigma_o \cdot 2^{-11} = (\beta+1)\cdot 2^{-11}$	to R2
6.	A⟶m	B.3		$\sigma_o \cdot 2^{-11}$ to B.3

Box 5.

1.	m⟶AcM	B.5	$\lvert\partial\rvert 2^{-11}$	to R2
2.	a⟶Ah	$-28\cdot 2^{-11}$	$(\lvert\partial\rvert-28)2^{-11}$	in R2
3.	C	10,1		

Box 6.

1.	m⟶AcM	B.5	$\lvert\partial\rvert\cdot 2^{-11}$	to R2
2.	R(8)		$\lvert\partial\rvert\cdot 2^{-19}$	in R2
3.	S⟶m	6,5		$\lvert\partial\rvert$ to (8-19)5
4.	m⟶Ac	B.2	b	to R2
5.	R(n)	$[\lvert\partial\rvert]$	$2^{-\lvert\partial\rvert}b$	in R2
6.	m⟶Ah	B.1	$s_o = a + 2^{-\lvert\partial\rvert}b$	in R2
7.	A⟶m	A.3		s_o to A.3

Box 7.

1.	m⟶AcM	A.3	$\lvert s_i\rvert$	to R2
2.	a⟶Ah	-2^{-1}	$\lvert s_i\rvert - 1/2$	in R2
3.	C	9,1		

Box 8.

1.	m⟶Ac	A.3	s_i	to R2
2.	L(1)		$s_{i+1} = 2s_i$	in R2
3.	A⟶m	A.3		s_{i+1} to A.3
4.	m⟶Ac	B.3	$\sigma_i\cdot 2^{-11}$	to R2
5.	a⟶Ah	-2^{-11}	$\sigma_{i+1} = (\sigma_i-1)2^{-11}$	in R2
6.	A⟶m	B.3		$\sigma_{i+1}\cdot 2^{-11}$ to B.3
7.	T	7,1		

Box 9.

1.	m⟶Ac	B.3	$\sigma\cdot 2^{-11}$	to R2
2.	R(28)		$\sigma\cdot 2^{-39}$	in R2
3.	S⟶m'	A.3		σ to (28-39)A.3
4.	Stop			

Box 10.

1.	m⟶Ac	B.1	a	to R2
2.	A⟶m	A.3		$s_o = a$ to A.3
3.	T	7,1		

In the coding of Box 1, the exponent is not cleared out of positions 28 through 39. These positions do not affect the answer. The numbering of the instructions in Box 1 is done hexadecimally. There are E instructions which corresponds to 14 decimally.

In the coding of problems for the computer, the numbering is done hexadecimally; therefore in all further examples the numbering will be hexadecimal.

In Box 9, where $\underline{s}$ and $\underline{\sigma}$ are combined, $\underline{s}$ is already residing in A.3. $\underline{\sigma}$ is brought into R2 and shifted right by 28 so that it is in positions 28 through 39 of R2. It is then sent to A.3 by means of a substitution instruction, and A.3 correctly contains

$$s(0\text{-}27),\ \sigma(28\text{-}39)$$

The floating point addition as set up would not be practical to incorporate into a large problem where many such additions are done. As coded, 49 instructions are used, several of which are lengthy shifts. In floating point routines, time becomes a determining factor and the routines must be constructed with that in mind. There are several ways in which the time involved in the present routine could be shortened. However, we are interested at this time in demonstrating floating point procedures without attempting to develop the most satisfactory scheme.

The present problem does not take into account a method of handling a number that is zero. A way of doing this is not to allow an exact zero, but to say that zero is to be represented as the fraction 1/2, with an appropriate negative exponent. The negative exponent needs to be at least 28 smaller than the smallest exponent encountered in the problem concerned. An addition would treat this number as zero in forming the sum. The fraction part as 1/2 is suggested so that all numbers are represented in the standard notation.

The code in final form contains 25 words since there are 49 instructions. If the code starts at zero, 25 words would occupy addresses 0 through 19, hexadecimally. B.1, B.2, B.3, B.4, and B.5 are the addresses 1A through 1E, and A.1, A.2, and A.3 are addresses 1F, 20, 21, respectively.

The code is:

0.	m⟶Ac	01F	R(1)	001
1.	A⟶m	01A	m⟶Ac	020
2.	R(1)	001	A⟶m	01B
3.	m⟶Ac	01F	L(28)	01C
4.	a⟶Ah	001	A⟶m	01C
5.	m⟶Ac	020	L(28)	01C
6.	a⟶Ah	001	A⟶m	01D
7.	m⟶Ac	01C	m⟶Ah-	01D
8.	A⟶m	01E	C	00C
9.	m⟶Q	01B	m⟶Ac	01A
A.	Q⟶m	01A	A⟶m	01B
B.	m⟶Ac	01D	A⟶m	01C
C.	m⟶AcM	01E	a⟶Ah	FE4
D.	C	018	m⟶AcM	01E
E.	R(28)	01C	S⟶m'	00F
F.	m⟶Ac	01B	R(∂)	000
10.	m⟶Ah	01A	A⟶m	021
11.	m⟶AcM	021	a⟶Ah	C00
12.	C	016	m⟶Ac	021
13.	L(1)	001	A⟶m	021
14.	m⟶Ac	01C	a⟶Ah	FFF
15.	A⟶m	01C	T	011
16.	m⟶Ac	01C	R(28)	01C
17.	S⟶m'	021	Stop	
18.	m⟶Ac	01A	A⟶m	021
19.	T	011		
1A.				
1B.				
1C.				
1D.				
1E.				
1F.	a, α			
20.	b, β			
21.	s, σ			

Instruction 5 of Box 6 becomes the right hand instruction of 00F. The substitution instruction (Box 6, Instruction 3) that substitutes the address into Instruction 5 of Box 6 must be an $s \longrightarrow m'$ instruction as is indicated. Instruction 2 of Box 6 must be changed to R(28) rather than R(8) to accommodate the $s \longrightarrow m'$.

Problem 9

The standard 40 bigit number (including sign) provides sufficient accuracy for must computation; however, certain problems may arise where added precision is necessary. To handle such cases, multiple precision routines must be used. These routines effect the basic operations with numbers that are 78, 117, or k·39 (k = 1, 2 ··· K) bigits in length. For the present purpose, which is to illustrate such methods, only double precision (numbers 78 bigits in length) is considered.

In the treatment of multiple precision numbers, some convention must be adopted for the sign bigits of the auxiliary components, the principal component having of course the same form as the standard size numbers. A convenient pattern is to set to 0 the sign bigits of all auxiliary components. Hence, for the double precision number $x \geq 0$, the representation is simply

$$x = x' + 2^{-39}x''$$

where x' is the principal component and x" the auxiliary one.

For $x < 0$, it should be represented as a 78 bigit complement, the sign bigit of the principal component being 1 and that of the auxiliary being 0 by our convention. This implies that the two parts of (2- x) are

$$2 - x' - 2^{-39} \quad \text{and}$$

$$1 - x''$$

The example chosen is double precision division, for it in itself includes a double precision multiplication and subtraction. The division is performed by forming first the reciprocal of the divisor to double precision, followed by a double precision multiplication with the dividend. We first consider double precision addition, subtraction and multiplication.

A double precision addition

$$s = x + y$$

is done by first adding the less significant components x" and y". The sum may be greater than 1. Recall that x" and y" each had a sign bigit

of 0 so that a 1 in the sign position of this sum indicates spillage rather than a negative number. This spillage corresponds to a carry into the 2^{-39} position of the more significant part of the sum; therefore, we may write

$$s'' = x'' + y'' \pmod 1$$

The more significant part of the sum is

$$s' = x' + y' + \epsilon_o$$

$$\epsilon_o = \begin{cases} 2^{-39} & \text{if carry present} \\ 0 & \text{if no carry present} \end{cases}$$

Finally, the double precision sum is

$$s = s' + 2^{-39}s''$$

In order to form a difference of two double precision numbers, the complement of the number being subtracted is first obtained. An addition is then performed as indicated above.

In the double precision multiplication, the product

$$p = xy$$

is to be formed. For simplicity of discussion, first assume $x,y \geq 0$. Algebraically, the multiplication is

$$p = (x' + 2^{-39}x'')(y' + 2^{-39}y'')$$

$$= x'y' + 2^{-39}x''y' + 2^{-39}x'y'' + 2^{-78}x''y''$$

Each term on the right has 78 bigits, so we may write the product (neglecting roundoff on the extreme right) as

$$p = (x'y')' + 2^{-39}(x'y')'' + 2^{-39}(x''y')' + 2^{-39}(x'y'')'$$

$$p' = (x'y')'$$

$$p'' = (x'y')'' + (x''y')' + (x'y'')'$$

p" is formed first. In the partial summing, carries may be produced that must be added into the 2^{-39} position of p'. The summing is done in two steps as

$$s = [(x'y')'' + (x''y')'] \pmod 1$$

with

$$\epsilon_o = \begin{cases} 1 & \text{if carry present} \\ 0 & \text{if no carry present} \end{cases}$$

$$p'' = [s + (x'y'')'] \pmod 1$$

$$\epsilon_1 = \begin{cases} 1 & \text{if carry present} \\ 0 & \text{if no carry present} \end{cases}$$

This completes the multiplication for $x,y \geq 0$ and

$$p'' = (x'y')'' + (x''y')' + (x'y'')' - \epsilon_o - \epsilon_1$$

$$p' = (x'y')' + 2^{-39}(\epsilon_o + \epsilon_1)$$

$$p = p' + 2^{-39}p''$$

In order to treat the double precision multiplication when either the multiplier or the multiplicand is negative, we refer to the algebraic derivation of the multiplication. (See chapter on Binary Arithmetic.) A product $\underline{uv}$ is formed as

$$p = (\xi_o + u)\left(\sum_{i=o}^{39} 2^i + 1 + v\right)$$

where

$$\xi_o = \begin{cases} 1 & \text{if } u \text{ is negative} \\ 0 & \text{if } u \text{ is positive} \end{cases}$$

The product expanded is

$$p = \xi_o \sum_{i=o}^{39} 2^i + \xi_o + \xi_o v + u \cdot \sum_{i=o}^{39} 2^i + u + uv$$

$$p = \xi_o v + 2^{40}u - u + u + uv + 2^{40}\xi_o$$

$$p \pmod 2 = \xi_o v + 2 - u + u + uv$$

If $\underline{u}$ is negative, $\xi_o = 1$ and a term $\underline{v}$ appears in the product. A correction of (2-v) is then necessary. For simple precision, if $\underline{v}$ is negative the terms (2-u) and $\underline{u}$ are generated during the multiplication and precisely compensate each other; hence, no correction term is necessary when $\underline{v}$ is negative. This compensation is not exact in double precision, and a small correction is required. Now in a double precision multiplication

$$p = xy,$$

a correction term of

$$2 - y$$

is necessary if $\underline{x}$ is negative (indicated by the sign of x'). All intermediate products involving x' have a correction added, namely the complement of the multiplicand. The terms involved are

$$(x'y')' \quad \text{and} \quad (x'y'')'$$

The term (x'y')" of course does not suffer any correction, and the corrections as done by the computer are, respectively,

$$(2\text{-}y') \quad \text{and} \quad (2\text{-}y'')$$

Combining these two terms appropriately one gets the correction as done by the computer for a negative multiplier x, namely

$$2 - y' + 2^{-39}(2-y'')$$

The true correction, however, should be

$$2 - y = 2 - y' - 2^{-39} + 2^{-39}(1-y'')$$

The most significant part of the correction term is 2^{-39} too large. It is adjusted by subtracting 2^{-39} from (x'y')'. The least significant part of the correction term is 1 too large. It is adjusted by setting the sign bigit of (x'y")' to 0. (Less pedagogically, but more concisely, it may be said that the computer correction is too large by 2^{-38}, and this is compensated by subtracting 2^{-39} twice.)

A negative multiplier which necessitates the above additional corrections may be detected by examining the sign of (x'y")'. y" always has a sign of 0; therefore, if

$$(x'y")' < 0, \text{ then } x < 0$$

and 2^{-39} is subtracted from (x'y')' and the sign of (x'y")' is set to 0. If

$$(x'y")' > 0, \text{ then } x > 0$$

and no correction is necessary. If

$$(x'y")' = 0$$

and if

$$y" = 0$$

then x' may be negative, and examining (x'y")' will not indicate this. However, in such a case, the correction as done by the computer is the precise one and no further steps are necessary.

When the multiplicand y is negative, the terms (x'y')' and (x"y')' suffer the standard correction by the computer (as a negative multiplicand is indicated by the sign of y'). We have seen above that the single multiplication process which forms the products (x'y')' and (x"y')' generates pairwise the terms

$$x', (2\text{-}x') \text{ and } x", (2\text{-}x")$$

The first pair compensate precisely; the second pair is really

$$2^{-39}(x"+2-x") = 2(2^{-39})$$

As before, this quantity must be subtracted from the collection to obtain the precise multiple product, and again this is accomplished by subtracting 2^{-39} from the more significant part and $\underline{1}$ from the less significant part. If

$$(x''y')' < 0 \quad \text{then} \quad y' < 0$$

and 2^{-39} is subtracted from $(x'y')'$ and the sign of $(x''y')'$ is set to $\underline{0}$.

If $(x''y')' \geq 0$ then no correction is needed.

A double precision product involving all necessary correction terms is done as follows:

Form $(x'y'')'$. If

$(x'y'')' < 0$ $\quad \epsilon_o = 2^{-39}$ then set sign of $(x'y'')'$ to $\underline{0}$.

$(x'y'')' \geq 0$ $\quad \epsilon_o = 0.$

Form $(x''y')'$. If

$(x''y')' < 0$ $\quad \epsilon_1 = 2^{-39}$ then set sign of $(x''y')'$ to $\underline{0}$.

$(x''y')' \geq 0$ $\quad \epsilon_1 = 0.$

Form $(x'y')'$ and $(x'y')''$. Then form the sum

$$s = (x'y'')' + (x''y')'.$$

If

$s \geq 1$ $\quad \epsilon_2 = 2^{-39}$ then set sign of $\underline{s}$ to $\underline{0}$.

$s < 1$ $\quad \epsilon_2 = 0.$

p'' is formed as

$$p'' = (x'y')'' + s.$$

If

$p'' \geq 1$ $\quad \epsilon_3 = 2^{-39}$ then set sign of p'' to $\underline{0}$.

$p'' < 1$ $\quad \epsilon_3 = 0.$

p' is formed as

$$p' = (x'y')' - \epsilon_o - \epsilon_1 + \epsilon_2 + \epsilon_3$$

and

$$p = p' + 2^{-39}p''.$$

We now return to the division process. The double precision quotient

$$Q = x/y$$

is to be formed. As a first step the reciprocal of $\underline{y}$ is obtained to 78

figures. The reciprocal of y is found by the iterative scheme

$$Z_o = 1/y'$$
$$Z_{i+1} = 2Z_i - yZ_i^2$$
$$\lim_{i\to\infty} Z_{i+1} = 1/y.$$

Such a scheme is error-squaring; therefore if the guess Z_o is precise to 39 bigits, Z_1 is precise to 78 bigits. The scheme is shown to be error-squaring by the following:

Multiply both sides of the above equation by $-y$. This gives

$$-yZ_{i+1} = -2Z_iy + y^2Z_i^2$$

Adding $\underline{1}$ to both members gives

$$1 - yZ_{i+1} = 1 - 2Z_iy + y^2Z_i^2$$
$$= (1 - yZ_i)^2$$

$(1-yZ_i)$ is the error in the i^{th} approximation. The error $(1-yZ_{i+1})$ is the error of the $(i+1)^{th}$ approximation which as indicated above is the square of the error of the i^{th} approximation.

The reciprocal of $\underline{y}$ cannot be determined directly as it is greater than $\underline{1}$. Hence, the reciprocal of the scaled quantity $2^{n+2}y$ is found where

$$1/2 \leq 2^n|y| < 1, \text{ hence } |(2^{n+2}y)^{-1}| < 1.$$

The <u>unscaled</u> quotient is obtained in two steps. First $\underline{x}$ may be multiplied by 2^n, inasmuch as $|x| < |y|$; after the division a left shift of two is then performed. The first guess Z_o is formed as

$$Z_o = 2^{-2}/2^ny' \qquad \text{Eq.(1)}$$

Z_o is precise to 39 figures; consequently Z_1 is precise to 78 figures. Z_1 involves a double precision multiplication in the term

$$2^ny \cdot Z_o^2$$

The subtraction

$$2Z_o - 2^2 \cdot 2^nyZ_o^2$$

is not a true double precision subtraction as $2Z_o$ contains just 39 figures. A double precision complement of $2^2 \cdot 2^nyZ_o^2$ must be taken however. Note that a factor 2^2 is incorporated into the subtrahend in the above subtraction.

This is necessary because of the 2^{-2} factor introduced in forming Z_o in Eq. (1). Using Z_1, the quotient

$$Q_1 = Z_1 \cdot x$$

is formed by a double precision multiplication. Then

$$Q = 2^2 Q_1$$

Since Q is formed by a left shift of 2, only 76 bigits are determined in Q rather than the desired 78.

We now discuss the flow diagram of Figures 17 and 18. Boxes 1 and 2 of the flow diagram adjust $\underline{x}$ and $\underline{y}$ so that

$$1/2 \leq |y| < 1$$

Box 4 stores $\underline{y}$ and Z_o^2 into the four appropriate locations to be used by the double precision multiplication routine. Since two double precision multiplications are required and since they are at two different locations on the flow diagram, the multiplication routine is arranged so that it can be used from any of several places. Four locations are reserved for the factors in the multiplication and upon completing the multiplication a variable remote connection is set as an exit point from the routine. Box 4 also sets the exit from the multiplication as $(\alpha) = (\alpha_1)$ which corresponds to the first instruction of Box 5.

Box 5 forms Z_1 and then sends Z_1 and $\underline{x}$ to the appropriate locations for the multiplication routine, and the exit is set as $(\alpha) = (\alpha_2)$ which corresponds to the first instruction of Box 6.

Box 6 shifts Q_1 left by 2 to give the desired Q.

The multiplication routine is contained in Boxes 7 through 14, numbering hexadecimally. The boxes follow the multiplication procedure as outlined; hence, no further discussion is necessary.

The storage of the problem is:

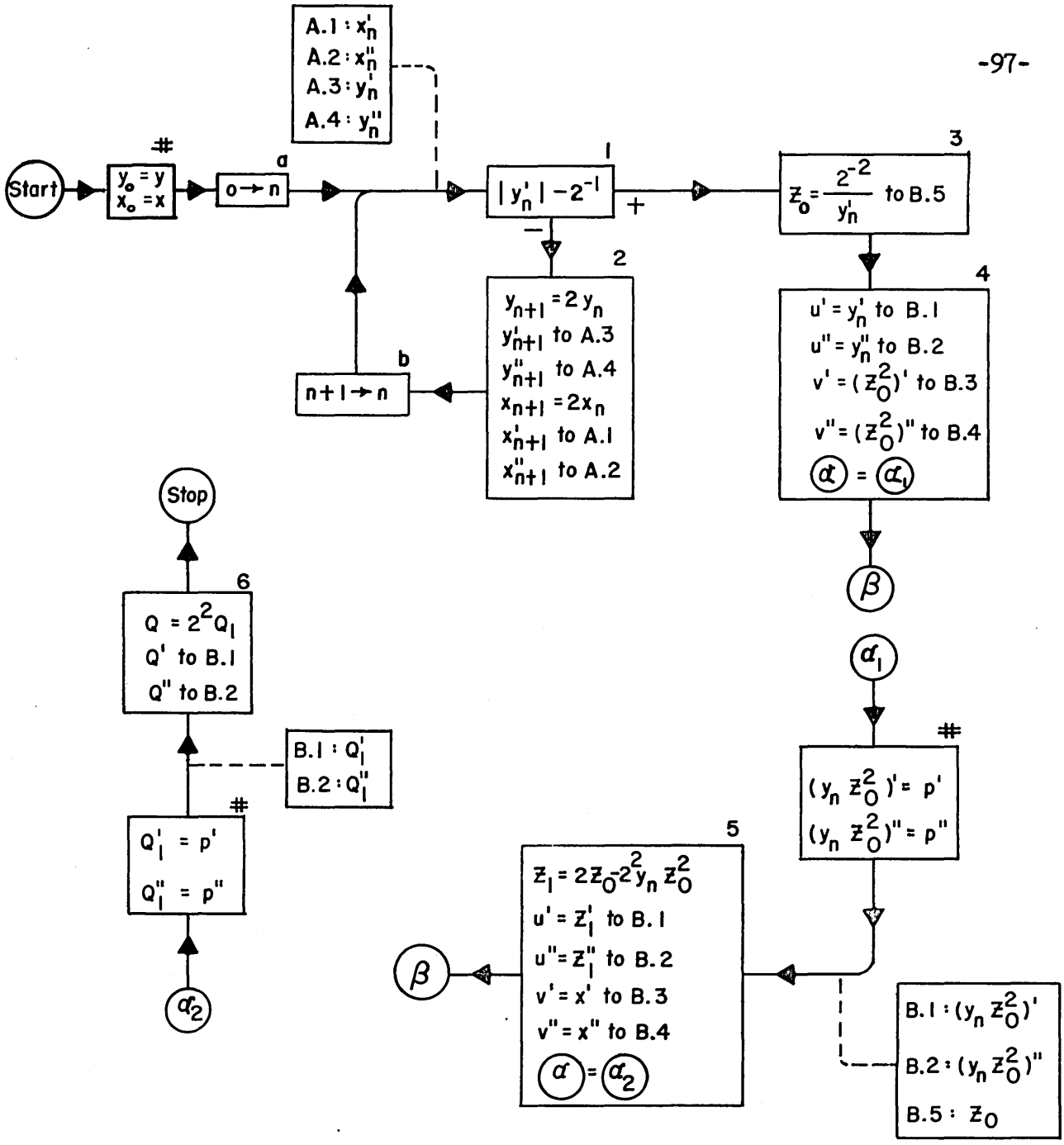
A.1: x'	A.3: y'
A.2: x"	A.4: y"

The number 2^{-39} is needed, and it is stored in A.5 as

$$A.5: \quad -2^{-39}$$

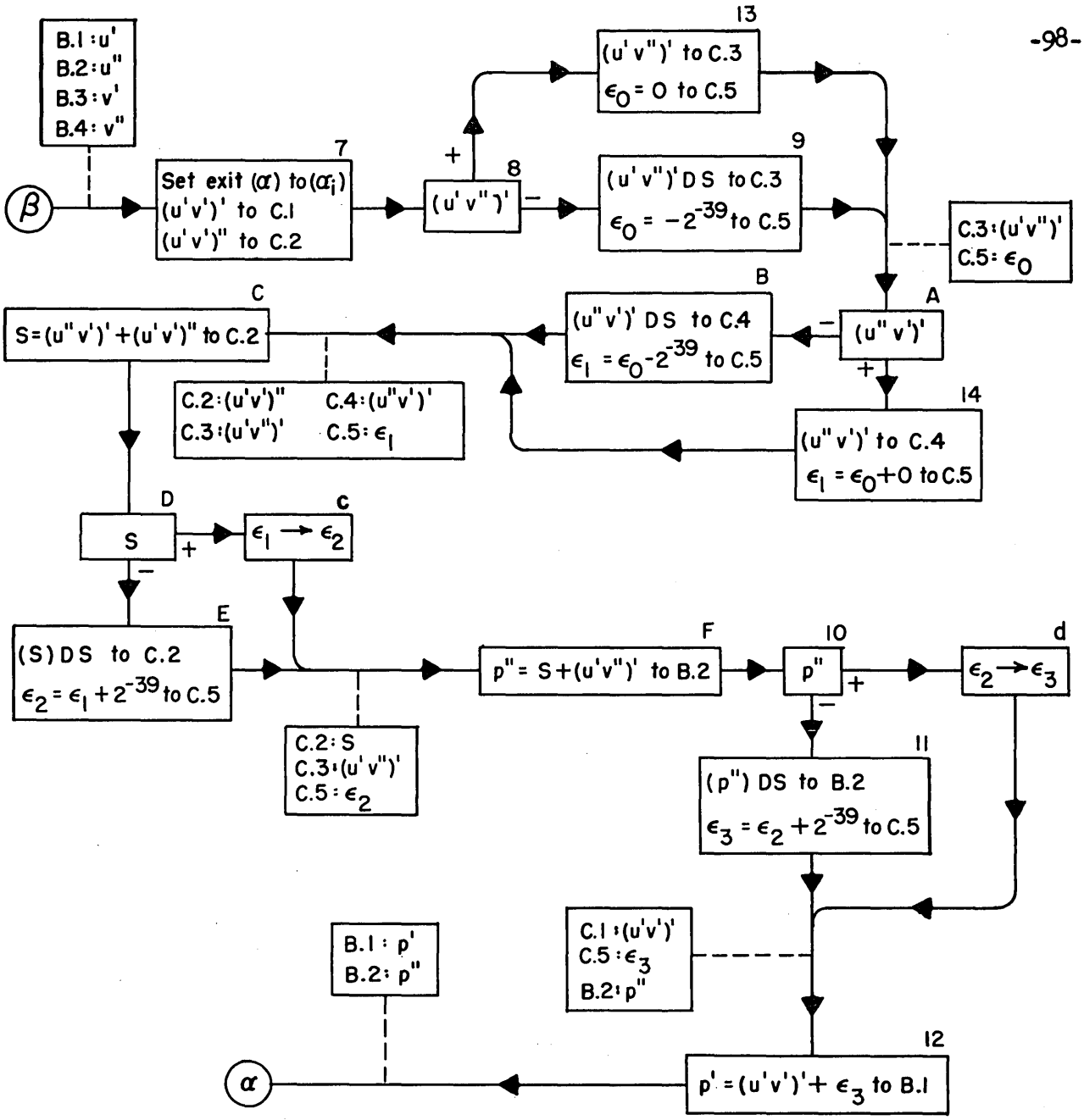
The addresses α_1 and α_2 need to be stored. They are stored as position marks at addresses A.6 and A.7.

$$A.6: \quad (\alpha_1)_o$$
$$A.7: \quad (\alpha_2)_o$$



DOUBLE PRECISION DIVISION

FIG. 17



DOUBLE PRECISION DIVISION

FIG. 18

Ten words of intermediate storage are needed for the computation. This storage is designated as

B.1	C.1
B.2	C.2
B.3	C.3
B.4	C.4
B.5	C.5

The coding is:

Box 1

1.	m⟶AcM	A.3	$\lvert y'_n \rvert$ to R2
2.	a⟶Ah	-1/2	$\lvert y'_n \rvert - 1/2$ in R2
3.	C	3,1	

Box 2

1.	m⟶Ac	A.4	y''_n to R2	
2.	L(1)		$2y''_n$ in R2	
3.	m⟶Q	A.3	y'_n to R4	
4.	L(1)		$2(y'_n + 2^{-39}y''_n)$ in R4 and R2	
5.	Q⟶m	A.3		y'_{n+1} to A.3
6.	R(1)			
7.	DS			
8.	A⟶m	A.4		y''_{n+1} to A.4
9.	m⟶Ac	A.2	x''_n to R2	
A.	L(1)		$2x''_n$ in R2	
B.	m⟶Q	A.1	x'_n to R4	
C.	L(1)		$2(x'_n + 2^{-39}x''_n)$ in R4 and R2	
D.	Q⟶m	A.1		x'_{n+1} to A.1
E.	R(1)			
F.	DS			
10.	A⟶m	A.2		x''_{n+1} to A.2
11.	T	1,1		

Box 3

1.	a⟶Ac	1/4	1/4 to R2	
2.	÷	A.3	$Z_o = 1/4 \div y'$ in R4	
3.	Q⟶m	B.5		Z_o to B.5

Box 4

1.	m⟶Q	A.3	y' to R4	
2.	Q⟶m	B.1		y' to B.1
3.	m⟶Q	A.4	y'' to R4	
4.	Q⟶m	B.2		y'' to B.2
5.	m⟶Q	B.5	Z_o to R4	
6.	X'	B.5	$(Z_o^2)'$ in R2, $(Z_o^2)''$ in R4	
7.	A⟶m	B.3		$(Z_o^2)'$ to B.3
8.	Q⟶m	B.4		$(Z_o^2)''$ to B.4
9.	m⟶Ac	A.6	$(\alpha_1)_o$ to R2	
A.	T	(β)		

Box 5

1.	m⟶Ac	B.2	$(yZ_o^2)''$ to R2	
2.	L(1)		$2(yZ_o^2)''$ in R2	
3.	m⟶Q	B.1	$(yZ_o^2)'$ to R4	
4.	L(2)		$2^2(yZ_o^2)'$ in R4, $2^3(yZ_o^2)''$ in R2	
5.	Q⟶m	B.1		$(2^2yZ_o^2)'$ to B.1
6.	R(1)		$(2^2yZ_o^2)''$ in R2	
7.	DS		$(2^2yZ_o^2)''$ in R2	
8.	A⟶m	B.2		$(2^2yZ_o^2)''$ to B.2
9.	m⟶Ac-	B.1	$-(2^2yZ_o^2)'$ to R2	
A.	m⟶Ac	A.5	$-(2^2yZ_o^2)' - 2^{-39}$ in R2	
B.	m⟶Ah	B.5	$Z_o - (2^2yZ_o^2)' - 2^{-39}$ in R2	
C.	m⟶Ah	B.5	$Z'_1 = 2Z_o - (2^2yZ_o^2)' - 2^{-39}$ in R2	
D.	A⟶m	B.1		Z' to B.1
E.	m⟶Ac-	B.2	$2 - (2^2yZ_o^2)''$ to R2	
F.	DS		$Z''_1 = 1 - (2^2yZ_o^2)''$ in R2	
10.	A⟶m	B.2		Z''_1 to B.2
11.	m⟶Ac	A.1	x' to R2	
12.	A⟶m	B.3		x' to B.3
13.	m⟶Ac	A.2	x'' to R2	
14.	A⟶m	B.4		x'' to B.4
15.	m⟶Ac	A.7	$(\alpha_2)_o$ to R2	
16.	T	(β)		

Box 6

1.	m⟶Ac	B.2	Q_1'' to R2	
2.	L(1)		$2Q_1''$ in R2	
3.	m⟶Q	B.1	Q_1' in R4	
4.	L(2)		Q' in R4, 2Q" in R2	
5.	Q⟶m	B.1		Q' to B.1
6.	R(1)		Q" in R2	
7.	DS			
8.	A⟶m	B.2		Q" to B.2
9.	Stop			

Box 7

1.	S⟶m	12,4	$(\alpha_i)_o$ in R2	α_i to (8-19)11,4
2.	m⟶Q	B.1	u' to R4	
3.	X'	B.3	(u'v')' in R2, (u'v')" in R4	
4.	A⟶m	C.1		(u'v')' to C.1
5.	Q⟶m	C.2		(u'v')" to C.2

Box 8

1.	m⟶Q	B.1	u' to R4	
2.	X	B.4	(u'v")' in R2	
3.	C	9,3		

Box 9

1.	m⟶Q	A.5	$\epsilon_o = -2^{-39}$ to R4	
2.	DS		(u'v")'DS in R2	
3.	A⟶m	C.3		(u'v")' to C.3
4.	Q⟶m	C.5		ϵ_o to C.5

Box A

1.	m⟶Q	B.2	u" to R4	
2.	X	B.3	(u"v')' in R2	
3.	C	B,3		

Box B

1.	m⟶Q	A.5	-2^{-39} to R4
2.	DS		(u"v')'DS in R2
3.	A⟶m	C.4	(u"v')' to C.4
4.	m⟶Ac	C.5	ϵ_o to R2
5.	m⟶Ah	800	$\epsilon_1 = \epsilon_o$ + (contents of R4)
6.	A⟶m	C.5	ϵ_1 to C.5

Box C

1.	m⟶Ac	C.4	(u"v')' to R2
2.	m⟶Ah	C.2	S = (u"v')' + (u'v')" in R2
3.	A⟶m	C.2	S to C.2

Box D

1.	C	F,1	

Box E

1.	DS		(S)DS in R2
2.	A⟶m	C.2	S to C.2
3.	m⟶Ac	C.5	ϵ_1 to R2
4.	m⟶Ah-	A.5	$\epsilon_2 = \epsilon_1 + 2^{-39}$ in R2
5.	A⟶m	C.5	ϵ_2 to C.5

Box F

1.	m⟶Ac	C.2	S to R2
2.	m⟶Ah	C.3	p" = S + (u'v")' in R2
3.	A⟶m	B.2	p" to B.2

Box 10

1.	C	12,1	

Box 11

1.	DS		(p")DS in R2
2.	A⟶m	B.2	p" to B.2
3.	m⟶Ac	C.5	ϵ_2 to R2
4.	m⟶Ah-	A.5	$\epsilon_3 = \epsilon_2 + 2^{-39}$ in R2
5.	A⟶m	C.5	ϵ_3 to C.5

Box 12

1.	m⟶Ac	C.1	$(x'y')'$ to R2
2.	m⟶Ah	C.5	$p' = (x'y')' + \epsilon_3$ in R2
3.	A⟶m	B.1	p' to B.1
4.	T	$[\alpha]$	

The double precision shift in Box 2 is done by placing twice the less significant part of the number into R2. Its first significant bigit is then in the sign position of R2. The more significant part of the number is put into R4. A left shift of 1 now shifts the 78 bigits correctly as the sign bigit position of R2 fills into 2^{-39} bigit position of R4. The quantity in R4 is stored. The quantity in R2 is then shifted right 1 and the sign bigit is set to 0. This is done to keep the less significant part of the number, the part in R2, in correct form.

In Box 5, the complement of $2^2yZ_o^2$ is needed. Recall that the complement of a 78 bigit number is

$$\begin{aligned} 2 - 2^2yZ_o^2 &= 2 - \left\{(2^2yZ_o^2)' + 2^{-39}(2^2yZ_o^2)''\right\} \\ &= 2 - (2^2yZ_o^2)' - 2^{-39} + 2^{-39}\left[1 - (2^2yZ_o^2)''\right] \end{aligned}$$

Since the complement is to be added to a standard 39 bigit number, the less significant part has only to be complemented as indicated and sent into storage. The more significant part is complemented as indicated and added to the $2Z_o$ and the result is sent to storage.

In the multiplication routine $(u'v'')'$ and $(u''v')'$ are formed using multiplication with round-off. This accounts for the possible contributions from the neglected terms involving the coefficient 2^{-78}. This does not, however, always give a correct round-off.

Note that Box 13 is not coded. It is not necessary to code it if the conditional transfer of Box 8 goes to Box 9, Instruction 3. Since $(u'v'')'$ is formed as a multiply with round-off, R4 contains 0. This 0 is set to ϵ_o and Instruction 4 of Box 9 stores it correctly. Instruction 3 of Box 9 stores the $(u'v'')'$. Similarly, Box 14 is not coded and the conditional transfer of Box A transfers into Instruction 3 of Box B.

Note the last two instructions of Box 4 and Box 5. Box 4 brings $(\alpha_1)_o$ into R2 and then transfers to the multiplication routine. Box 5 brings $(\alpha_2)_o$ into R2 and then transfers to the multiplication routine.

The first instruction of the multiplication routine then substitutes the address α_1 or α_2 as the case may be, into the transfer instruction at the end of the multiplication routine.

There are, in all, 107 instructions in the code; which is 54 words. The code is to start at Word 0; therefore Words 0 through 35, hexadecimally, are the code. Words 36 through 3C are A.1 through A.7, respectively. Words 3D through 41 are B.1 through B.5, and Words 42 through 46 are C.1 through C.5, respectively.

The coding paired into words is:

0.	m⟶AcM	038	a⟶Ah	C00
1.	C	00A	m⟶Ac	039
2.	L(1)	001	m⟶Q	038
3.	L(1)	001	Q⟶m	038
4.	R(1)	001	DS	000
5.	A⟶m	039	m⟶Ac	037
6.	L(1)	001	m⟶Q	036
7.	L(1)	001	Q⟶m	036
8.	R(1)	001	DS	000
9.	A⟶m	037	T	000
A.	a⟶Ac	200	÷	038
B.	Q⟶m	041	m⟶Q	038
C.	Q⟶m	03D	m⟶Q	039
D.	Q⟶m	03E	m⟶Q	041
E.	X'	041	A⟶m	03F
F.	Q⟶m	040	m⟶Ac	03B
10.	T	020	m⟶Ac	03E
11.	L(1)	001	m⟶Q	03D
12.	L(2)	002	Q⟶m	03D
13.	R(1)	001	DS	000
14.	A⟶m	03E	m⟶Ac-	03D
15.	m⟶Ah	03A	m⟶Ah	041
16.	m⟶Ah	041	A⟶m	03D
17.	m⟶Ac-	03E	DS	000
18.	A⟶m	03E	m⟶Ac	036
19.	A⟶m	03F	m⟶Ac	037
1A.	A⟶m	040	m⟶Ac	03C
1B.	T	020	m⟶Ac	03E
1C.	L(1)	001	m⟶Q	03D

1D.	L(2)	002	Q⟶m	03D
1E.	R(1)	001	DS	000
1F.	A⟶m	03E	Stop	
20.	S⟶m	035	m⟶Q	03D
21.	X'	03F	A⟶m	042
22.	Q⟶m	043	m⟶Q	03D
23.	X	040	C	025
24.	m⟶Q	03A	DS	000
25.	A⟶m	044	Q⟶m	046
26.	m⟶Q	03E	X	03F
27.	C'	028	m⟶Q	03A
28.	DS	000	A⟶m	045
29.	m⟶Ac	046	m⟶Ah	800
2A.	A⟶m	046	m⟶Ac	045
2B.	m⟶Ah	043	A⟶m	043
2C.	C	02F	DS	000
2D.	A⟶m	043	m⟶Ac	046
2E.	m⟶Ah-	03A	A⟶m	046
2F.	m⟶Ac	043	m⟶Ah	044
30.	A⟶m	03E	C'	033
31.	DS	000	A⟶m	03E
32.	m⟶Ac	046	m⟶Ah-	03A
33.	A⟶m	046	m⟶Ac	042
34.	m⟶Ah	046	A⟶m	03D
35.	T'	[ ]		
36.	A.1	x'		
37.	A.2	x"		
38.	A.3	y'		
39.	A.4	y"		
3A.	A.5	-2^{-39}	FFF ••• FF	
3B.	A.6	$(\alpha_1)_o$	0001000010	
3C.	A.7	$(\alpha_2)_o$	0001B0001B	
3D.	B.1			42. C.1
3E.	B.2			43. C.2
3F.	B.3			44. C.3
40.	B.4			45. C.4
41.	B.5			46. C.5

Problem 10

The problems previously discussed have all been of an analytical character where the efficiency of solution is dependent upon the speed and flexibility of the arithmetic unit. We now consider a problem of a combinatorial nature which falls into a class of problems where the efficiency of solution depends on the flexibility of the logical control. The problem is a simple sorting procedure.

A set of N numbers, subject to no degree of monotony whatever, is to be sorted into a monotonic decreasing sequence. In order to simplify the discussion, we assume that the number of numbers to be sorted is

$$N = 2^P$$

where P is a positive integer.

The sorting is accomplished by repeated meshings of groups of numbers. Meshing is the process of combining groups of elements (numbers) in a prescribed fashion. For the present sorting procedure we are meshing groups two at a time. Two groups, each monotonic decreasing, are meshed into a single monotonic decreasing group; e.g., groups ξ and η of length $\underline{b}$ and $\underline{c}$ elements, respectively, (where the elements of ξ and η are in a monotonic decreasing sequence) are meshed into a group

$$\nu = \xi + \eta$$

of length $\underline{b + c}$ elements where ν is also a monotonic decreasing sequence. Since we have restricted the N numbers to be sorted to be

$$N = 2^P$$

we may without further loss of generality say that any two groups to be meshed are to contain the same number of elements.

The procedure is as follows: Consider the original sequence of numbers as N groups, where each group contains one element. These N groups are then meshed two at a time into N/2 groups each containing two elements. The N/2 groups are meshed two at a time into N/4 groups each containing four elements. This meshing process is continued until the sorting is complete (one group of N elements is formed). In each of the meshings the monotonic decreasing sequence is preserved. Hence, for the various meshings there are $N/2^i$ groups of 2^i elements each, where i $(=1,2\cdots N/2)$ specifies the particular meshing.

The meshing of two groups ξ and η is done as follows. Each group contains J numbers and the numbers x_i belong to ξ, and y_j belong to η. The groups are monotonic decreasing so

$$x_i \geq x_{i+1} \quad \text{and} \quad y_j \geq y_{j+1} \qquad \text{where } i,j\ (=1,2\cdots J)$$

The groups ξ and η are to be meshed into a group ν with elements called v_n where

$$v_n \geq v_{n+1} \qquad n\ (=1,2\cdots 2J)$$

The elements x_1 and y_1 are compared. Then

$$\text{if} \quad (1) \quad x_1 \geq y_1, \quad v_1 = x_1$$

$$\text{or if} \quad (2) \quad x_1 < y_1, \quad v_1 = y_1$$

If (1) holds, then x_2 is compared with y_1. Then

$$\text{if} \quad (3) \quad x_2 \geq y_1, \quad v_2 = x_2$$

$$\text{or if} \quad (4) \quad x_2 < y_1, \quad v_2 = y_1$$

However, if (2) holds rather than (1), x_1 is compared with y_2. Then

$$\text{if} \quad (5) \quad x_1 \geq y_2, \quad v_2 = x_1$$

$$\text{or if} \quad (6) \quad x_1 < y_2, \quad v_2 = y_2$$

The meshing of elements x_i and y_j follows the above:

$$\text{if} \quad (a) \quad x_i \geq y_j, \quad v_n = x_i \qquad (n = i + j - 1)$$

$$\text{or if} \quad (b) \quad x_i < y_j, \quad v_n = y_j$$

If (a) holds i and n are increased by 1 and the process is repeated. If (b) holds j and n are increased by 1 and the process is repeated. The meshing continues until either all of the numbers x_i or all of the numbers y_j are incorporated into ν. The remaining elements of the non-exhausted set are then directly included as the last elements of ν.

A meshing of two groups, each containing J elements, needs at most $2J - 1$ comparisons of the elements to complete the meshing.

The number of elements involved in a sorting may often exceed the capacity of the electrostatic memory; hence, we consider the problem which requires the magnetic drum. However, we further simplify the discussion and assume that each drum track contains 64 $(=2^6)$ words rather than the actual 50 words.

Once the sorting procedure given here is understood, it is easily generalized to any number of elements and to any number of words per drum track; e.g., 50 in our instance.

The N numbers subject to no degree of monotony whatever are stored on the drum on M consecutive tracks. The numbers on the drum are considered as two sets, X_1 and Y_1, where X_1 is the first N/2 numbers on the drum and Y_1 is the remainder. X_1 and Y_1 each contain N/2 groups of one number. The groups of X_1 are meshed with the groups of Y_1 to form a set V_1 of N/2 groups of two numbers each. To accomplish this initial step, the first track (64 numbers) of X_1 and the first track of Y_1 are brought into the memory. The first number of X_1 is meshed with the first number of Y_1 and the two are stored properly in the electrostatic memory. This is repeated with the second elements of the sets, and so on. When 64 numbers have been meshed into groups of two, the 64 numbers are sent to the first drum track of the second set of M tracks on the drum; when 64 more numbers have been meshed they are then sent to the drum, and so on, until the entire set X_1 has been meshed with Y_1. Whenever the 64 numbers from either the set X_1 or Y_1 have been exhausted, another track of 64 numbers of the appropriate set is brought into the memory. The set V_1 consists of N/2 groups (each of two elements) where each group is a monotonic decreasing sequence.

The set V_1 is now considered as two sets X_2 and Y_2, where X_2 is the first N/2 numbers and Y_2 the remaining numbers. X_2 and Y_2 each contain N/4 groups (of two elements) and each group has the desired monotony. The groups of the set X_2 are meshed with the groups of the set Y_2 to form a set V_2 of N/4 groups (of four elements) where each group is a monotonic decreasing sequence.

We then have the following inductive process: Two sets of numbers X_p and Y_p each contain $N/2^p$ groups (of 2^{p-1} elements). The groups of X_p are meshed with the groups of Y_p to form the set V_p, where V_p contains $N/2^p$ groups (of 2^p elements). The set X_{p+1} is

the first N/2 numbers of V_p, and Y_{p+1} the remaining numbers. X_{p+1} and Y_{p+1} each contain $N/2^{p+1}$ groups (of 2^p numbers).

For a further discussion and elaboration of the sorting procedure we draw the flow diagram.

The flow diagram contains three induction loops. They are:

(i) the induction concerned with the mesh cycles

(ii) that concerned with the meshing of a group within the sets during any mesh cycle

(iii) that concerned with the transfer of elements between the memory and the drum

Eleven distinct indices (variables of induction) are needed in the flow diagram to describe the inductions.

The index $\underline{p}$ (=0,1,2$\cdots$P) describes the induction over the mesh cycles. It is used in connection with the sets X_p and Y_p. It keeps account of the mesh cycle. $\underline{p}$ has no relevance other than as an index, and it need not be stored.

The index $\underline{n}$ (=1,2$\cdots$N), where $\underline{N}$ is the total number of elements being sorted, indicates the current number of elements that have been meshed during any mesh cycle $\underline{p}$. It is also used in a discrimination to indicate the completion of the p^{th} mesh cycle; therefore, $\underline{n}$ is a stored quantity.

The indices $\underline{i}$, $\underline{j}$, and $\underline{k}$ describe the induction concerned with the meshing of the groups within the two sets.

The index $\underline{k}$ (=1,2,$2^2\cdots$) indicates the number of elements in the groups of the two sets X_p and Y_p. $\underline{k}$ and $\underline{p}$ are simply related: during mesh cycle $\underline{p}$ the number of elements in the groups within X_p and Y_p is $k = 2^p$.

The indices $\underline{i}$ and $\underline{j}$ indicate the elements x_i and y_j of the groups within the sets X_p and Y_p. The indices $\underline{i}$ and $\underline{j}$ are used in discriminations with $\underline{k}$ to indicate the completion of the meshing of any two groups within the sets; hence, $\underline{i}$, $\underline{j}$, and $\underline{k}$ are all stored quantities. Rather than using $\underline{i}$ and $\underline{j}$ as indices which range over i,j (=1,2$\cdots$k),

we let $\underline{i}$ and $\underline{j}$ be such that i,j (=1,2$\cdots$N/2). That is, $\underline{i}$ and $\underline{j}$ range over the total number of elements of X_p and Y_p. The discrimination of $\underline{i}$ and $\underline{j}$ cannot then be done directly with the index $\underline{k}$, since they are not reset to $\underline{1}$ at the time they become equal to $\underline{k}$; in fact, they continue increasing until they reach N/2. To accomplish the desired discrimination, an index K (=k, 2k, 3k$\cdots$N/2) is introduced; and when

$$1 \leq i,\ j \leq k \qquad \text{then } K = k$$
$$k+1 \leq i,\ j \leq 2k \qquad \text{then } K = 2k$$

and so on, until K = N/2.

At the completion of each mesh cycle $\underline{p}$, the index $\underline{k}$ is doubled; i.e., when $\underline{p}$ is increased by $\underline{1}$ to become $\underline{p+1}$, then $k = 2^p$ is increased to $k = 2^{p+1}$. This index $\underline{k}$ is used to determine the completion of the sorting. The sorting is complete when p = P, at which time $k = 2^P = N$; hence, a discrimination on k - N becomes positive for the first time when $k = 2^p$, and the process is terminated.

The indices $\underline{i'}$, $\underline{j'}$, and $\underline{n'}$ are the indices describing the induction concerned with the drum and i', j', n' (=1,2$\cdots$64). The indices i' and j' indicate when the 64 elements x_i or y_j which are in the memory are meshed. They also keep account of which two elements of the 64 elements x_i and y_j are being meshed. Whenever i' or j' reaches 64, a new track of elements x_i or y_j, respectively, is brought from the drum into the memory. The index n' indicates the number of elements x_i and y_j that have been meshed and stored in locations in the memory. When n' reaches its maximum value, the 64 elements which have been meshed and stored in the electrostatic memory are subsequently sent to the drum. The indices i', j', and n' are needed in discriminations and in addresses; hence, they are stored numbers. There are three indices concerned with the drum which are, in themselves, addresses. They are Tx, Ty, and Tv. Tx is the address of the drum track which contains the 64 elements of the set X_p that are to be sent into the memory. Ty is the corresponding drum address for Y_p. The index Tv is the address of the drum track upon which the 64 meshed elements are to be stored. Now that we have defined the necessary indices, the flow diagram of Figure 19 may be examined in detail.

In what follows, decimal and hexadecimal numbers both enter into the discussion. The hexadecimal numbers usually refer to instructions and box numbers, hence entering only in the role of "labels" or "names." The decimal numbers are usually used where the numerical character of the number is significant. However, at places where there might be confusion if the number is intended to be decimal it is underlined.

Boxes 1, 2, and 3 set up the necessary indices. Boxes 4, 5, 6, 7, 8, 10, and 11 are the boxes of the meshing of the groups within the sets. Box 12 is an alternative box that indicates when the N elements have all been meshed. Boxes A, B, C, D, E, and F are the boxes concerned with the transfer of numbers between the drum and the memory. Boxes 13, 15, and 16 set up necessary values at the completion of one meshing of the sets X_p and Y_p in order to start the next cycle in the meshing. Box 14 determines when the entire process is completed.

Box 1 sets the index $k = (1)_o$ since the first meshing is in groups of one element. It sets the initial drum addresses for Tx, Ty, and Tv. It also sets the address $(\beta) = (\beta_1)$. This is discussed in more detail when Boxes 15 and 16 are discussed.

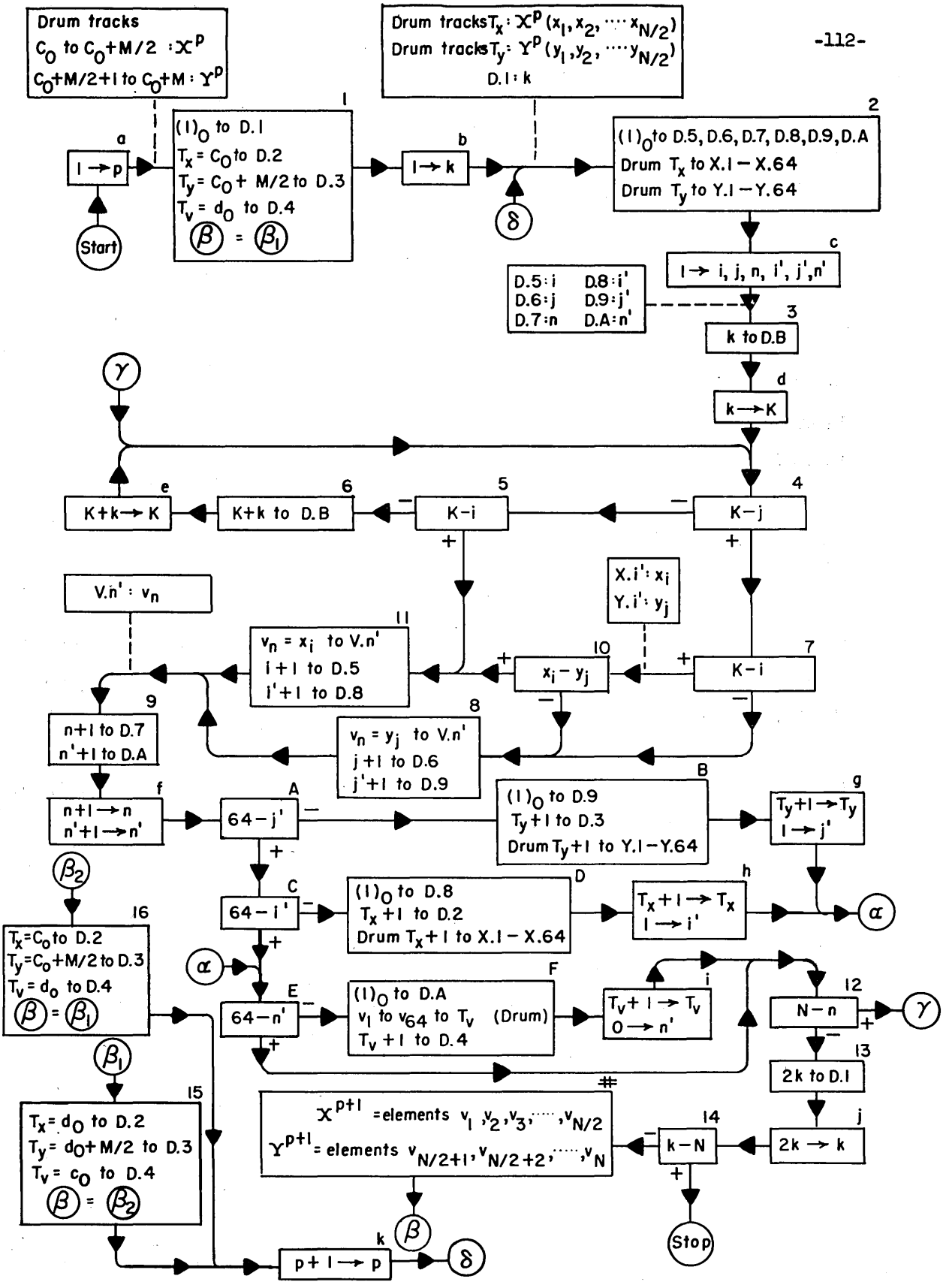
Box 2 sets up the indices k, j, n, i', j', and n'. These indices are all set to $(1)_o$. This box also sends the contents of tracks Tx and Ty into the memory.

Box 3 sets up an index K = k.

Box 4 is the alternative box that indicates when all of the elements y_j of a particular group of the set Y_p have been meshed. Boxes 5 and 7 indicate when all of the elements x_i of a particular group of the set X_p have been meshed. If the elements of the two groups have not been exhausted, the control proceeds to Box 10 to determine which is the larger, x_i or y_j.

If $x_i \geq y_j$, from Box 10 the control proceeds to Box 11, and $v_n = x_i$. i and i' are increased by $(1)_o$. If $x_i < y_j$, from Box 10 the control proceeds to Box 8, and $v_n = y_j$. j and j' are increased by $(1)_o$.

If all of the elements of a particular group of the set Y_p are meshed and stored, and the corresponding elements of the set X_p are not, the discrimination of Box 4 is negative and that of Box 5 is still positive; hence the control proceeds to Box 11, where the element $v_n = x_i$ is stored.



A SIMPLE SORTING PROCEDURE

FIG. 19

This condition holds until all of the elements of the particular group of the set X_p have been incorporated into the meshed sequence. A similar condition holds for the entry of the control from Box 7 to Box 8. In this instance, the elements of a particular group of the set X_p have all been exhausted and those of the set Y_p have not.

The Alternative Box 12 determines when N elements have been meshed. The control proceeds to Box 13 when this obtains, and the control proceeds to Box 4 when the meshing is not complete.

The Alternative Box A determines when $\underline{64}$ elements of Y_p have been meshed. If they have, Box B sends $\underline{64}$ new elements y_j into the memory.

Box C and Box D determine if $\underline{64}$ elements of the set X_p are exhausted, and if they are, $\underline{64}$ new elements x_i are sent to the memory.

Box E determines when $\underline{64}$ elements have been meshed and stored in the memory. Box F subsequently stores the $\underline{64}$ elements onto the drum.

Box 14 terminates the sorting process when k = N. However, if the sorting is not complete, Box 15 or Box 16 sets up the new initial drum addresses for subsequent meshing. Recall that in Box 1 the address $(\beta) = (\beta_1)$ was set up. This means that upon the first traversal through Box 14 the control proceeds to Box 15 as is desired. In Box 15 the address $(\beta) = (\beta_2)$ is set up so that on the next traversal of Box 14 the control proceeds to Box 16 where the address $(\beta) = (\beta_1)$ is restored, and so on, until the sorting is complete. Upon the completion of either Box 15 or Box 16, the control returns to Box 2, where the i, j, n, i', j', and n' indices are reset to $(1)_o$ in order to repeat the induction process.

The storage needed for the problem is as follows: The quantity $(1)_o$ is needed and

$$\text{B.1:} \quad (1)_o$$

The four initial drum track addresses are stored, scaled by 2^{-27} and

$$\text{B.2:} \quad c_o \cdot 2^{-27}$$
$$\text{B.3:} \quad (c_o + M/2)2^{-27}$$
$$\text{B.4:} \quad d_o \cdot 2^{-27}$$
$$\text{B.5:} \quad (d_o + M/2)2^{-27}$$

The quantity $1 \cdot 2^{-27}$ is needed for altering the drum track addresses and

$$\text{B.6:} \quad 1 \cdot 2^{-27}$$

Three memory addresses are needed. They are base addresses for the storage of the numbers x_i, y_j, and v_n; and they are designated $(X.0)_o$, $(Y.0)_o$ and $(V.0)_o$. The storage is

B.7: $(X.0)_o$

B.8: $(Y.0)_o$

B.9: $(V.0)_o$

where the address $(X.i')_o = (X.0)_o + (i')_o$, and X.i': x_i. Similarly, Y.j': y_j and V.n': v_n. The number $(\underline{64})_o$ is needed for discriminations and

B.A: $(\underline{64})_o$

The total number of elements $\underline{N}$ is needed and

B.B: $(N)_o$

The drum instructions occupy full words where bigits 28-39 specify an address to which the control transfers upon completion of the drum instructions. The addresses for these transfers need to be stored. Four such addresses are needed and they are

B.C: $(\text{Box } 12,\ 1)2^{-39}$

B.D: $(\text{Box } 2,\ 10)2^{-39}$

B.E: $(\text{Box } 3,\ 1)2^{-39}$

B.F: $(\text{Box E},\ 1)2^{-39}$

The addresses β_1 and β_2 are needed. They are stored as position marks in

B.10: $(\beta_1)_o$

B.11: $(\beta_2)_o$

Eleven words of intermediate storage are needed during the course of the computation. They are designated as D.1, D.2 ··· D.9, D.A, D.B. The required electrostatic storage for the numbers being meshed is 192 locations; the drum storage is 2M tracks.

The coding is:

Box 1

1.	m⟶Ac	B.1	$(1)_o$ to R2	
2.	A⟶m	D.1		$(1)_o$ to D.1
3.	m⟶Ac	B.2	$c_o \cdot 2^{-27}$ to R2	
4.	A⟶m	D.2		$T_x = c_o \cdot 2^{-27}$ to D.2
5.	m⟶Ac	B.3	$(c_o + M/2)2^{-27}$ to R2	
6.	A⟶m	D.3		$T_y = (c_o + M/2)2^{-27}$ to D.3
7.	m⟶Ac	B.4	$d_o \cdot 2^{-27}$ to R2	
8.	A⟶m	D.4		$T_v = d_o \cdot 2^{-27}$ to D.4

Box 1 (Cont.)				
9.	m⟶Ac	B.10	$(\beta_1)_o$ to R2	
A.	S⟶m	14,4		β_1 to (8-19)14,4
Box 2				
1.	m⟶Ac	B.1	$(1)_o$ to R2	
2.	A⟶m	D.5		$(1)_o$ to D.5
3.	A⟶m	D.6		$(1)_o$ to D.6
4.	A⟶m	D.7		$(1)_o$ to D.7
5.	A⟶m	D.8		$(1)_o$ to D.8
6.	A⟶m	D.9		$(1)_o$ to D.9
7.	A⟶m	D.A		$(1)_o$ to D.A
8.	m⟶Ac	D.2	T_x to R2	
9.	m⟶Ah	B.D	$T_x + (\text{Box } 2,10)2^{-39}$ in R2	
A.	HS⟶m	2,F		
B.	m⟶Ac	D.3	T_y to R2	
C.	m⟶Ah	B.E	$T_y + (\text{Box } 3,1)2^{-39}$ in R2	
D.	HS⟶m	2,11		
E.	D⟶m	X.1		$(x_1$ to $x_{\underline{64}})$ to X.1 to X.64
F.	$[T_x$	2,10]		
10.	D⟶m	Y.1		$(y_1$ to $y_{\underline{64}})$ to Y.1 to Y.64
11.	$[T_y$	3,1]		
Box 3				
1.	m⟶Ac	D.1	$(k)_o$ to R2	
2.	A⟶m	D.B		$(k)_o$ to D.B
Box 4				
1.	m⟶Ac	D.B	$(K)_o$ to R2	
2.	m⟶Ah-	D.6	$(K-j)_o$ in R2	
3.	C	7,1		
Box 5				
1.	m⟶Ac	D.B	$(K)_o$ to R2	
2.	m⟶Ah-	D.5	$(K-i)_o$ in R2	
3.	C	11,1		
Box 6				
1.	m⟶Ac	D.B	$(K)_o$ to R2	
2.	m⟶Ah	D.1	$(K+k)_o$ in R2	
3.	A⟶m	D.B		$(K+k)_o$ to D.B
4.	T	4,1		

Box 7

1.	m⟶Ac	D.B	$(K)_o$ to R2
2.	m⟶Ah-	D.5	$(K-i)_o$ to R2
3.	C	10,1	

Box 8

1.	m⟶Ac	B.8	$(Y.0)_o$ to R2
2.	m⟶Ah	D.9	$(Y.j')_o = (Y.0)_o + (j')_o$ in R2
3.	S⟶m	8,D	$Y_{j'}$ to (8-19)8,D
4.	m⟶Ac	D.6	$(j)_o$ to R2
5.	m⟶Ah	B.1	$(j+1)_o$ in R2
6.	A⟶m	D.6	$(j+1)_o$ to D.6
7.	m⟶Ac	D.9	$(j')_o$ to R2
8.	m⟶Ah	B.1	$(j'+1)_o$ in R2
9.	A⟶m	D.9	$(j'+1)_o$ to D.9
A.	m⟶Ac	B.9	$(V.0)_o$ to R2
B.	m⟶Ah	D.A	$(V.n')_o = (V.0)_o + (n')_o$ in R2
C.	S⟶m	8,E	V·n' to (8-19)8,E
D.	m⟶Ac	[Y.j']	$v_n = y_j$
E.	A⟶m	[V.n']	v_n to V.n'

Box 9

1.	m⟶Ac	D.7	$(n)_o$ to R2
2.	m⟶Ah	B.1	$(n+1)_o$ in R2
3.	A⟶m	D.7	$(n+1)_o$ to D.7
4.	m⟶Ac	D.A	$(n')_o$ to R2
5.	m⟶Ah	B.1	$(n'+1)_o$ in R2
6.	A⟶m	D.A	$(n'+1)_o$ to D.A

Box A

1.	m⟶Ac	B.A	$(\underline{64})_o$ to R2
2.	m⟶Ah-	D.9	$(\underline{64}-j')_o$ in R2
3.	C	C,1	

Box B

1.	m⟶Ac	D.3	T_y to R2
2.	m⟶Ah	B.6	$T_y + 1$ in R2
3.	A⟶m	D.3	$T_y + 1$ to D.3

Box B (Cont.)

4.	m⟶Ah	B.F	$T_y + 1 + (\text{Box E,1})2^{-39}$ in R2
5.	HS⟶m	B,9	
6.	m⟶Ac	B.1	$(1)_o$ to R2
7.	A⟶m	D.9	$(1)_o$ to D.9
8.	D⟶m	Y.1	$(y_j$ to $y_{j+\underline{64}})$ to Y.1 to Y.64
9.	$[T_y + 1$	E,1]	

Box C

1.	m⟶Ac	B.A	$(\underline{64})_o$ to R2
2.	m⟶Ah-	D.8	$(\underline{64}\text{-i}')_o$ in R2
3.	C	E,1	

Box D

1.	m⟶Ac	D.2	T_x to R2
2.	m⟶Ah	B.6	$T_x + 1$ in R2
3.	A⟶m	D.2	$T_x + 1$ in D.2
4.	m⟶Ah	B.F	$T_x + 1 + (\text{Box E,1})\,2^{-39}$ in R2
5.	HS⟶m	E,9	
6.	m⟶Ac	B.1	$(1)_o$ to R2
7.	A⟶m	D.8	$(1)_o$ to D.8
8.	D⟶m	X.1	$(x_i$ to $x_{i+\underline{64}})$ to X.1 to X.64
9.	$[T_x + 1$	E,1]	

Box E

1.	m⟶Ac	B.A	$(64)_o$ to R2
2.	m⟶Ah-	D.A	$(64\text{-n}')_o$ in R2
3.	C	12,1	

Box F

1.	m⟶Ac	D.4	T_v to R2
2.	m⟶Ah	B.6	$T_v + 1$ in R2
3.	A⟶m	D.4	$T_v + 1$ to D.4
4.	m⟶Ah-	B.6	T_v in R2
5.	m⟶Ah	B.C	$T_v + (\text{Box 12,1})2^{-39}$ in R2
6.	HS⟶m	10,A	
7.	m⟶Ac	B.1	$(1)_o$ to R2
8.	A⟶m	D.A	$(1)_o$ to D.A
9.	m⟶D	V.1	$(v_n$ to $V_{n+64})$ to T_v
A.	$[T_v$	12,1]	

Box 10

1.	m⟶Ac	B.7	$(X.0)_o$ to R2
2.	m⟶Ah	D.8	$(X.i')_o = (X.0)_o + (i')_o$ in R2
3.	S⟶m	10,7	X.i' to (8-19)10,7
4.	m⟶Ac	B.8	$(Y.0)_o$ to R2
5.	m⟶Ah	D.9	$(Y.j')_o = (Y.0)_o + (j')_o$ in R2
6.	S⟶m	10,8	Y.j' to (8-19)10,8
7.	m⟶Ac	[X.i']	x_i to R2
8.	m⟶Ah-	[Y.j']	$x_i - y_j$ in R2
9.	C	11,1	
A.	T	8,1	

Box 11

1.	m⟶Ac	B.7	$(X.0)_o$ to R2
2.	m⟶Ah	D.8	$(X.i')_o = (X.0)_o + (i')_o$ in R2
3.	S⟶m	11,D	X.i' to (8-19)11,D
4.	m⟶Ac	B.9	$(V.0)_o$ to R2
5.	m⟶Ah	D.A	$(V.n')_o = (V.0)_o + (n')_o$ in R2
6.	S⟶m	11,E	V.n' to (8-19)11,E
7.	m⟶Ac	D.5	$(i)_o$ to R2
8.	m⟶Ah	B.1	$(i+1)_o$ in R2
9.	A⟶m	D.5	$(i+1)_o$ to D.5
A.	m⟶Ac	D.8	$(i')_o$ to R2
B.	m⟶Ah	B.1	$(i'+1)_o$ in R2
C.	A⟶m	D.8	$(i'+1)_o$ to D.8
D.	m⟶Ac	[X.i']	$v_n = x_i$ to R2
E.	A⟶m	[V.n']	v_n to V.n'
F.	T	9,1	

Box 12

1.	m⟶Ac	B.B	$(N)_o$ to R2
2.	m⟶Ah-	D.7	$(N-n)_o$ in R2
3.	C	4,1	

Box 13

1.	m⟶Ac	D.1	$(k)_o$ to R2
2.	L(1)	1	$(2k)_o$ in R2
3.	A⟶m	D.1	$(2k)_o$ to D.1

Box 14

1.	m⟶Ac	D.1	$(k)_o$ to R2
2.	m⟶Ah-	B.B	$(k-N)_o$ in R2
3.	C	16,A	
4.	T	[β]	

Box 15

1.	m⟶Ac	B.4	$T_x = d_o \cdot 2^{-27}$ to R2
2.	A⟶m	D.2	T_x to D.2
3.	m⟶Ac	B.5	$T_y = (d_o + M/2)2^{-27}$ to R2
4.	A⟶m	D.3	T_y to D.3
5.	m⟶Ac	B.2	$T_v = c_o \cdot 2^{-27}$ to R2
6.	A⟶m	D.4	T_v to D.4
7.	m⟶Ac	B.11	$(\beta_2)_o$ to R2
8.	S⟶m	14,4	β_2 to (8-19)14,4
9.	T	2,1	

Box 16

1.	m⟶Ac	B.2	$c_o \cdot 2^{-27}$ to R2
2.	A⟶m	D.2	$T_x = c_o \cdot 2^{-27}$ to D.2
3.	m⟶Ac	B.3	$(c_o + M/2)2^{-27}$ to R2
4.	A⟶m	D.3	$T_y = (c_o + M/2)2^{-27}$ to D.3
5.	m⟶Ac	B.4	$d_o \cdot 2^{-27}$ to R2
6.	A⟶m	D.4	$T_v = d_o \cdot 2^{-27}$ to D.4
7.	m⟶Ac	B.10	$(\beta_1)_o$ to R2
8.	S⟶m	14,4	β_1 to (8-19)14,4
9.	T	2,1	
A.	Stop		

Recall that the magnetic drum instructions each occupy a full word. The drum instructions are:

"m⟶D	BD	Read 50 successive words from the memory starting with the word at the address specified by bigits 8-19 of the instruction. Write these 50 words into the drum on the track specified by bigits 20-27. Then transfer the control to the left-hand instruction of the word at the address specified by the bigits 28-39.
D⟶m	BC	Read the 50 words from the track of the drum specified by bigits 20-27 of the instruction. Write these words into 50 successive memory locations starting with the address specified by bigits 8-19. Then transfer the control to the left-hand instruction of the word at the address specified by bigits 28-39."

For the present problem we assume that 64 words are transferred, rather than the 50 expressed by the instructions.

Instructions E and F of Box 2 comprise a drum instruction. In the final coding these two instructions must be in the same word. Instructions E and F are interpreted as: Read 64 words from track Tx of the drum, and write them into the memory at the addresses X.1 through X.64; then transfer the control to Instruction 10 of Box 2. This means that Instruction 10 of Box 2 must appear on the left side of an instruction word in the final coding.

Note that Instruction F of Box 2, the right-hand 20 bigits of the drum instruction, is formed in R2 by Instructions 8 and 9 and then sent to F by an HS$\longrightarrow$m instruction. This is necessary since Tx is a variable address. (In Box 2, Tx may be either c_o or d_o.) There is no instruction that will modify only bigits 20-27 of a word in the memory, so one method of altering the drum track address is to modify bigits 20-39 of the drum instruction by an HS$\longrightarrow$m instruction. This method necessitates storing the address which is to constitute bigits 28-39, the transfer portion, of the instruction. Instruction 8 of Box 2 brings the track $Tx \cdot 2^{-27}$ into R2. Instruction 9 adds to this the address of (Box 2, Instruction 10)2^{-39}. The half-word substitution is then effected by Instruction A. In the final coding this must be an HS$\longrightarrow$m' instruction.

Instructions 10 and 11 of Box 2 also comprise a drum instruction where the right-hand 20 bigits, Instruction 11, are generated as discussed for the previous drum instruction.

Instructions 8 and 9 of Box B, Instructions 8 and 9 of Box D, and Instructions 9 and A of Box F are drum instructions. Note in Box B and Box D, where the coding would normally end with a transfer instruction to send the control to Box E, Instruction 1, and in Box F, where the coding would normally end with a transfer to Box 12, Instruction 1, that the drum instruction performs this function. When possible then, it is useful to incorporate the drum instructions at points where transfers must normally take place.

The drum instructions in Boxes B, D, and F are similar in treatment to the previous discussion; hence the only further comment needed is that the drum instruction in Box F is an m$\longrightarrow$D instruction.

In the pairing of the coding into words one has to ascertain that Box 2, Instructions E and 10; Box 3, Instruction 1, Box D, Instruction 8; Box 12, Instruction 1; Box E, Instruction 1; and Box F, Instruction 9, all are the left-hand instructions of their respective instruction words.

We begin the coding at Word 000. There are, in all, 153 instructions, which is 76 1/2 code words. The code would normally occupy Words 000 through 04C hexadecimally. However, four "dummy" instructions need to be inserted to obtain the correct positioning of those instructions which must begin on the left. This adds two words to the code, and it occupies Words 000 through 04E.

The constant storage begins at 04F. The 17 words of B storage occupy locations 04F through 05F. The 11 words of intermediate storage occupy Words 060 through 06A.

The routine and storage occupy 107 words of the memory 000-06A. Numerical values are inserted for the addresses $(X.0)_o$, $(Y.0)_o$ and $(V.0)_o$. They are chosen as:

$$(X.0)_o = (06A)_o$$
$$(Y.0)_o = (0AA)_o$$
$$(V.0)_o = (0EA)_o$$

The algebraic addresses are left for the drum tracks as they depend in part on the total number of numbers being sorted. The quantity $(N)_o$ which is the total number of numbers is also left in algebraic notation.

The coding, with the necessary "dummy" instructions, is:

0.	m⟶Ac	04F	A⟶m	060
1.	m⟶Ac	050	A⟶m	061
2.	m⟶Ac	051	A⟶m	062
3.	m⟶Ac	052	A⟶m	063
4.	m⟶Ac	05E	HS⟶m'	044
5.	m⟶Ac	04F	A⟶m	064
6.	A⟶m	065	A⟶m	066
7.	A⟶m	067	A⟶m	068
8.	A⟶m	069	m⟶Ac	061
9.	m⟶Ah	05B	HS⟶m'	00C
A.	m⟶Ac	062	m⟶Ah	05C
B.	HS⟶m'	00D	(D S 000)	"dummy"

C.	D⟶m	06B	00000	
D.	D⟶m	0AB	00000	
E.	m⟶Ac	060	A⟶m	06A
F.	m⟶Ac	06A	m⟶Ah-	065
10.	C	014	m⟶Ac	06A
11.	m⟶Ah-	064	C	038
12.	m⟶Ac	06A	m⟶Ah	060
13.	A⟶m	06A	T	00F
14.	m⟶Ac	06A	m⟶Ah-	064
15.	C	033	m⟶Ac	056
16.	m⟶Ah	068	S⟶m'	01B
17.	m⟶Ac	065	m⟶Ah	04F
18.	A⟶m	065	m⟶Ac	068
19.	m⟶Ah	04F	A⟶m	068
1A.	m⟶Ac	057	m⟶Ah	069
1B.	S⟶m	01C	m⟶Ac	000
1C.	A⟶m	000	m⟶Ac	066
1D.	m⟶Ah	04F	A⟶m	066
1E.	m⟶Ac	069	m⟶Ah	04F
1F.	A⟶m	069	m⟶Ac	058
20.	m⟶Ah-	068	C	026
21.	m⟶Ac	062	m⟶Ah	054
22.	A⟶m	062	m⟶Ah	05D
23.	HS⟶m'	025	m⟶Ac	04F
24.	A⟶m	068	(D S 000)	"dummy"
25.	D⟶m	0AB	00000	
26.	m⟶Ac	058	m⟶Ah-	067
27.	C	02C	m⟶Ac	061
28.	m⟶Ah	054	A⟶m	061
29.	m⟶Ah	05D	HS⟶m'	02B
2A.	m⟶Ac	04F	A⟶m	067
2B.	D⟶m	06B	00000	
2C.	m⟶Ac	058	m⟶Ah-	069
2D.	C	040	m⟶Ac	063
2E.	m⟶Ah	054	A⟶m	063
2F.	m⟶Ah-	054	m⟶Ah	05A

30.	HS⟶m'	032	m⟶Ac	04F
31.	A⟶m	069	(D S 000)	"dummy"
32.	m⟶D	0EB	00000	
33.	m⟶Ac	055	m⟶Ah	067
34.	S⟶m	036	m⟶Ac	056
35.	m⟶Ah	068	S⟶m'	036
36.	m⟶Ac	000	m⟶Ah-	000
37.	C	038	T'	015
38.	m⟶Ac	055	m⟶Ah	067
39.	S⟶m	03E	m⟶Ac	057
3A.	m⟶Ah	069	S⟶m'	03E
3B.	m⟶Ac	064	m⟶Ah	04F
3C.	A⟶m	064	m⟶Ac	067
3D.	m⟶Ah	04F	A⟶m	067
3E.	m⟶Ac	000	A⟶m	000
3F.	T'	01C	(00000)	"dummy"
40.	m⟶Ac	059	m⟶Ah-	066
41.	C	00F	m⟶Ac	060
42.	L(1)	001	A⟶m	060
43.	m⟶Ac	060	m⟶Ah-	059
44.	C	04E	T	000
45.	m⟶Ac	052	A⟶m	061
46.	m⟶Ac	053	A⟶m	062
47.	m⟶Ac	050	A⟶m	063
48.	m⟶Ac	05F	HS⟶m'	044
49.	T	005	m⟶Ac	050
4A.	A⟶m	061	m⟶Ac	051
4B.	A⟶m	062	m⟶Ac	052
4C.	A⟶m	063	m⟶Ac	05E
4D.	HS⟶m'	044	T	005
4E.	STOP			
4F.	$(1)_o$			
50.	$c_o \cdot 2^{-27}$			
51.	$(c_o + M/2)2^{-27}$			
52.	$d_o \cdot 2^{-27}$			

53. $(d_o + M/2)2^{-27}$
54. $1 \cdot 2^{-27}$
55. $(X.0)_o = (06A)_o$
56. $(Y.0)_o = (0AA)_o$
57. $(V.0)_o = (0EA)_o$
58. $(64)_o = (040)_o$
59. $(N)_o$
5A. $12,1 = (040)2^{-39}$
5B. $2,10 = (00D)2^{-39}$
5C. $3,1 = (00E)2^{-39}$
5D. $E,1 = (02C)2^{-39}$
5E. $(\beta_1)_o = (CA045)_o$
5F. $(\beta_2)_o = (CB049)_o$
60. D.1
61. D.2
62. D.3
63. D.4
64. D.5
65. D.6
66. D.7
67. D.8
68. D.9
69. D.A
6A. D.B

The first drum instruction (Box 2, Instructions E and F) would not normally have been in one word in the paired coding. A "dummy" instruction, DS000, was inserted on the right-hand side of Word 00B in order to position the drum instruction correctly in Word 00C. The right 20 bigits of the drum instruction are not indicated as they are supplied from the problem. In punching a tape, five 0's could be punched for right-hand portion of Word 00C.

Upon positioning 00C correctly, the next drum instruction, Word 00D, and the first instruction of Box 3, Word 00E, are in the correct position.

The drum instruction in Box B, Instructions 8 and 9, also needed a "dummy" instruction inserted as the right-hand instruction of Word 024 to position the drum instruction correctly into Word 025. Similarly, the drum instruction in Box F, Instructions 9 and A, needs a "dummy" instruction inserted in the right-hand side of 031 to position the drum instruction correctly into 032. Instructions 1 of Boxes 12 and E need to be left-hand instructions since they are entered by the transfer portion of drum instructions. Box E is in the correct position as it begins on the left of Word 02C; however, Box 12 does not naturally begin on the left, hence a dummy instruction (00000) is inserted into 03F' following the last instruction of Box 11. Box 12 then begins on the left of Word 040 as is desired. The dummy instruction may be inserted as all 0's since the instruction is never executed by the control as Box 11 ends in a transfer instruction.

$(\beta_1)_o$ and $(\beta_2)_o$ are stored as

$$05E: \quad (\beta_1)_o = (CA045)_o$$

$$05F: \quad (\beta_2)_o = (CB049)_o$$

rather than as addresses. This is done since the entrances (β_1) and (β_2) which are Box 15,1 and Box 16,1 do not both begin on the same side of their respective words. The addresses (β_1) and (β_2) are supplied to Box 14, Instruction 4 (Word 044') by an HS$\longrightarrow$m' instruction; hence the order as well as the address is modified appropriately.

The sorting procedure as presented is valid only if all of the numbers have the same sign (i.e., either all positive or all negative). If numbers of mixed sign are to be sorted, Box 10 would need to be modified as numbers of opposite sign could presently cause spillage.

Problem 11

We evaluate and tabulate a sequence of values for sin x where the argument x is not given in any systematic order. The values of x are punched on paper tape for use in the sine computation. When sin x is determined for each value of x, it is stored with its argument as one word. The first 20 bigits (0-19) store x and bigits (20-39) store sin x. The values of x and sin x are then printed and punched by the flexowriter.

The method used for evaluating sin x is the Taylor's series expansion of the function.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

The following induction describes the series:

$$\sigma_1 = x \qquad \sum_1 = x$$

$$\sigma_3 = -\sigma_1 \cdot \frac{x^2}{3 \cdot 2} \qquad \sum_3 = \sum_1 + \sigma_3$$

$$\vdots$$

$$\sigma_{j+2} = -\sigma_j \frac{x^2}{(j+1)(j+2)} \qquad \sum_{j+2} = \sum_j + \sigma_{j+2}$$

$$\vdots$$

$$\lim_{j \to \infty} \sum_j = \sin x$$

For the example it is assumed that $0 \leq x < 1$, where x is in radians. It then follows that $\sin x < 1$.

From the induction process it is seen that the formation of the term σ_{j+2} involves a division by $(j+1)(j+2)$. Since j is an integer, the division cannot be done directly. To allow this division, j is scaled by 2^{-n}, determined by

$$2^{-n} j_{max} < 1$$

As the σ_{j+2} is desired as an unscaled quantity, the numerator is scaled by the same factor as is the denominator which gives the resultant quotient unscaled. In order to preserve significant figures, σ_{j+2} is formed as follows:

$$\sigma_{j+2} = -\sigma_j \frac{2^{-n}x}{2^{-n}(j+1)} \cdot \frac{2^{-n}x}{2^{-n}(j+2)}$$

The induction is terminated when the difference between two successive terms is less than a predetermined amount δ, where the size of δ is determined by the number of figures desired in the approximation to sin x. The difference between two successive approximations is the term σ_j. The discrimination is on the quantity

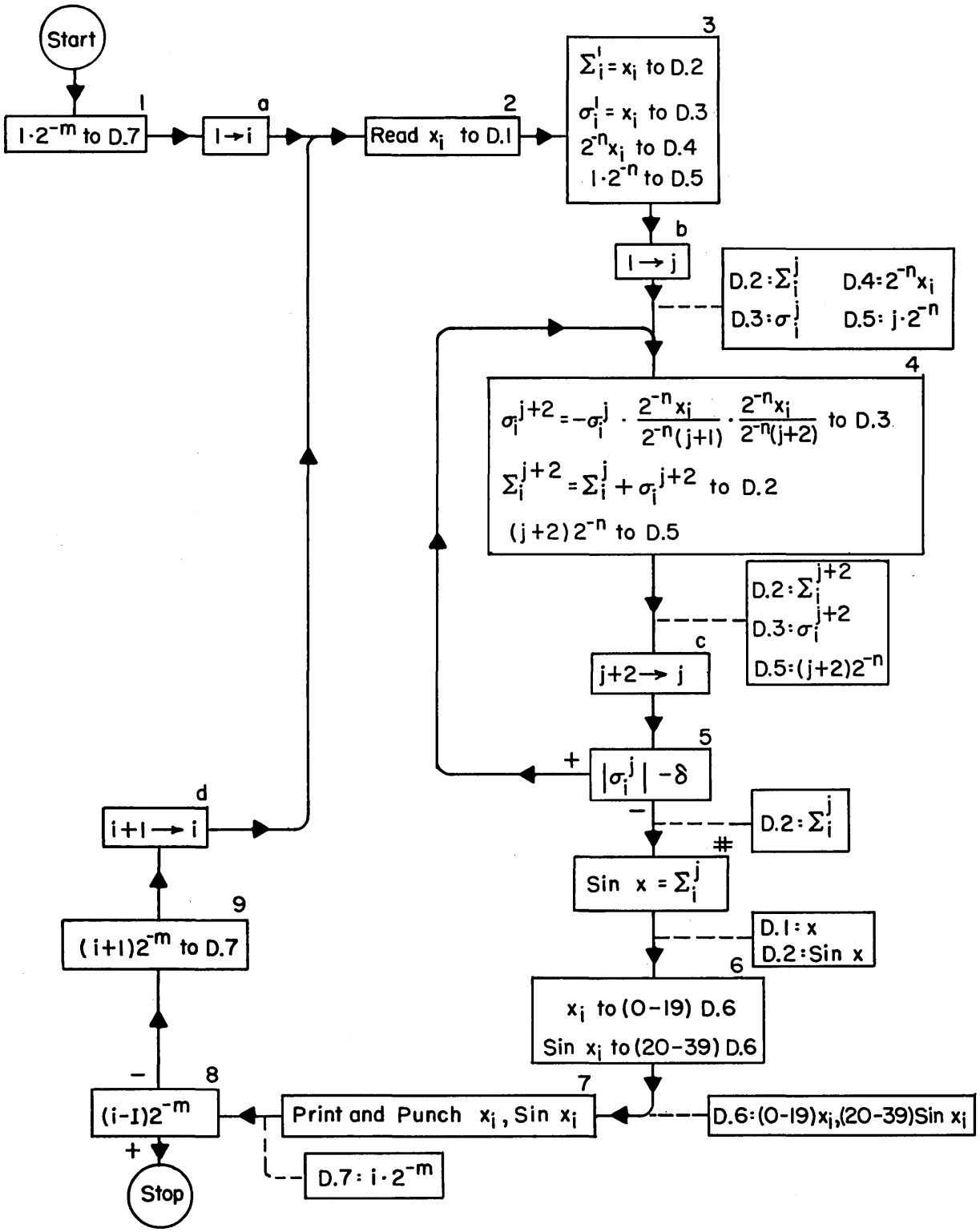
$$|\sigma_j| - \delta$$

The absolute value of σ_j is used, since σ_j may be positive or negative.

The storage needed is as follows: The constants $1 \cdot 2^{-n}$ and δ are stored at B.1 and B.2, respectively. The number I, representing the total number of values of the argument x, is stored at B.3 as $I \cdot 2^{-m}$, and $1 \cdot 2^{-m}$ is stored at B.4 where 2^{-m} is such that $I \cdot 2^{-m} < 1$. The values $x_1, x_2, x_3 \cdots x_I$ are punched onto paper tape as input data. Seven intermediate storage locations are needed. They are designated as D.1, D.2 ··· D.7.

No explanatory remarks are needed for the flow diagram which is shown in Figure 20, so we turn directly to the coding.

Box 1				
1.	m→Ac	B.4	$1 \cdot 2^{-m}$ to R2	
2.	A→m	D.7		$1 \cdot 2^{-m}$ to D.7
Box 2				
1.	Read	D.1		x_i to D.1
Box 3				
1.	m→Ac	D.1	x_i to R2	
2.	A→m	D.2		$\sum_i^1 = x_i$ to D.2
3.	A→m	D.3		$\sigma_i^1 = x_i$ to D.3
4.	R(n)	n	$2^{-n}x_i$ in R2	
5.	A→m	D.4		$2^{-n}x_i$ to D.4
6.	m→Ac	B.1	$1 \cdot 2^{-n}$ to R2	
7.	A→m	D.5		$1 \cdot 2^{-n}$ to D.5



SINE by TAYLOR'S SERIES EXPANSION

FIG. 20

Box 4

1.	m⟶Ac	D.5	$j \cdot 2^{-n}$ to R2	
2.	m⟶Ah	B.1	$(j+1)2^{-n}$ in R2	
3.	A⟶m	D.5		$(j+1)2^{-n}$ to D.5
4.	m⟶Ac-	D.4	$-2^{-n}x_i$ to R2	
5.	÷	D.5	$-\frac{x_i}{j+1}$ in R4	
6.	X	D.3	$-\sigma_i^j \frac{x_i}{j+1}$ in R2	
7.	A⟶m	D.3		$-\sigma_i^j \frac{x_i}{j+1}$ to D.3
8.	m⟶Ac	D.5	$(j+1)2^{-n}$ to R2	
9.	m⟶Ah	B.1	$(j+2)2^{-n}$ in R2	
A.	A⟶m	D.5		$(j+2)2^{-n}$ to D.5
B.	m⟶Ac	D.4	$2^{-n}x_i$ to R2	
C.	÷	D.5	$\frac{x_i}{j+2}$ in R4	
D.	X	D.3	$\sigma_i^{j+2} = -\sigma_i^j \frac{x}{j+1} \cdot \frac{x}{j+2}$ in R2	
E.	A⟶m	D.3		σ_i^{j+2} to D.3
F.	m⟶Ah	D.2	$\sum_i^{j+2} = \sum_i^j + \sigma_i^{j+2}$ in R2	
10.	A⟶m	D.2		$\sum_i^{j+2}$ to D.2

Box 5

1.	m⟶AcM	D.3	$\lvert \sigma_i^j \rvert$ to R2
2.	m⟶Ah-	B.2	$\lvert \sigma_i^j \rvert - \delta$ in R2
3.	C	Box 4,1	

Box 6

1.	m⟶Ac	D.1	x_i to R2	
2.	HS⟶m	D.6		x_i to (0-19)D.6
3.	m⟶Ac	D.2	$\sin x_i$ to R2	
4.	R(20)	20	$2^{-20} \sin x_i$ in R2	
5.	HS⟶m'	D.6		$\sin x_i$ to (20-39)D.6

Box 7

1.	Flexoprint	D.6	$(0\text{-}19)x_i (20\text{-}39)\sin x_i$ to Printer
2.	Punch	D.6	$(0\text{-}19)x_i (20\text{-}39)\sin x_i$ to Punch

Box 8

1.	m⟶Ac	D.7	$i \cdot 2^{-m}$ to R2	
2.	m⟶Ah-	B.3	$(i-I)2^{-m}$ in R2	
3.	C	Box A,1		

Box 9

1.	m⟶Ac	D.7	$i \cdot 2^{-m}$ to R2	
2.	m⟶Ah	B.4	$(i+1)2^{-m}$ in R2	
3.	A⟶m	D.7		$(i+1)2^{-m}$ to D.7
4.	T	Box 2,1		

Box A

1. Stop

The coding needed in Box 2 is merely the read instruction. The read instruction does the following:

> Read the next word to come under the reading head of the photo-electric reader and send the word to the memory at the address specified with the instruction.

In Box 3, Instruction 4 specifies only a right shift of n places. In an actual problem the scaling factor 2^{-n} would be determined and the numerical value of n would be inserted as the address of the R(n) instruction. Box 6 stores the x_i and sin x_i into one word D.6 by making use of the HS⟶m and HS⟶m' instructions. Instruction 2 of Box 6 stores the first 20 bigits of x_i into bigits 0-19 of D.6. This instruction does not alter bigits 20-39 of D.6. Instructions 4 and 5 store the first 20 bigits of sin x_i into bigits 20-39 of D.6. Since the HS⟶m' order replaces bigits 20-39 of m by bigits 20-39 of R2, the number in R2 must be positioned so that the 20 bigits to be sent to m are in bigits 20-39 of R2. Instruction 4 shifts sin x right 20 bigits so that the 20 most significant bigits of sin x are in (20-39)R2. Instruction 5 is then an HS⟶m' D.6 which stores sin x into (20-39)D.6. Box 7 requires two instructions, one to print D.6 and one to punch D.6

In this example the HS⟶m and HS⟶m' instructions were used to store half-precision (20 bigits) numbers, as compared to Problem 10 where they were used in modifying instructions.

The pairing of the code into words should present no difficulties. If the code sequence is started at address 000 the paired coding is:

0.	m⟶Ac	019	A⟶m	020
1.	Read	01A	m⟶Ac	01A
2.	A⟶m	01B	A⟶m	01C
3.	R(n)	(n)	A⟶m	01D
4.	m⟶Ac	016	A⟶m	01E
5.	m⟶Ac	01E	m⟶Ah	016
6.	A⟶m	01E	m⟶Ac-	01D
7.	÷	01E	X	01C
8.	A⟶m	01C	m⟶Ac	01E
9.	m⟶Ah	016	A⟶m	01E
A.	m⟶Ac	01D	÷	01E
B.	X	01C	A⟶m	01C
C.	m⟶Ah	01B	A⟶m	01B
D.	m⟶AcM	01C	m⟶Ah-	017
E.	C	005	m⟶Ac	01A
F.	HS⟶m	01F	m⟶Ac	01B
10.	R(20)	014	HS⟶m'	01F
11.	Flexoprint	01F	Punch	01F
12.	m⟶Ac	020	m⟶Ah-	018
13.	C'	015	m⟶Ac	020
14.	m⟶Ah	019	A⟶m	020
15.	T	001	Stop	
16.		$1\cdot2^{-n}$		
17.		§		
18.		$I\cdot2^{-m}$		
19.		$1\cdot2^{-m}$		
1A.				
1B.				
1C.				
1D.				
1E.				
1F.				
20.				

Problem 12

During the course of a lengthy computation it is desirable to make a periodic record of the contents of the memory. This record should be in a form that can be read back into the memory. Then, in the event of a computer malfunction which causes a computational error, one has only to read the last record of the memory contents back into the computer and resume the computation. If such a record is not available, the computation often has to be restarted from the beginning; and several hours, or even several days, of computational time may be lost. These periodic records of the memory contents help to keep the time lost due to computational errors at a minimum.

Such periodic records also increase the flexibility of the computer, for it becomes a simple task to interrupt a problem at any stage of the computation and start computation on a different problem. To interrupt a problem, one has only to record the memory contents and to know the instruction with which the control is to resume the computation. To resume, the record is read back into the memory and the control is sent to the desired starting instruction.

A magnetic tape unit has been adapted to the computer as an auxiliary input-output device for making these periodic records of the memory contents. A further discussion of the magnetic tape unit and its operational procedures is given in the chapter on operating procedures.

In this problem we outline two routines which are concerned with the magnetic tape unit. The first of these routines transfers the contents of the memory except for the routine itself to the magnetic tape. The second of the routines transfers the contents of the magnetic tape into the memory at the addresses specified by the routine.

Routine 1: Memory to magnetic tape.

This routine reads successively the words in the memory beginning with the first word beyond this routine and ending with the last word (1023) of the memory. As these words are read from the memory they are written onto the magnetic tape in a serial fashion beginning at a premarked section of the magnetic tape (details are discussed in the chapter on operating procedures).

A sum is formed of the contents of the memory (excluding this routine). The sum is:

$$S_1 = \sum_{i=c_o}^{1023} m_i$$

where c_o is the address of the first recorded word and m_i is the word at address $\underline{i}$ in the memory. This sum is recorded on the magnetic tape immediately following the word m_{1023}, and the sum is also printed. The sum is formed as a checking procedure for the magnetic tape unit. When the words on the tape are read back into the memory, the memory is summed and this sum must agree with the sum made at the time the contents of the memory were sent to the tape. If the two do not agree, an error has occurred and the record sent to the tape has not been transmitted correctly into the memory.

The inductive procedure should cause no difficulty, so we turn directly to the flow diagram in Figure 21. Box 1 sets up the initial values of the induction. Box 2 sends the word m_i to the magnetic tape. The partial summation

$$\sum_i = \sum_{i-1} + m_i$$

is also formed. Note in Box 2 the expression

$$[\text{delay } L(40)]$$

This has the following meaning: Each Q⟶t instruction is preccded by an L(40) shift instruction. During the traversal of this routine by the control, the magnetic tape is running continuously, and the L(40) instruction gives a certain spacing between words on the tape. This spacing is necessary to insure accurate transmission at some later occasion of the data from the tape back into the memory. Again this is discussed more thoroughly in the chapter on operating procedures.

Note in Alternative Box 3 how the induction is terminated. The discrimination is upon

$$(M\cdot i+1)2^{-10} \quad \text{where } M\cdot i(= c_o,\ c_o+1 \cdots 1023)$$

Now when

$$M\cdot i < 1023$$

$$M\cdot i+1 < 1024$$

and

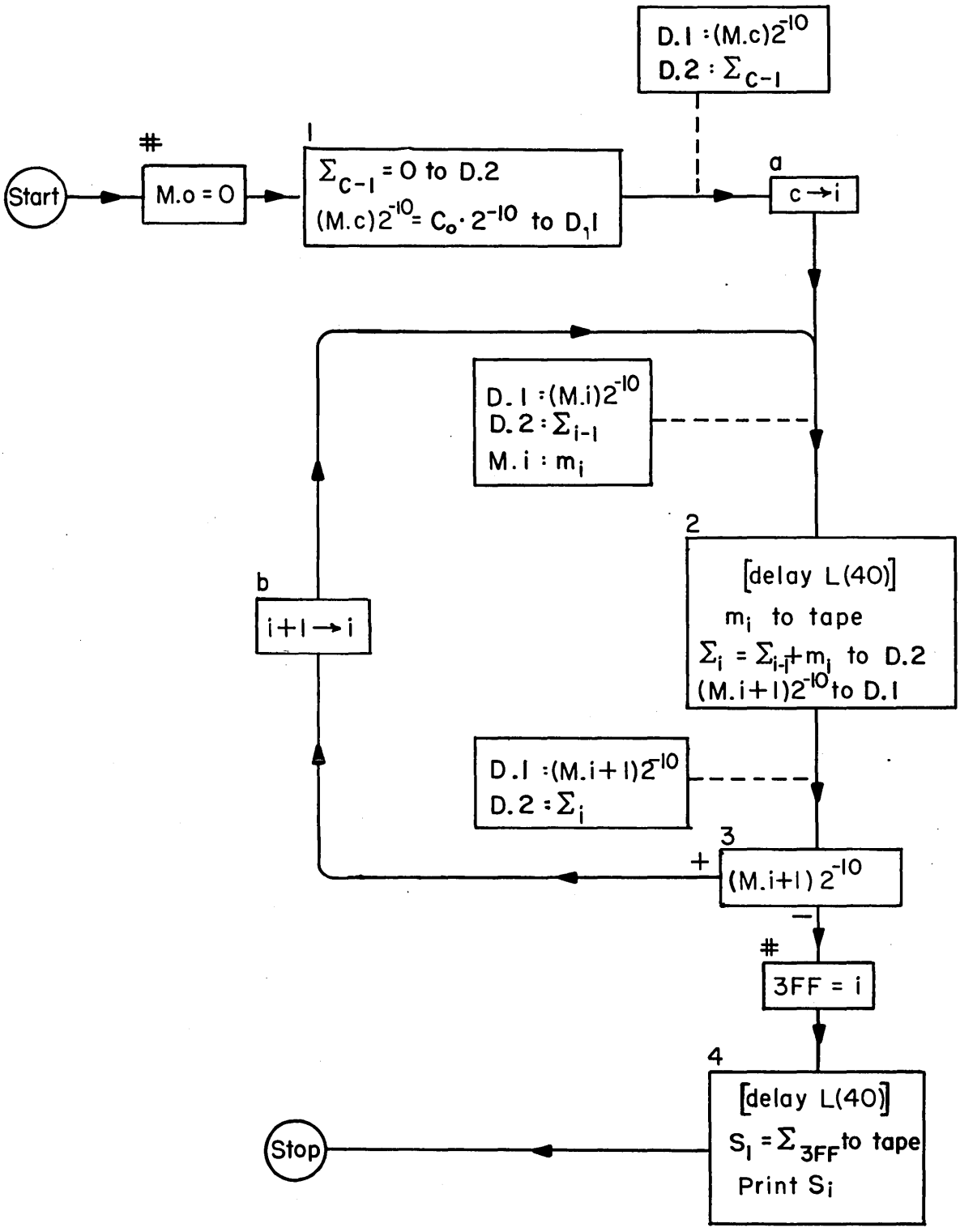
$$(M\cdot i+1)2^{-10} < 1$$

However, when

$$M\cdot i = 1023$$

which means that the last word in the memory has been sent to the magnetic tape

$$M\cdot i+1 = 1024$$



MEMORY TO MAGNETIC TAPE

FIG. 21

and

$$(M\cdot i+1)2^{-10} = 1$$

which appears in the computer as a negative number and the control proceeds to Box 4. This discrimination really allows the positive discriminating quantity to increase until it exceeds for the first time the allowed range for numbers in the computer. The effect to the computer is a change in the sign bigit of the number upon which the discrimination is made.

Box 4 sends the summation $\sum_{3FF}$ ($3FF \equiv 1023$) to the tape and also prints the sum.

The only storage needed in the problem is for two intermediate values of the computation. These values are the address M.i and the partial summation $\sum_i$. They are stored in D.1 and D.2, respectively.

The coding of the problem is:

Box 1

1.	a⟶Ac	0	$\sum_{c-1} = 0$ to R2	
2.	A⟶m	D.2		$\sum_{c-1} = 0$ to D.2
3.	a⟶Ac	$c_o \cdot 2^{-10}$	$(M.c)2^{-10} = c_o \cdot 2^{-10}$ to R2	
4.	A⟶m	D.1		$(M.c)2^{-10}$ to D.1

Box 2

1.	R(9)	9	$(M.i)2^{-19}$ in R2	
2.	S⟶m	2,5		M.i to (8-19)5
3.	L(40)			
4.	m⟶Ac	D.2	$\sum_{i-1}$ to R2	
5.	m⟶Q	[M.i]	m_i to R4	
6.	m⟶Ah	800	$\sum_i = \sum_{i-1} + m_i$ in R2	
7.	A⟶m	D.2		$\sum_i$ to D.2
8.	Q⟶t			m_i to tape
9.	m⟶Ac	D.1	$(M.i)2^{-10}$ to R2	
A.	a⟶Ah	$1\cdot 2^{-10}$	$(M\cdot i+1)2^{-10} = (M.i)2^{-10} + 1\cdot 2^{-10}$ in R2	

Box 3.

1.	C	1,4	

Box 4

1.	L(40)			
2.	m⟶Q	D.2	$S_1 = \sum_{1023}$ to R4	
3.	Q⟶t			S_1 to tape
4.	Flexoprint	D.2		S_1 to printer
5.	Stop			

In Box 1 the starting address $(c_o)2^{-10}$ is stored as the address portion of an a⟶Ac instruction. The instruction clears R2 and brings $c_o \cdot 2^{-10}$ into positions 0-11 of R2. An a⟶Ac instruction may often be utilized in this manner for storing and forming addresses.

Since the address as formed is

$$(M \cdot i)2^{-10}$$

it cannot directly be used in conjunction with an S⟶m instruction, as the bigits of an address to be substituted must appear in R2 as

$$(M.i)2^{-19} \text{ or } (M.i)2^{-39}$$

Instruction 1 of Box 2 shifts $(M.i)2^{-10}$ right by nine places so that the bigits in R2 are

$$(M.i)2^{-19}$$

Consequently, the instruction that receives this address must reside on the left-hand side of the instruction-pair.

Instruction 6 of Box 2 adds m_i to the quantity $\sum_{i-1}$ which is in R2 as the result of Instruction 4. Instruction 6 reads

m⟶Ah 800

Recall that any of the add orders (orders 1-8 of the vocabulary, Table I) treat R4 as a memory location with the address 2048 = 800 hexadecimally. m⟶Ah 800 adds the contents of R4 into R2. Now R4 contains m_i as the result of Instruction 5, so that

$$\sum_i = \sum_{i-1} + m_i$$

is formed in R2 as desired.

Instruction 8 of Box 2 is the Q⟶t instruction. The instruction is

"Q⟶t AD Write the number in R4 onto the magnetic tape."

The quantity m_i to be sent to the tape is in R4 as the result of Instruction 5 of Box 2. The address portion of the Q⟶t instruction has no relevance (the address is usually set to 000 for convenience; it may, however, be set to any value).

Instructions 9 and A of Box 2 form $(M.i+1)2^{-10}$, in R2. Rather than storing $(M.i+1)2^{-10}$ into D.1, it is left in R2 for the discrimination of Box 3, Instruction 1. The conditional transfer of Box 3, if effective, sends the control to Box 1, Instruction 4, where the contents

of R2, $(M.i+1)2^{-10}$ are sent to storage. We saw previously that upon entry into Box 2 from Box 1, the quantity $(M.i)2^{-10}$ was in R2. Box 2 is also entered from the plus branch of Alternative Box 3, and from this entry the quantity $(M.i)2^{-10}$ is correctly in R2.

Box 4, Instructions 1, 2, and 3 send $\sum_{3FF}$ onto the magnetic tape. Again, as in Box 2, an instruction L(40) precedes the Q⟶t instruction.

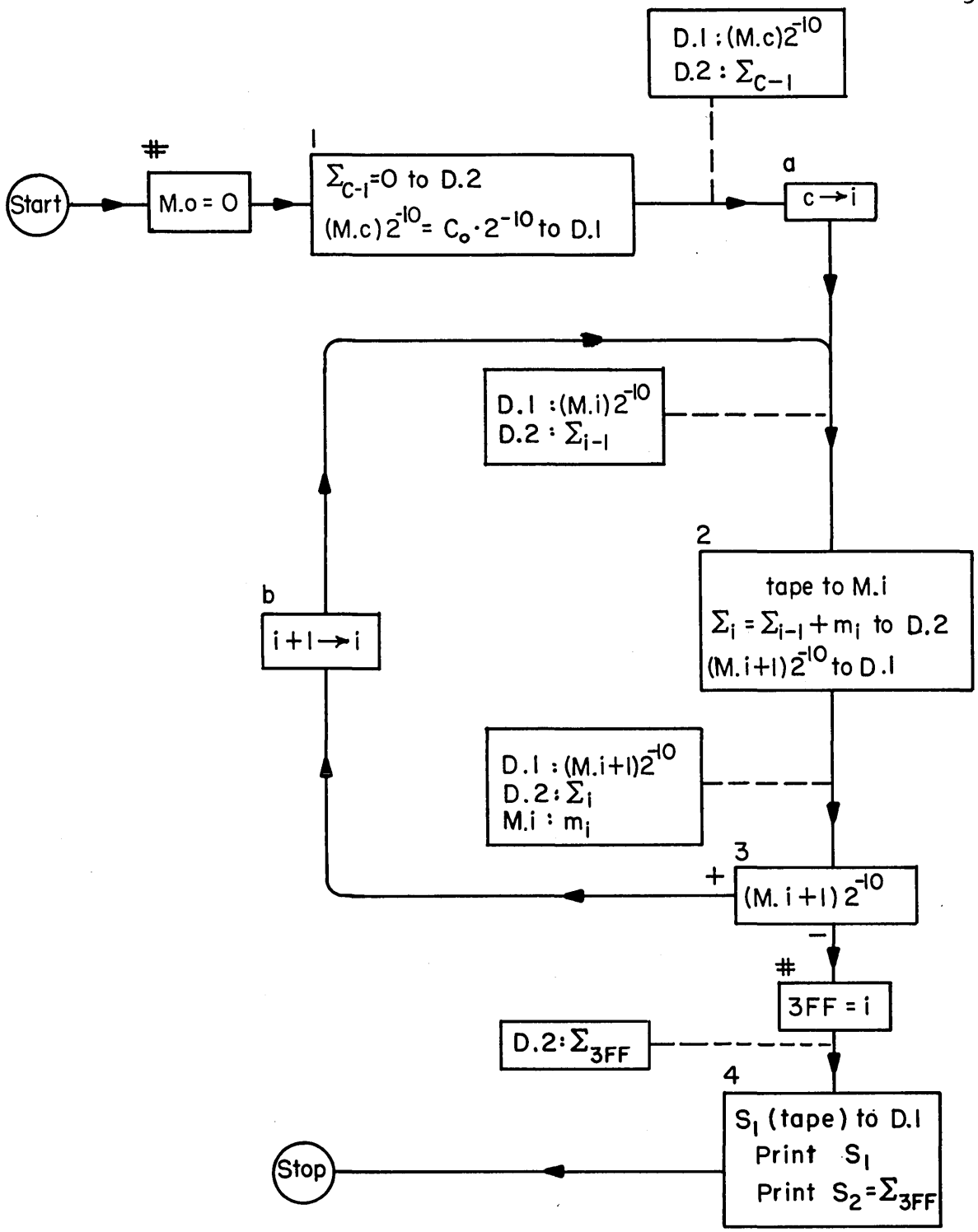
The routine as outlined is to be coded beginning with Word 000. The paired coding occupies Words 000 through 009 and the storage needed is designated as 00A and 00B. The initial address c_o is then 00C. The paired coding is:

0.	a⟶Ac	000	A⟶m	00B
1.	a⟶Ac	018	A⟶m	00A
2.	R(9)	009	S⟶m	004
3.	L(40)	028	m⟶Ac	00B
4.	m⟶Q	[000]	m⟶Ah	800
5.	A⟶m	00B	Q⟶t	000
6.	m⟶Ac	00A	a⟶Ah	002
7.	C'	001	L(40)	028
8.	m⟶Q	00B	Q⟶t	000
9.	Flexoprint	00B	Stop	
A.				
B.				

The left-hand instruction of Word 001 sets up the initial address c_o. It is to be $(00C)2^{-10}$ which is $(018)2^{-11}$; hence, the address of the instruction is 018.

Routine 2: Magnetic tape to memory.

This routine is to be used in conjunction with Routine 1. It reads successively the words from the magnetic tape (which had been written onto the tape by utilizing Routine 1) and writes them into the memory at the addresses that they had originally occupied. Routine 1 sent Words 00C through 3FF onto the tape; therefore, this routine reads the words from the tape and writes them into the memory at the addresses 00C through 3FF.



MAGNETIC TAPE TO MEMORY

FIG. 22

After the words m_i where i (= 12, 13 ··· 1023) are sent to the memory, a sum

$$S_2 = \sum_{i=c_o}^{1023} m_i \qquad c_o = 12 \text{ (dec.)}$$

is formed and printed. Also printed is the word immediately following m_{1023} on the magnetic tape. The latter is S_1, the sum of the memory contents (hence the sum of the words on the tape) when the tape record was made. The sums S_1 and S_2 are identical, if no errors have been made by the computer or the magnetic tape. The procedures to be followed if S_1 and S_2 do not agree are outlined in the chapter on operating procedures.

The flow diagram shown in Figure 22 is so similar to the flow diagram of Routine 1 that we turn directly to the coding without further comment.

The coding is:

Box 1				
1.	a⟶Ac		$\sum_{c-1} = 0$	to R2
2.	A⟶m	D.2		$\sum_{c-1}$ to D.2
3.	a⟶Ac	$c_o \cdot 2^{-10}$	$(M.c)2^{-10} = c_o \cdot 2^{-10}$	to R2
4.	A⟶m	D.1		$(M.c)2^{-10}$ to D.1
Box 2				
1.	R(9)		$(M.i)2^{-19}$	in R2
2.	S⟶m	2,5		M.i to (8-19)5
3.	S⟶m	2,7		M.i to (8-19)7
4.	t⟶Q		m_i	to R4
5.	Q⟶m	[M.i]		m_i to M.i
6.	m⟶Ac	D.2	$\sum_{i-1}$	to R2
7.	m⟶Ah	[M.i]	$\sum_i = \sum_{i-1} + m_i$	in R2
8.	A⟶m	D.2		$\sum_i$ to D.2
9.	m⟶Ac	D.1	$(M.i)2^{-10}$	
A.	a⟶Ah	$1 \cdot 2^{-10}$	$(M \cdot i+1)2^{-10} = (M \cdot i)2^{-10} + 1 \cdot 2^{-10}$	in R2
Box 3				
1.	C	1,4		
Box 4				
1.	t⟶Q		S_1	to R4
2.	Q⟶m			S_1 to D.1
3.	Flexoprint	D.1		S_1 to Printer
4.	Flexoprint	D.2		S_2 to Printer
5.	Stop			

In the formation of each successive term of the partial summation $\sum_i$, in Box 2, Instructions 6, 7, and 8, the contribution m_i is added from its memory location M.i rather than from R4 where it also exists. The checking obtained by this summing process is more complete than if m_i were added from R4.

The t⟶Q instruction which is Instruction 4 of Box 2 and Instruction 1 of Box 4 is:

"t⟶Q AC Replace the number in R4 by the first word to come under the reading head of the magnetic tape reader."

Again, as in the Q⟶t instruction, the address of the instruction has no relevance. Note that the L(40) instruction which preceded each Q⟶t instruction is not used with the t⟶Q instructions.

In the paired coding, Instructions 5 and 7 of Box 2 must be left-hand instructions since the address M.i which is being substituted is in R2 as

$$(M.i)2^{-19}$$

In Box 4, Instructions 3 and 4 print the summations S_1 and S_2. A visual check is then made of the numbers rather than allowing the computer to do the comparison. This has the added feature that these two numbers printed may also be checked against the number S_1 which was printed when the tape record was made.

This routine is coded into Address 000 and occupies Words 000 through 009. D.1 and D.2 are designated as 00A and 00B, respectively. Again, c_o is 00C. The paired coding is:

0.	a⟶Ac	000	A⟶m	00B
1.	a⟶Ac	018	A⟶m	00A
2.	R(9)	009	S⟶m	004
3.	S⟶m	005	t⟶Q	000
4.	Q⟶m	[000]	m⟶Ac	00B
5.	m⟶Ah	[000]	A⟶m	00B
6.	m⟶Ac	00A	a⟶Ah	002
7.	C'	001	t⟶Q	000
8.	Q⟶m	00A	Flexoprint	00A
9.	Flexoprint	00B	Stop	
A.				
B.				

We have in this problem taken the liberty of incorporating checking features into the two related routines without either discussing the need for such checking features or discussing what the procedures are if this checking indicates an error in the transmission. This checking is such an integral part of the routines which make use of the magnetic tape unit that we do not feel that the routines should be presented without including them.

Problem 13

We develop a routine for the synchroprinter, the high-speed page printer that has been adapted to the computer as a part of the output equipment. The synchroprinter has a maximum operating speed of 36,000 characters per minute. The ordinal numbers 0, 1, 2 ··· 9; the letters A, B ··· F; a decimal point; and a minus sign are the eighteen distinct characters that may be printed. A line at a time is printed, where a line consists of 40 characters. Recall that the synchroprint order reads:

"Sync Print CE To be used in a subroutine which simultaneously prints $\underline{m}_i$, $\underline{m}_{i+1}$, $\underline{m}_{i+2}$ and $\underline{m}_{i+3}$; $\underline{i}$ is to be communicated to the routine (high speed)."

Inasmuch as four words are printed simultaneously, it is not surprising that a special routine is required. Further discussion of the synchroprinter is given in the chapters IV and VI on The Computer and Operating Procedures, respectively.

In order to achieve the high speed of operation, the printer operates as follows:

To print an aggregate of forty digits (a line) there are eighteen distinct print cycles. All the F's of the aggregate are printed simultaneously in Cycle 1, all the E's of the aggregate are printed simultaneously in Cycle 2, and so on to Cycle 16 which prints all the 0's, to Cycle 17 for the decimal points, and to Cycle 18 for the minus signs. Since there are these eighteen distinct cycles, one has only to supply the digital information which corresponds to the cycle. That is, during Cycle 1, only the digital information for the F's is needed, and so on. This information is obviously binary. For Cycle 1 it is either to print an F in a particular digit position, or not to print it. The line of print is 40 digits and a register contains 40 bigits, so a register may supply the binary data (either print or do not print) to the printer for each cycle. The register R2 is used for this purpose. During the i^{th} print cycle i (= 1,2 ··· 18) an appropriate number which specifies the digit positions to be printed is brought into R2. A 0 in any position of the number in R2 corresponds to the presence of the character of the i^{th}

cycle in the respective digit position of the line, whereas a 1 indicates the absence of the corresponding character.

For simplicity of design, the paper feed is vertically down. Hence, to achieve a conventional listing, the characters must be inverted and left,right interchanged, so that the leftmost bigit of R2 corresponds to the rightmost bigit of the print line while 2^{-39} of R2 corresponds to the leftmost bigit of the print line.

The procedure to print a line corresponding to four 10-digit (10-tetrad) words is as follows:

The four words are fanned out into an 18 x 40 array which occupies 18 successive memory locations. The rows of the array (the eighteen locations) correspond to the characters of the printer. The columns of the array correspond to the digit position within the line of print. The first row of the array corresponds to the minus sign, the second to the decimal point, the third to the 0, the fourth to the 1, and so on, through the 18th row which corresponds to the F. Column 0 corresponds to digit position 39 of the line, column 1 to digit position 38, and so on, through column 39, which corresponds to digit position 0 of the line.

We define an element of the array as a_{ij}, where i corresponds to the row of the array and j corresponds to the column. If

$$a_{ij} = 0$$

the i^{th} character is to be printed in column j (digit position 39 - j). If

$$a_{ij} = 1$$

the character is not to be printed. No column of the array may contain more than one 0; that is, only one character may be printed in any digit position. However, if a column contains 1's only, then no character is printed in the corresponding digit position.

The elements of the array are initially set to 1. The first tetrad of the first word is examined and found to have the value i, then a 0 is inserted into the appropriate element $a_{i,39}$. The second tetrad is examined and a 0 is inserted into the corresponding element $a_{i,38}$; and so on, until the forty tetrads of the four words have been examined and 0 has been inserted into the appropriate elements of the array.

The inductive process of fanning the four words into the array is described as follows: The elements of the 18 x 40 array are initially set to 1. The insertion of zeros into elements in the two rows of the array corresponding to the minus sign and the decimal point is treated apart from the induction. Hence, we may regard the rows as being specified by the values of the tetrads with

$$0 \leq i \leq F.$$

The tetrads of the words must be isolated to obtain the values i. They are isolated as follows: The four words are specified as

$$m_k \qquad k\ (=0,1,2,3).$$

In each word there are ten tetrads

$$i_{k,n} \qquad n\ (=0,1,2\cdots 9).$$

The induction for isolating the tetrads of any word m_k is over the index n and it is

$$c_{k,-1} = m_k$$
$$c_{k,n} = 2^4 c_{k,n-1} \text{ (fractional part)}$$
$$i_{k,n} = 2^4 c_{k,n-1} \text{ (integer part)}$$

where $0 \leq n \leq 9.$

After the row i is determined, the column j must be determined so that the element a_{ij} may be set to 0. The column j is easily seen to be given by

$$j = 39 - (10k + n)$$

We specify the i^{th} row of the array as r_i. Then after determining the appropriate i and j values we have only to perform the operation

$$r_i - 2^{-j}$$

to set the element a_{ij} to 0.

The printing sequence proper, which is carried out after the array is formed, may now be given. Within the sequence, each of the eighteen print cycles is determined by a print order. The first print order actuates the printer and the remaining seventeen print orders act in a timing capacity to keep the printer and computer in synchronization. Once the printer has been actuated it proceeds through its eighteen

cycles at a fixed rate independently of the computer. Each of the seventeen print orders must be given before the printer is ready to perform that particular cycle. The order has the effect that the computer waits for the printer until the cycle is complete and then proceeds to the next instruction of the sequence. The printer operates at a speed of roughly 1.5 milliseconds between its print cycles. The print sequence must have no more than 1.5 milliseconds elapse between successive print orders.

Immediately preceding each print order, the appropriate word of the array is brought into R2. Cycle 1 prints the F's so that Word 18 of the array is the first word to be brought into R2. It is followed by a print order which actuates the printer and executes Cycle 1. Word 17 of the array is brought in and the succeeding print order executes Cycle 2 and prints the E's. This continues until the eighteen print cycles have been completed.

Even though the eighteen distinct characters may not all appear in any given printed line, it is necessary that eighteen print orders corresponding to the eighteen characters be given. Those characters that do not appear have their respective row in the array containing all 1's so that nothing is printed during the corresponding print cycle.

We now turn to the flow diagram shown in Figure 23. The storage needed is as follows: The four words m_o, m_1, m_2, and m_3 which are to be printed are stored in D.1, D.2, D.3, and D.4, respectively. The eighteen words needed for the array are designated (the addresses are hexadecimal):

E.1: r_-
E.2: $r.$
E.3: r_o
E.4: r_1
.
.
.
E.12: r_F

The following constants are needed

B.1: -2^{-39}
B.2: $2^{-7} + 2^{-19} + 2^{-39}$
B.3: 0

Three initial addresses are stored. They are

B.4: $(D.1)_o$

B.5: $(E.3)_o$

B.6: $(E.12)_o$

$(D.1)_o$ is the base address for the four words to be printed. $(E.3)_o$ is the base address to which $\underline{i}$ is added to form the address of r_i. $(E.12)_o$ is the base address used in the printing sequence. Four words of intermediate storage are needed. They are designated as D.5, D.6, D.7, and D.8.

Boxes 1, 2, and 3 of the flow diagram set the eighteen rows of the array to all $\underline{1}$'s. Boxes 4 through A form a double induction that records 0's into the appropriate elements a_{ij} of the array. Boxes B, C, D, and E are the print sequence proper.

Box 1 sets the initial index of $I \cdot 2^{-7}$ for storing $\underline{1}$'s into the rows r_i. Box 2 stores -2^{-39} into the rows

$$r_{I-i} \qquad \text{where } i\ (=0,1,2 \cdots 17, \text{ decimally})$$

The discrimination of Box 3 is on

$$(I - i - 2)2^{-7}$$

Immediately after

$$r_o = -2^{-39}$$

is stored, (i = 17, dec.) $\underline{i}$ is increased by $\underline{1}$; hence the quantity

$$I - 1 - (i+1)2^{-7}$$

is correctly negative for the first time as

$$I = 18 \cdot 2^{-7}, \text{ dec.}$$

Box 4 sets up the initial conditions for the induction over $\underline{k}$. It sends the initial address $(D.1)_o$ to D.7 where it becomes $(D.1+k)_o$ as k = 0 initially. It also sends the number

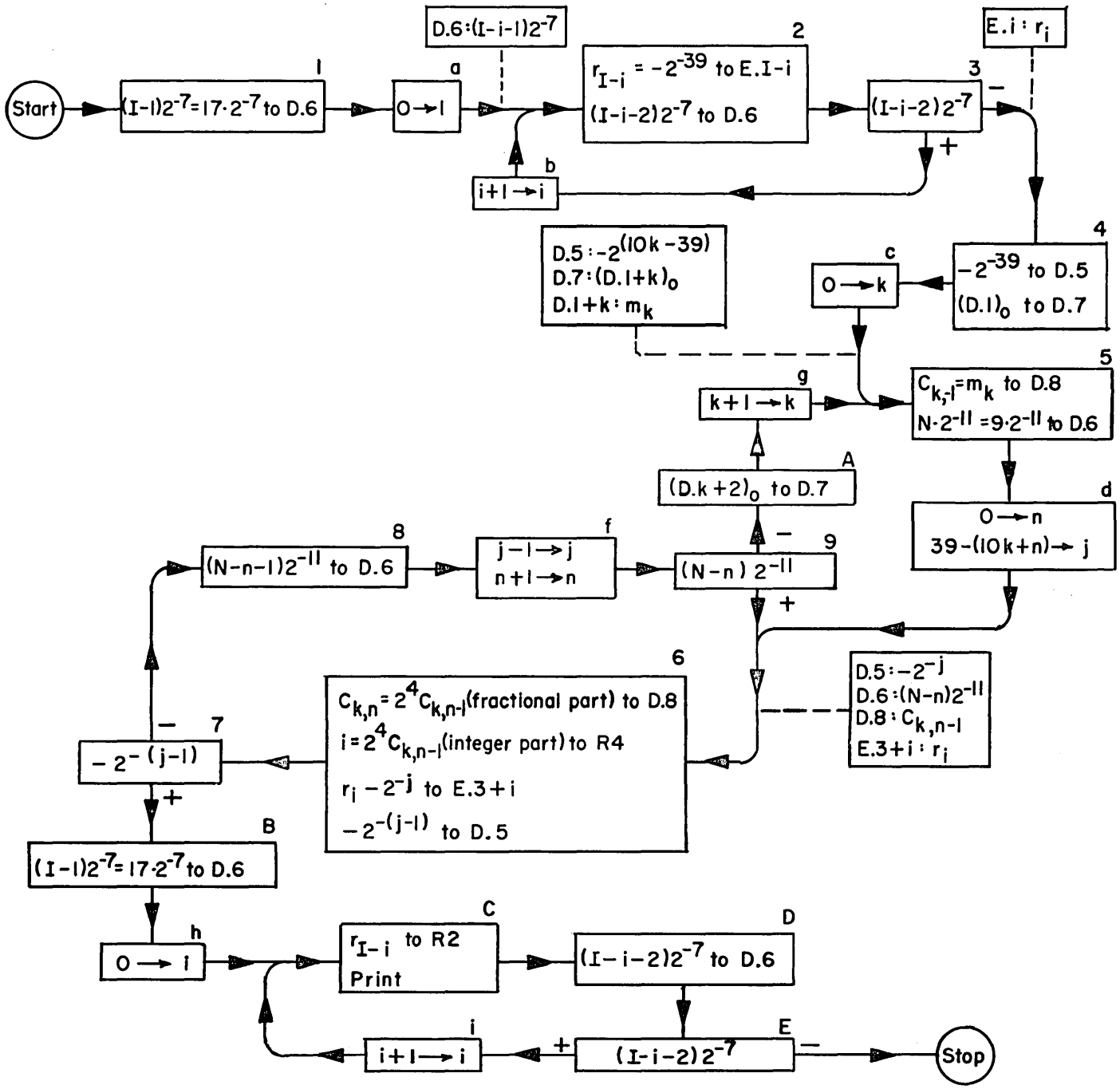
$$-2^{-39} \text{ to D.5}$$

where it is to become

$$-2^{-j} \qquad j = 39 - (10k + n)$$

$\underline{k}$ and $\underline{n}$ are both initially $\underline{0}$; hence $\underline{j}$ is initially 39, as is desired.

Box 5 sets up the induction over $\underline{n}$. The word m_k becomes $c_{k,-1}$, and $N \cdot 2^{-11}$ is set to $9 \cdot 2^{-11}$. Box 6 forms $c_{k,n}$ and $\underline{i}$ by shifting $c_{k,n-1}$



A SYNCHROPRINTER ROUTINE

FIG. 23

left four places. $\underline{i}$ is in R4 as $2^{-39}i$ and $c_{k,n}$ is in R2. The appropriate element a_{ij} is set to $\underline{0}$ by the operation

$$r_i - 2^{-j} \text{ to E.3+1}$$

Note that

$$E\cdot 3 + i: \quad r_i \qquad \text{where } i\ (=0,1\cdots F)$$

E.1 and E.2 contain r_- and $r.$ of the array, and they do not enter into this print routine, but they must exist as all $\underline{1}$'s. Alternative Box 7 terminates the double induction and sends the control to the print sequence. The discrimination is on

$$-2^{-j}$$

This quantity appears negative to the computer until $j = -1$, at which time -2^{-j} appears as $\underline{0}$ in the computer. It is then a positive number with respect to discrimination and the control is sent to the start of Box B. Note in Box 6 that $\underline{j}$ is decreased to $j - 1$ after the operation

$$r_i - 2^{-j} \text{ to E.3+}\underline{i}$$

When $j = 0$ the last step of the induction is completed and a $\underline{0}$ is stored in the leftmost bigit of the row r_i. $\underline{j}$ is then decreased to $j^- - 1 = -1$ and the quantity

$$-2^{-j}$$

becomes positive for the first time.

Boxes B, C, D, and E bring out the rows of the array and print them, starting with r_I, which corresponds to the character F, and decreasing to r_o, which corresponds to character $\underline{0}$. Print orders are given corresponding to the rows $r.$ and r_-, even though these characters are not printed by the routine. After the print order for r_- has been given, the discrimination of Box E is negative for the first time and the routine terminates.

The coding of the routine is:

Box 1

1. a⟶Ac $11\cdot 2^{-7}$ $11\cdot 2^{-7}(=17\cdot 2^{-7}$ dec.) to R2

Box 2

1. m⟶Ah B.6 $(I-1)2^{-7} + (E.I)_o = 11\cdot 2^{-7} + (E.12)_o$ in R2
2. S⟶m 2,4 E.I-i to (8-19)2,4
3. m⟶Q B.1 -2^{-39} to R4

Box 2 (Cont.)

4.	Q⟶m	[E.I-i]	$r_i = -2^{-39}$ to E. I -i
5.	m⟶Ah-	B.2	$(I-i-2)2^{-7} + (E.\ I-i-1)_o$ in R2

Box 3

1.	C	2,2	

Box 4

1.	Q⟶m	D.5	-2^{-39} to D.5
2.	m⟶Ac	B.4	$(D.1)_o$ to R2
3.	A⟶m	D.7	$(D.1)_o$ to D.7

Box 5

1.	m⟶Ac	D.7	$(D.k+1)_o$ to R2
2.	S⟶m	5,5	D.k+1 to (8-19)5,5
3.	a⟶Ac	$9\cdot2^{-11}$	$N\cdot2^{-11} = 9\cdot2^{-11}$ to R2
4.	A⟶m	D.6	$N\cdot2^{-11}$ to D.6
5.	m⟶Ac	[D.k+1]	$c_{k,-1} = m_k$ to R2
6.	A⟶m	D.8	$c_{k,-1}$ to D.8

Box 6

1.	m⟶Ac	D.8	$c_{k,n-1}$ to R2
2.	m⟶Q	B.3	$\underline{0}$ to R4
3.	L(4)	4	$i\cdot2^{-39}$ in R4; $c_{k,n}$ in R2
4.	A⟶m	D.8	$c_{k,n}$ to D.8
5.	m⟶Ac	B.5	$(E.3)_o$ to R2
6.	m⟶Ah	800	$E.3\cdot2^{-19} + (E.3+i)2^{-39}$ in R2
7.	S⟶m	6,9	E.3+i to (8-19)6,9
8.	S⟶m	6,B	E.3+i to (8-19)6,B
9.	m⟶Ac	[E.3+i]	r_i to R2
A.	m⟶Ah	D.5	$r_i - 2^{-j}$ in R2
B.	A⟶m	[E.3+i]	$r_i - 2^{-j}$ to E.3+i
C.	m⟶Ac	D.5	-2^{-j} to R2
D.	L(1)	1	$-2^{-(j-1)}$ in R2
E.	A⟶m	D.5	$-2^{-(j-1)}$ to D.5

Box 7

1.	C	B,1	

Box 8

1.	m⟶Ac	D.6	$(N-n)2^{-11}$ to R2	
2.	a⟶Ah	-2^{-11}	$(N-n-1)2^{-11}$ in R2	
3.	A⟶m	D.6		$(N-n-1)2^{-11}$ to D.6

Box 9

1.	C	6,1		

Box A

1.	m⟶Ac	D.7	$(D.k+1)_o$ to R2	
2.	m⟶Ah	B.2	$(D.k+2)_o$ in R2	
3.	A⟶m	D.7		$(D.k+2)_o$ to D.7
4.	T	5,1		

Box B

1.	a⟶Ac	11.2^{-7}	$11\cdot2^{-7}$ (= $17\cdot2^{-7}$ dec.) to R2	

Box C

1.	m⟶Ah	B.6	$(I-1)2^{-7} + (E.I)_o = 11\cdot2^{-7} + (E.12)_o$ in R2	
2.	S⟶m	C,4		E.I-i to (8-19)C,4
3.	A⟶m	D.6		$(I-i-1)2^{-7} + (E.I-i)_o$ to D.6
4.	m⟶Ac	[E.I-i]	r_{I-i} to R2	
5.	Syncprint			

Box D

1.	m⟶Ac	D.6	$(I-i-1)2^{-7} + (E.I-1)_o$ to R2	
2.	m⟶Ah-	B.2	$(I-i-2)2^{-7} + (E.I-i-1)_o$ in R2	

Box E

1.	C	C,2		
2.	Stop			

In the induction storing -2^{-39} to all r_i, the register R2 is needed only in forming $(I-i-1)2^{-7}$ and in forming the addresses $(E.I-i)_o$. These two operations may be performed simultaneously and the quantities $(I-i-1)2^{-7}$ and $(E.I-i)_o$ are left in R2 throughout the induction. Therefore the quantity $11\cdot2^{-7}$ ($=17\cdot2^{-7}$ dec.) need only be sent to R2 in Box 1, and it is not stored into D.6. During the traversal of Box 2, R2 contains

$$(I-i-1)2^{-7} + (E.I-i)_o$$

Instruction 5 subtracts the contents of B.2 from R2. B.2 contains the constant

$$2^{-7} + 2^{-19} + 2^{-39}$$

so that the subtraction gives

$$(I\text{-}i\text{-}2)2^{-7} + (E.I\text{-}i\text{-}1)_o$$

in R2 as is desired. The quantity -2^{-39} that is sent to all addresses E.I-1 is stored from R4. The only instruction needed in Box 3 is the conditional transfer as the quantity $(17\text{-}i\text{-}2)2^{-7}$ upon which the transfer acts is in R2 from Box 2.

In Box 4, Instruction 1 stores -2^{-39} to D.5 where it becomes -2^{-j} The quantity -2^{-39} exists in R4 as a result of Box 2.

Instruction 2 of Box 6 sends $\underline{0}$ to R4 and Instruction 3, an L(4), isolates $\underline{i}$ in R4 as

$$2^{-39}i$$

The quantity $i \cdot 2^{-39}$ is added from R4 into the $(E.3)_o$ in R2 by making use of the m⟶Ah 800 instruction where the address 800 refers to R4. Instructions 7 and 8 must both be S⟶m' instructions in the final code since the pertinent address in R2 is

$$(E.3+i)2^{-39}$$

In Box C where the print order is given the scheme used in Box 2 of having the index and the address in one word as

$$(I\text{-}i\text{-}1)2^{-7} + (E.I\text{-}i)_o$$

is utilized. In this instance, however, the word cannot be left in R2 during the induction as the rows r_i to be printed must be brought into R2. Instruction 5 is the print. Since $\underline{i}$ is initially $\underline{0}$ the rows of the array r_{I-i} are correctly brought into R2 beginning with r_F.

Box D subtracts

$$2^{-7} + 2^{-19} + 2^{-39}$$

from

$$(I\text{-}i\text{-}1)2^{-7} + (E.I\text{-}i)_o$$

and leaves the result in R2. Box E then needs only the conditional transfer order. As long as $(I\text{-}i\text{-}2)2^{-7}$ is a positive number the transfer sends the control to Instruction 2 of Box C.

The coding contains 49 instructions, which is 24 1/2 words. We start the code at Word 000. Upon examination it is revealed that Instructions 9 and B of Box 6 naturally occur as left-hand instructions in the final code. It is necessary for them to be on the right; therefore a dummy instruction must be inserted for positioning. This gives 25 words of code which occupy addresses 000 through 018 in the memory. The 18 words of the array occupy addresses 019 through 02A. The 6 words of B storage are then in addresses 02B through 030, and the 8 words of D storage are in addresses 031 through 038.

The coding is:

000	a⟶Ac	110	m⟶Ah	030
001	S⟶m	002	m⟶Q	02B
002	Q⟶m	000	m⟶Ah-	02C
003	C	001	Q⟶m	035
004	m⟶Ac	02E	A⟶m	037
005	m⟶Ac	037	S⟶m	007
006	a⟶Ac	009	A⟶m	036
007	m⟶Ac	000	A⟶m	038
008	m⟶Ac	038	m⟶Q	02D
009	L(4)	004	A⟶m	038
00A	m⟶Ac	02F	m⟶Ah	800
00B	S⟶m'	00C	S⟶m'	00D
00C	(DS	000)	m⟶Ac	000
00D	m⟶Ah	035	A⟶m	000
00E	m⟶Ac	035	L(1)	001
00F	A⟶m	035	C	014
010	m⟶Ac	036	a⟶Ah	FFF
011	A⟶m	036	C	008
012	m⟶Ac	037	m⟶Ah	02C
013	A⟶m	037	T	005
014	a⟶Ac	110	m⟶Ah	030
015	S⟶m	016	A⟶m	036
016	m⟶Ac	000	Syncprint	000
017	m⟶Ac	036	m⟶Ah-	02C
018	C	015	Stop	

019 r_{-}
01A $r_{.}$
01B r_{o}
01C r_{1}
01D r_{2}
01E r_{3}
01F r_{4}
020 r_{5}
021 r_{6}
022 r_{7}
023 r_{8}
024 r_{9}
025 r_{A}
026 r_{B}
027 r_{C}
028 r_{D}
029 r_{E}
02A r_{F}
02B B.1: -2^{-39}
02C B.2: $2^{-7} + 2^{-19} + 2^{-39}$
02D B.3: 0
02E B.4: $(D.1)_{o} = (00031)_{o}$
02F B.5: $(E.3)_{o} = (0001B)_{o}$
030 B.6: $(E.12)_{o} = (0002A)_{o}$
031 D.1
032 D.2
033 D.3
034 D.4
035 D.5
036 D.6
037 D.7
038 D.8

III. BINARY ARITHMETIC

We begin the study of arithmetic as it relates to the computer by discussing (i) the allowed ranges of numbers and (ii) the treatment of negative numbers.

The allowed number range may be approached in two ways. There is the so-called "floating binary point" method and the "fixed binary point" method. We have adopted the latter approach; however, a few cursory remarks may be made about the former.

The "floating binary point" allows each number to be expressed as a fraction and a characteristic. That is, the binary number 1011.1101 would be expressed as (0.10111101, +100) where the 100 is the positive exponent of 2 associated with the number. An argument in favor of such a scheme is that it alleviates the scaling considerations at the coding stage which one otherwise encounters in working with a fixed binary point. It is felt, however, that scaling is not a serious problem and that the time spent in arranging suitable scale factors is small in comparison to the total time spent in preparing an interesting problem for the computer. Two definite arguments against the floating binary point are: (i) It increases the complexity of the computer which in turn increases maintenance difficulties. (ii) It increases the time necessary to perform each operation. In many problems that are contemplated the time required for their solution is a principal factor; hence advantages of speed are important.

In the "fixed binary point" method the binary point in the present computer is taken between the first and second bigits from the left. The binary point might have been fixed between any other bigit pair. This fixed binary point places an upper limit on the size of a number in the computer.

Since it is necessary to be able to distinguish between positive and negative numbers, and since their treatment has a direct bearing on the allowed range of numbers, we digress temporarily and discuss the "sign" of a number.

Although there are many possibilities for the representation of numbers in the computer, we consider the two most common ones: (i) "signed" numbers and (ii) "complement" numbers. In the first

scheme the leftmost bigit would indicate the "sign". The sign bigit would be a 0 or 1 according as the number is non-negative or negative. In each instance the sign bigit is followed by the actual numerical bigits. Clearly, in this case the magnitude of all numbers would be less than 1.

In the second scheme of "complement" numbers, since

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1,$$

we write x as

$$x \equiv \sum_{i=0}^{N} 2^i + 1 + x - 2^{N+1};$$

then take for our representation of x

$$x(\text{mod } 2^{N+1}) \equiv \sum_{i=0}^{N} 2^i + 1 + x.$$

For positive x, that is $x \geq 0$, the above equation gives:

$$x(\text{mod } 2^{N+1}) \equiv x.$$

If also, $|x|<1$, the leftmost bigit contains a 0 as in the preceding scheme. For the negative values of x, $0 > x > -1$, the integral part of the number's representation is a sequence of 1's, $(N+1)$ in length, followed by a fractional part equal to $(1-|x|)$. Since the computer contains numbers modulo 2, it contains the complete fractional part and the first integer to the left of the binary point; hence the leftmost bigit contains a 1. Therefore, in the complement scheme, if $|x| < 1$, the "sign" of a number may be identified by examining the leftmost bigit. This is not a true "sign" and the bigit has numerical significance. However, for convenience it is called the sign bigit.

In either the "signed" number representation or the "complement" number representation $|x| < 1$, and the "sign" of the number is determined by the leftmost bigit.

Since the sign of a number is identified by examining the sign bigit, we are naturally led to treating zero as positive for computational purposes. Since a 1 in the sign bigit indicates a negative

number and since the sign bigit also has numerical significance, one may interpret 1 in the sign bigit followed by all 0's as the integer -1 and operate with it accordingly. The allowed number range in the computer is then $-1 \leq x < 1$.

All numbers are of the form:

$$x(\text{mod } 2^{N+1}) \equiv \sum_{i=0}^{N} 2^i + 1 + x,$$

where N may be any suitably chosen value. For the discussion of addition and subtraction it suffices to take N=0 and to consider a negative number as represented by its complement with respect to 2. For the multiplication process, $N \geq 39$. The details are considered presently. Since the computer contains numbers modulo 2, we actually see x or $(2-|x|)$ according as x is positive or negative, and we refer either to the number or its complement with respect to 2. However, the existence of the (N+1) bigits left of the binary point is implied.

Shifting

Shifting is one of the more basic operations the computer performs and perhaps should be the first of the arithmetic operations discussed. The left and right shift provide a means of multiplying by 2^n where $-40 \leq n \leq 40$.

Recall that x is represented as:

$$x(\text{mod } 2^{N+1}) \equiv \sum_{i=0}^{N} 2^i + (1+x).$$

Performing a left shift of n places, $0 \leq n \leq 40$, gives:

$$2^n x = 2^n \sum_{i=0}^{N} 2^i + 2^n(1+x)$$

$$2^n x(\text{mod } 2^{N+1}) \equiv \sum_{i=n}^{N} 2^i + 2^n + 2^n x$$

$$= \sum_{i=0}^{N} 2^i + (1+2^n x)$$

which conforms with the adopted complement notation.

As previously stated, all numbers in the computer must have a numeric value less than 1; therefore, for this "power" shift to be a legitimate operation

$$2^n|x| < 1,$$

or

$$|x| < 2^{-n}$$

In the computer where the left shift takes place modulo 2, the sign bigit is treated as a numerical bigit, and at each step of the shift the 2^{-1} bigit shifts into the sign bigit. After an n-fold shift where $|x| < 2^{-n}$, the shifted number still has the proper sign representation as is indicated by the algebraic representation.

There are other schemes of left shifting; for example, where the sign is not affected and numerical bigits are lost from the 2^{-1} bigit position. For purposes of power shifting this scheme is comparable to the scheme adopted. However, when one uses shifting facilities to separate a multiplex of numbers stored as a 40-bigit aggregate our scheme allows much more flexibility. This is not the place for a discussion of non-standard operation; hence we delay the discussion of shifting as it applies to such cases until a later time.

Performing a right shift of n places, $0 \leq n \leq 40$, gives:

$$2^{-n}x = 2^{-n}\sum_{i=o}^{N} 2^i + 2^{-n}(1+x) \qquad N > n$$

$$= \sum_{i=o}^{N} 2^{i-n} + 2^{-n} + 2^{-n}x$$

$$= \sum_{i=o}^{N-n} 2^i + \left(\sum_{i=1}^{n} 2^{-i} + 2^{-n} + 2^{-n}x\right)$$

$$= \sum_{i=o}^{N-n} 2^i + (1 - 2^{-n} + 2^{-n} + 2^{-n}x)$$

$$= \sum_{i=o}^{N-n} 2^i + (1 + 2^{-n}x)$$

$$N - n = N' > 0$$

$$2^{-n}x \pmod{2^{N'+1}} \equiv \sum_{i=o}^{N'} 2^i + (1 + 2^{-n}x)$$

which conforms with our complement notation.

Phenomenologically, one may say that in a right shift the sign bigit fills into the bigit positions that are vacated by the shift. The output to the right of the 2^{-39} position is still available elsewhere, but is of no concern in the present discussion.

For examples of shifting, consider a left shift of 4 and a right shift of 4 where $\underline{x}$ is in each case a negative number. A negative $\underline{x}$ is used as it provides the more interesting example. The shift examples are considered modulo 2 as this is the computer representation.

(i) shift $\underline{x}$ left 4,

$$x = -0.00001011$$
$$2-|x| = 1.11110101$$
$$2^4(2-|x|) = 11111.01010000$$
$$2^4(2-|x|) \bmod 2 = 1.01010000 \quad \text{equivalent to the signed number } -0.10110000$$

(ii) shift right 4,

$$x = -0.10101011$$
$$2-|x| = 1.01010101$$
$$2^{-4}(2-|x|) = 1.111101010101 \quad \text{then truncating}$$
$$\text{gives } 2^{-4}(2-|x|) = 1.11110101 \quad \text{equivalent to the signed number } -0.00001011$$

In the right shift the resulting number may be in error by at most $\underline{1}$ in the rightmost position because of the truncation. One can reduce this truncation error by introducing a "round-off" scheme in the right shift.

Addition and Subtraction

Consider the sum $S = (x+y)$. Not only must $|x|, |y| < 1$, but $|S| < 1$. $(x+y)$ is represented in complement notation as

$$\sum_{i=0}^{N} 2^i + 1 + x + \sum_{i=0}^{N} 2^i + 1 + y = 2 \sum_{i=0}^{N} 2^i + 2 + (x+y)$$

$$\sum_{i=0}^{N+1} 2^i + (1+x+y) = \sum_{i=0}^{N+1} 2^i + (1+S).$$

Hence S is of the same form as x and y. If $(x+y) \geq 0$, then

$$\left\{ \sum_{i=0}^{N+1} 2^i + (1+S) \right\} \bmod 2^{N+1} \equiv S \geq 0.$$

If $(x+y) < 0$, then the result is:

$$\sum_{i=0}^{N+1} 2^i + (1-|S|).$$

In either case we have the correct interpretation. Since the result is viewed modulo 2, we may set N=0 in the above equations without affecting the results. Therefore, in addition and subtraction numbers are of the form (2+u), where $-1 \leq u < 1$.

$$(2+x) + (2+y) = 4 + (x+y) = 4 + S$$

and if $(x+y) \geq 0$, then $(4+S) \bmod 2 \equiv S$. On the other hand, if $(X+y) < 0$, then $(4-|S|) \bmod 2 \equiv 2-|S|$. Therefore, the signs of $\underline{x}$ and $\underline{y}$ do not alter the process and one may, by means of addition, effect either sums or differences.

Clearly, if it is desired to form the difference (x-y) of two numbers $\underline{x}$ and $\underline{y}$ where their representations are

$$\sum_{i=0}^{N} 2^i + (1+x) \quad \text{and} \quad \sum_{i=0}^{N} 2^i + (1+y),$$

we must first represent $-\underline{y}$ in this notation which is

$$\sum_{i=0}^{N} 2^i + (1-y).$$

This is referred to as the complement of $\underline{y}$ with respect to 2^N. For subtraction it suffices to be able to form the complement of numbers with respect to 2.

To form the complement (2-y), write

$$2 - y = (2-2^{-n}-y) + 2^{-n}$$

where $\underline{n}$ is the rightmost bigit position. Since

$$2 - 2^{-n} = \sum_{i=0}^{n} 2^{-i},$$

$(2-2^{-n}-y)$ is the reflection of each bigit of $\underline{y}$; that is, where there is a $\underline{1}$ in $\underline{y}$ there will be a $\underline{0}$ in $(2-2^{-n}-y)$. The complement is completed by adding 2^{-n} to the difference. For example,

$$\begin{array}{rll} 2 - 2^{-12} & = & 1.1111\ 1111\ 1111 \\ - y & = & \underline{-0.1101\ 0110\ 1011} \\ \text{which reflects each bigit of y:} & & 1.0010\ 1001\ 0100 \\ \text{Adding } \underline{1} \text{ into the rightmost bigit} & & \underline{\qquad\qquad\qquad\quad 1} \\ \text{gives the complement} \quad 2 - y & = & 1.0010\ 1001\ 0101 \end{array}$$

This method of reflecting each bigit and adding 1 into the rightmost bigit position is, in essence, the method by which the computer forms complements.

Examples of addition and subtraction are:

(1) $x = 0.00101011$; $y = 0.01000111$; form $S = x + y$

$$\begin{aligned} x &= 0.00101011 \\ y &= \underline{0.01000111} \\ x + y = S &= 0.01110010 \end{aligned}$$

(2) $x = 0.10101101$; $y = 0.11010110$; form $S = x - y$

$$\begin{aligned} x &= 0.10101101 \\ 2 - y &= \underline{1.00101010} \\ 2 + (x-y) = 2 - |S| &= 1.11010111 \end{aligned}$$

Multiplication

We consider the multiplication of y, a 39-bigit multiplicand and sign, by x, a 39-bigit multiplier and sign. The product P is a 78-bigit product and sign. It has previously been stated that $|x|$, $|y| < 1$; therefore it follows that $|P| < 1$. Here is an advantage of placing the binary point to the left of the first numerical bigit. If $|x|$, $|y| > 1$ were allowed, the product P could be greater than either factor, and P would have its binary point in a position different from either that of x or y.

To develop a multiplication scheme, consider two numbers x and y where $|x|$, $|y| < 1$. Since the complement notation is used, their product is:

$$P = \left\{\sum_{i=0}^{N} 2^i + (1+x)\right\}\left\{\sum_{i=0}^{N} 2^i + (1+y)\right\}$$

$$= (1+x)(1+y)+(1+x)\cdot\sum_{i=0}^{N} 2^i + (1+y)\cdot\sum_{i=0}^{N} 2^i + \sum_{i=0}^{N} 2^i \cdot \sum_{i=0}^{N} 2^i$$

$$= 1 + x + y + xy + \sum_{i=0}^{N} 2^i + x\cdot\sum_{i=0}^{N} 2^i + \sum_{i=0}^{N} 2^i + y\cdot\sum_{i=0}^{N} 2^i + \sum_{i=0}^{N} 2^i\cdot\sum_{i=0}^{N} 2^i$$

Using the relation

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1,$$

One obtains:

$$P = 1 + x + y + xy + 2^{N+1} - 1 + 2^{N+1}x -$$
$$- x + 2^{N+1} - 1 + 2^{N+1}y - y + 2^{2N+2} - 2\cdot 2^{N+1} + 1.$$

Collecting terms:

$$P = 2^{2N+2} + 2^{N+1}(x+y) + xy.$$

Since

$$2^{2N+2} = \sum_{i=0}^{2N+1} 2^i + 1$$

rewrite the product as:

$$P = 2^{N+1}(x+y) + \sum_{i=0}^{2N+1} 2^i + (1+xy).$$

Either $(x+y) = 0$, or $|x+y| \geq 2^{-39}$; hence, if we choose $N=39$, $2^{N+1}(x+y)$ is either 0 or greater than 2. Since the computer contains numbers modulo 2, $2^{40}(x+y) \bmod 2 \equiv 0$, and we see P as :

$$P \bmod 2 \equiv 2 + xy,$$

the correct complement notation.

The scheme as outlined is not desirable for the computer as it considers x mod 2^{40} which implies that the multiplication is a 78-step process rather than the conventional 39 steps.

One may modify the scheme so that it treats only the fractional part (but not the sign bigit) of the multiplier $\underline{x}$. Here, $\underline{x}$ has the representation $(\xi_o + x)$ where $\xi_o = 0$ if $x \geq 0$ and $\xi_o = 1$ if $x < 0$; i.e., the complement of $\underline{x}$ with respect to $\underline{1}$ if $\underline{x}$ is negative. By a procedure similar to the above, one finds

$$(\xi_o + x)\left(\sum_{i=0}^{39} 2^i + 1 + y\right) = 2^{40}\xi_o + 2^{40}x + \xi_o y + xy.$$

Rewrite the product P as

$$P = 2^{40}\xi_o + 2^{40}(x - 2^{-39}) + 2 + \xi_o y + xy.$$

Then, as in the preceding case, consider P mod 2 and

$$P \bmod 2 \equiv 2 + \xi_o y + xy.$$

If $x \geq 0$, then $\xi_o = 0$ and

$$P \bmod 2 \equiv 2 + xy,$$

the correct product using complement notation. If $x < 0$, then $\xi_o = 1$ and $P \bmod 2 \equiv 2 + y + xy.$

Clearly, one needs to subtract $\underline{y}$ to gain the desired product. An additional step is required in this scheme if the multiplier is negative, namely adding the complement of the multiplicand $\underline{y}$ to the product.

The multiplication is accomplished by examining the multiplier a bigit at a time, beginning with the least significant bigit, and performing the indicated operation. If the multiplier bigit is a $\underline{1}$, the multiplicand is added into the partial product; then the sum and multiplier are shifted right one place. If the multiplier bigit is a $\underline{0}$, the partial product and multiplier are merely shifted right one. The multiplication involves 40 steps; the first 39 steps either add the multiplicand to the partial product and shift the sum right one unit, or merely shift the partial product right one unit according as the examined bigit of the multiplier is $\underline{1}$ or $\underline{0}$. The 40^{th} step adds the complement of the multiplicand to the partial product or does nothing according as the sign bigit of the multiplier is $\underline{1}$ or $\underline{0}$.

The computer can only perform operations modulo 2; therefore some way is needed of simulating the multiplicand modulo 2^{40}. To find a suitable method, we examine whether there is a simple relation between the sign of the partial products, as viewed in the computer, and the sign of the multiplicand for the scheme discussed immediately above. We now prove that after the first $\underline{1}$ is encountered in the examination of the successive bigits of the multiplier (prior to that the partial product is zero), the signs of the multiplicand and the partial product agree.

Assume the partial product p_i is of the form:

$$p_i = \sum_{i=o}^{N} 2^i + (1+b) \qquad \text{where} \quad |b| < 1;$$

if the $(i+1)^{th}$ bigit of the multiplier is a $\underline{1}$

$$2p_{i+1} = \sum_{i=o}^{N} 2^i + (1+b) + \sum_{i=o}^{N} 2^i + (1+y)$$

$$= 2\sum_{i=o}^{N} 2^i + (2+b+y)$$

$$p_{i+1} = \sum_{i=o}^{N} 2^i + (1+\frac{b+y}{2}).$$

Now $|b| < 1$ and $|y| < 1$; therefore $|b+y|/2 = |b'| < 1$ and

$$p_{i+1} = \sum_{i=o}^{N} 2^i + (1+b'). \qquad \text{Eq. (1)}$$

For the case where the $(i+1)^{th}$ bigit of the multiplier is a 0, it is easy to see that b' of Eq. (1) is equal to b/2. The partial product is originally 0, but after the first 1 appears in the multiplier, the partial product p is:

$$2p = \sum_{i=o}^{N'} 2^i + 1 + y \qquad |y| < 1$$

$$p = \sum_{i=o}^{N-1} 2^i + 1 + y/2 \qquad |y/2| < 1$$

Therefore, by induction all succeeding partial products are of the form:

$$p_i = \sum_{i=o}^{N'} 2^i + 1 + b \qquad |b| < 1$$

Inasmuch as the various increments to the partial product all have the same sign, namely that of the multiplicand, and since it has been shown that $|b'| < 1$ for all possibilities, it is clear that the sign of the partial product agrees with that of the multiplicand (again, after the first 1 appears in the multiplier). Hence, if it is arranged so that this condition is satisfied in the course of multiplication as done by the computer, then one has simulated the multiplicand modulo 2^{40} and the above scheme may be adopted.

It turns out, however, that multiplication as done by the computer may cause the sign bigit to change; consequently it must be arranged to keep it invariant after the first 1 of the multiplier appears. To see that the sign bigit may change if no precautions are taken, consider the magnitude of the p_i's:

$$2p_{i+1} = p_i + y \qquad \text{where } |y| < 1$$

$$p_{i+1} = \frac{p_i+y}{2}$$

$$|p_i+y| \leq |p_i| + |y| < 2$$

$$\frac{|p_i+y|}{2} < 1$$

and

$$|p_{i+1}| < 1.$$

Since $p_o = 0$, by induction all $|p_i| < 1$. Although $|p_i| < 1$, $2|p_i|$ is not necessarily less than one, but $2|p_i| < 2$. At each step $2|p_i|$ is formed and then shifted right one unit. This implies that in forming $2p_i$ one does not lose significant bigits of the partial product, but the "sign" bigit may be lost. The loss of the "sign" bigit is the result of the addition at each step being done modulo 2.

The multiplication of a 39-bigit number by a 39-bigit number gives a 78-bigit product. When one is interested in single precision operation, i.e., operation with 39-bigit numbers, the 78-bigit product is 'rounded-off" to 39 bigits. That is, the 78-bigit product is approximated by a 39-bigit product. There are several methods for doing "round-off" that are applicable to our needs. We have chosen for multiplication the scheme in which all bigits beyond and including the n^{th} bigit are ignored and the n^{th} bigit is set to a 1. At this point we do not plan to argue the validity of this round-off scheme. We may, however, state that the scheme is unbiased, and it has a variance of $1/3 \cdot 2^{2n}$.

The multiplication may be summarized as follows: There are 39 steps in which the multiplier is examined a bigit at a time. At each examination the multiplicand is added to the partial product or nothing is done, according as the multiplier bigit is a 1 or a 0. In either case the result is shifted right one unit and the process is repeated for 39 steps. When the first 1 appears in the multiplier, the sign bigit of the partial product is, on this and all subsequent steps, set equal to the sign bigit of the multiplicand. The 40^{th} step either adds in the complement of the multiplicand or does nothing, according as the sign of the multiplier is a 1 or a 0. And at the end of the 40^{th} step the 39^{th} bigit of the product is set to a 1 if the multiplication is done with round-off; or nothing is done if the multiplication is without round-off.

We consider two examples of multiplication. For simplicity we use three-bigit multipliers and multiplicands. Both examples are with negative multiplicands as this affords the most interesting cases. The first

example has a positive multiplier and the product is rounded-off to three bigits. This round-off to three bigits, of course, tends to give a more distorted product than would occur in the computer where the product is rounded-off to 39 bigits. The second example has a negative multiplier; hence, as a correction, the complement of the multiplicand is added to the product in the last step. This example considers multiplication without round-off.

Example 1:

$x = 0.111 = 7/8$ $\qquad y = 1.001 = -7/8$

$xy = 1.001111 = -49/64$

$xy(ro) = 1.001 = -7/8$ (The round-off scheme used is to set the 2^{-3} position to a 1. In this instance it is a 1; hence no change is made.)

$y = 1.001$

$p_o = 0.000$ $\qquad x_o = 0.111$

(i)

$p_o = 0.000$
$+y = 1.001$
$2p_1 = 1.001$
$p_1 = 1.1001$ $\qquad x_1 = 0.011$

(ii)

$p_1 = 1.1001$
$+y = 1.001$
$2p_2 = [1]0.1011$
$p_2 = 1.01011$ $\qquad x_2 = 0.001$

(iii)

$p_2 = 1.01011$
$+y = 1.001$
$2p_3 = [1]0.01111$
$p_3 = 1.001111$ $\qquad x_3 = 0.000$

(iv) $\qquad P = p_3(ro) = 1.001$

Step (i): Initially ($p_o = 0$). The rightmost bigit of the multiplier is examined. Since it is a 1, y is added to p_o to give $2p_1$. We have a negative multiplicand; hence, from this step on, the sign of the partial product is set to the sign of the multiplicand. $2p_1$ is shifted right one place

to give p_1, and the sign of p_1 is set to a 1. x_o is shifted right one place to form x_1, which again has a 1 in the rightmost position.

Step (ii): y is added to p_1 to form $2p_2$. (Note that in adding $(y+p_1)$ $2p_2$ is written as [1] 0.1011. The 1 does not exist in the computer as it adds modulo 2; hence the 1 is shown in brackets and does not enter into the product.) $2p_2$ is shifted right to form p_2, and the sign bigit is set to a 1. x_1 is shifted right to give x_2.

Step (iii): Identical in procedure to Step (ii).

Step (iv): x_3 is examined and the rightmost bigit (the original sign of the multiplier) is a 0; hence no correction term is needed. Round-off is indicated; hence the right-hand three bigits are truncated and the 2^{-3} bigit is set to a 1. In this instance it is a 1; therefore no action is required.

Example 2:

$x = 1.101 = -3/8$ $\quad y = 1.011 = -5/8$

$xy = 0.001111 = 15/64$

$y = 1.011$
$p_o = 0.000$ $\quad x_o = 1.101$

(i)
$p_o = 0.000$
$+y = 1.011$
$2p_1 = 1.011$
$p_1 = 1.1011$ $\quad x_1 = 1.110$

(ii)
$p_1 = 1.1011$
$2p_2 = 1.1011$
$p_2 = 1.11011$ $\quad x_2 = 1.111$

(iii)
$p_2 = 1.11011$
$+y = 1.011$
$2p_3 = [1]1.00111$
$p_3 = 1.100111$ $\quad x_3 = 1.111$

(iv)
$p_3 = 1.100111$
$+(2-y) = 0.101$
$P = [1]0.001111$
$= 0.001111$

Steps (i) and (iii): These are identical in procedure to the preceding example.

Step (ii): The multiplier bigit is a 0; hence p_1 is shifted right one unit to form p_2.

Step (iv): The rightmost bigit of x_3 is a 1 indicating the complement correction. (2-y) is added to p_3 to give the correct product, P. (If round-off had been indicated, the right-hand three bigits would now be truncated and the 2^{-3} bigit of the product set to a 1.

Division

The division scheme adopted for the computer is a pseudo-non-restoring scheme. Before discussing the scheme, we compare a true non-restoring scheme with the more familiar restoring type of division.

For simplicity of discussion, we assert that x, the dividend, and y, the divisor, are positive. Further we assert that for any division scheme

$$|x| < |y| < 1$$

(all numbers in the computer must be less than 1).

In the restoring scheme, the divisor is continually subtracted from the partial remainder (the dividend on the first step) until the remainder is less than the divisor. The number of such subtractions is then recorded in the appropriate position in the quotient. The partial remainder is then shifted left one unit and the process is repeated.

In the non-restoring scheme the divisor is subtracted from the partial remainder (the dividend on the first step) until the partial remainder becomes negative. The number of such subtractions is then recorded in the appropriate position in the quotient. The partial remainder is then shifted left one unit, but now the divisor is added to the partial remainder until the partial remainder again becomes a positive quantity. The number of such additions is then appropriately positioned and subtracted from the existing partial quotient.

These two sequences are then repeated ad infinitum with the sign of the partial remainder being either positive or negative. The quotient is formed by a succession of additions and subtractions.

If we consider the binary base, a well ordered division may have only one addition or subtraction for each fixed quotient position. This may be seen most clearly by referring again to the restoring scheme. If the dividend is initially less than the divisor, then for any fixed quotient position there may be at most (m-1) subtractions (where m is the number base) before the partial remainder becomes smaller than the divisor. In the non-restoring scheme it is not necessary to have more than (m-1) subtractions or additions for a fixed quotient position, as it suffices to know that the dividend is less than the divisor. Since (m=2) for the binary case, one addition or subtraction suffices for each quotient position.

An example of a well-ordered non-restoring division in binary form is:

$$15/64 \div 3/4 = 5/16$$

$$\begin{array}{r|l} & (1x2^{0})-(1x2^{-1})-(1x2^{-2})+1x2^{-3})-(1x2^{-4}) \\ 0.11 & 0.001111 \\ & \underline{1.01} \quad \text{(i)} \\ & 1.011111 \\ & \underline{0.011} \quad \text{(ii)} \\ & 1.110111 \\ & \underline{0.0011} \quad \text{(iii)} \\ & 0.000011 \\ & \underline{1.11101} \quad \text{(iv)} \\ & 1.111101 \\ & \underline{0.000011} \quad \text{(v)} \\ & 0.000000 \end{array}$$

Collecting terms of the quotient gives:

$$\begin{aligned} 1 \times 2^{0} + 1 \times 2^{-3} &= 1.0010 \\ -(1x2^{-1}+1x2^{-2}+1x2^{-4}) &= -(\underline{0.1101}) \\ 5/16 &= 0.0101 \end{aligned}$$

Step (i): The sign of the divisor and dividend (partial remainder) are the same. The first quotient position is chosen as the 2^{0} position; hence a 1 is recorded and the divisor

is subtracted from the dividend (the subtraction is done using complement notation).

Step (ii): The partial remainder is now negative; hence its sign differs from that of the divisor. $-\underline{1}$ is recorded in the 2^{-1} quotient position and the (divisor) x 2^{-1} is added to the partial remainder. In the computer the partial remainder is shifted left one unit rather than shifting the divisor right one unit as it is added. In essence the two are equivalent; however the former is more advantageous with respect to computer operation.

Step (iii): The partial remainder is still negative, a $-\underline{1}$ is inserted into the 2^{-2} quotient position, and the (divisor) x 2^{-2} is added to the partial remainder.

Step (iv): The partial remainder is positive; hence a $\underline{1}$ is recorded in the 2^{-3} position of the quotient and the (divisor) x 2^{-3} is subtracted from the partial remainder.

Step (v): The partial remainder is negative, so $-\underline{1}$ is recorded in the 2^{-4} position of the quotient. The (divisor) x 2^{-4} is added to the partial remainder giving a new partial remainder of $\underline{0}$ which terminates the division.

Step (vi): The indicated additions and subtractions in the quotient are performed. The result is the desired quotient.

Note that the restriction of treating $\underline{x}$ and $\underline{y}$ as positive numbers is not necessary in the non-restoring scheme as the sign of the partial remainder (x, initially) may be either positive or negative. It is not needed to know the specific sign of each factor but only the relation between the sign of the divisor and dividend. Hence, in further discussion no sign restrictions are necessary.

As each step of the quotient involves an addition or a subtraction, the true non-restoring scheme would necessitate a second register that had all the complications associated with the adding facilities. There is, however, a simple relationship between the true non-restoring quotient that is written as a series of $\underline{1}$'s and $-\underline{1}$'s and a pseudo-non-restoring quotient obtained by replacing the $-\underline{1}$ by $\underline{0}$ wherever it occurs. This relation, first shown by von Neumann, may be found as follows:

Write the true quotient Q in non-restoring form as:

$$Q = 2^{o}\lambda_{o} + 2^{-1}\lambda_{1} + \ldots + 2^{-n}\lambda_{n} + r_{n}$$

where λ_i may be ± 1 and r_n may be positive or negative. Using the transformation $\lambda_i = 2c_i - 1$ where $c_i = 0$ if $\lambda_i = -1$ and $c_i = 1$ if $\lambda_i = 1$, one obtains:

$$Q = 2^{o}(2c_{o}-1) + 2^{-1}(2c_{1}-1) + \ldots + 2^{-n}(2c_{n}-1) + r_{n}$$

$$= 2(2^{o}c_{o}+2^{-1}c_{1}+\ldots+2^{-n}c_{n}) - (2^{o}+2^{-1}+\ldots+2^{-n}) + r_{n}$$

If we assert that the pseudo-quotient C is:

$$C = 2^{o}c_{o} + 2^{-1}c_{1} + \ldots + 2^{-n}c_{n},$$

then, since

$$-(2^{o}+2^{-1}+\ldots+2^{-n}) = -\sum_{i=o}^{n} 2^{-i} = -(2-2^{-n}),$$

$$Q = 2C - 2 + 2^{-n} + r_{n},$$

$$2 + Q = 2C + 2^{-n} + r_{n}.$$

If we form the pseudo-quotient C, multiply it by 2 (a simple left shift), and add 2^{-n}, the result is (2+Q) which is the correct complement notation with respect to 2. In our instance $2^{-n} = 2^{-39}$ (the rightmost bigit position).

The 2^{-39} that is introduced is, in effect, round-off of the same type as that used in multiplication.

The pseudo-non-restoring scheme is the one actually used in the computer.

For an example of division, divide

$$49/128 \div -7/8 = -7/16$$

Divisor	Partial Remainder	Quotient
$y = 1.001 = -7/8,$	$x = r_o = 0.0110001 = 49/128,$	0
(i)	$r_o = 0.0110001$ $+y^o = 1.001$ $r_1 = 1.1000001$	0.

(ii)
$$\begin{array}{rll} 2r_1 & = 1\ 1.0000010 & \\ +(\underline{2-y}) & = \quad 0.111 & \\ \hline r_2 & = \quad 1.1110010 & 0.1 \end{array}$$

(iii)
$$\begin{array}{rll} 2r_2 & = 1\ 1.1100100 & \\ +(\underline{2-y}) & = \quad 0.111 & \\ \hline r_3 & = \quad 0.1010100 & 0.11 \end{array}$$

(iv)
$$\begin{array}{rll} 2r_3 & = 0\ 1.0101000 & \\ \underline{+y} & = \quad 1.001 & \\ \hline r_4 & = \quad 0.0111000 & 0.110 \end{array}$$

(v)
$$\begin{array}{rll} 2r_4 & = 0\ 0.1110000 & \\ \underline{+y} & = \quad 1.001 & \\ \hline r_5 & = \quad 0.0000000 & 0.1100 \end{array}$$

(vi)
$$C = 0.1100$$

$$Q = 2C + 2^{-4} = 1.1001 = -7/16$$

Step (i): The sign of the partial remainder (dividend at this step) and the sign of the divisor are different; hence the divisor (y) is added to the partial remainder (r_o) and a $\underline{0}$ is recorded in the quotient.

Step (ii): The sign of r_1 and $\underline{y}$ are the same; hence the complement of $\underline{y}$ is added to $2r_1$ and a $\underline{1}$ is recorded in the quotient.

Step (iii): The sign of r_2 and $\underline{y}$ are the same; hence the complement of $\underline{y}$ is added to $2r_2$ and a $\underline{1}$ is recorded in the quotient.

Step (iv): The sign of r_3 and $\underline{y}$ are different; hence $\underline{y}$ is added to $2r_3$ and a $\underline{0}$ is recorded in the quotient.

Step (v): The sign of r_4 and $\underline{y}$ are different; hence $\underline{y}$ is added to $2r_4$ and a $\underline{0}$ is recorded in the quotient. (r_5=0) so the division steps are completed.

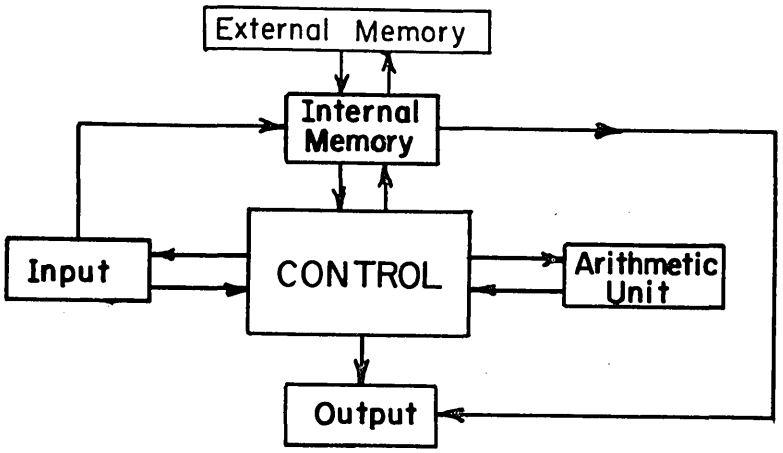
Step (vi): Shift C, the quotient resulting from the first 5 steps, left one place and add 2^{-4}. This gives the true Q.

The computer would not terminate, as we have done, when the remainder is $\underline{0}$. It would carry the division out to 40 steps rather than 5, and then insert a $\underline{1}$ into the 2^{-39} position. Obviously this does not give an exact answer. In fact, the computer quotient for the given example would be $Q = 1.1000111\ldots111, = -(7/16 + 2^{-39})$.

IV THE COMPUTER

Block Diagram

In this part we discuss in more detail the various components of the computer and the various interactions between them. We begin with a simple block diagram of the computer:



The block diagram shows the components with their various interconnections. Some of these connections are for logical (non-arithmetical) operations and others to transfer numerical data from one component to another. It is observed that the control is the central agency in the organization and directs the operation of the other components. It signals the input to read new information into the internal memory and receives a signal when the operation is completed. The control directs the internal memory to provide the next order to be executed; further, it transfers numbers from the memory to the arithmetic unit, and conversely. The control directs the transfer of numbers between the internal and external memory. It supplies the sequence of pulses and voltage changes to the arithmetic unit to effect the various mathematical operations. Finally, it instructs the output to punch a paper tape and print page information from the memory for external use.

Arithmetic Unit

We follow the same pattern as in the Introduction and begin with the arithmetic unit. A schematic cross-section of the arithmetic unit proper is shown:

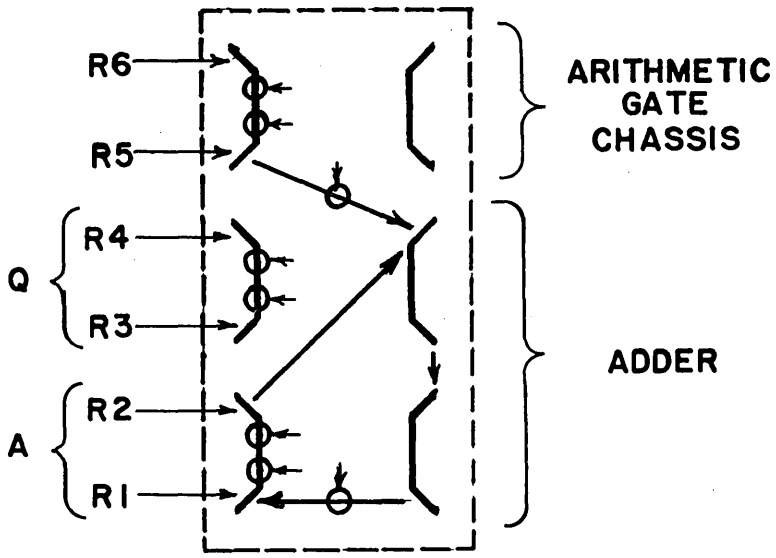


Fig. 1 Schematic Cross-Section of Arithmetic Unit. Circles with the small arrows indicate gate tubes, or electronic switches. Also shown are the interconnections for the addition process.

The six registers, R1, R2,...R6, are mounted in pairs on three horizontal, three-dimensional chassis, a type proposed by Bigelow. It is sometimes convenient to refer to the pair, R1, R2, by the single letter A (for accumulator); R3, R4 are designated by Q (for quotient register). R1 and R3 provide a method for the shifting of numbers in R2 and R4, respectively, so it is quite natural to think of the two doublets of registers, A and Q, as single entities. However, R5 and R6 are not so interconnected; in fact they perform quite different functions. Nevertheless, it is compact to have them also juxtaposed.

Opposite the three chassis of registers are three other sets, quite similar in appearance. The lower two constitute the adder proper; the topmost is called the arithmetic gate chassis.

We discuss first the registers. Each register is a set of 40 flip-flops. Between the two rows of flip-flops in a chassis are two other rows of tubes. These are the so-called gate tubes (electronic

switches) and allow for four different types of switching action. (Each tube contains two halves which can be used independently; such tubes are sometimes called double triodes.)

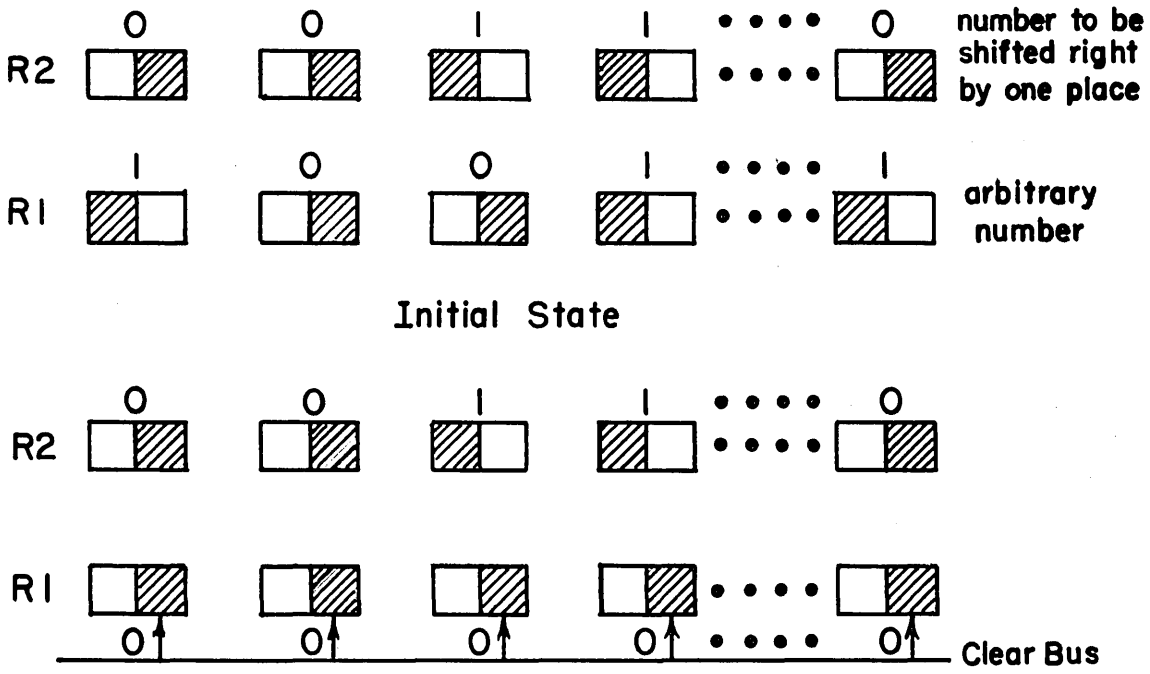
A flip-flop is a relatively simple electronic circuit containing a tube consisting of two separate parts, such that either one half is conducting current and the other is cut off, or the converse. These two modes of operation correspond to two stable configurations, and one state is said to represent a "0", the other a "1". A flip-flop is schematically drawn as a rectangle of two squares, one being shaded to indicate conduction. We adopt the following convention:

A small neon is connected to each flip-flop; "off" corresponds to a 0 and "on" to a 1.

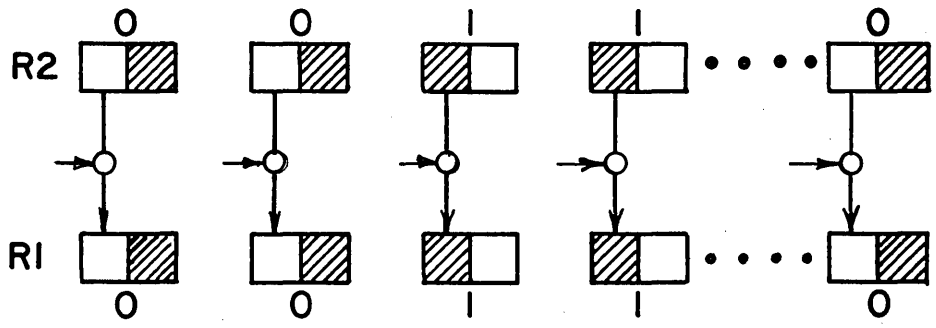
As mentioned in the Introduction there are two alternative methods for transferring information from one set of flip-flops to another. Consider two sets of flip-flops, A and B. There exist circuits--gating schemes--whereby it is possible to transfer information from A⟶B independent of the previous states of the individual flip-flops of B. The alternative procedure would be to first reset all of B to 0's and then cause only those flip-flops of B to be set to 1 whose corresponding flip-flops in A contain 1. Quite clearly, B could be first reset (or "cleared") to all 1's and then the 0's from A could be transferred to the corresponding flip-flops of B. The latter method with both schemes of "clearing" and gating is used in the computer.

We indicate diagrammatically how a number 0011···0 in, say, R2 is shifted to the right by one binary place. R1 initially contains an arbitrary number from some previous operation. (See Figure below.)

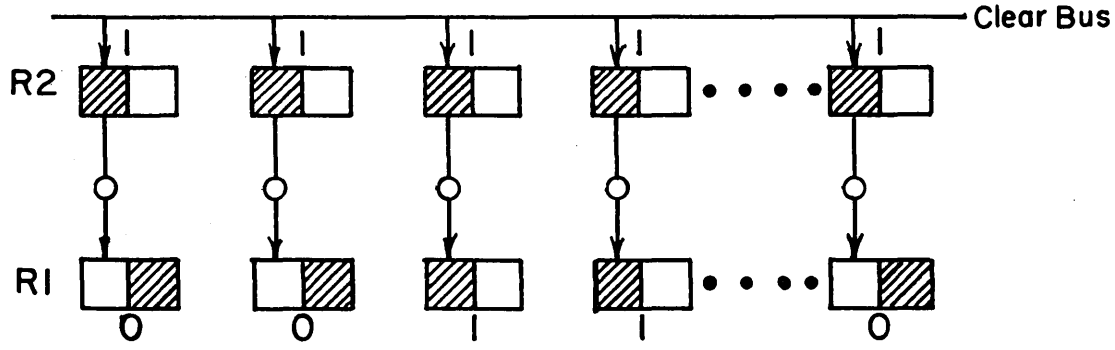
As a result of the four steps, the number originally in R2 has been shifted to the right by one binary place. It is observed that the leftmost flip-flop of R1, the flip-flop of the sign bigit, has an additional gate leading to the "sign" flip-flop of R2, as is of course required



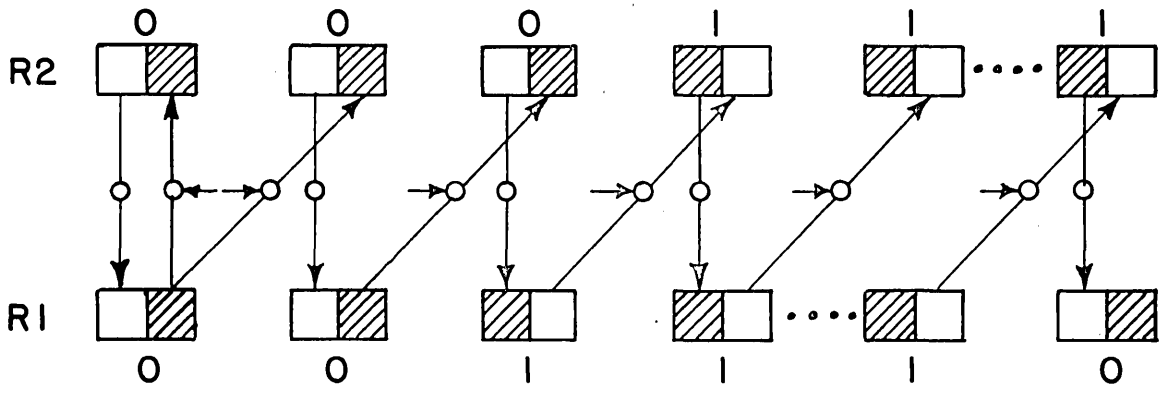
Step 1. Clear R1 to ZEROS by voltage pulse on Clear Bus. Symbol = CoR1



Step 2. Flip-flops of R2 containing "1" cause corresponding flip-flops of R1 to set to "1" when voltage pulse is applied to gate tubes.



Step 3. Clear R2 to "1" Symbol = C 1 R2



Step 4. Flip-flops of R1 containing "0" cause corresponding flip-flops of R2 to set to "0".

to propagate the sign bigit. With the aid of a third set of gate tubes connected diagonally to the left, shifting to the left by one binary place is essentially the same sequence as in the above, except that in Step 4 the third set of gates would be pulsed.

It is convenient to label the sequence of toggles in a register by 0,1,...39 starting from the left, so that there is a one-to-one correspondence between a flip-flop and the magnitude of the exponent of that binary place; e.g., 0R2 designates the sign flip-flop of R2, (0-7)R1 refers to the first eight flip-flops of R1.

The chassis with R3 and R4 has a similar set of gate connections. In fact, whenever a shift occurs in A the same process occurs in Q; both multiplication and division processes make use of the simultaneous shifting. Furthermore, it is desirable in some instances not to lose the information which would otherwise disappear by truncation at the ends of A. In order to retain the information, flip-flop 0R1 is connected to 39R4, and the information being truncated at the left of A is introduced at the right in Q. The information being truncated

at the left of Q is lost. Symbolically,

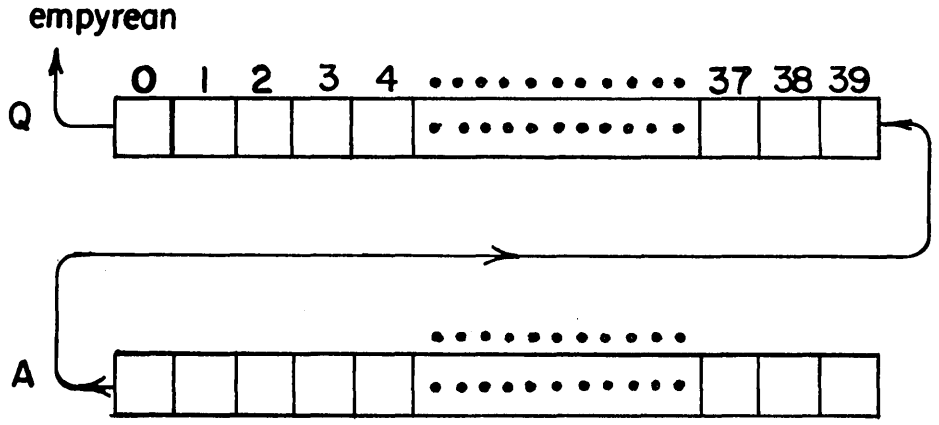


Fig. 2 Nature of left shift operation, showing interconnection of A and Q.

The sign flip-flop of A, OR2, is treated the same as the others of R2; i.e., the original sign of a number in R2 gets shifted, along with the numerical part. This type of shift operation facilitates the separation of multiple stored numbers.

In the right shift operations, Q again acts as a reservoir for the bigits spilling out of A. Here the bigits are introduced at the left in Q, beginning with the "sign" flip-flop, OR4. Diagrammatically,

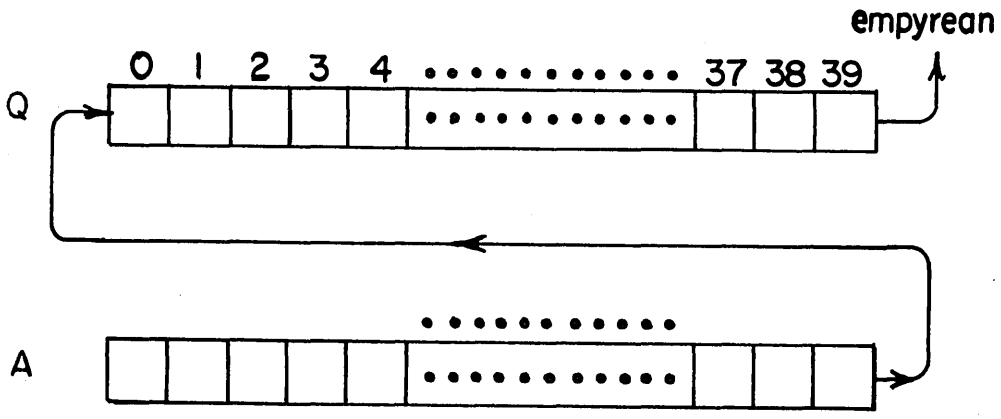


Fig. 3 Nature of right shift operation

Thus we can imagine that for the left shifts, Q is the continuation of

A on the left, and for the right shifts, Q is the extension of A on the right. For the right shift operation, it is of course necessary that the original sign bigit of R2 propagate. For example, a right shift by five binary places of the complement number, say 1001...in R2, results in 111111001.....

The Addition Process

A schematic drawing of the addition process is given in Figure 4.

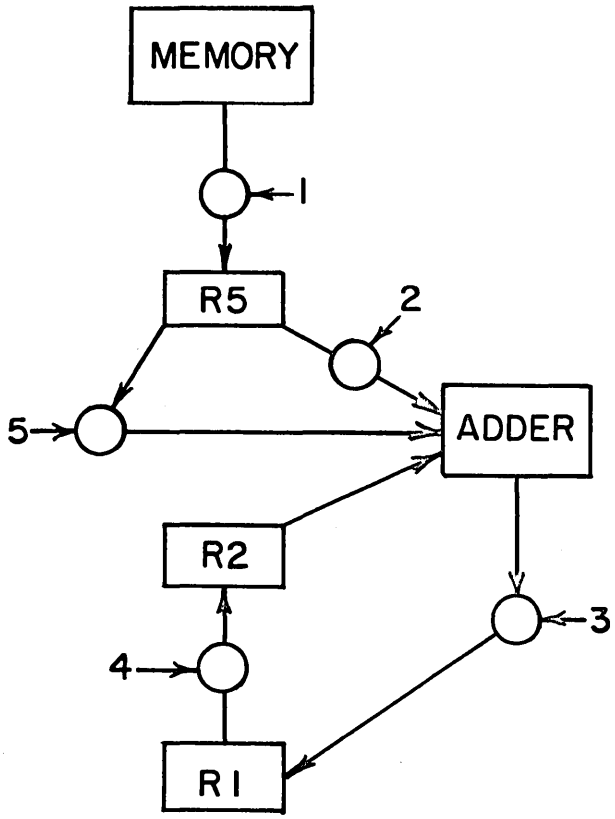


Fig. 4 Schematic cross-section of the arithmetic unit that participates in the addition process. As usual, circles indicate gate tubes. The small arrows represent symbolically the signals that stimulate the gating action. The clearing actions associated with each gating action are not shown.

The 40 stages of the various registers are represented by simple squares. As before, circles represent "gates". The two inputs to the adder are from R2 and R5. R2 is statically connected to the adder, so that its contents are always sent there, irrespective of whether or not an addition operation is being pursued. The number to be added to that in R2 comes initially from some memory location into R5. The pulse, indicated by 2 in Figure 4, gates this number into the adder. There the sum is formed. During this process, in preparation for receiving the sum, R1 is cleared. Finally gating action 4 transfers the sum from R1⟶R2. This latter gating action involves a displacement of the bigits to the right by one. In order to keep the position of the binary point unchanged, gating action 3 effects a shift of one to the left. An alternative scheme would be to have gating action 3 bring the sum into R1 without any shift. Then transfer to R2 with a right shift; return to R1 directly; but then go back to R2 with a left shift. This doubling back costs two extra clearing and gating actions. In place of this we have introduced another set of gates, in which the sum is brought into R1 displaced once to the left; then a single transfer to R2 completes the process. It should be mentioned that it is necessary to have an extra flip-flop, eR1, beyond OR1, which connects to OR2, the sign flip-flop of R2.

We have seen earlier that the subtraction d=(a-b) may be performed by adding to a the complement of b. We have also indicated that the complement information is quite naturally available in a set of flip-flops. Indeed, if a set of gates is connected to the adder from the side of the toggles opposite that normally used in addition, we can perform subtraction. Gating action 5, of Figure 4, transfers the complement of the number in R5 to the adder; the result (here, the difference) again appears finally in R2. Thus the addition and subtraction processes differ only in the choice of gating action 2 or 5, respectively. Whenever the "complement" gate 5 is used it must, of course, be accompanied by the insertion of a 1 into the 39th stage of the adder in order to obtain the true complement of the number in R5. This insertion is effected by stimulating a carry input into the 39th stage of the adder.

Multiplication

Inasmuch as multiplication is a series of additions, the nature of the addition process dictates in large part the role of the various registers in the multiplication process. When the multiply order is given, it is assumed that the multiplier factor is already residing in R4 as a result of a preceding instruction or of an earlier arithmetical operation. The address associated with this order refers to the memory location containing the multiplicand. The operation begins with the transfer of the multiplicand from the memory to R5; simultaneously R2 is cleared in preparation for the successive partial products. We distinguish two types of multiplication:

(i) no round-off, in which the full 78 bigits and sign are available, the significant portion appears in R2, and the right half appears in R4;

(ii) round-off, in which the first 39 bigits rounded-off are in R2. The remaining portion of the product is truncated.

In both types of multiplication the first step is the examination of the bigit in 39R4, the rightmost bigit of the multiplier. If it is a 1, an addition of the multiplicand and the partial product (at first, 0) is performed. R2 and R4 are then shifted to the right by one place. In the event that the bigit is a 0, R2 and R4 shift without an addition. The succeeding bigit of the multiplier is now examined in R4 and an addition is performed if the bigit is 1. Because of the preceding right shift of the partial product in R2, the direct addition of the multiplicand to it is appropriately placed. Note that the bigits being shifted out of R2 are no longer involved in the partial product sum. In the case of "nro" (no round-off) they are introduced into R4 at the left, where room is being made available by the right shifting of the multiplier. In "ro" multiplication, R4 is empty at the end of the process. The final step in the process involves the "multiplicand correction" (as discussed in the section on binary arithmetic) in the event the multiplier is negative, and the round-off procedure if the latter is indicated.

The successive additions that occur in forming the partial products differ in one respect from the single addition process associated

with the addition orders. In the latter case it will be recalled that the gate connection from the adder to R1 was such that the output of the adder was displaced one to the left, so that in the subsequent right-diagonal transfer from R1 → R2 the binary point is unchanged. In the multiplication process, a right shift of one is precisely what is needed of the partial sum; hence the gating from the adder to R1 is direct; i.e., the i^{th} stage of the adder is connected to the i^{th} stage of R1, and the subsequent transfer from R1 → R2 introduces the desired right shift by one.

In the control panel immediately to the left of the adder chassis is a six stage binary counter called the operations counter. At the beginning of the multiplication process, this counter is set to 23, and each cycle of the multiplication adds 1. It is arranged so that the iterative routine is interrupted after the counter reaches 63; i.e., the counter is filled with 1's. The full counter then terminates the routine, stimulates the multiplicand correction in the event of a negative multiplier, and finally initiates the round-off procedure if indicated. The sign bigit of the multiplier is at this time residing in 39R4 and is detected there.

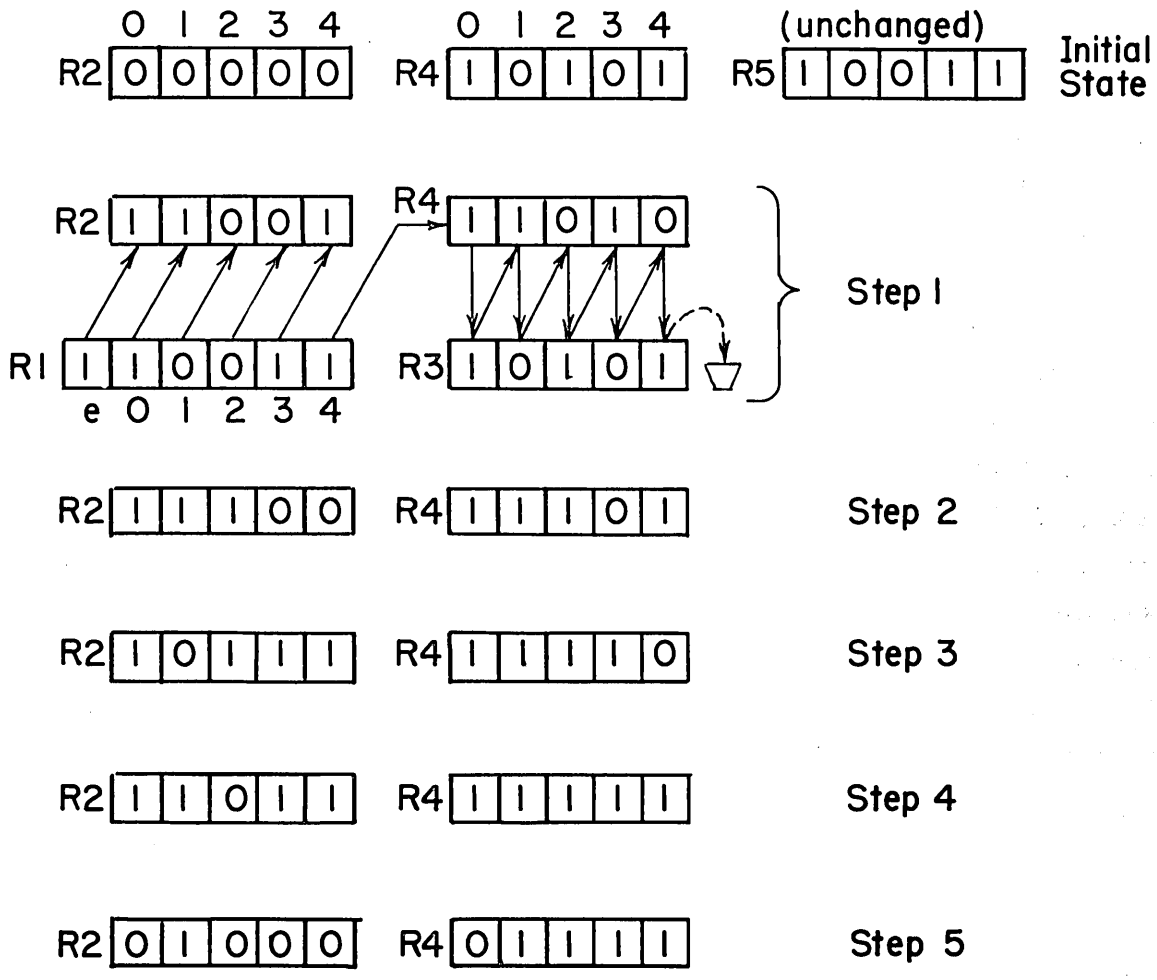
We conclude the discussion of the multiplication by an example with "nro". The particular problem is

$$(-\tfrac{13}{16}) \times (-\tfrac{11}{16}) = \tfrac{143}{256}$$

in binary form: $(-0.1101) \times (-0.1011) = (0.10001111)$

in complement form: $(1.0011) \times (1.0101) = (0.10001111)$

The first row of the sketch shows the initial configuration. In Step 1 we have included R1 and R3 to show their respective gate connections to R2 and R4. There is no connection from the adder to eR1; it is set to correspond to OR5. In the subsequent steps only the principals, R2 and R4, are shown. In the example we assume that the arithmetic unit has only 5 stages instead of the actual 40.



At the end of Step 4 the iterative procedure is completed, and the sign of the multiplier is by now at the extreme right flip-flop. Step 5 is a true addition of the complement of the multiplicand, inasmuch as the multiplier is negative. Simultaneously R4 is shifted to the right by one so that the right half of the final product is properly positioned. For reasons of uniformity OR4 is always set to 0 in this step, irrespective of the true sign of the product.

If the multiplication were rounded-off, the rightmost flip-flop of R2 would always be set to 1 and R4 would contain all 0's.

Division

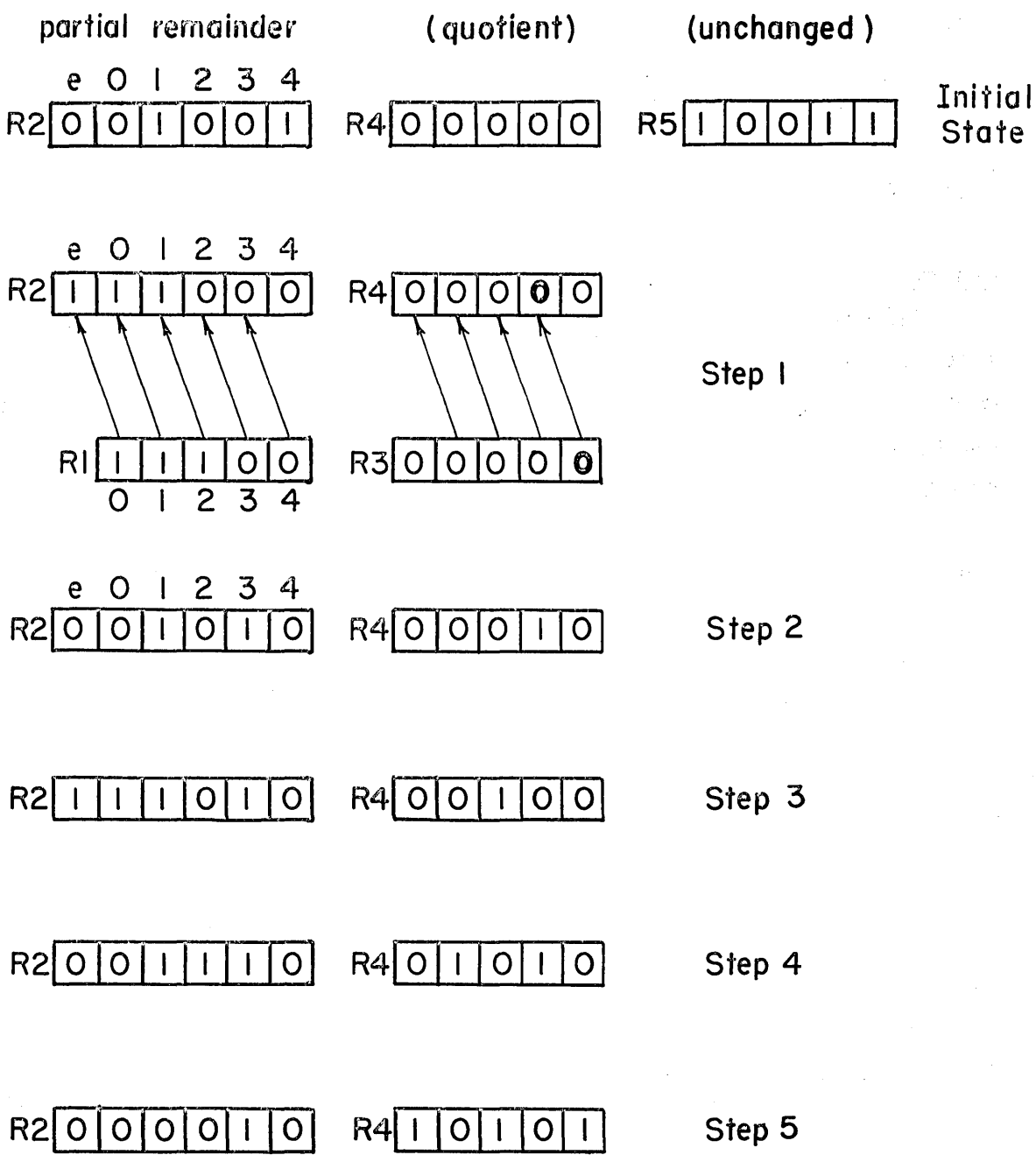
We now discuss the various steps of the division process. It will be recalled that we use a so-called "pseudo-non-restoring" type of division rather than the usual "restoring" form. It is assumed that the dividend is already in place in A as a result of a previous instruction or operation. The first step is to transfer the divisor from some memory location, specified by the address part of the divide instruction, to R5. The signs of the divisor and dividend are then compared. If they agree the complement of the divisor is added to the dividend; and accordingly a 1 is set into 39R4, the register which eventually contains the whole of the quotient. On the other hand, if the signs of the two terms differ, the divisor is added directly to the dividend, and 39R4 is left undisturbed. Inasmuch as R4 was cleared to 0's at the start, if 39R4 is left undisturbed this corresponds to the insertion of a 0. Q is then shifted one to the left. By virtue of the gate connections used here, in particular the fact that the transfer from R1 → R2 is diagonally left, the partial remainder appears in R2 already shifted to the left by one. The signs of the partial remainder and the divisor are again compared and 39R4 again set appropriately. This process is done 40 times. In this manner the pseudo-quotient is obtained. We have seen that the pseudo-quotient is simply related to the true quotient. Finally, the round-off is performed.

Inasmuch as the desired shift of the partial remainder is to the left, it is necessary to have an extra flip-flop precede 0R2 in order not to lose the sign of the partial remainder. It is designated as eR2. Further, along with the preparatory step of securing the divisor, it is necessary to set eR2 to agree with 0R2. At the completion of the operation, Q contains the rounded-off quotient and A has twice the remainder.

As an illustrative example, we consider a four-bigit division:

$$\underset{\text{Binary}}{0.1001/\text{-}0.1101} = (\tfrac{9}{16})/(\tfrac{13}{16}) = \underset{\text{Computer}}{0.1001/1.0011}$$

At the start of the process, R2 contains the dividend, R5 the divisor, and R4 is cleared to 0's. eR2 is made the same as 0R2, in this case 0. The first sign comparison of eR2 and 0R5 shows disagreement; hence the contents of R5 are sent to the adder directly, and a 0 is


partial remainder
(quotient)
(unchanged)
e 0 1 2 3 4
R2 0 0 1 0 0 1
R4 0 0 0 0 0
R5 1 0 0 1 1
Initial State
e 0 1 2 3 4
R2 1 1 1 0 0 0
R4 0 0 0 0 0
R1 1 1 1 0 0
R3 0 0 0 0 0
0 1 2 3 4
Step 1
e 0 1 2 3 4
R2 0 0 1 0 1 0
R4 0 0 0 1 0
Step 2
R2 1 1 1 0 1 0
R4 0 0 1 0 0
Step 3
R2 0 0 1 1 1 0
R4 0 1 0 1 0
Step 4
R2 0 0 0 0 1 0
R4 1 0 1 0 1
Step 5

set in the rightmost flip-flop of R4 and transferred, via R3, one place to the left. In the second sign comparison, the signs agree and the complement of the contents of R5 is sent to the adder directly and a 1 appears in R4, etc. At Step 5, R4 contains 10100. (The last stage is always 0 at the completion of intermediate steps.) The round-off procedure corresponds to setting the rightmost flip-flop to 1, and the quotient is 1.0101 (= -0.1011). Twice the remainder resides in R2 because of the shift occurring in each addition process.

Memory

The memory (internal and external) component of the computer provides the storage facility for numbers and instructions. The internal memory is electrostatic storage and the external memory is magnetic drum storage. In what follows reference to "memory" refers to the internal memory and reference to "drum" implies external memory.

The memory consists of 40 cathode ray tubes (crt), commercially available two inch tubes, type 2BP1. Each tube is mounted in a separate metal container, together with some associated electronic circuitry. The units have been designed so that they may be easily connected into the computer, or easily removed in case of malfunction and replaced by tested spares. The ensemble is located immediately above the arithmetic unit.

Each unit of the memory communicates with one, and only one, stage of the arithmetic unit; that is to say, the 40 units of the memory are connected in parallel with the 40 stages of the arithmetic component.

Each unit has a capacity of 1024 bigits. These are arranged in a 32 x 32 square array. If the various positions are numbered from 0——1023, clearly it requires 10 bigits ($2^{10} = 1024$) to specify a location or, as it is commonly called, an address. Once an address is specified, all units switch to the corresponding position in their square arrays, and communicate simultaneously to the arithmetic unit the corresponding bigits.

Data sent to the memory, either initially as input material or during the course of computation, must be continuously regenerated in order to be retained effectively. Indeed, the cathode ray tubes are continually regenerating the contained information unless interupted to go through an action cycle when the arithmetic unit asks for a new order pair or number, or else when the memory is to receive new information. After the interruption the memory returns to regeneration.

Without entering into a discussion of the theory of storage tubes, let us make a few simplified remarks on "writing" and "reading" of information in crt.

(i) Writing: the prescription for inserting a 0 at some location is to turn the beam on for a few microseconds. To write a 1, the beam is turned on for a few microseconds exactly as in writing a 0; but then the beam is displaced a few spot diameters and kept on a few microseconds longer in the new position. In either case, the procedure is independent of what conditions existed beforehand; in other words, there is nothing required that corresponds to erasing.

(ii) Reading: the beam is turned on for a few microseconds in the undisplaced position. If a 0 is residing there, there will be a small negative pulse on the pickup screen on the outside face of the tube. On the other hand, if a 1 were there the pulse on the pickup screen would be positive. These pulses are amplified and used to set flip-flops accordingly. We discuss this presently; however, it might be mentioned here that, in the event of a 0, the associated negative pulse turns the beam off before it is displaced; hence the 0 at that spot is not destroyed and is available for repeated consultations. The positive pulse does not turn the beam off until the beam is displaced; hence the 1 is intact also.

A very much simplified logical diagram of the memory system is shown in Figure 5.

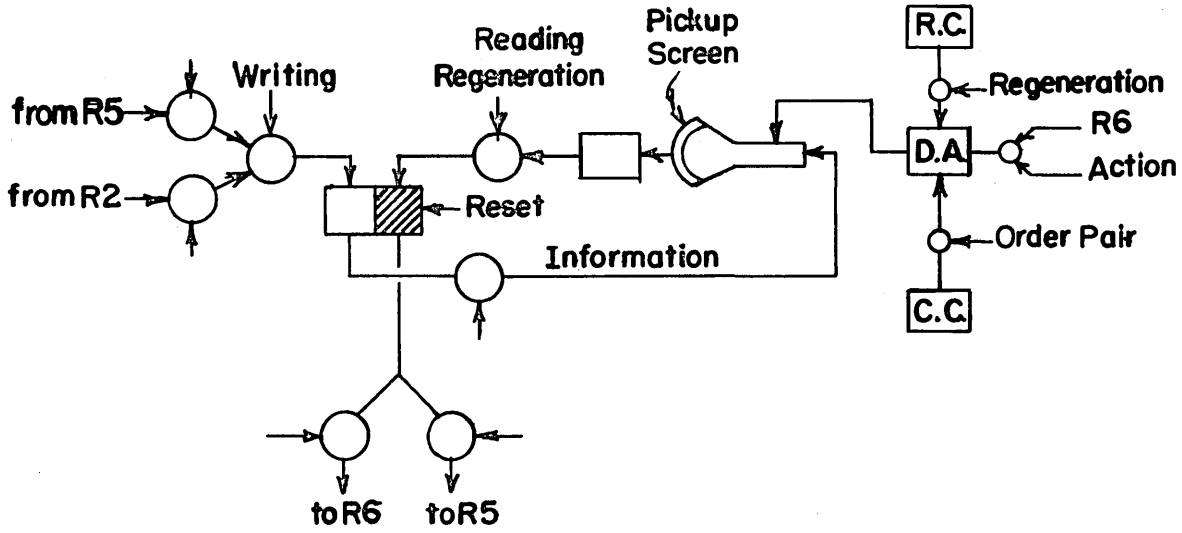


Fig. 5 Memory System. Abbreviations: D.A. deflection adder
R.C. regeneration counter
C.C. control counter

Only one of the 40 cathode ray tubes with its associated amplifier and flip-flop is shown. The deflection adder is a device that converts a 10-bigit number into a pair of voltages which are applied to the deflecting plates of the crt. There are three inputs (via gate tubes) into the deflection adder. Normally, the regeneration counter is sending its systematic addresses to it. When an action cycle is called for, the deflection adder receives an address either from the control counter or from R6 in preparation for activity at the location specified by them.

In a regeneration cycle, an address from the regeneration counter is sent to the deflection adder and there converted into a deflection voltage on the crt. The electron beam is then turned on to read the information at that spot. An amplified positive pulse from the pickup plate, corresponding to a 1, will set the flip-flop and allow the beam to stay on in its slightly displaced position; thus a 1 is rewritten in that spot. If the pulse is negative, the flip-flop is not set; the beam is turned off before it gets displaced; and a 0 is rewritten. In the meantime, the regeneration counter is advanced by one; the flip-flop is then reset; and the cycle is repeated for the succeeding spot. In this way, the complete pattern is continuously regenerated.

At some point in this process let us assume that an action cycle is demanded and that this action is to read a number from the memory to the arithmetic unit, into either R5 or R6. There is an interlock (not shown in the diagram) which allows the regeneration process to complete the present cycle; but in the next cycle, instead of gating an address to the deflection adder from the regeneration counter, the address is either taken from R6 or from the control counter, according as an order is being executed or a new order pair is being asked for. Reading proceeds and the flip-flop is either set to the 1 state or left undisturbed. The information, in addition to being sent back into the crt, is also gated into R5 or R6 as desired, by means of the gates shown in the diagram.

If the action cycle calls for writing into the memory, either from R2 or R5, the corresponding gates are opened and again the flip-flop is set or left undisturbed according as the bigit is 1 or 0. Here, too, the flip-flop controls the length of time the beam is on, hence whether

it is to "write" a 1 or 0.

There exists a variety of possible paths of communication between the various registers of the arithmetic unit and the memory. Obviously, R6 must be able to receive order pairs from the memory; it suffices that this connection is unilateral. R2 must be able to send to and receive from the memory; similarly, R4. Finally R5 needs to receive from the memory (for example, in multiplication). The scheme adopted is shown in Figures 6 and 7.

In the first are shown the gate connections from the memory. R6 connection is straightforward and requires no additional comments. A number from the memory is gated into R2 by first being gated into R5, from there to the adder, then to R1, and finally to R2; the last having been previously cleared or not as desired. R4 communicates with the memory via R5.

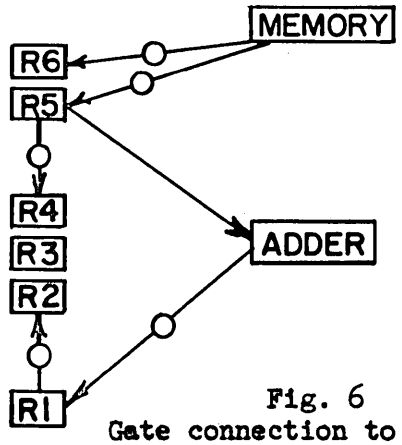


Fig. 6
Gate connection to the arithmetic unit from the memory.

The connections to the memory are shown in Figure 7. R2 and R5 communicate directly with the memory; R4 reaches the memory via R5. There exists a certain amount of flexibility in the gate connections from R2 and R5 to the memory. It is possible to send a composite word to the memory, one part being from R2 and the remainder from R5. This arrangement is useful in the substitution order where it is desired to change the address part of an order residing in the memory by an address at the moment in R2. This is executed by first bringing all of the word from the memory into R5, then sending all but the old address part back;, the new address being supplied from R2, where the appropriate set of 12 gates is opened. Use is also made of this flexibility of composition in the half-word substitution.

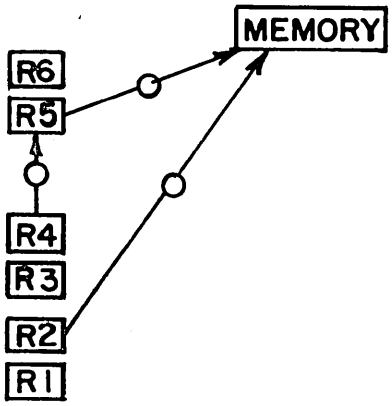


Fig. 7
Gate connections to the memory from the arithmetic unit.

The external memory is a magnetic drum system built for the computer by Engineering Research Associates, Inc., of St. Paul, Minnesota. The drum proper is a precision cylinder whose surface carries a magnetizable iron oxide. The cylinder is 8 1/2 inches in diameter and 15 inches in length. The drum cylinder is completely enclosed in a housing on which are mounted 202 magnetic heads for reading and writing information on the drum. When in operation with the computer, the drum is continuously rotating at 3450 rpm. The drum is mounted with the associated electronic gear in a 7 foot cabinet which is approximately 5 feet wide and 30 inches deep.

The drum has a capacity of 10,000 forty-bigit words. However, these words are not singly addressed and the communication between the drum and the memory is in blocks of fifty words. The addressing is done by 200 drum tracks where each contains fifty words arranged serially around the periphery of the cylinder. A separate magnetic head is associated with each drum track. There are 202 magnetic heads in all; two of these are for indexing purposes and the rest are concerned with the 200 storage tracks.

Due to peculiarities in the ERA logical design of the drum, the track addresses range from 0-255 with certain addresses being omitted. Table III shows the correspondence between the ordinal numbers and the actual track addresses. There are, however, routines in existence which allow one to address the drum tracks sequentially as addresses 0-C7 (0-199, decimally) in the process of coding. Since the communication with the drum is by tracks where any block of 50 words comes from a single track (one magnetic head), we observe that the drum is a serial storage system in contrast to the parallel storage of the memory.

It requires between four and five revolutions of the drum to read or write a track of words. The drum speed of 3450 rpm gives a drum period of 17 milliseconds, so that it requires between 68 to 85 milliseconds for 50 words to be read from, or written onto, the drum. This is, on the average, 78.5 milliseconds per 50 words.

The drum instructions each require a <u>full</u> word for their expression. The drum orders are:

0	0	19	20	32	40	4B	60	64	80	7D	A0	96	C0	AF	E0
1	1	1A	21	33	41	4C	61	65	81	7E	A1	97	C1	B0	E1
2	2	1B	22	34	42	4D	62	66	82	7F	A2	98	C2	B1	E2
3	3	1C	23	35	43	4E	63	67	83	80	A3	99	C3	B2	E3
4	4	1D	24	36	44	4F	64	68	84	81	A4	9A	C4	B3	E4
5	5	1E	25	37	45	50	65	69	85	82	A5	9B	C5	B4	E5
6	6	1F	26	38	46	51	66	6A	86	83	A6	9C	C6	B5	E6
7	7	20	27	39	47	52	67	6B	87	84	A7	9D	C7	B6	E7
8	8	21	28	3A	48	53	68	6C	88	85	A8	9E	C8	B7	E8
9	9	22	29	3B	49	54	69	6D	89	86	A9	9F	C9	B8	E9
A	A	23	2A	3C	4A	55	6A	6E	8A	87	AA	A0	CA	B9	EA
B	B	24	2B	3D	4B	56	6B	6F	8B	88	AB	A1	CB	BA	EB
C	C	25	2C	3E	4C	57	6C	70	8C	89	AC	A2	CC	BB	EC
D	D	26	2D	3F	4D	58	6D	71	8D	8A	AD	A3	CD	BC	ED
E	E	27	2E	40	4E	59	6E	72	8E	8B	AE	A4	CE	BD	EE
F	F	28	2F	41	4F	5A	6F	73	8F	8C	AF	A5	CF	BE	EF
10	10	29	30	42	50	5B	70	74	90	8D	B0	A6	D0	BF	F0
11	11	2A	31	43	51	5C	71	75	91	8E	B1	A7	D1	C0	F1
12	12	2B	32	44	52	5D	72	76	92	8F	B2	A8	D2	C1	F2
13	13	2C	33	45	54	5E	74	77	94	90	B4	A9	D3	C2	F3
14	14	2D	34	46	56	5F	76	78	95	91	B5	AA	D4	C3	F4
15	15	2E	35	47	58	60	78	79	98	92	B8	AB	D8	C4	F8
16	16	2F	36	48	5A	61	7A	7A	99	93	B9	AC	D9	C5	F9
17	17	30	37	49	5C	62	7C	7B	9C	94	BC	AD	DA	C6	FA
18	18	31	38	4A	5E	63	7E	7C	9D	95	BD	AE	DB	C7	FB

Table III

"m⟶D	BD	Read 50 successive words from the memory starting with the word at address specified by bigits 8-19 of the instruction. Write these 50 words into the drum on the track specified by bigits 20-27. Then transfer the control to the left-hand instruction of the word at the address specified by the bigits 28-39.
D⟶m	BC	Read the 50 words from the track of the drum specified by bigits 20-27 of the instruction. Write these words into 50 successive memory locations starting with the address specified by bigits 8-19. Then transfer the control to the left-hand instruction of the word at the address specified by bigits 28-39."

An example of a drum instruction in hexadecimal notation is

BD 137 29 2BF.

This is interpreted as: Read 50 words from the memory beginning with the word at address 137. Write these 50 words into the drum at track 29. Upon completion of the instruction the control transfers to the left-hand instruction of the word at address 2BF in the memory.

During a drum instruction R4 serves as a transition register between the parallel storage of the memory and the serial storage of the drum. That is, in transmitting to the drum each word is brought into R4 from the memory (parallel) and then shifted out of R4 to the drum (serial). In transmitting from the drum each word shifts into R4 (serial) and then is stored from R4 into the memory (parallel).

In order to transmit 50 words between the memory and the drum there must be a register or a counter which specifies the appropriate memory addresses. The control counter is used for this purpose. This means then that the control counter contains, at the completion of the transmission of the 50 words, the address of the 50th memory word concerned with the instruction. This, in general, is not the address of the next instruction word to be brought into R6; hence the drum instruction ends in a transfer which sets the control counter to the desired address for the next instruction word of the code sequence.

In the use of auxiliary equipment such as the drum, it is desirable to incorporate some sort of checking feature. The checking of the drum is by summing procedures similar to those used in loading. That is, when 50 words are transmitted from the memory to the drum, a sum of the words is formed and stored in an appropriate location. Upon transmitting this track of information back into the memory, a sum is again formed

and checked against the previously formed sum. It was initially intended that this summing be done entirely by programmed routine; however, it was observed that summing could be done electronically on the D⟶m instruction with practically no additional equipment; hence this feature was incorporated as follows: If in the D⟶m instruction one writes the initial memory address m as m + 800, the sum of the 50 words is accumulated in R2. R2 is not cleared to zero prior to the start of the summing; hence the sum is added into the contents of R2. At the completion of the instruction, the sum is left in R2 and may be checked with further programming. One still needs the summing routine for the m⟶D instruction.

Input-Output

The input component exists in two forms. There is the photo-electric paper tape reader and the magnetic tape unit. All input to the computer is initially via the photo-electric reader.

For input by the photo-electric reader, information on the paper tape is punched transversely in groups of four bigits, called tetrads. Usually a decimal digit or a logical character is represented by a single tetrad. For each separate decimal digit, the true binary representation is used where a punched hole corresponds to a 1 and a blank to a 0. Clearly, the true binary representation of a sequence of decimal digits is not given by the sequence of tetrads (cf. page 56). However, the conversion to the true binary number is quite simple and is done by the computer through a conversion routine before the actual computation starts.

We distinguish two methods of reading information from the paper tape into the memory. There is, first, an initial loading process which begins by setting the control counter to the desired initial address. The first word (10 tetrads) from the paper tape is transmitted by the reader into R5. The space symbol which terminates each word initiates the transfer of the word from R5 to the memory location specified by the control counter. The control counter is advanced one, the second word is read and transmitted to the second memory location, etc. The end of the loading process is indicated by the presence of two consecutive space symbols. The control counter resets to the initial address, the first order pair may then be brought into R6, and the problem started.

R5 has been made into a shifting register by making use of a short term memory facility afforded by a simple resistance-capacity circuit connected between each stage of R5. The speed of the photo-electric reader is sufficiently slow compared to electronic speeds that it is possible to scan the transverse series of holes of a tetrad and still have time to shift R5 four times per tetrad. In this way the parallel information in tetradic form is converted into a strictly serial pattern.

The use of R5 in association with the reader affords two desirable features. First, the functioning of the memory is divorced from that of the arithmetic unit so that, in the event of some malfunction, isolation of the difficulty is greatly facilitated. Second, since each word passes through R5 en route to the memory, it may be added into R2 so that during the loading process R2 acts as an accumulator of partial sums. At the completion of the loading the number residing in R2 is the sum of the contents of the tape, and it may be compared with a known correct value. This provides a useful preliminary check of the reader and associated electronics.

The second method of reading from the paper tape is, of course, the single read instruction which transfers a word (the next one in the series) from the tape to the memory location specified by the address part of that instruction. The use of this instruction in a small induction loop makes it possible to read whole blocks of words from the tape to the memory.

The magnetic tape unit serves as an input and output device. The magnetic tape drive is a standard audio-broadcast unit that was purchased from the Ampex Electric Corporation, San Carlos, California. The tape drive with our own associated electronic gear is mounted in a console cabinet of approximate dimensions 3 feet high by 2 feet wide by 2 feet deep. The unit is used as a single channel serial system where the magnetic tape reels contain 1200 feet of 1/4 inch wide Scotch Sound Recording Tape.

The reels of magnetic tape are, in general, premarked into sections which will accommodate 1024 forty-bigit words. There are fifteen such sections on a 1200 foot reel. The markings dividing these sections are short lengths made transparent by removing the magnetizable material from the tape.

Since the unit is used only as an input-output device, there is no automatic addressing of the fifteen marked sections, and there are only manual searching facilities.

The manual searching is afforded by a photo-cell hooked into the tape drive mechanism and a fast forward and reverse for driving the tape. The fast forward and reverse allows one to advance or reverse the tape at a speed of roughly four seconds per block of 1024 words. The photo-cell actuates a brake whenever a transparent length of tape passes in front of it. With this, one can then advance or reverse a tape as many blocks as desired.

The operating speed of the tape is 15 inches per second. The packing density of the tape is 72.6 zeros per inch, or 57.1 ones per inch, which is an average of 64.8 bigits per inch. The time required to record a memory load onto the tape is 40.9 seconds, if the information is all zeros, or 51 seconds if the information is all ones. This gives an average record time of 45.9 seconds per memory load.

The magnetic tape unit has no completely automatic load feature as does the reader; hence all information from the magnetic tape is read into the computer by a programmed routine. The tape order, reading from tape to R4, is:

"t⟶Q AC Replace the number in R4 by the first word to come under the reading head of the magnetic tape reader."

To insure accurate reading of data from the tape to the computer, a timing feature must be incorporated in the writing process, i.e., in the computer to tape routine. This feature is a time delay between the transmission of successive words from the computer to the tape, and it is accomplished by an L(40) instruction given prior to each Q⟶t instruction. This delay in recording on the magnetic tape gives adequate spacing between words to insure proper transmittal by the tape "call" routine which does not include the L(40) delay.

As in the drum, a checking feature has been incorporated into the magnetic tape routines by summing. In the computer to tape routine, the words sent to the tape are summed. The sum is printed and recorded on the tape as the last word of the record. Upon "calling" the information back into the computer via the tape to computer routine, the contents of the tape are summed except for the last word. The sum is then compared with the last word of the record; the last word being the sum formed when the record was made.

Output from the computer may be accomplished by four mechanisms. There is the magnetic tape already discussed, the Synchroprinter, a high-speed page printer; the Flexoprinter, a slow-speed page printer; and the Flexopunch, a slow-speed tape punch. No further comments are needed for the magnetic tape unit; hence we turn to the printers and punch.

The Synchroprinter is a high-speed page printer that was purchased from the ANelex Corporation, Concord, New Hampshire. The Synchroprinter and its associated electronic gear are mounted in a cabinet of approximate dimensions 5 feet 6 inches high by 1 foot 10 inches wide by 1 foot 7 inches deep. The printer has a maximum operating speed of fifteen lines per second which is 36,000 characters per minute.

The characters that may be printed are the ordinal numbers 0,1,2 ··· 8,9; the letters A,B ··· F; a decimal point; and a minus sign. The printer achieves its speed by printing a line at a time where a line consists of 40 characters; these may be four 10-digit numbers or any other aggregate. The printer operates on the following principle: There are 40 type wheels, each containing the 18 available characters. The 40 wheels are rigidly mounted on a metal cylinder. All of the 0's, 1's, 2's, etc., of the 40 wheels are aligned. This cylinder rotates at a constant speed whether the printer is being actuated or not. During any one revolution of the cylinder a line may be printed. In printing an aggregate of 40 characters all of the 0's of the aggregate are printed simultaneously, then the 1's, the 2's, and so on, until after one revolution of the type cylinder the 40 characters of the line are printed.

There are two apparent methods of operating such a printer. The first is to supply the correct digital information to all 40 type wheels simultaneously and then allow each wheel to print at the proper time. As is known, a 40-bigit register may represent only 10 coded-decimal or hexadecimal characters; hence to represent 40 such characters, four standard registers would be needed. Although this method is very simple from a coding viewpoint the electronic gear involved makes such a scheme prohibitive.

The second method and the one adopted for the printer involves very little additional electronic equipment. Inasmuch as the 0's of a line are printed simultaneously and then the 1's, the 2's, and so on,

only the 0's digital information needs to be supplied to the appropriate type wheels when the 0's are to be printed, and similarly for the remaining digits. During the 0 print cycle the information that needs to be supplied to each type wheel is binary, i.e., either print or do not print. Since a register contains 40 bigits, and since a line for the printer is 40 characters, a register may supply the necessary binary information to the print wheels. The register R2 is used for this purpose.

To print an aggregate of 40 digits, the 40 digits are first represented by an 18-row, 40-column matrix (i.e., 18 consecutive memory locations) where the rows represent the 18 characters present on a print wheel, and the columns correspond to the digit position in the aggregate. For electronic convenience a 0 in any element corresponds to the presence of a digit and a 1 corresponds to the absence of that digit. As an example, consider a 4-row, 6-column matrix where the number 302132 is represented. It is:

0: 1 0 1 1 1 1
1: 1 1 1 0 1 1
2: 1 1 0 1 1 0
3: 0 1 1 1 0 1

where rows correspond to the digits 0 → 3 in order from top to bottom, and the leftmost column corresponds to the most significant digit position. To represent an 18 x 40 array or matrix in the computer 18 words of storage are required. After such an array has been formed a line may be printed. Row 0 is brought into R2 for the 0 print cycle, row 1 for the 1 print cycle, row 2 for the 2 print cycle, and so on.

A timing problem is involved, as only about 1.5 milliseconds exist between adjoining print cycles once the printer is actuated. The print order itself acts as a timing element. To print a line 18 print orders are given as part of a subroutine. The first of the 18 actuates the printer and the rest act in a timing capacity. It is necessary that the time elapsing between successive print orders be less than 1.5 milliseconds, and for safety it is recommended that the time be kept somewhat less. When each print order is given the appropriate row of the matrix must be in R2.

Although the described scheme complicates the print subroutine it is felt that the reliability obtained by including no new electronic gear certainly justifies the added complications of the coding.

The matrix is formed in the computer so that the first row corresponds to the minus sign, the second row to the decimal point, the third row to digit 0, the fourth row to digit 1 ••• the 17th row to the letter E, and the 18th row to the letter F. The type is arranged on the print cylinder so that the sequence of printing the characters is F, E, D, C ••• 3, 2, 1, 0, ·, -. This means that the words corresponding to the rows of the matrix must be brought into R2 beginning with row 18 (the letter F) and ending with row 1 (the minus sign).

The paper feed for the printer operates from top to bottom past the print cylinder. The first line printed then appears at the bottom of a column of lines. In order to have the first line printed appear at the top of a column of lines (as it customarily does) the type characters on the wheels have been inverted. If the mirror image of a 40-digit aggregate is then printed it comes out of the printer inverted, but upon turning the copy upright one has a conventional listing which for a column of lines would read from top to bottom and from left to right. To print a mirror image of the aggregate the order of the columns of the array is reversed; i.e., the rightmost column corresponds to the most significant digit and the leftmost column to the least significant digit. The 4 x 6 matrix of the previous example for the number 302132 should be formed as

1 1 1 1 0 1
1 1 0 1 1 1
0 1 1 0 1 1
1 0 1 1 1 0

The print order is:

Sync Print CE To be used in a subroutine which prints simultaneously $\underline{m}_i$, $\underline{m}_{i+1}$, $\underline{m}_{i+2}$, $\underline{m}_{i+3}$; $\underline{i}$ must be supplied to the routine.

The address bigits of the print instruction have no relevance with respect to the instruction.

An example of a Synchroprint routine is given as Problem 13 of Chapter II. There is, in addition to the high-speed printer, a modified Teletype page printer that has an operating speed of 396 characters (36 10-digit words and spaces) per minute. The printer is

modified to 16 characters; the ordinal numbers 0,1,2 ··· 9, and the letters A,B,C ··· F. This printer is actuated by the print order "Flexoprint EA Print m on the page printer (slow speed)."

The reason for retaining this printer in addition to the Synchroprinter is that one may print directly any word in the memory. To print a word via the Synchroprinter involves a routine, while the Teletype printer needs only an instruction. Whenever any volume of printing is desired, however, the faster Synchroprinter is used.

The Flexowriter punch allows one to punch information from the computer onto paper tape for subsequent use. The punch is a modified Flexowriter punch for five hole paper tape. Its speed of operation is 869 characters (79 10-digit words and accompanying spaces) per minute.

The punch order is:

"Punch CF Punch m on paper tape."

Due to the very slow speed of the punch, the magnetic tape is used whenever practicable for output needed in a form to be used as subsequent input.

Control

The control is the agency which directs the various activities of the computer. Some parts of the control relate specifically to the detail operation of the various components, such as the memory control concerned with the regeneration of stored information. To some extent these have been discussed under the respective headings in previous sections. Here we propose to consider some of the more general features of the control.

The instructions for the computer are of the one-address type; i.e., an order is associated with a single address referring to some memory location that contains a number upon which the specific order is to operate. This system of instructions is much simpler in structure than some proposed schemes for other computers. There have been proposals for four-address instructions; the first two addresses specifying the two factors of an operation (say in multiplication, the multiplicand and multiplier), the third referring to the destination for the result, and the last to the location in the memory of the next instruction. We do not cite the various advantages for the several proposals except to remark that simplicity is a rather compelling argument.

The normal word length in the memory is 40 bigits. An instruction is 20 bigits, so that instructions are stored in pairs. Of the 20 bigits, 8 are used for specifying the order, and 12 remain for the address. Actually 10 suffice with our present memory capacity of 1024 (= 2^{10}), so that 2 bigits are available for future expansion or for some other purpose.

The 8 bigits describing an order are initially punched onto a paper tape as two tetrads. In principle any of the 16 possibilities 0,1,2...9, A,B...F might be used for each tetrad. Thus a maximum of 256 possibilities is available. Our present feeling is that the number of useful orders will not exceed 36; thus only letters in pairs are used to designate an order. This is useful in coding.

Let us begin at some point in the cycle of activity and describe the sequence of events that leads back to the same point; after that we indicate with the aid of some logical diagrams how some of these things are accomplished.

Assume that a pair of orders has just been brought into R6. The order part of the left-hand instruction must be interpreted and the corresponding sequence of pulses and voltage changes provided. At the same time the address part is sent to the deflection adder of the memory in preparation for communication with the memory. When this instruction is completed, the control then examines the instruction residing in the right half of R6 and takes the necessary measures to execute it. In the meantime, the control counter is advanced by one so that when the right hand instruction is completed the next order pair can be brought to R6, and thus complete the cycle.

It is convenient to subdivide this part of the control into three sections: The first is concerned with the interpretation of the eight bigits as a specific order, and is called the order matrix. The second, called the operations control, provides a set of pulses for executing a given order. The third, the instruction control, deals with the "red tape" associated with doing the left half of an order pair, then the right half, and then seeking a new order pair.

The Order Matrix: Inasmuch as it has been decided to use only letters (and not include decimal digits) to specify orders, each tetrad of a pair begins with a 1 (letters correspond to the digits 10-15). Therefore, of the eight digits, only six are used to discriminate among

the various orders. To simplify the discussion, assume we are concerned with only two bigits. (The case for six is an obvious extension.) These two bigits are in two flip-flops of R6; and imagine further that in each flip-flop two wires tap in at symmetrical points of the flip-flop as shown diagrammatically in Figure 8. If I has a 0, A has a definite voltage V, and B has another definite voltage V'; if I has a 1, the voltages are interchanged, that on A is V' and on B it is V. The voltages on C and D depend on the contents of II in precisely the same way.

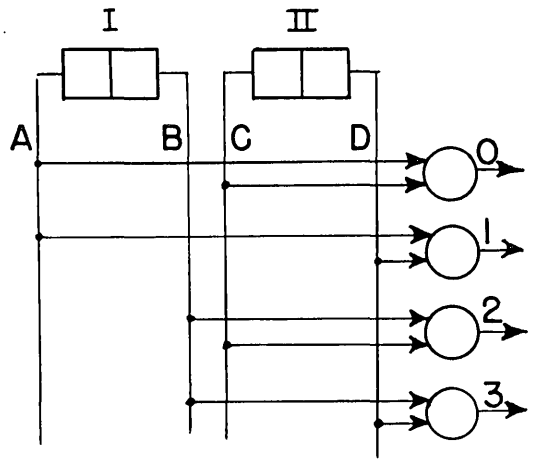


Fig. 8 A two stage order matrix.

Consider next a two level "and-gate" with the following properties: If, and only if, the input voltages are both V, a signal is given to the output. We now construct four such "and-gates" with inputs from the set A,B,C,D; the specific connections are shown in the diagram. Clearly, if the contents of I and II are 0,0 the above condition is satisfied for only the topmost gate and a pulse is given out along the 0 output. Similarly, if the contents are 0,1 a pulse goes out along the 1 output, etc. To envisage the actual order matrix, imagine that there are 6 flip-flops with various connections to 36 "and-gates" of level six; i.e., six conditions must be satisfied to stimulate an output. Thus from a series of bigits we actuate a unique line corresponding to that particular set.

The Operations Control: The operations control is essentially a pulse generator producing a sequence of seven pulses. Four of these pulses are of fixed length; the remaining three may be variable. The necessity for pulses of variable duration stems from the fact that the time required for certain operations is somewhat indeterminate. For example, if an action cycle is required of the memory at some moment, it is necessary to wait until the memory completes its present regeneration cycle before going into action. Inasmuch as the waiting period is somewhat arbitrary, the time from the instant the action cycle is requested to completion is slightly indefinite. The comple-

tion of the operation terminates the pulse and the operations control then generates the next pulse.

Some of the more complex orders require more than just one sequence of such pulses; hence one of the provisions made is to permit the operations control to go through its paces the required number of cycles. On the other hand, some of the simpler orders do not need the full complement of seven pulses and, in the interest of speed, provision is made to terminate the sequence at some earlier point.

We now consider a very much simplified example of an order, by way of illustrating how an actuated line from the output of the order matrix and the signals from the operations control combine to execute the given order. Say the order is a shift to the left by one place of a number in R2. A series of "and-gates" of level two are connected to the output line from the order matrix that corresponds to this order. The output line is thus a common static input to all of these gates. The second inputs are the various timed pulses from the operations control. These connections are shown in Figure 9.

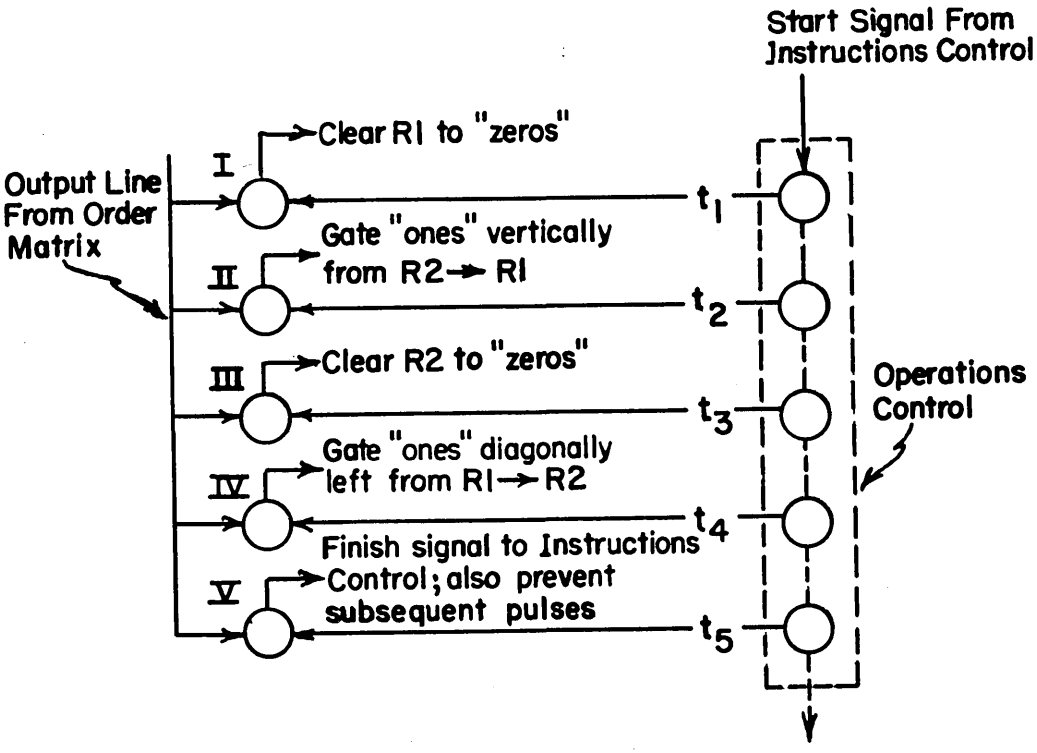


Fig. 9 Gate connections for a simplified order.

When the first signal t_1 is produced, conditions at gate I are satisfied and an output signal is produced and is sent to the clear bus of R1. Its effect is to set all the flip-flops of R1 to the 0 state. After a short delay, pulse t_2 is produced and directed to gate II. This output sets those flip-flops of R1 to 1 to match the corresponding flip-flops of R2 or, simply said, 1's are gated into R1 from R2 vertically. The subsequent steps are obvious.

The Instruction Control: It includes the following functions:

(i) Communication with the memory to obtain the next order pair. Signals must be given to clear R6, to send the address from the control counter to the deflection adder, and to transfer the order pair from the memory to R6.

(ii) Transfer of the order part of the left instruction to the order matrix and of the address to the deflection adder of the memory; upon completion to examine the instruction in the right half of R6.

(iii) Sending a start signal to the operations control.

(iv) In the event that the left order is a transfer order, the sequence is interrupted, the new order pair is brought into R6, and a new sequence of instructions is started. There is also provision to skip the left order for those cases where the transfer is to begin a new sequence of instructions with the right half of an order pair.

(v) Finally, it must advance the control counter by one after each order pair, and also receive the finish signal from the operations control.

In order to make convenient gate connections between the various functions of the control, a collection of vertical bus wires is accessible in the control panel immediately to the left of the registers. A cross-sectional layout of the arrangement is shown in Figure 11. The notation is as follows:

$C_i RJ$	clear RJ (J=1,2,...6) to i (i=0,1);
t_n	n^{th} timed signal (n=1,2,...6);
$RJRJ' \begin{cases} L_i \\ R_i \\ S_i \end{cases}$	gate i from RJ → RJ' either $\begin{cases} \text{left diagonally;} \\ \text{right " ;} \\ \text{straight;} \end{cases}$

Hold	allows variation in length of t_2 and/or t_3;
Finish	finish signal from operations control to instruction control;
Set Trans FF	sets a flip-flop in instruction control to indicate transfer to new sequence of instructions;
Set Rt Trans FF	sets a flip-flop in instruction control to indicate transfer to new sequence beginning with right half;
Cycle Input	input to operations control to repeat sequence of timed signals;
Start Toggle $\underline{0}$	a special timed signal which permits cycling operations control twice in a given order.

Vertical Buses of Order Gates

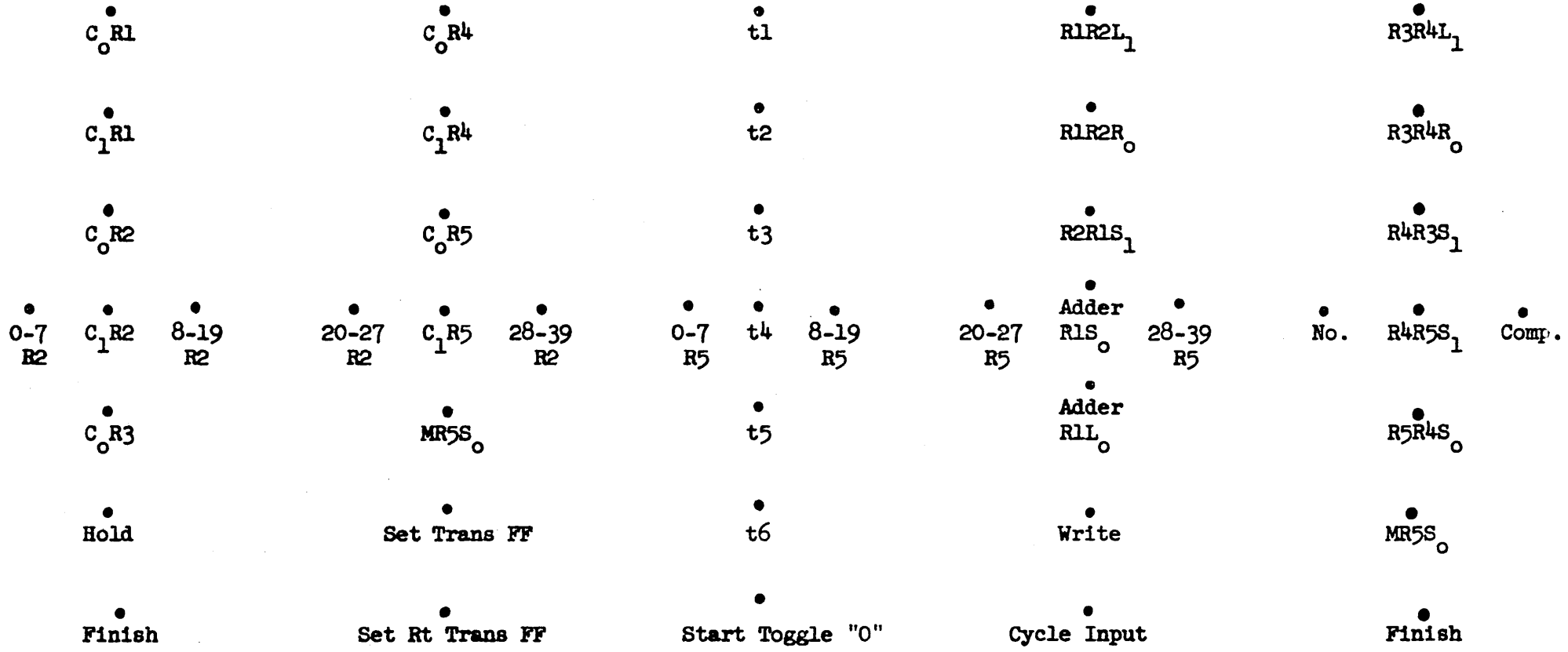


Figure 11

V. DESCRIPTIVE CODING AND SUBROUTINES

Recall from Chapter II that the steps in the preparation of a code of a problem are:

1. The logical coding is first prepared. In this coding the logical rather than the computer symbols are used. Each box of the flow diagram is treated independently and the instructions within the box are numbered consecutively beginning with 1. Indexed Latin letters are used to indicate the addresses of the necessary storage of the problem.
2. The computer code is then prepared. In this coding the instructions are paired into words and these instruction words are sequenced and numbered (addressed) according to their subsequent residence in the memory. The computer symbols for the orders are written in place of the logical symbols. Numerical addresses are assigned to the storage, and the addresses of instructions referring to storage are modified accordingly.
3. The computer code is checked so that any errors may be corrected before the code is punched onto paper tape for subsequent input to the computer.

As one examines these steps in detail, the question quite naturally arises as to whether the computer might be instructed to carry out part of the coding process. The question can be answered in the affirmative, and the purpose here is to describe a method for coding in which the computer is instructed to carry out all of Step 2 of the coding procedure.

The method is by no means unique. The motivation for its choice is found in the desire to use the computer as an aid in constructing a usable code which is tailored in the manner described in Chapter II, and to relieve the person preparing the code of much of the routine work involved, and possibly to reduce the number of errors.

The method in general is as follows: A logical code using a prescribed set of symbols and following a prescribed set of rules is prepared. These symbols identify the various kinds of storage of the problem (e.g., numerical constants or logical quantities) and the addresses of the various instructions of the problem. This logical code is checked for errors and after any needed corrections are inserted, a punched tape of

this logical code is prepared. This tape is then used as input data by a routine designed to assemble a computer code from this material.

The assembly routine reads the individual instructions from the logical code tape and pairs these instructions properly into instruction words; assigns addresses to these instruction words; and stores them into the proper location. The absolute (numerical) addresses of the storage of the problem are assigned by the assembly routine, and the instructions referring to this storage have their addresses translated accordingly. The addresses of instructions that do not refer to storage (i.e., instructions that refer to other instructions) are also translated into their absolute value. When this computer code is completely assembled it is punched onto paper tape or written onto magnetic tape by the assembly routine; a printed copy is also produced.

This method of coding has been given the name descriptive coding since many of the identifying symbols used in the logical coding are descriptive in nature.[1]

We now turn to the discussion of the descriptive coding, and we establish the necessary rules and define the symbols needed to carry out such a coding. The assembly routine is not discussed in detail since its complexities are beyond the scope of a manual of this type.

In the preparation of any code which is to be modified and assembled through an assembly routine, the flexibility of the coding (i.e., the freedom of choice of symbols and the amount and different kinds of information which can be specified in a descriptive instruction) is dependent upon the number of bigits that are allowed to express each instruction. Clearly the more bigits allowed, the greater is the flexibility.

It was found that the normal instruction length of twenty bigits was adequate to achieve a code by means of such an assembly routine, which was comparable to a tailored code both in number of words of code and subsequent running time of the problem. The first two tetrads of the twenty bigits specify the order using the standard vocabulary symbols; the remaining three tetrads are for the address. There are two advantages in having the descriptive instructions conform as much as possible

---

[1] The method was developed by Eugene H. Herbst, John B. Jackson, and Mark B. Wells, of the Los Alamos Electronic Computer Group.

to the familiar logical instructions. First, by remaining within the framework of the logical coding, a relatively small number of new symbols and new rules for coding need to be introduced. Second, the work in preparing the descriptive code is no greater than the usual logical code and the labor of the tape preparation for the descriptive code is comparable to that involved in the preparation of the tape for a tailored computer code.

The descriptive coding is prepared from a flow diagram. No modifications of the flow diagram are necessary and it is as discussed in Chapter II.

In the descriptive coding (as in the usual logical coding) each box of the flow diagram is coded as though it were independent of the remainder of the diagram. The only interdependence of boxes of coding is through transfer and substitution instructions. These are discussed presently. The instructions written for each operation box are numbered consecutively, starting with 1, and the numbering is done hexadecimally, as shown in Example 1.

Example 1

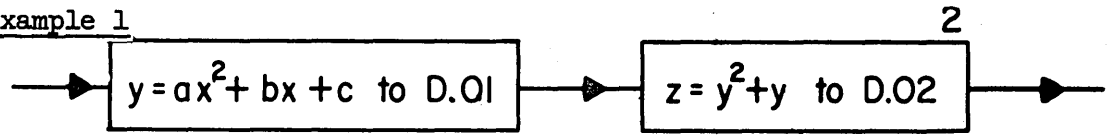


Figure 1

Storage chart: B.01: x C.01: a D.01: y
C.02: b D.02: z
C.03: c

The coding is:

Box 1

1.	m⟶Q	C.01	a to R4	
2.	X	B.01	ax in R2	
3.	m⟶Ah	C.02	ax + b in R2	
4.	L(40)	028	ax + b in R4	
5.	X	B.01	ax^2 + bx in R2	
6.	m⟶Ah	C.03	y = ax^2 + bx + c in R2	
7.	A⟶m	D.01		y to D.01

Box 2

1.	m⟶Q	D.01	y to R4
2.	X	D.01	y^2 in R2
3.	m⟶Ah	D.01	$z = y^2 + y$ in R2
4.	A⟶m	D.02	z to D.02

The addresses that can occur in instructions must be classified and a set of symbols may be used to represent each class so that the assembly routine may interpret and modify the various addresses correctly. Addresses of instructions fall into four general classes. They are:

(i) Addresses that refer to numerical storage.

(ii) Addresses that do not play a normal address role, as in R(n), L(n), a⟶Ac, and a⟶Ah instructions.

(iii) Address that refer to instructions within the same operation box.

(iv) Addresses that refer to instructions in other operation boxes.

Each class may be divided into as many subclasses as is deemed necessary. Let us examine each class of addresses.

Recall that there are two kinds of storage requirements for a problem, static storage and dynamic storage. The static storage is that storage which originates with the problem and remains unmodified throughout the course of the computation. The dynamic storage is that storage which originates from computation within the problem.

For simplicity of addressing, the static storage has been assigned the four symbols:

B.i
7.i
C.i
A.i

i (= 1, 2 ••• FF)

255 words may be stored on each set of addresses. The sets have the following significance. B storage is that static storage which originates with the problem as Binary numbers; hence, any constants which are given in a problem as binary numbers are referred to by B.i addresses, and are listed sequentially as B.i storage. 7 storage is very similar to B storage in that the numbers to be stored in 7.i storage are also given in binary form. The 7 storage has significance

with respect to subroutines, and it is discussed more appropriately in the section on subroutines. The letter C designates static storage that is to originate with the problem as decimal numbers and is to be Converted to binary numbers by the assembly routine. The letter A designates the static storage that contains Addresses (numbers corresponding to addresses) which are to be used by substitution instructions in modifying other instructions during the course of the computation.

The symbol D.i i (= 1, 2•••FF)

is used for Dynamic storage and 255 words of D storage are **allowed.**

We now examine more closely the storage requirements of Example 1. We may assume that the number x is given as a binary number; therefore it is placed in B storage and indicated as

B.01: x

The constants, a, b, and c, are assumed to be numbers which are originally given as decimal numbers and which are to be converted to binary numbers by the computer during the process of preparing the code through the assembly routine. a, b, and c are listed in C storage as

C.01: a
C.02: b
C.03: c

The dynamic storage consists of storage for the quantities y and z which are formed during the computation; hence two dynamic storage locations are needed, and

D.01: y
D.02: z

The second class of addresses, those that do not play a normal address role, have the proper numerical address inserted in the descriptive code; e.g., Box 1, Instruction 4, reads

4. L(40) 028

where 028 is the correct hexadecimal address for a left shift of forty places. As a further illustration consider the use of an a⟶Ac instruction to bring 2^{-1} into R2. The instruction reads

a⟶Ac 400

where 400 corresponds to 2^{-1} when brought into R2. If for any reason it is desirable to insert an instruction which contains an absolute address, such an address should be used in the descriptive coding (except

in transfer and substitution instructions) and the assembly routine will not alter it; e.g., the instruction Q⟶A (m⟶A 800) has its special address 800 inserted in the descriptive coding.

The third class of addresses, those addresses of instructions that refer to other instructions Enclosed within the same operation box, are designated by the symbol E. Such an address

E.i i (= 1, 2 ... FF)

may range over 255 instructions of an operation box. This is a partial restriction on the number of instructions in an operation box. Although an operation box may have more than 255 instructions, no instruction may refer to any instruction beyond number 255 of the same operation box. The E.i address is used primarily in substitution instructions. Such an address has special use with other instructions. In fact, we shall see in the discussion of subroutines that the E.i address is used in transfer instructions. The following example illustrates the use of E.i addresses.

Example 2

The flow diagram of Example 2 shows only that portion of an induction loop in which the sequence of quantities z_i (i = 0,1•••I-1) are formed and stored in the memory at addresses D.20+i hence

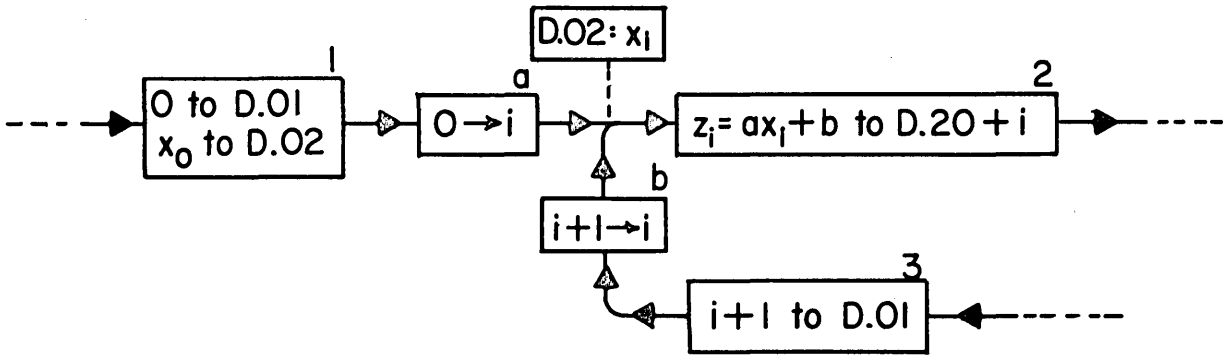


Figure 2

A.01: AAD20AAD20

B.01: 0
B.02: (1)$_o$
B.03: x_o

C.01: a
C.02: b

D.01: i
D.02: x_i
⋮
D.20: z_o
D.21: z_1
⋮
D.20+i: z_i

The coding is:

Box 1.

1.	m⟶Q	B.01	0 to R4	
2.	Q⟶m	D.01		0⟶i to D.01
3.	m⟶Q	B.03	x_o to R4	
4.	Q⟶m	D.02		$x_o \longrightarrow x_i$ to D.02

Box 2.

1.	m⟶Ac	A.01	AAD20AAD20 to R2	
2.	m⟶Ah	D.01	AAD20+iAAD20+i in R2	
3.	S⟶m	E.07		D.20+i to address of Instruction 7
4.	m⟶Q	D.02	x_i to R4	
5.	X	C.01	ax_i in R2	
6.	m⟶Ah	C.02	$z_i = ax_i + b$ in R2	
7.	A⟶m	[D.20+i]		z_i to D.20+i

Box 3.

1.	m⟶Ac	D.01	$(i)_o$ to R2	
2.	m⟶Ah	B.02	$(i+1)_o$ in R2	
3.	A⟶m	D.01		$(i+1)_o \longrightarrow (i)_o$ to D.01
4.	T	02,1		

In the storage required, the numbers 0, $(1)_o$, and x_o are originally stored as binary numbers; hence B storage is used. The numbers a and b are decimal numbers to be converted into binary numbers by the assembly routine; consequently they are stored in C storage. $(i)_o$ and x_i are stored in dynamic, D storage. We assume after the initial traversal that x_i is sent to D.02 from a portion of the routine not shown. The choice of D.20 as the starting address for the z_i is arbitrary, and any block of I locations would suffice for that D storage.

The A storage is used to store the initial address D.20 from which all addresses D.20+i are formed (Instructions 1 and 2, Box 2). Note that D.20 is stored in A.01 as

A.01: AAD20AAD20.

It is stored as an instruction-word where the two instructions are identical. This is true in general: that all A storage is stored as instruction-words where the two instructions of the word are identical and the address of the instructions is the desired descriptive address. The choice of the order that appears in the instruction word depends on the use of the particular word of A storage. The choice of the order AA in this instance is significant in that the assembly routine deletes the AA from each instruction at the time the D.20 is assigned its absolute value. For example, suppose that the absolute address corresponding to D.20 is 154. The A storage before and after modification by the assembly routine is:

A.01: AAD20AAD20 A.01: 0015400154

The order AA is the only order that is deleted from A storage when the storage is modified.

In the coding of Example 2, the first two instructions of Box 2 form $(D.20+i)_o$ in R2. Instruction 3 reads

S⟶m E.07

Hence, the address of Instruction 7 is replaced by the number in R2 which is D.20+i. Note that the order S⟶m is used rather than S⟶m'. This is always the case, not only for S⟶m but also for T, C, and HS⟶m. All transfer and substitution instructions whose addresses refer to other instructions are coded as the unprimed order; that is, the order that refers to a left-hand instruction of an instruction-word. The assembly routine then modifies the order if a modification is necessary.

The fourth class, those addresses of instructions that refer to instructions in other operation boxes, are addresses of transfer instructions and substitution instructions. Transfer instructions and substitution instructions are the only instructions whose addresses may refer to instructions of other operation boxes than the one containing the instruction.

Transfer instructions act in two ways as connecting links between operation boxes. These are the fixed connection and the variable remote connection. We treat the fixed connections first.

A transfer instruction that is a fixed connection has as its address the operation box number and the instruction number of that box

into which the transfer is to send the control. The first two of the three address tetrads are used for the operation box number. The remaining tetrad is used to specify the instruction number within the box. As an illustration, Instruction 4, Box 2, of Example 2, reads

4. T 02,1

which is a transfer of the control to Box 2, Instruction 1.

Recall that on a flow diagram the flow lines enter at the beginning of a box. If the coding strictly followed the flow diagram, a transfer instruction would always be to the first instruction of an operation box. However, it has been shown in previous codings that it is often possible to save an instruction or two by transferring the control into one of the first few instructions of a box or one of the last few instructions of the preceding box (cf. Page 72, Problem 6, Box 6, Instruction 1).

A transfer can refer to any one of the first seven instructions of the operation box to which the transfer is effected, or it can refer to any one of the last seven instructions of the preceding box. The operation box number specified in the address of a transfer instruction is the box of the flow diagram which is entered by the flow line indicating the transfer. A number 1, 2, ··· 7 in the third address tetrad indicates a transfer into the corresponding instruction of the box. A number F(=-1), E(=-2), D(=-3) ··· 9(=-7) indicates a transfer into the corresponding instruction of the preceding box; e.g.,

CA20,3(T 20,3) reads: Transfer the control to Operation Box 20, Instruction 3.

CA25,E(T 25,E) reads: Transfer the control to Operation Box 25, Instruction -2, which is the next to last instruction of the preceding box. The preceding box is not necessarily Box 24.

The address of a conditional transfer instruction, where the (+) exit is a fixed connection, is formed in the same manner as the address of a transfer instruction.

Example 3 illustrates transfer instructions acting as fixed connectors.

Example 3

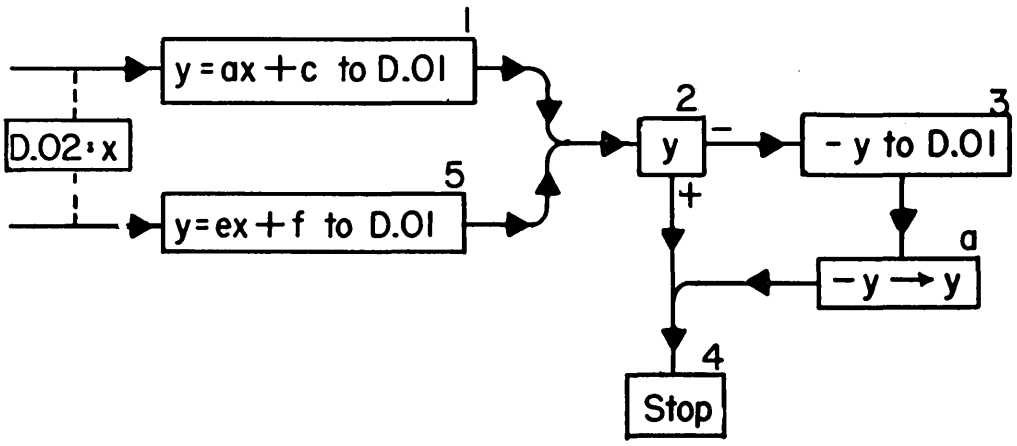


Figure 3

C.01: a D.01: y
C.02: c D.02: x
C.03: e
C.04: f

We assume that x is formed in a part of the routine not shown and is stored in D.02. **The coding is:**

Box 1.

1.	**m⟶Q**	**C.01**	**a to R4**	
2.	**X**	**D.02**	**ax in R2**	
3.	**m⟶Ah**	**C.02**	**y = ax + c in R2**	
4.	**A⟶m**	**D.01**		**y to D.01**

Box 2.

1.	m⟶Ac	D.01	**y to R2**
2.	C	04,1	

Box 3.

1.	m⟶Ac-	D.01	**-y to R2**	
2.	A⟶m	D.01		**-y to D.01**

Box 4.

1. Stop

Box 5.

1.	m⟶Q	C.03	e to R4
2.	X	D.02	ex in R2
3.	m⟶Ah	C.04	**y = ex + f in R2**
4.	T	02,F	

The conditional transfer instruction of Box 2 reads C 04,1 which is a conditional transfer to Box 4, Instruction 1. The transfer instruction of Box 5 reads T 02,F which is a transfer to Box 2, Instruction -1. This is a transfer to the last instruction of the preceding box, in this case Box 1.

Substitution instructions may also have an address consisting of an operation box number and an instruction number. However, the substitution instructions can modify any one of the <u>first</u> fifteen instructions of any operation box other than the box containing the substitution instruction. Note that this treatment differs from the transfer instructions.

Recall on a flow diagram that a set of variable remote connections is indicated by a Greek letter in a circle as an exit, and the same Greek letter with identifying subscripts in a circle at each entrance point. See Figure 4.

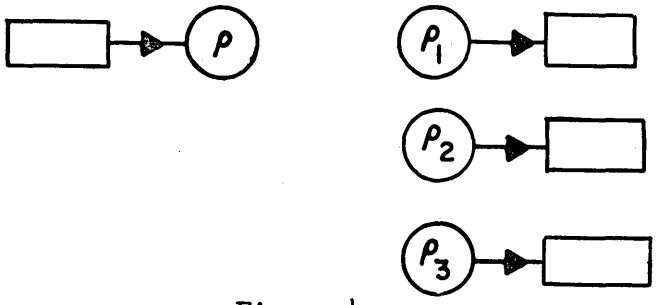


Figure 4

In the preparation of a logical code, the transfer instruction indicating the exit (ρ) written as

T [ρ]

is used to <u>identify</u> the particular remote exit. It is the location in the memory where the transfer order of the exit resides and not to be interpreted as the address part of the transfer instruction.

The addresses corresponding to the entrances (ρ_1), (ρ_2), and (ρ_3) are provided to the exit [ρ] from the appropriate positions of the flow diagram (cf. Chapter II, Problem 7, pp. 53 ff). The various (ρ_1) are supplied to T [ρ] by substitution instructions, S⟶m,

In the descriptive coding each <u>set</u> of variable remote connections is represented by a symbol

F.i i(= 01,02 ···)

where the <u>i</u> is distinct for each <u>set</u>. (Greek letters do not exist

in the vocabulary. We use them in the discussion and in flow diagrams for simplicity of notation.) These instructions concerned with such a set (both the transfer instruction which is the exit and the various substitution instructions which supply addresses to the transfer instructions) have as their address the symbol F.i corresponding to the particular set. Example 4 illustrates this.

Example 4

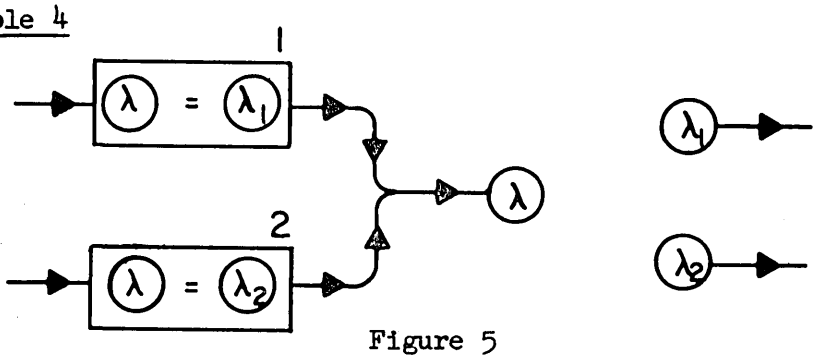


Figure 5

Since (λ_1) and (λ_2) are addresses they are to be stored in A storage as instruction words. However, for this example we do not discuss the A storage in detail, and we merely indicate

A.01: $(\lambda_1)_o$

A.02: $(\lambda_2)_o$

We designate the set of variable remote connections by F.01. The coding is:

Box 1

1.	m⟶Ac	A.01	$(\lambda_1)_o$ to R2	
2.	S⟶m	F.01		λ_1 to address of F.01
3.	L(0)	000		
4.	T	F.01		

Box 2

1.	m⟶Ac	A.02	$(\lambda_2)_o$ to R2	
2.	S⟶m	F.01		λ_2 to address of F.01
3.	T	01,4		

Instructions 2 and 4 of Box 1, and Instruction 3 of Box 2, are those instructions concerned with the set of variable remote connections F.01; hence they have as their address F.01. Note that Instruction 3 of Box 1 is L(0). This insertion is necessary as no substitution instruction may modify the instruction immediately following. The L(0) serves as a "dummy-do-nothing" instruction which separates by one the substitution instruction and the instruction that it is to modify.

The (λ_1) and (λ_2) are indicated on the flow diagram as entrances into operation boxes; therefore the addresses corresponding to (λ_1) and (λ_2) are usually the addresses of the first instruction of their indicated operation box. The address portion of the words in A storage corresponding to (λ_1) and (λ_2) are treated in the same manner as the address of a fixed connection transfer. Therefore, if (λ_1) corresponds to Box 5, Instruction 1, the address portion of (λ_1) in A storage would be

A.01: ··· 05,1 ··· 05,1

The exit, $T[\lambda]$, of the variable remote connection must transfer the control at different stages of the problem to the various (λ_i) associated with the remote connection. The addresses corresponding to the (λ_i) are usually distinct. When the computer code is formed by the assembly routine, there is no assurance that the instructions to which the (λ_i) refer will all occupy the same side of their respective instruction words. In order that the $T[\lambda]$ shall have the flexibility that enables it to transfer the control to either side of an instruction word, the transfer order as well as the address must be modified. To accomplish this, each (λ_i) is stored as a transfer instruction, and the assembly routine modifies the order if necessary when the absolute address corresponding to (λ_i) is assigned. A half-word substitution instruction, HS⟶m, is then used rather than S⟶m, as indicated in Example 4, to supply to the exit $T[\lambda]$ the appropriate $T[\lambda_i]$. Example 5 illustrates three sets of variable remote connections and the proper A storage associated with them.

Example 5

The necessary storage is:

A.01:	**CA031CA031**	**D.01:**	**x**
A.02:	**CA041CA041**		
A.03:	**CA051CA051**		
A.04:	**CA061CA061**		
A.05:	**CC091CC091**		
A.06:	**CC0A1CC0A1**		

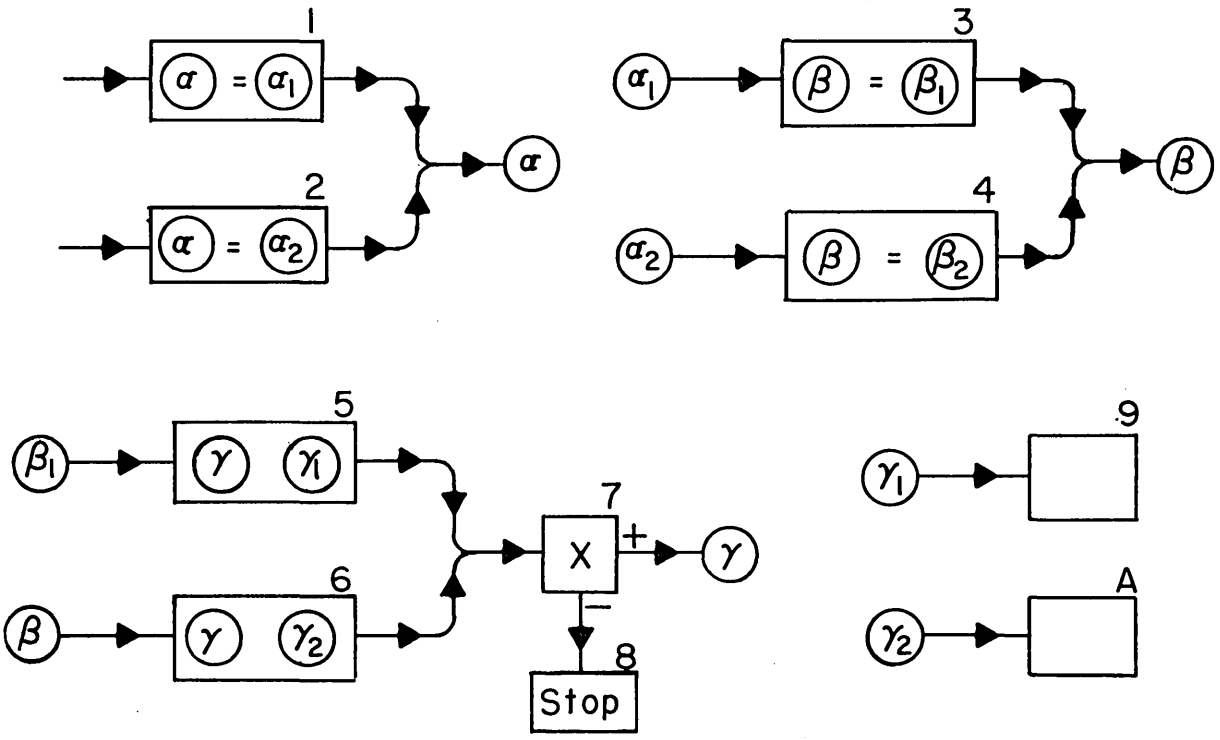


Figure 6.

We assume that x is formed in another part of the routine and stored in D.01. We designate by F.01 the set of variable remote connections (α), by F.02 the set (β), and by F.03 the set (γ). The coding is:

Box 1.

1.	m⟶Ac	A.01	$(CA03,1)_o$ to R2	
2.	HS⟶m	F.01		CA03,1 to F.01
3.	L(0)	000		
4.	T	F.01		

Box 2.

1.	m⟶Ac	A.02	$(CA04,1)_o$ to R2	
2.	HS⟶m	F.01		CA04,1 to F.01
3.	T	01,4		

Box 3.

1.	m⟶Ac	A.03	$(CA051)_o$ to R2	
2.	HS⟶m	F.02		CA05,1 to F.02
3.	L(0)	000		
4.	T	F.02		

Box 4.

1.	m⟶Ac	A.04	$(CA06,1)_o$ to R2	
2.	HS⟶m	F.02		CA06,1 to F.02
3.	T	03,4		

Box 5.

1.	m⟶Ac	A.05	$(CC09,1)_o$ to R2	
2.	HS⟶m	F.03		CC09,1 to F.03

Box 7.

1.	m⟶Ac	D.01	x to R2
2.	C	F.03	

Box 8.

1. Stop

Box 6.

1.	m⟶Ac	A.06	$(CC0A,1)_o$ to R2
2.	T	07,F	

Instructions 2 and 4 of Box 1, and Instruction 2 of Box 2, are those concerned with the set of variable remote connections (α) ≡ F.01; therefore, those instructions have the address F.01. Similarly, Instructions 2 and 4 of Box 3, and Instruction 2 of Box 4, have the address F.02; and Instruction 2 of Box 5, and Instruction 2 of Box 7, have the address F.03.

Instruction 2 of Box 7 is a conditional transfer instruction; hence those instruction words in A storage which are to be substituted into it are themselves conditional transfer instructions as shown in A.05 and A.06.

Note the use of the HS⟶m instructions in the substitutions concerned with the variable remote connections.

The sequence in which the operation boxes are coded is 1, 2, 3, 4, 5, 7, 8, 6, which is the order in which the computer code is to be sequenced. It is always true that the sequencing of the operation boxes in the descriptive coding must correspond to the sequencing necessary in the computer code regardless of the numbering of the boxes on the flow diagram. The number assigned to each box on the flow diagram is, however, the number to be used in the address of instructions referring to the box.

In Box 6 of Example 5, Instruction 2 is a transfer to 07,F which is a transfer of the control into the last instruction of the box immediately

preceding Box 7. In this case, the transfer is to Box 5, Instruction 2, since Box 5 is the box in the coded sequence which immediately precedes Box 7.

The assembly routine treats the variable remote connections as follows: The A storage concerned is altered to its absolute address and the transfer order contained is modified, if necessary. Whenever the assembly routine encounters a substitution instruction with an address F.i, the absolute address of the associated transfer instruction (the transfer instruction with the same F.i address) is determined and that address is inserted into the substitution instruction.

It is often useful to be able to store numbers from R2 into D storage by using substitution instructions. To do this, the substitution instruction is given the appropriate D.i address; however, the substitution order must be written as the desired primed or unprimed order. For example, consider that bigits (20-39) of R2 are to be sent to bigits (20-39) of D.05. The descriptive instruction effecting this would be

HS⟶m' D05 which is FDD05

Similarly, to store bigits (8-19) of R2 into bigits (8-19) of D.0A, the instruction reads

S⟶m D.0A which is FAD0A

In a substitution instruction with a D address the assembly routine never modifies the order part of the instruction.

Since the substitution instructions may have box numbers as addresses and since substitution instructions may refer to D storage, it is necessary to restrict the total number of operation and alternative boxes of any one problem to CF boxes, which decimally is 207 boxes in all.

There are occasions when it is necessary to know in advance whether an instruction is to occupy the left or right-hand instruction of a word in the computer code. In fact, it may be necessary to position certain instructions on a fixed side of an instruction word; e.g., at the completion of a drum instruction, the control is transferred to the left-hand instruction of the word specified by bigits (28-39) of the drum instruction; hence, the instruction to which the transfer is desired must be in the left-hand side of its respective instruction word. Further, the drum instruction itself must occupy a full word in the computer code so

that this instruction must always begin on the left. In order that instructions, where necessary, can be positioned with the desired parity (i.e., left or right) a symbol is provided in the descriptive code so that the computer code of any operation box can be started on the left of an instruction word. As soon as the first instruction of a box is fixed on the left, the parity of all instructions within the box is known immediately. By inserting a "dummy-do-nothing" L(0) as a first instruction, one may change the parity of all succeeding instructions.

The descriptive code tape is composed of the descriptive coding and the static storage (i.e., A, B, 7, and C storage) of the problem. All of the descriptive coding and any identifying symbols for the tape which refer to the descriptive coding are punched as five character words. The C, B, and 7 storage and any corresponding identifying symbols are punched as ten character words.

The sequencing of the data on the code tape is as follows:

In order that the assembly routine can assign the absolute addresses to the various instructions and the storage, the initial absolute address for the code must be specified. It is the first word that is punched on the tape, and it is a five-character word. For example, if the assembled code is to begin at address 25E, the first word of the tape would be

0025E

A descriptive code may be assembled into an absolute code starting at any initial address with the restriction that the code with A, C, B, and 7 storage must not exceed address 37C (892 decimally).

Immediately following the initial address on the tape is the descriptive coding. The sequencing of the boxes of descriptive code as punched on the tape specifies the linear sequencing of the assembled code. Preceding the instructions of each box, the box number is punched onto the tape as a five-character word where the word consists of three zeros followed by the box number. For example, consider a descriptive coding of two operation boxes where the assembled code is to begin at address 052. The descriptive coding and the corresponding code tape is:

Box 1

1. m⟶Ac D.01
2. m⟶Ah D.02
3. A⟶m D.03

Box 2

1. m⟶Q B.01
2. X D.03
3. A⟶m D.04

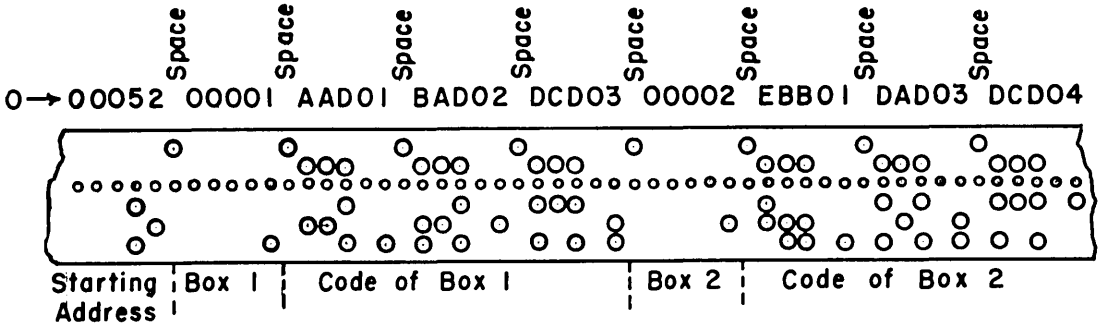


Figure 7

All of the instructions of the boxes with the corresponding box numbers are punched onto the tape in this fashion. Recall that the boxes of code are not necessarily sequential according to box number, but sequential according to linear ordering in the assembled code. The box number that precedes each box of instructions corresponds to the box number as shown on the flow diagram.

Immediately following the last instruction of the descriptive coding, the box numbers only of the associated subroutines are punched on the tape in the order corresponding to the linear sequencing of the subroutines in the assembled code. As before, these box numbers are five-character words. We defer any further discussion of this until the section on subroutines, at which point the reasons for listing the subroutine box numbers are discussed.

The five-character word

00C00

follows the subroutine box numbers on the tape. If no subroutines are associated with the descriptive code, the word 00C00 follows the last instruction of the descriptive coding. The word 00C00 indicates the completion of the descriptive coding.

The A storage punched as five-character words follows the word 00C00 on the tape. For example, consider a descriptive coding where the A storage is

A.01: CA041CA041

A.02: CC227CC227

A.03: AAD05AAD05

The section of the descriptive tape corresponding to this would be:

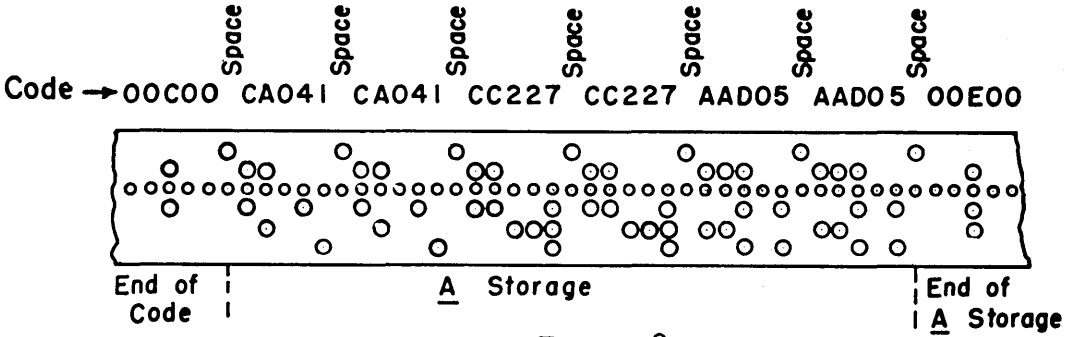


Figure 8.

Following the A storage on the tape is the five-character word

00E00

which indicates the end of the A storage. If there is no A storage the word 00E00 immediately follows the word 00C00 on the tape.

The numerical storage of the problem is punched onto the tape following the word 00E00. This storage is punched as ten-character words. Each group of storage is punched in order of ascending addresses and is terminated by two adjacent spaces on the tape. The C storage is the first group of storage punched on the tape. The last word of C storage is followed by two adjacent spaces. The B storage is then punched on the tape and it is followed by two spaces. Next is the 7 storage on the tape. The 7 storage terminates the descriptive code tape and at least five spaces must follow the last word of 7 storage on the tape.

At one stage in the evolution of the descriptive coding a word 8000000000 was used in lieu of the two adjacent spaces separating the groups of numerical storage on the descriptive tape. Hence, between the C and B storage, between the B and 7 storage, and following 7 storage, was the word 8000000000. The present assembly routine allows the use of this word 8000000000 in the aforementioned manner; therefore, this is an optional method of separating and identifying the groups of storage.

In the event that a storage group is not used in a descriptive coding, the spaces signifying the end of the groups of storage are treated as follows:

The omission of 7 storage effects no changes and the last group of storage on the tape, whether it is C, B or 7, is followed by at least five adjacent spaces.

If there is no C storage, the word 00E00 is followed by two spaces and then the B storage.

If there is no B storage, one additional space symbol must be used in conjunction with the two adjacent space symbols signifying the end of the C storage (whether or not any C storage is actually present). In other words, if B storage is omitted three adjacent spaces are used to signify the end of C storage and the absence of B storage.

In the alternative method where the word 8000000000 indicates the end of each group of storage, even though a group of storage is not present its terminating word is included on the tape to indicate the end of, or absence of, a particular group. Example 6 illustrates a three box code, and its descriptive code tape.

Example 6

The example forms an approximation to e^{-x} for $0 < x < 1$ from the expression

$$e^{-x} = \lim_{n \to \infty} \left[\frac{1 - \frac{x}{2n}}{1 + \frac{x}{2n}} \right]^n$$

where for this example we choose n = 32, and

$$e^{-x} \simeq \left[\frac{1 - \frac{x}{64}}{1 + \frac{x}{64}} \right]^{32}$$

The flow diagram is:

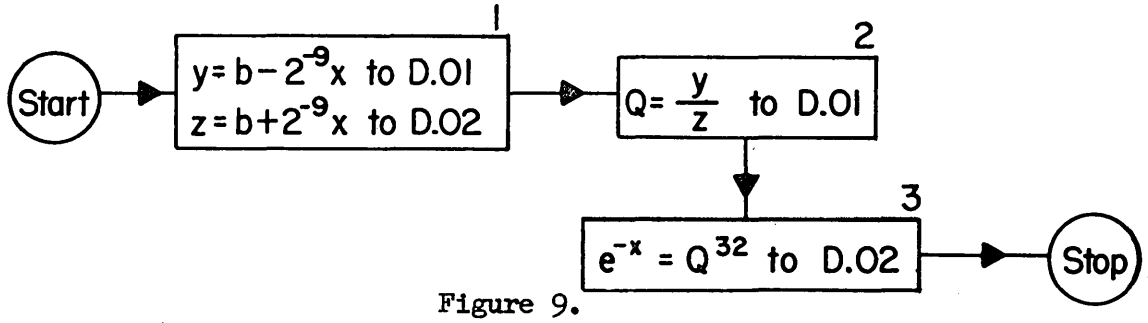


Figure 9.

C.01: x D.01:
B.01: $b = 2^{-1}$ D.02:

The coding is:

Box 1				
1.	m⟶Ac	C.01	x to R2	
2.	R(9)	009	$2^{-9}x$ in R2	
3.	A⟶m	D.01		$2^{-9}x$ in D.01
4.	m⟶Ah	B.01	$z = b + 2^{-9}x$ in R2	
5.	A⟶m	D.02		z to D.02
6.	m⟶Ac	B.01	b to R2	
7.	m⟶Ah-	D.01	$y = b - 2^{-9}x$ in R2	
8.	A⟶m	D.01		y to D.01
Box 2				
1.	m⟶Ac	D.01	y to R2	
2.	÷	D.02	Q = y/z in R2	
3.	A⟶m	D.01		Q to D.01
Box 3				
1.	m⟶Q	D.01	Q to R4	
2.	X	D.01	Q^2 in R2	
3.	A⟶m	D.01		Q^2 to D.01
4.	m⟶Q	D.01	Q^2 to R4	
5.	X	D.01	Q^4 in R2	
6.	A⟶m	D.01		Q^4 to D.01
7.	m⟶Q	D.01	Q^4 to R4	
8.	X	D.01	Q^8 in R2	
9.	A⟶m	D.01		Q^8 in R2
A.	m⟶Q	D.01	Q^8 to R4	
B.	X	D.01	Q^{16} to R4	
C.	A⟶m	D.01		Q^{16} to D.01
D.	m⟶Q	D.01	Q^{16} to R4	
E.	X	D.01	$e^{-x} = Q^{32}$ in R2	
F.	A⟶m	D.02		e^{-x} to D.02
10.	Stop			

The code is to be assembled starting at address 297. The descriptive code tape is shown in Figure 10. x in C.01 is set to 0.5 for the tape.

Note: Tape is continuous, but has been broken for illustrative purpose.

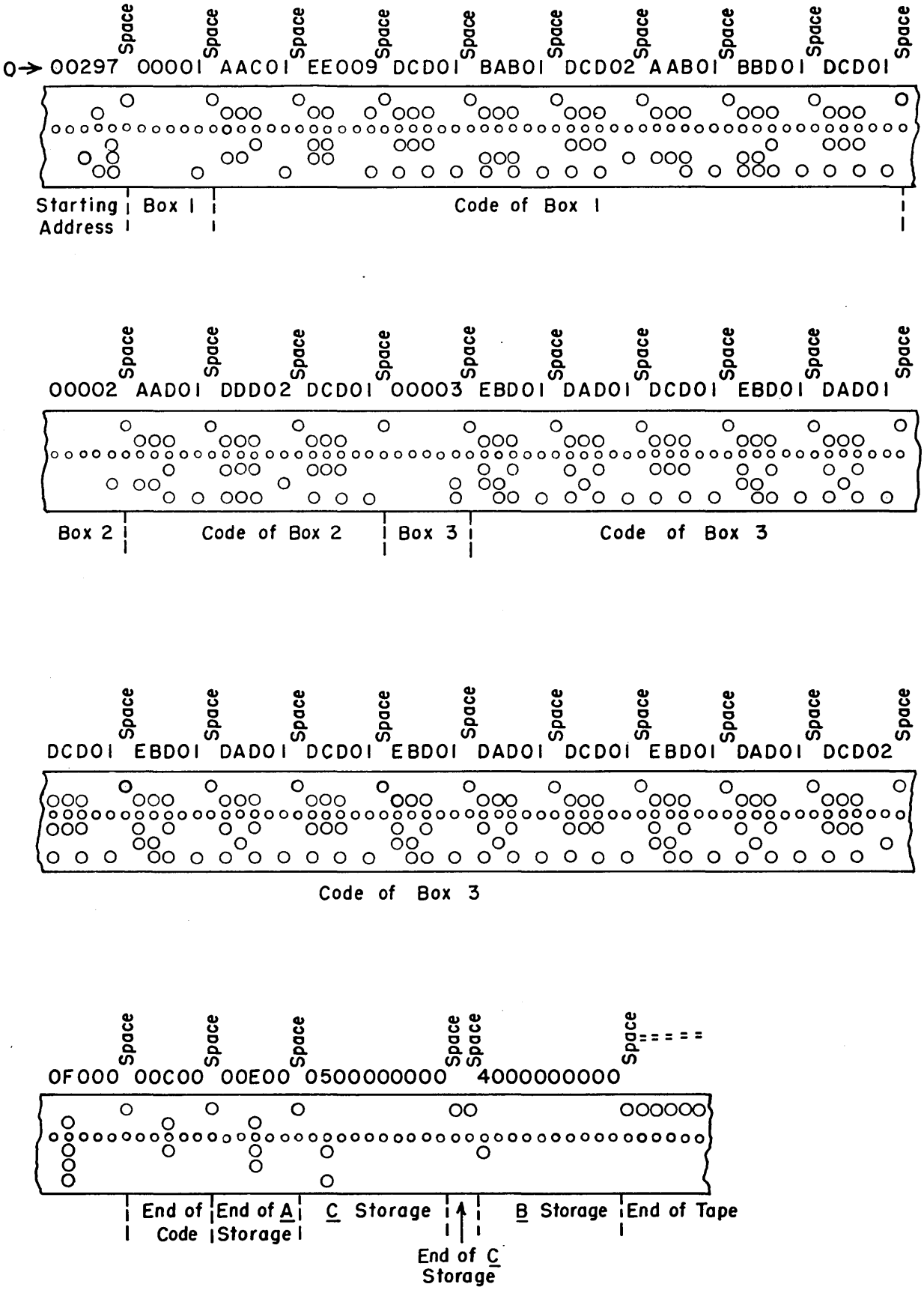


FIG. 10

Since there are no subroutines associated with the code, the word 00C00 follows the last instruction of Box 3; and since there is no A storage the word 00E00 follows 00C00. No 7 storage is contained in the coding; hence the five spaces follow the B storage.

As previously mentioned, there is a symbol which indicates to the assembly routine that the first instruction of a box is to be on the left side of an instruction pair. It is included in the word that specifies the box number and is the character 4 for the middle tetrad; e.g., suppose that in the code of Example 6, Box 2 is to begin as a left-hand instruction. The word on the descriptive tape specifying the box number would be

00402

When the code is processed by the assembly routine and a box number word with a 4 in the middle tetrad is encountered, the following occurs: If the last instruction of the previous box was assembled as a right-hand instruction, the first instruction of the box concerned naturally becomes a left-hand instruction of its instruction-word, and the assembly routine proceeds accordingly. If the last instruction of the preceding box was assembled as a left-hand instruction, the assembly routine completes the word by inserting a "dummy-do-nothing" instruction of L(0) into the right-hand instruction position. The first instruction of the box concerned is then assembled as a left-hand instruction of the succeeding word. If the flow diagram indicates a transfer of the control to a box that must begin as a left-hand instruction, one cannot use the flexibility and convenience afforded by a transfer into one of the last seven instructions of the preceding box. This restriction arises because of the "dummy" L(0) instruction that may be inserted.

Another symbol may be incorporated in the word specifying the box number. This is a character 8 as the first tetrad of the word. This symbol causes the assembly routine to interrupt the assembly process and to stop the computer. The need for such a symbol is covered in the discussion of methods of alteration of the descriptive code in the chapter on Operating Procedures.

A frequent occasion where it is necessary to have a box begin with a left-hand instruction is in the use of drum instructions which we now examine in detail.

The drum instruction, since it is a full word, necessitates special treatment both in the descriptive code and by the assembly routine. As previously mentioned, the drum instruction must be coded in the descriptive coding so that it naturally starts with the left-hand instruction of an instruction word in the assembled code. The drum instruction is, however, coded as two descriptive instructions. The first instruction is the drum order, and the descriptive address for the associated block of fifty words in the memory. The second instruction specifies the associated drum track in the order position and the address position contains the descriptive address for the transfer of the control upon completion of the drum instruction.

The descriptive address for the associated fifty words in the memory may refer to any of the storage; hence it may be an A.i, C.i, B.i, 7.i, or D.i address; the address may be an E.i if it is desired to have the drum communicate with a block of fifty words contained in the same box as the drum instruction; the address may be inserted as an absolute address if desired; or the address may be supplied to the drum by a substitution instruction in conjunction with addresses in A storage.

The associated drum track address is either inserted into the descriptive coding as a pseudo-absolute address or is supplied from a coded routine. The pseudo-absolute addresses range from 00 to C7, corresponding to the two hundred tracks of the drum (0-199, decimally). Unfortunately, the drum tracks are not addressed sequentially from 00 through C7, but range from 00 through FF (0-255, decimally); hence the expression "pseudo-absolute" is used for inserted drum addresses. The assembly routine modifies the pseudo-absolute address to the actual value in the range 00 through FF. The address to which the control is to transfer upon completion of a drum instruction is treated in the same manner as are the addresses of transfer instructions. The address may specify a box number and one of the first seven instructions of the box or one of the last seven instructions of the preceding box. The address may also be specified by an E.i address if the transfer is within the operation box containing the drum instruction. The transfer, however, is automatically to the left-hand instruction of a word; hence that instruction must be positioned appropriately.

We now give three examples (7, 8, and 9) illustrating the treatment of the drum instruction.

Example 7

Three operation boxes are given. There are two drum instructions. One sends fifty words from D storage to the drum. The second reads fifty words from the drum into the just vacated D storage of the memory.

The flow diagram is:

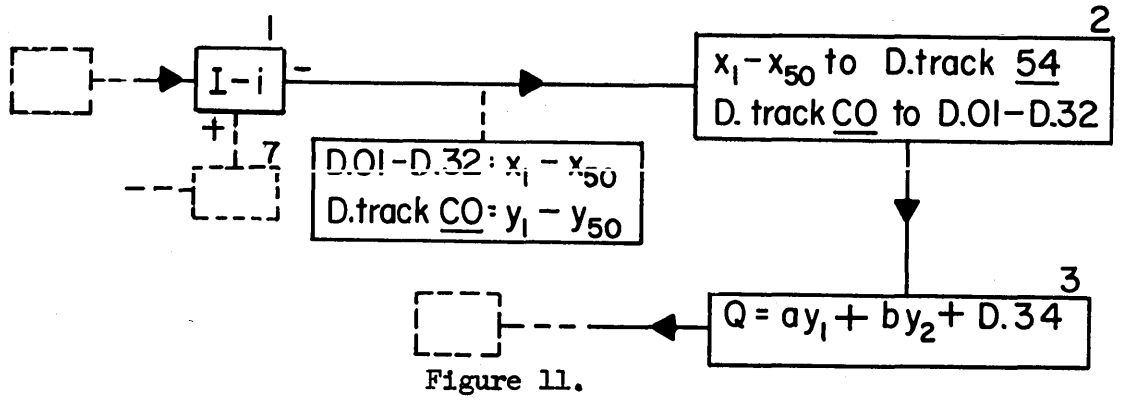


Figure 11.

B.01: a D.01 - D.32: $x_1 - x_{50}$
B.02: b D.33: i
B.03: I D.34:

The coding is:

Box 1

1.	m⟶Ac	B.03	I to R2	
2.	m⟶Ah-	D.33	I - i in R2	
3.	C	07,1		

Box 2

1.	m⟶D	D.01		$x_1 - x_{50}$ to D.track 54
2.	54	E.03		
3.	D⟶m	D.01		$y_1 - y_{50}$ to D.01-D.32
4.	CO	03,1		

Box 3

1.	m⟶Q	D.01	y_1 to R4	
2.	X	B.01	ay_1 to R2	
3.	A⟶m	D.34		ay_1 to D.34
4.	m⟶Q	D.02	y_2 to R4	
5.	X	B.02	by_2 to R2	
6.	m⟶Ah	D.34	$Q = ay_1 + by_2$ in R2	
7.	A⟶m	D.34		Q to D.34

Instructions 1, 2, 3, and 4 of Box 2 are the two drum instructions; hence Instructions 1 and 3 must be left-hand instructions in their respective instruction-words in the assembled code. This is done by arranging Box 2 so that it begins with a left-hand instruction; i.e., on the descriptive code tape 00402 is punched for the box number word. Instruction 2 specifies that Track 54 is the pseudo-track number. This is modified to Track 69 (the absolute address) by the assembly routine. The transfer indicated by the address of Instruction 2 is to E.03; hence the control is to transfer to Instruction 3 of Box 2. The instruction to which the transfer is effected must be on the left side in the assembled code and since Instruction 1, Box 2, begins on the left of a word, Instruction 3 does also. Instruction 4 of Box 2 specifies the pseudo-track number C0 which the assembly routine modifies to F1, the corresponding absolute track address. The address specifies a transfer to Box 3, Instruction 1. Box 3 must then be coded so that it begins with a left-hand instruction. In this example we see that this is taken care of, since Box 2 ends with a right-hand instruction. If Box 3 did not naturally begin with a left-hand instruction, it would have to be so arranged by punching the box number for Box 3 as 00403.

Example 8

In this example fifty words of code in the memory are to be replaced by fifty words from the drum where the fifty memory words are contained in the same box as the drum instruction. The quantity i, that eventually becomes the drum track number, is formed in a part of the routine not coded, and is stored in D.01 as

$$D.01: \quad i \cdot 2^{-27}$$

The drum instruction upon completion is to transfer to the first instruction of the fifty words which have been called into the memory.

The coding is:

Box 1

1.	m⟶Ac	D.01	$i \cdot 2^{-27}$ to R2	
2.	A⟶m	D.02		$i \cdot 2^{-27}$ to D.02
3.	m⟶Ac	E07	D⟶m E09 00E09 to R2	
4.	S⟶m'	D.02		E09 to (28-39)D02

```
5.  m⟶Ac     D.02     i(20-27 )E09(28-39)  to R2
6.  HS⟶m     E08                           i(20-27) E09(28-39) to E07
7.  D⟶m      E09
8.  [00      E09]
9.
A.
B.
C.
```

The descriptive tape has the box number 00401. Instructions 1, 2, 3, and 4 form the drum track address. Instruction 2 sends $i \cdot 2^{-27}$ to D.02. Instructions 3 and 4 then combine the address part (the address specifying the transfer) of the instruction with the track address in D.02. Note that Instruction 4 is written as S⟶m' D.02. It is written as the primed instruction since the substitution is into the right-hand side of a word of D storage. (Note that this differs from the case where a substitution is made into instructions, cf. page 212) Since Instruction 1 of the box is on the left, the drum instruction (Instruction 7) and the instruction to which the transfer is effected (Instruction 9) are left-hand instructions as desired.

Example 9

In this example, fifty words of code on the drum are to replace fifty words of code in the memory, where both the words in the memory and those on the drum correspond to one or more complete boxes of code. Again, only the box containing the drum instruction is coded. We assume the words to be replaced in the memory begin with Box 2C, Instruction 1, and the drum track concerned has the pseudo-track number A1. The address corresponding to Box 2C, Instruction 1, is stored in A storage in a transfer or substitution instruction word and is

A.01: CA2C1CA2C1

The coding is:

Box 1

```
1.  m⟶Ac    A.01     CA2C1CA2C1  to R2
2.  S⟶m     E.03                          2C,1  to (8-19) Instr.3
3.  D⟶m     [2C,1]
4.   A1      54,1
```

Instruction 1 brings the address for the drum instruction into R2. This address was stored in A.01 as part of a transfer instruction so that it could be stored as a box number and instruction number. Instruction 2 is an S⟶m E.03 which supplies the address 2C,1 to Instruction 3, the drum instruction. Recall that a substitution instruction is not supposed to substitute into an immediately following instruction. However, in this instance, we know that the drum instruction **begins** as the left-hand instruction of a word; hence, the substitution instruction cannot be in the same instruction word as the drum instruction and the substitution as indicated is permissible. The address written in the drum instruction is irrelevant; hence, any address may be placed there. Instruction 4 contains the pseudo-track address A1 and the address of Box 54, Instruction 1, to which the control is to transfer upon completion of the drum instruction. Box 1 must begin with a left-hand instruction to position the drum instruction correctly; therefore, the Box 1 code word is

00401

Box 2C as it originally is coded must begin with a left-hand instruction; hence the Box 2C code word is

0042C

The control is to transfer to Box 54, Instruction 1, upon completion of the drum instruction; hence Box 54 must begin with a left-hand instruction and its code word is

00454

The assembly routine modifies the pseudo-track number A1 to the corresponding absolute track address, CB.

For a further discussion of the drum one should consult the chapter on **The Computer.**

It is desirable to have a printed copy of the assembled code so that one may know the absolute addresses of the storage and the instructions in order to "debug" the assembled code for subsequent running. It is important that this printed copy is in a form that is easily read and understood. To produce such a copy a printing routine using the Synchroprinter has been included in the assembly routine. It provides the following data:

The first line of the printed listing contains five 3-character numbers which are the absolute addresses corresponding to

A.00 C.00 B.00 7.00 D.00

respectively. If any group of storage is not contained in the coding, the address for that group is the same as the initial address of the succeeding group. Consider that an assembled code has the following absolute initial addresses

A.00 = 201 7.00 = 23B
C.00 = 205 D.00 = 2D5
B.00 = 221

The first line of the listing would be

201 205 221 23B 2D5

Following the first line is the listing of the code proper. One has the option of a listing of five or six columns. The five-column listing contains, in order of columns from left to right on the page,

1. the box number
2. the descriptive instruction number
3. the absolute instruction-word number (address) as assigned by the assembly routine
4. the instruction with its absolute address as assigned by the assembly routine
5. the descriptive address of the instruction as coded in the descriptive coding

The six-column listing contains the five columns as listed above and a sixth column that is:

6. the contents of the B or C storage specified in the address of the instruction.

Following the listing of the code is a listing of A, C, B and 7 storage, respectively. The C, B, and 7 storage listing is a four-column listing where the columns are:

1. classification of storage
2. the descriptive address of the storage
3. the absolute location address as assigned by the assembly routine
4. the numerical quantity as stored at the address concerned

Example 10 illustrates the 5-column page listing.

Example 10

Consider the descriptive code of Example 1 and assume that it has been assembled in the memory beginning at address 000. The listing given of the assembled routine is:

	005	005	008	009	009
01	01	000	EB006	C01	
	02		DA009	B01	
	03	001	BA007	C02	
	04		DE028		
	05	002	DA009	B01	
	06		BA008	C03	
	07	003	DC00A	D01	
02	01		EB00A	D01	
	02	004	DA00A	D01	
	03		BA00A	D01	
	04	005	DC00B	D02	
C	01	006	a		
C	02	007	b		
C	03	008	c		
B	01	009	x		

The code contains no A or 7 storage; hence the first line corresponds to

C.00 C.00 B.00 D.00 D.00

In lines 2 through 11, inclusive, the numbering in the first and second columns corresponds to the numbering on the descriptive coding. The third column contains absolute location addresses; hence each address corresponds to an instruction-pair in column 4; i.e., word 000 is

000: EB006DA009

The descriptive addresses as given in column 5 are the same as those in the instructions in the descriptive coding.

If we set

a = 4040000000

b = 2190000000

x = 4000000000

a 6-column listing of the first three instructions would be

01	01	000	EB006	C01	4040000000
	02		DA009	B01	4000000000
	03	001	BA007	C02	2190000000
	.	.	.	.	.
	.	.	.	.	.

The contents of C01 and C02 as listed would be the converted number (the binary equivalent of the decimal input) in the C storage.

If the coding had contained A storage, for example

A.01: CA02,1 CA02,1,

the listing of it would be

A 01 006 CB003CB003 02,1

where the first four columns are as before, and the fifth column gives the relative address.

The method of descriptive coding is easily generalized to incorporate the use of subroutines; hence it is appropriate that subroutines are discussed in conjunction with the descriptive coding.

As a person gains in experience in coding it becomes apparent to him that from one problem to another there are certain basic sequences of instructions that are very similar. For example, two different problems might, at some phase of their computation, involve taking the square root of some number or group of numbers. The two sequences of instructions for the square root would generally contain identical orders, while the corresponding addresses would be different. Routines such as the conversion routine as discussed in Chapter II would be an integral part of most problems, and from problem to problem these routines would differ only in the addresses of their instruction sequences, while the order patterns would be the same. In fact, it is true that most of the routines coded in Chapter II would occur as parts of larger problems.

Since these routines or sections of code that repeatedly appear in problems can be coded in a way such that the addresses of the instructions can easily be modified to any desired addresses, it becomes possible to incorporate such routines directly into the code of any problems without having to rewrite their instructions. We call any section of code a subroutine if it is coded in a way that it can be incorporated into any problem without having to rewrite the coding. Consequently, a library of subroutines, or more precisely a library of punched tapes of subroutines, has been compiled. These punched tapes may be incorporated directly into any desired problem. There is a card indexing system for the library where each subroutine has a card on file which gives complete information about the particular routine. We defer further discussion of this and return to the coding of subroutines.

We have already discussed how any problem code, including all of its necessary storage, may be assembled from a descriptive code tape into any absolute addresses in the memory, excluding addresses 37C to 3FF. Further, we have seen how one can, by altering only the initial word on the tape, form different instruction sequences in the memory, where the order patterns are the same but the corresponding addresses differ. This is precisely the kind of thing that is desired for subroutines. Each subroutine is coded descriptively as though it were a problem complete with storage. In fact, each subroutine does constitute a complete problem, in the sense that it starts with certain initial conditions and leads to a clearly defined conclusion.

The descriptive coding of a subroutine differs in several ways from the coding of a normal problem, and we now discuss these differences. In the coding of a subroutine the boxes of code must be numbered consecutively starting with 1, where the numbering corresponds to the linear sequencing of the boxes on the descriptive code tape. For ease of use it is desirable to code a subroutine as one box whenever practicable.

Only one set of variable remote connections is allowed, and this set pertains to the exit from the subroutine. The details of this are discussed presently.

All of the static storage necessary in the subroutine is included on the descriptive code tape of the subroutine with the condition that neither A nor C storage is allowed. Any storage that would normally correspond to C storage is converted and stored in the subroutine as B or 7 storage. Storage that would normally correspond to A storage must have special treatment, in that the storage must exist as instructions in the descriptive code. This is illustrated by later examples.

There are, in general, two kinds of dynamic storage associated with a subroutine. These are the dynamic storage that originates from within the code of the subroutine and the dynamic storage that originates in the problem apart from the subroutine, but is pertinent in the subroutine. Although this latter storage is static with respect to the subroutine, it is, however, dynamic storage in the overall problem and is treated as such in the subroutine. For example, in a square root subroutine, the dynamic storage originating from within the routine is the

storage arising from intermediate values in the iterative process and the storage for the successive iterates. The dynamic storage arising apart from the routine is the storage for the number whose square root is desired. This number comes from the problem and is present at the time of entry into the square root subroutine.

All storage is addressed as in a problem. That is, the addresses of each group of B.i, 7.i, and D.i storage are consecutive addresses beginning with i = 01.

We now have the situation that a subroutine coded by the descriptive method with the above mentioned restrictions can be coded as an independent problem into any desired addresses in the memory. The next step is to have the assembly routine specify the desired addresses.

In the flow diagram of a problem, boxes should be included for the subroutines of the problem although they do not need to indicate in detail the computation of the subroutine. These boxes need to be assigned numbers on the flow diagram where the only restriction is that a subroutine that contains several boxes must be assigned a corresponding group of consecutive numbers. The numbers assigned on the flow diagram to the boxes of subroutines will not, in general, be the same as those indicated on the subroutines' descriptive code tapes. Note that this differs from the treatment of the problem proper.

Recall that on the descriptive code tape the box numbers corresponding to the subroutines are first punched following the main problem code and prior to the code word 00C00. These box numbers correspond to the box numbers as assigned by the particular flow diagram. They will replace the box numbers as given originally on the subroutine tapes.

We now describe the method by which the assembly routine integrates the subroutines into the problem. The descriptive tapes corresponding to the subroutines are arranged in the order in which they are to appear in the computer. It is recommended that a single tape containing all of the desired, properly analyzed subroutines be prepared from the separate tapes. After the descriptive tape of the problem, including storage, is initially processed by the assembly routine, the computer stops so that the subroutines may be inserted. The subroutine tape is placed in the reader and the assembly process is continued. The code of each subroutine is assembled in order following the code of the problem. The storage associated with each subroutine is treated as follows:

The static storage associated with each subroutine is included on its descriptive tape. The storage of each subroutine is not directly added to the storage of the problem as this, in general, would lead to duplication of storage. For example, the number 0 might be already stored in B storage in the problem, and in the B storage of several of the subroutines. The 0 need only be stored once, however, and the other storage of 0's is needless duplication. To circumvent this, as each word of B storage of a subroutine is incorporated into the storage of the problem, it is compared with all existing C and B storage in the problem; and if it is identical to any existing C or B storage it is not stored. However, all of the descriptive addresses of the subroutine that referred to the discarded word of storage are modified to refer to the already existing word. If the subroutine word of B storage is not identical with any existing C or B storage in the problem, the word of storage of the subroutine is added to the existing B storage of the problem and the addresses of the pertinent instructions are accordingly modified. We see then that after the assembly process is completed there is no duplication of storage due to the B storage of subroutines. This, however, leads us to the meaninful purpose of 7 storage.

The 7 storage existing in a problem is not compared with the B storage of the incorporated subroutines. Any 7 storage existing in subroutines is directly added to the existing 7 storage of the problem. The need for such a group of storage becomes apparent as one works with subroutines, and it is illustrated in a subroutine example.

This completes the discussion of how the subroutines are incorporated into a problem and all that remains is to discuss the means of entry into and exit from these subroutines.

These connecting links of a subroutine are analagous to those of some of the orders of the vocabulary, so we first discuss the more familiar order in the vocabulary.

Consider, for the discussion, that a multiplication is to be performed. The multiplication order supplies the multiplicand, but the multiplier must be already in R4. This latter fact is accomplished by coding that precedes the multiplication order. The sequencing by the control counter brings the multiplication instruction into R6, the control register, so that it can be performed. The address associated

with the multiply order specifies the location of the multiplicand. Upon the completion of the multiplication, the product resides in R2. The exit from the multiplication is provided by the address which is in the control counter, the next instruction in the code sequence. We naturally expect the entry into and exit from a subroutine to be more complex than for a simple multiplication since a subroutine is a sequence of instructions rather than a single instruction. However, as in the multiplication order, the number or numbers that are to be operated upon by the subroutine must be in locations specified by the subroutine prior to entry into the routine. (In the multiplication, the multiplier is in R4, the multiplicand is at the address of the instruction.) These connecting addresses are certain dynamic storage locations, D.i, and the precise D.i addresses are specified on the library index card of the subroutine. The necessary numbers are sent to the appropriate D.i addresses by code prior to entry of the subroutine. After the necessary numbers are stored, the actual entry into the subroutine is initiated.

The entry into a subroutine from any location in a problem is treated as a fixed connection. The box numbers of a subroutine are indicated on the flow diagram; hence one need only indicate a transfer to the starting box and instruction of the subroutine in question.

When the subroutine is performed, a number or set of numbers is formed as the results (the product in the multiplication is in R2). These numbers are then stored in other D.i addresses specified by the subroutine. These D.i addresses are shown on the subroutine index card.

Prior to entry into the subroutine, the desired exit is established. At each point of entry it is known where the control is to proceed upon exit. This exit is established by a set of variable remote connections. The variable transfer is contained in the subroutine and follows the last pertinent instruction of the subroutine. Recall that associated with each set of variable remote connections is an F.i symbol used in addressing, and the variable transfer associated with the set has this F.i address. In the coding of a subroutine this variable exit is always coded as a transfer (T or C) with the address F00. The assembly routine then adjusts the F00 to the proper F.i

address. The F.i address for the subroutines follow in sequence the F.i addresses of the problem proper. There are two methods by which substitution instructions may refer to the variable exit of a subroutine, and these methods are illustrated by the examples.

The fixed connection transfer which indicates the entry into the routine and the variable connection transfer (the address of which is established prior to entry) play the role in a subroutine that the control counter plays in the performance of a single instruction of an instruction sequence.

Upon exit from the subroutine (the return of the control to the problem proper) the results from the computation are in the specified D.i addresses from which they may be used in the succeeding code. (In the multiplication the product is in R2 for subsequent use.)

We see that from the way subroutines are used in a problem there is a close analogy to the use of the standard vocabulary of the computer. It is natural then, from the coding viewpoint, to consider the subroutines as a generalization of the computer vocabulary. The subroutine library index cards constitute the vocabulary of subroutines.

Two samples are now given in order to illustrate some actual subroutines. Accompanying the subroutines are duplicates of their library index cards.

Subroutine S-251.1: Random Number Generation

The generation of the random numbers is accomplished by an iterative scheme which is called "The Middle Squaring Process". The process generates successive iterates from a given initial number. The present routine starts with a 38-bigit number and generates 38 bigit iterates. The formation of the $(i+1)^{st}$ iterate from the i^{th} iterate is

$$x_{i+1} = (20\text{-}57)x_i^2 \equiv \boxdot x_i^2$$

That is, the 38 bigit x_i when squared gives a 76 bigit product, x_i^2, and x_{i+1} is comprised of bigits (20-57) of x_i^2, where the 20th bigit corresponds to the 2^{-1} position of x_{i+1}. All iterates are positive. We illustrate the subroutine in conjunction with two boxes, corresponding to the code of the problem, that represent the point of entrance and the point of exit.

The flow diagram is:

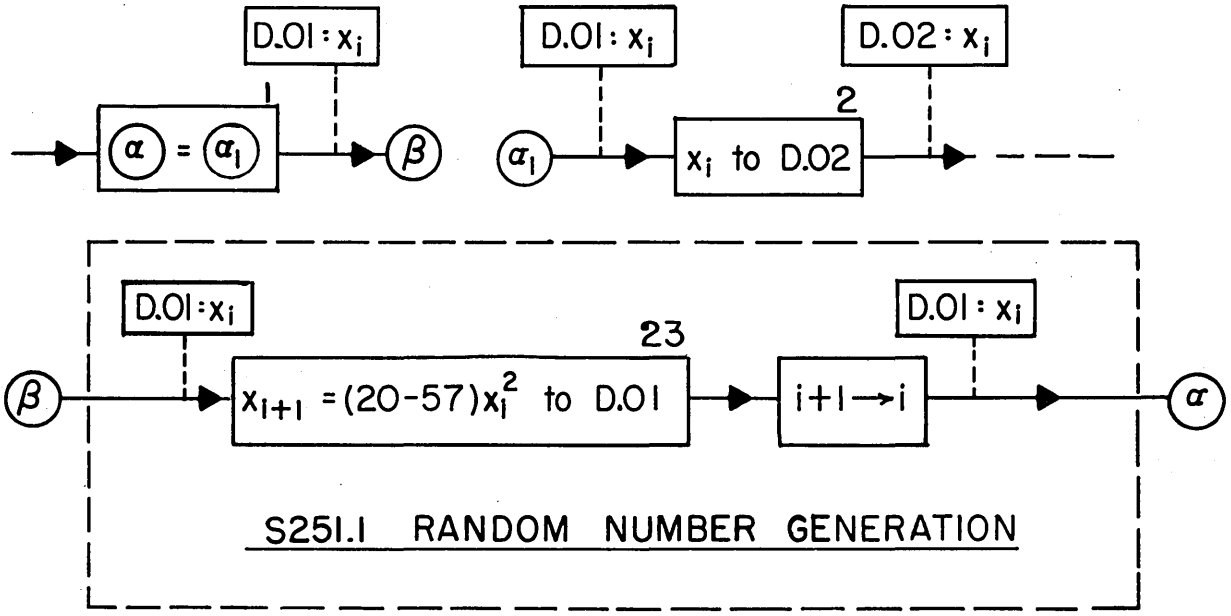


Figure 12

The section of the flow diagram enclosed in the dotted lines would not normally be drawn in complete detail with a problem, but would be drawn as

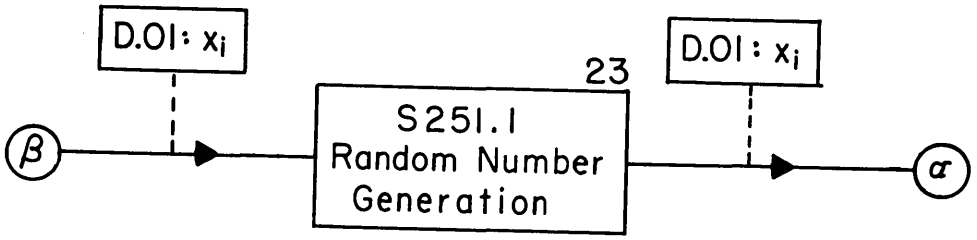


Figure 13

The complete diagram is included now for clarity of coding. Boxes 1 and 2 are coded in two ways to illustrate two alternative methods of entering a subroutine. Box 23 is the subroutine itself. The necessary static storage for the problem (Boxes 1, 2) is:

A.01: CA02,1 CA02,1

No C, B, or 7 storage is needed for the problem. Two D addresses, D.01 and D.02, are used. D.01 contains x_i which was stored in D.01

at a portion of the problem prior to Box 1 and not shown on the flow diagram. We assume for the coding that three sets of variable remote connections exist in the problem proper (they are not shown on the flow diagram). The set of variable remote connections concerned with the subroutine is the fourth set and has the address F.04 associated with it. The coding of Box 1 and Box 2 is:

Box 1

1.	m⟶Ac	A.01	CA021CA021 to R2
2.	HS⟶m	F.04	
3.	T	23,2	

Box 2

1.	m⟶Ac	D.01
2.	A⟶m	D.02

In Box 1 the address for the exit of the subroutine is brought into R2. This address is then substituted into the variable transfer F.04, the exit of the subroutine. Recall that the exit of the subroutine is originally coded with the address F00; however, the assembly routine modifies it to its correct F.i address, which in this case is F.04 (F.01, F.02, and F.03 exist in the problem proper). The fixed connection transfer is to the second instruction of the subroutine (CA23,2) rather than the first instruction. The reason for this is discussed after the code for the subroutine is illustrated.

The second way in which Boxes 1 and 2 may be coded is as follows:

Box 1

1.	m⟶Ac	A.01	CA02,1 CA02,1 to R2
2.	T	23,1	

Box 2

1.	m⟶Ac	D.01
2.	A⟶m	D.02

In Box 1, the address for the exit of the subroutine is brought into R2 and then, without effecting the substitution, the transfer to the subroutine into its first instruction is made. Without further comment let us examine the code of the subroutine proper.

Subroutine Box 1

1.	HS⟶m	F00		
2.	m⟶Q	D01	x_i to R4	
3.	X'	D01	x_i^2 in R2 and R4	
4.	A⟶m	D01		$(0\text{-}39)x_i^2$ to D01
5.	L(1)	001		
6.	m⟶Ac	D01	x_i^2 in R2 and R4	
7.	R(22)	016	$(18\text{-}57)x_i^2$ in R4	
8.	m⟶Ac	800	$(18\text{-}57)\ x_i^2$ in R2	
9.	L(1)	001	$(19\text{-}57)\ x_i^2$ in R2	
A.	DS	000	$x_{i+1} = (20\text{-}57)x_i^2$ in R2	
B.	A⟶m	D01		x_{i+1} to D01
C.	T	F00		

We observe that the first instruction is a half-word substitution to F00; that is, to the exit transfer. This accounts for the two methods of coding Box 1. In the first coding of Box 1, the substitution instruction was performed prior to entry into the subroutine; hence the entry transfer was to the second instruction. In the second coding of Box 1, the instruction word comprising the exit from the subroutine is brought into R2 and then, without making the substitution, the transfer to the subroutine is effected. The exit word, however, still resides in R2 and the initial instruction of the subroutine accomplishes the substitution to establish the desired exit.

Instructions 2 and 3 form x_i^2 as a 78-bigit number. Bigits (20-57) are to be isolated by shifting. Recall that a double precision product has a 0 in the sign position of R4. Instructions 4, 5, and 6 eliminate this 0 so that the subsequent right shift of 22 in Instruction 7 combines the sections of x_i^2 into R4 as $(18\text{-}57)x_i^2$. Instructions 9 and A then complete the process by forming

$$x_{i+1} = \boxdot x_i^2$$

Although the subroutine is indicated as Box 23 on the flow diagram, it is coded as Box 1 in its descriptive code. And, as previously mentioned, the assembly routine makes the necessary adjustments of the box numbers of the subroutines.

As in this subroutine, all subroutines are coded so that the first instruction is

HS⟶m F00

There are, subsequently, two methods of entry into the routine. If the exit to the subroutine is set up prior to entry into the routine, the fixed connection transfer to the subroutine bypasses the first instruction and enters into the second (the subroutine index card should be consulted for exceptions to this rule). Or if the instruction word for the exit to the routine is brought into R2 immediately prior to entry into the routine, the transfer into the routine is to the first instruction of the routine (again consult library index card for exceptions).

We include a copy of the library index card for the subroutine example, in order to illustrate the kinds of information listed. For complete details, the description of the subroutine library filing system should be consulted.

The card reads as follows:

S 251.1 RANDOM NUMBER GENERATION (Middle Squaring)

This routine forms a sequence of 38-bigit pseudo-random numbers by a middle squaring process. The tested base number is sent to D.01. The hexadecimal number 10BBBFA4DE gives 718,627 iterates and then degenerates to 0.

1. Number of operation boxes: 1
2. (a) Number of code words: 6 (dec.); 6 (hex.)
   (b) Number of code words plus B and 7 storage: 6 (dec.); 6 (hex.)
3. D storage needed: D.01
4. Prior to entry the operand must be sent to D.01
5. (a) D.01 and R2 contain new random number upon exit
   (b) Input number is destroyed
6. Entry: Box 1, Instruction 1
   Exit: CA
7. Legal spillage: Instructions 5 and 9

We see that the card first gives a brief description of the routine. Then, in order, it gives:

1. The number of operation boxes, so that the necessary box numbers may be assigned on the flow diagram.
2. (a) The number of code words, so that the words of code in the subroutines may be included in estimates of problem code length.
   (b) The number of code words plus B and 7 storage, so that total word length estimates of problem may be made.
3. D storage needed. This is important, since the D storage shown here must be empty or irrelevant upon entry into routine (except for that D storage which has numbers pertinent to routine).

4. Numbers required for routine, and D storage to which they must be sent prior to entry into routine.
5. (a) D storage in which results are located upon exit from routine.
   (b) Limitations of routine.
6. (a) Instruction into which entry is made. If exit is set up prior to entry into routine, the instruction into which entry is made is one beyond that listed.
   (b) Specifies whether exit is CA or CC, so that corresponding orders may be stored as the exit words in A storage.
7. Legal spillage indicates which instructions in the routine allow numbers to exceed the range $-1 \leq n < 1$. This information is useful in "debugging" procedures and is discussed elsewhere.

Subroutine 116.1: Integer Conversion from Binary to Decimal

This routine is used to convert a binary integer, N, scaled as $N \cdot 2^{-39}$, into its decimal equivalent. The allowable range of N as an integer is $0 \leq N < 10^9$.

The conversion is effected by subtracting the binary equivalents of the successive powers of ten (i.e., 10^8, $10^7 \cdots 10^1$) from N the appropriate number of times and recording the number of subtractions of each power of ten as a decimal digit in its proper position. The inductive process is:

$$\begin{aligned} N_o &= N \\ N_1 &= N_o - a_o 10^8 \\ N_2 &= N_1 - a_1 10^7 \\ &\vdots \\ N_{i+1} &= N_i - a_i 10^{8-i} \\ N_9 &= N_8 - a_8 10^o = 0 \end{aligned}$$

The a_i's are in the range

$$0 \leq a_i \leq 9$$

and each a_i is chosen so that

$$N_i - a_i \cdot 10^{8-i} \geq 0$$

but

$$N_i - (a_i+1)10^{8-i} < 0$$

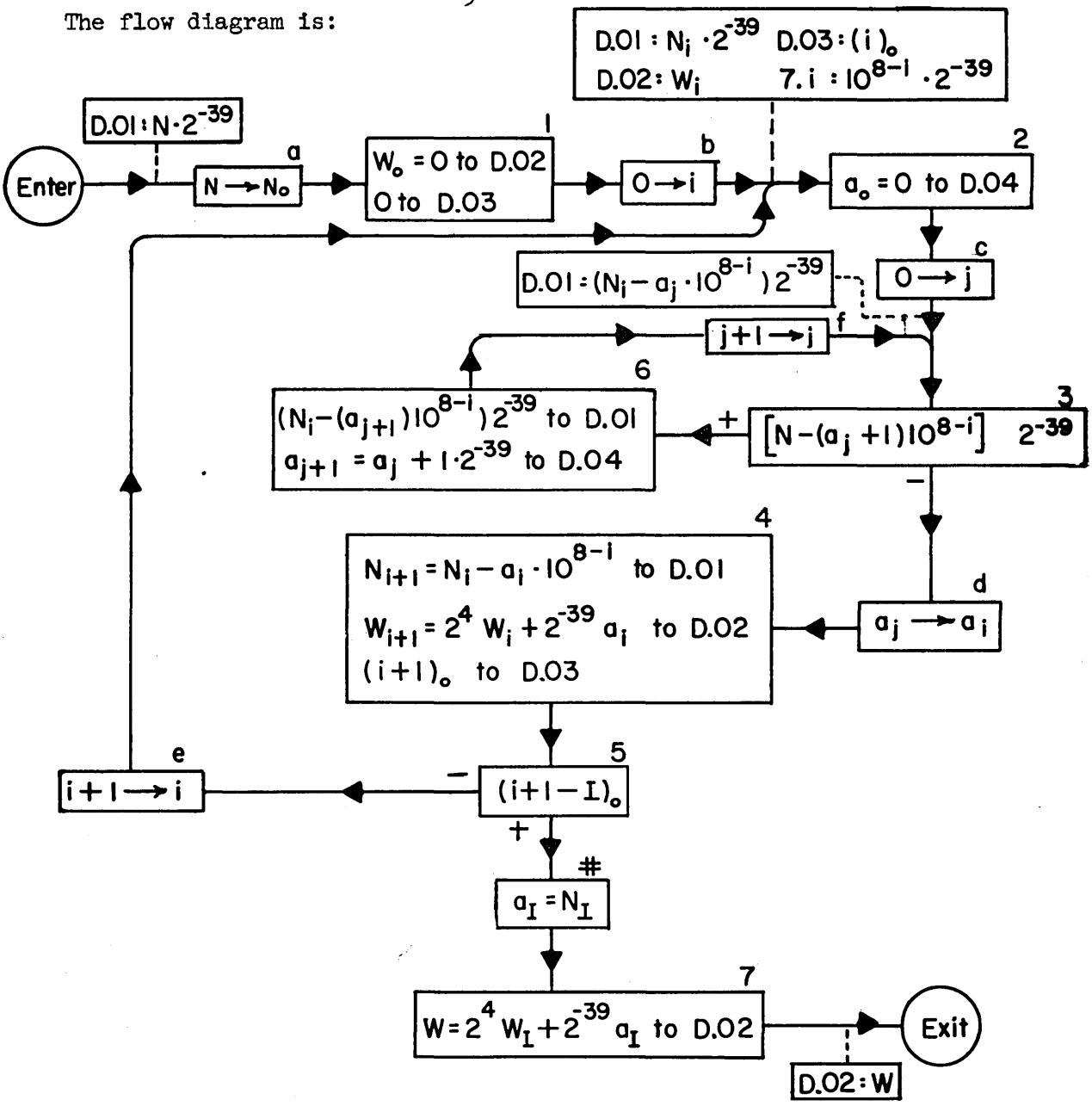
The converted number is then

$$a_o 10^8 + a_1 10^7 + \cdots + a_7 10^1 + a_8$$

Each decimal digit is represented as a tetrad; hence the actual formation of the nine decimal digit integer is described as

$$
\begin{aligned}
w_o &= 0 \\
w_1 &= 2^4 w_o + 2^{-39} \\
&\vdots \\
w_{i+1} &= 2^4 w_i + 2^{-39} \\
&\vdots \\
w_9 &= 2^4 w_8 + 2^{-39} a_8 \\
w &= w_9 = \text{decimal number}
\end{aligned}
$$

The flow diagram is:



S116.1 INTEGER CONVERSION

Figure 14.

The necessary storage is:

B.01:	$(1)_o$	7.01:	$10^8 \cdot 2^{-39}$	**D.01:**	
B.02:	$1 \cdot 2^{-39}$	7.02:	$10^7 \cdot 2^{-39}$	**D.02:**	
		⋮			
B.03:	$I = (8)_o$	7.08:	$10^1 \cdot 2^{-39}$	**D.03:**	
				D.04:	

The flow diagram is drawn as a double induction loop. The primary induction is over the index i and forms

$$N_{i+1} = N_i - a_i 10^{8-i} \quad \text{and}$$

$$w_{i+1} = 2^4 w_i + 2^{-39} a_i$$

The secondary induction is over the index j; and although the induction index is on j, the end result of the induction is the formation of a_i.

Note in the storage of the subroutine that the various powers of ten, $10^{8-i} \cdot 2^{-39}$ are stored in 7 storage. This means that these numbers will be added to the 7 storage of any problem containing the routine, and they will be in eight consecutive locations. It is necessary that the addresses be consecutive, since the appropriate 10^{8-i} are located by an index

$$i \ (=0 \cdots 7)$$

In order that the address 7.i may be formed, a base address 7.01 needs to be stored. This would normally be stored in A storage; since no A storage is allowed in subroutines, the base address is stored in the body of the code.

Although the flow diagram contains seven operation boxes, it is coded as one, as it is desirable to keep the number of boxes of a subroutine to a minimum.

The coding is:

Subroutine Box 1

(box 1)	1.	HS⟶m	F00		
	2.	a⟶Ac	000	0 to R2	
	3.	A⟶m	D.02	$w_o = 0$	$w_o \longrightarrow w_i$ to D.02
	4.	A⟶m	D.03		$0 \longrightarrow (i)_o$ to D.03
(box 2)	5.	m⟶Ac	E.23	$(7.01)_o$ to R2	
	6.	m⟶Ah	D.03	$(7.01+i)_o$ in R2	

	7.	S⟶m	E.0A	7.01+i to (8-19)A
	8.	a⟶Ac	000	$a_o = 0$ to R2
	9.	A⟶m	D.04	$a_o \longrightarrow a_j$ to D.04
(box 3)	A.	m⟶Q	[7.01+i]	$10^{8-i} \cdot 2^{-39}$ to R4
	B.	m⟶Ac	D.01	$(N_i - a_j \cdot 10^{8-i})2^{-39}$ to R2
	C.	m⟶Ah-	800	$(N_i - (a_j+1)10^{8-i})2^{-39}$ in R2
	D.	C	E.1A	
(box 4)	E.	m⟶Ah	800	$N_{i+1} = N_i - a_i 10^{8-i}$ in R2
	F.	A⟶m	D.01	N_{i+1} to D.01
	10.	m⟶Ac	D.02	w_i to R2
	11.	L(4)	004	$2^4 w_i$ in R2
	12.	m⟶Ah	D.04	$w_{i+1} = 2^4 w_i + a_i \cdot 2^{-39}$ in R2
	13.	A⟶m	D.02	w_{i+1} to D.02
	14.	m⟶Ac	D.03	$(i)_o$ to R2
	15.	m⟶Ah	B.01	$(i+1)_o$ in R2
	16.	A⟶m	D.03	$(i+1)_o$ to D.03
(box 5)	17.	m⟶Ah-	B.03	$(i+1-I)_o$ in R2
	18.	C	E.1E	
	19.	T	E.05	
(box 6)	1A.	A⟶m	D.01	$(N_i - (a_j+1)10^{8-i})2^{-39}$
	1B.	m⟶Ac	D.04	a_j to R2 to D.01
	1C.	m⟶Ah	B.02	$a_{j+1} = a_j + 2^{-39}$ in R2
	1D.	T	E.09	
(box 7)	1E.	m⟶Ac	D.02	w_I to R2
	1F.	L(4)	004	$2^4 w_I$ in R2
	20.	m⟶Ah	D.01	$w = 2^4 w_I + a_I \cdot 2^{-39}$
	21.	A⟶m	D.02	w to D.02
	22.	T	F00	
	23.	m⟶Ac	7.01	} "A storage"
	24.	m⟶Ac	7.01	}

In the coding the box numbers as indicated on the flow diagram are indicated with the code for ease of discussion.

In (box 1) the first instruction is the HS⟶m F00 which is in all subroutines. Instruction 5, the first instruction of (box 2) is m⟶Ac E23. Instructions E23 and E24 each contain AA701 and it is

desired that m⟶Ac E23 bring E23 and E24 into R2 as

AA701 AA701.

To accomplish this, E23 and E24 must be assembled as one word and not as parts of two words. E23 has the same parity as the first instruction of the subroutine; hence the subroutine is coded to begin as a left-hand instruction in the assembled code. This positions E23 and E24 in the same word. The descriptive code tape of the subroutine begins with the word

00401

to accomplish this positioning.

Since the coding is done as one box, the transfer instructions which are fixed connectors contain E.i addresses rather than box number addresses. For example, Instruction D, which represents the conditional transfer from (box 3) to (box 6) on the flow diagram, is coded as C E.1A. Instruction E1A then corresponds to the first instruction of (box 6).

The index card for the subroutine is:

S 116.1 INTEGER CONVERSION

This routine converts any binary integer N, scaled as $N \cdot 2^{-39}$, to its decimal equivalent w by a scheme of subtracting powers of ten. N must be in the range $0 \leq N < 10^9$.

1. Number of operation boxes: 1
2. (a) Number of words of code: 19 (dec.); 13 (hex.)
   (b) Number of words of code and B and 7 storage: 30(dec.);1E(hex.)
3. D storage: D.01 D.04
4. Prior to entry D.01 must contain $N \cdot 2^{-39}$
5. (a) w is in D.02 and R2 upon exit
   (b) $N \cdot 2^{-39}$ is destroyed
6. (a) Entry: Box 1, Instruction 1
   (b) Exit: CA
7. Legal spillage: none

B.01:	0000100001	7.01:	0005F5E100	$= 10^8 \cdot 2^{-39}$
B.02:	0000000001	7.02:	0000989680	·
B.03:	0000800008	7.03:	00000F4240	·
		7.04:	00000186A0	·
		7.05:	0000002710	·
		7.06:	00000003E8	·
		7.07:	0000000064	·
		7.08:	000000000A	$= 10^1 \cdot 2^{-39}$

Since the subroutine library is dynamic and continually growing and being improved, no attempt will be made here to catalogue the existing subroutines. However, in any problem being prepared for computation, the subroutine library should be consulted at the time the flow diagram is drawn in order that any desired subroutines might be incorporated into the problem.

The composition of a subroutine descriptive code tape differs slightly from that of a regular problem. The first word (five character) on a subroutine tape is always a Box 1 code word

00001 or 00401,

the latter if the subroutine must start as a left-hand instruction. The first instruction after this code word is always the substitution

FCF00

This is followed by the descriptive code of the first box and all subsequent boxes punched as five-character words, as with a tape of a problem. Immediately following the last instruction of the routine is the code word

00E00

The code word 00C00 is omitted, since no A storage is allowed in a subroutine. Following the word 00E00, the B storage is punched on the tape. (Recall that no C storage is allowed.) The B storage is terminated by two adjacent spaces, and the 7 storage is punched following these two spaces. The last word of the tape (whether it is the end of 7 storage, the end of B storage if no 7 storage is included, or the code word 00E00 if neither B nor 7 storage is needed) is followed by two adjacent spaces. If no B storage is needed and if 7 storage is present, the two adjacent spaces indicating the end of the B storage are nevertheless included immediately following the code word 00E00. Example 11 illustrates sections of three subroutine tapes containing the storage and the appropriate spaces.

Example 11

Each tape begins with the last instruction of the subroutine which for our example is the exit transfer, T F00.

The first subroutine has both B and 7 storage, namely

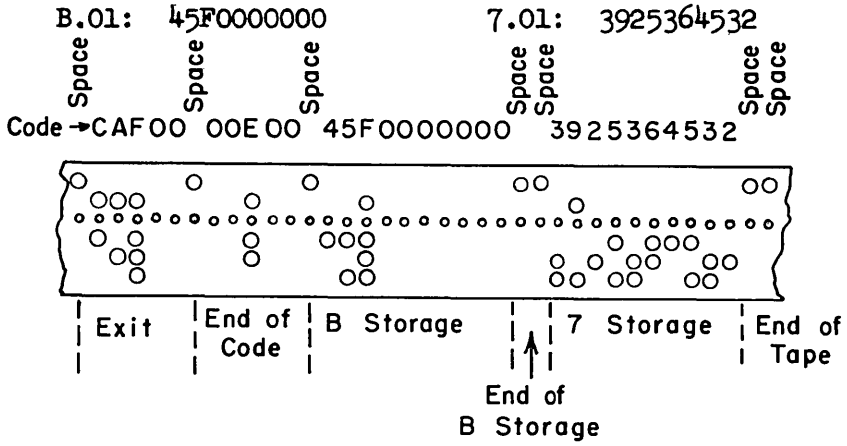


Figure 15.

The second subroutine has only 7 storage:

7.01: F439B7CD32

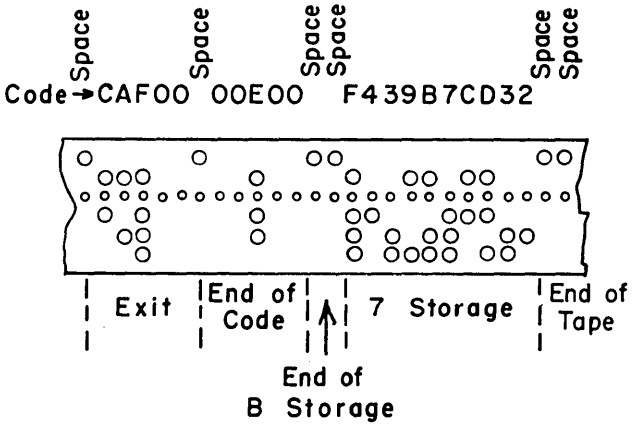


Figure 16.

There is no storage for the third subroutine.

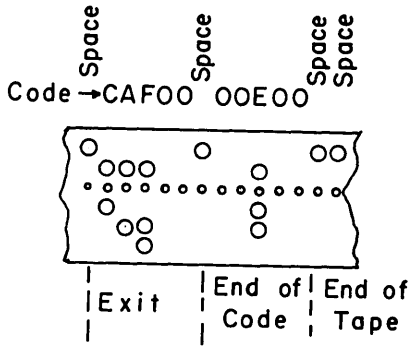


Figure 17.

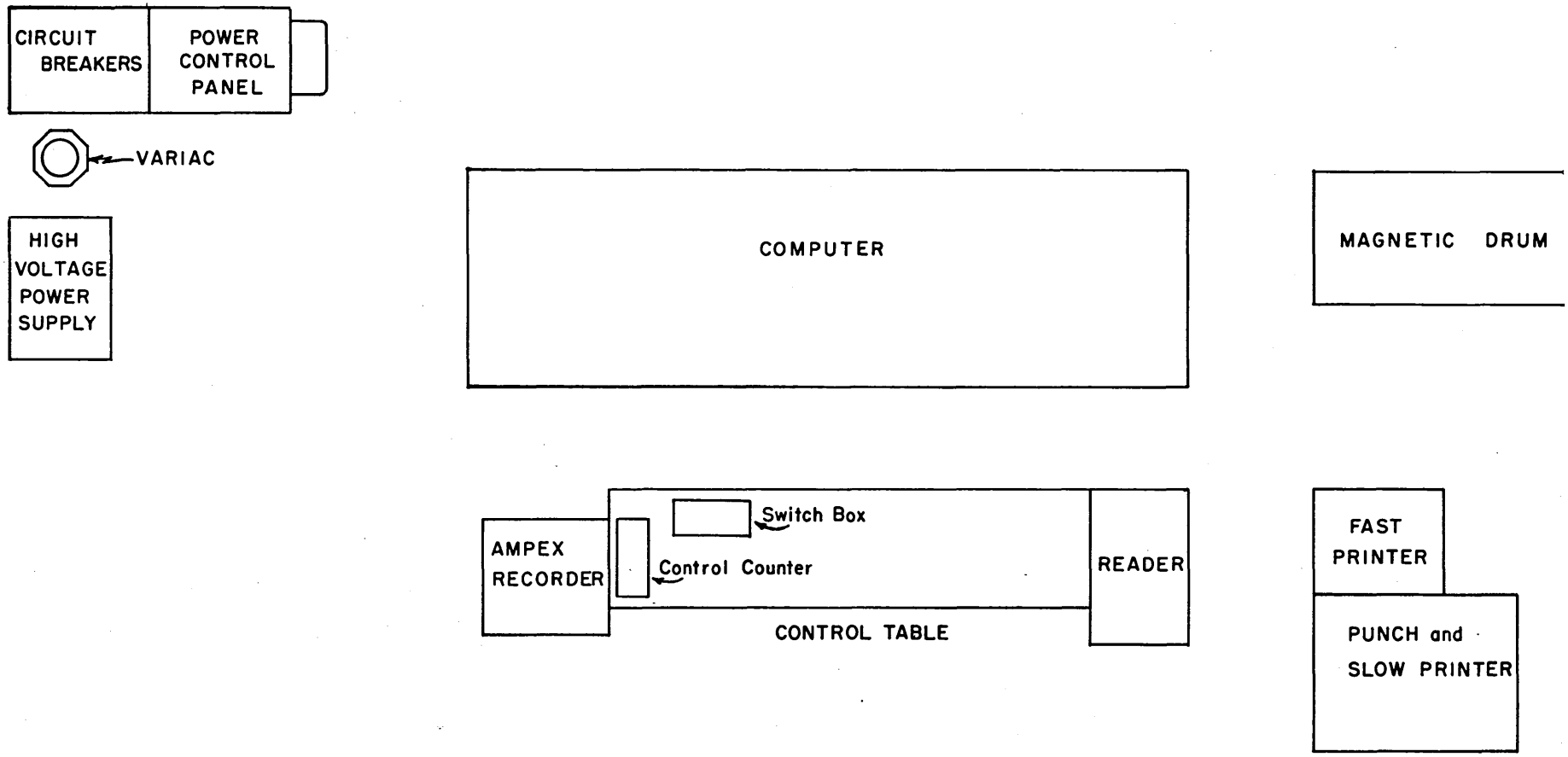
VI. OPERATING PROCEDURES

In this chapter on operating procedures we present the discussion in four sections. First, the functions of the indicator lights and switches of the control panel are discussed so that one has at his disposal the necessary mechanics for operating the computer. The second and main section is the preparation and debugging of a problem. The discussion of the preparation begins with the descriptive code of the problem being complete. The code is carried through its assembly and then the debugging procedures are discussed. The third section returns to the discussion of the computer and it brings out in some detail the role of the various registers. The fourth section contains some miscellaneous information such as the "audio-monitor"; the "memory monitors"; the magnetic tape and Synchroprinter procedures, etc.

In order to give one a better mental picture of the ensuing discussion, Figures 1, 2, and 3 have been included. Figure 1 shows a floor layout of the computer and its auxiliary equipment. The figure is not drawn to scale but it serves to show all of the auxiliary equipment and its position relative to the computing unit. Figures 2 and 3 give a schematic view of the front and back of the computer. These figures show the position of the various registers, the control system, and the electrostatic memory. Now, keeping these three figures in mind, we turn to the operating panel.

The operating panel has been kept in a simplified form for ease of operation. The panel consists of ten display lights and ten switches for setting the counter (shown as the control counter in Figure 1); the memory clear switch (shown in Figure 2); two lights for the function gates (mounted atop the switch box shown in Figure 1); and six operating switches (mounted on the switch box shown in Figure 1) designated in order from left to right as:

1. the load switch
2. the "red" breakpoint switch
3. the "green" breakpoint switch
4. the perform order switch
5. the manual-automatic switch
6. the start next order switch



FLOOR LAYOUT of COMPUTER and AUXILIARY EQUIPMENT

FIG. 1

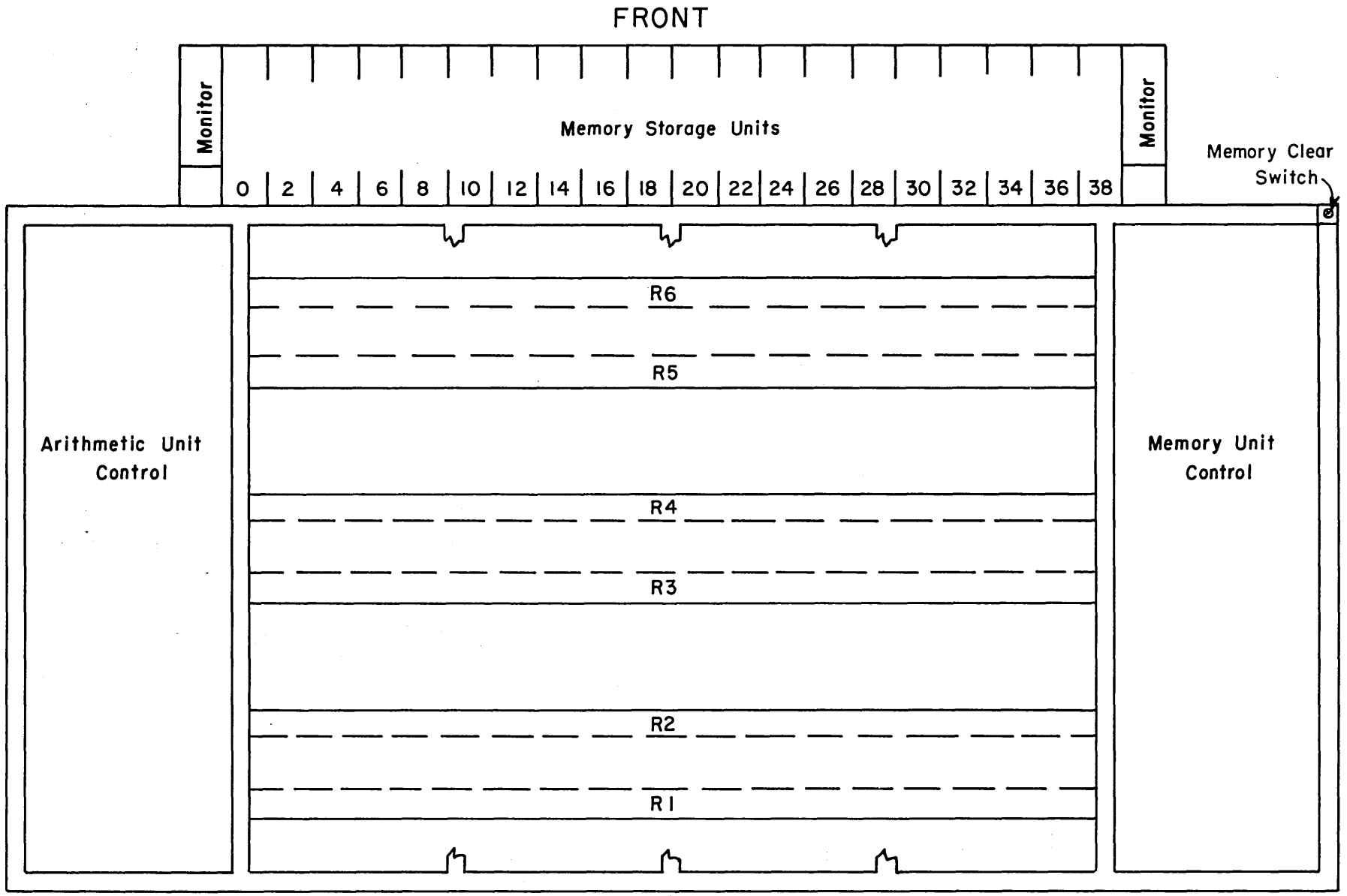

FRONT
Monitor
Memory Storage Units
Monitor
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
Memory Clear Switch
Arithmetic Unit Control
R6
R5
R4
R3
R2
R1
Memory Unit Control

FIG. 2

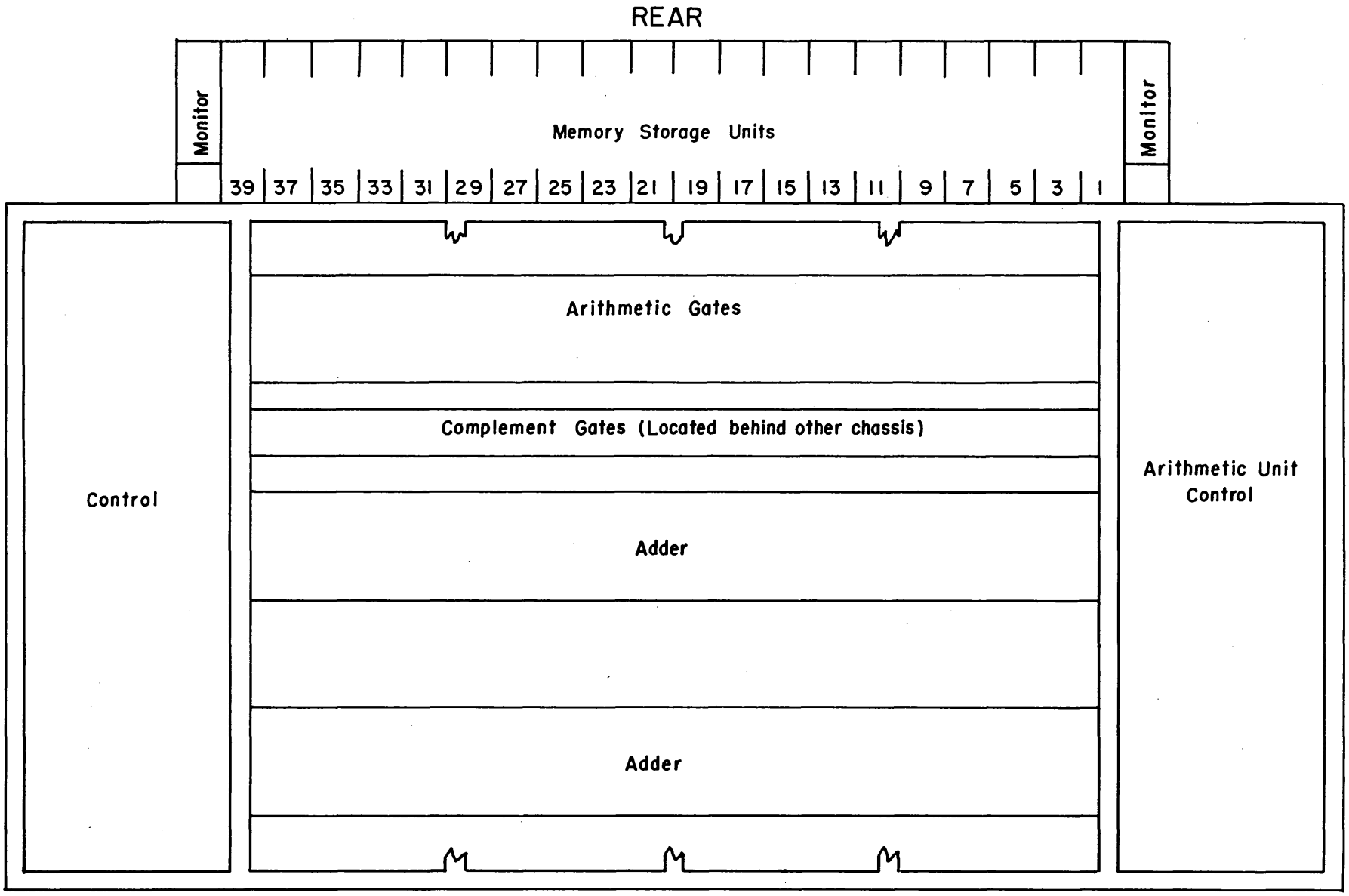

REAR
Monitor
Memory Storage Units
Monitor
39 37 35 33 31 29 27 25 23 21 19 17 15 13 11 9 7 5 3 1
Control
Arithmetic Gates
Complement Gates (Located behind other chassis)
Adder
Adder
Arithmetic Unit Control

FIG. 3

The display neons for the various registers have not been brought out to the panel but are physically located with their register. They are readily visible from the operating panel table. In line with this, the monitor tubes for visible memory display are mounted in the memory rack rather than on the operating panel (See Figures 2 and 3).

The control counter display lights and selector switches are laid out on a panel as shown in Figure 4.

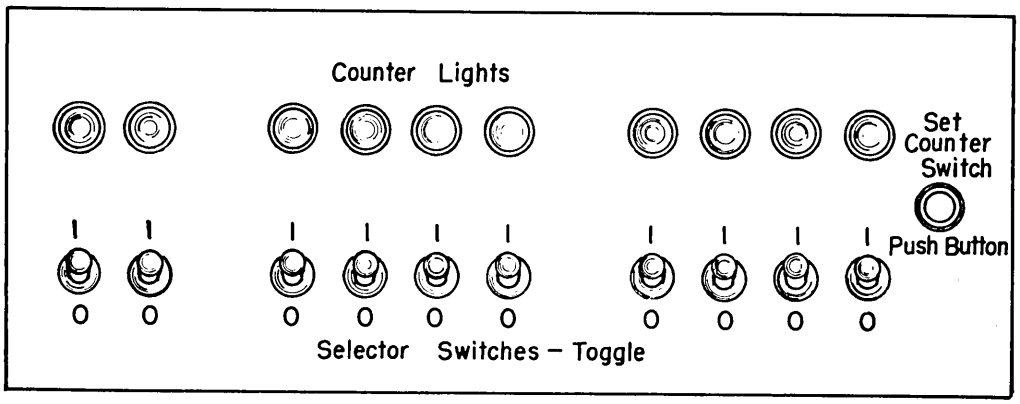


Figure 4.

The control counter is the mechanism used to sequence the instruction-words. The control counter normally contains the address of the forth-coming instruction word to be brought into the R6 (control) register. Since the control counter handles addresses, it counts from 000 to 3FF, which requires a ten-stage counter. Inasmuch as the counter is the sequencing mechanism, we easily see how transfer instructions are accomplished, namely that the address of the transfer instruction is sent to the control counter. (The right-left selection is done through the function gates, which are discussed presently.) If the computer is stopped, the operator may manually effect a transfer of the control to any address by using the selector switches. The control counter (hence the control) is set to any desired address by setting the selector switches to the address and then depressing the "set counter" switch. The control counter lights indicate the address to which the counter is set.

In addition to being the control sequencing mechanism, the control counter is used in conjunction with the magnetic drum instructions. It indicates in sequence the fifty memory addresses associated with the

instruction. The counter is also used in the loading process; here the counter indicates the address of the memory to which the next word from the reader is sent. We discuss the loading process presently.

Prior to using the computer, the operator usually clears (sets to all zeros) the memory of any previous code or data. The memory is cleared to zeros by depressing the "memory clear" switch located on the front section of the computer in the upper right-hand corner of the arithmetic unit frame. This switch is separated from the operating panel so that it will not be pushed inadvertently during the course of a computation. Its location is shown in Figure 2.

The two function gate lights are mounted on a panel immediately above the six operating switches. These are display lights for the function gates, a set of gates which allows, in turn, each instruction of the word in R6 to be connected into the control circuitry in order to be performed. The function gate lights indicate which instruction in R6 is connected into the control circuitry. When the left-hand light is on, the left-hand instruction in R6 is connected into the control circuitry and, similarly, the right-hand light corresponds to the right-hand instruction. In general, if the computer is stopped and an instruction pair is in R6, the instruction corresponding to the function gate light setting has already been performed by the control. The function gate lights are shown in Figure 5.

In a transfer instruction, the control selects the left or right side by opening the corresponding function gates. There is no switch for setting the gates manually, but as we shall see this is not necessary.

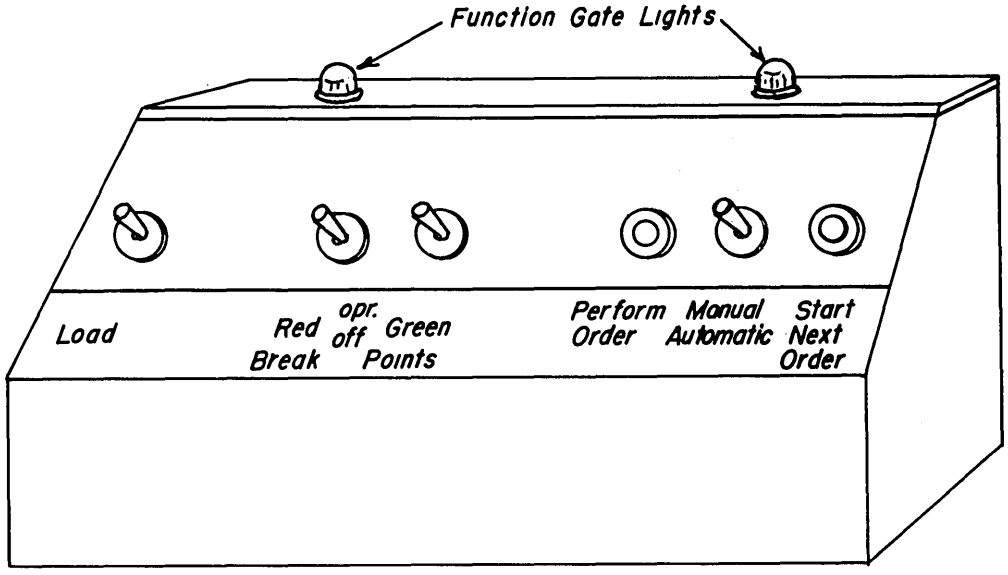


Figure 5.

We now turn to the six operating switches shown in Figure 5 and discuss first the "load" switch and the loading process. Prior to loading a tape into the memory, one first clears the memory to zeros by depressing the memory clear switch and then sets the tape in the photo-electric reader. When a tape of data is punched for use in the computer, the first word of the data should be preceded by five or six inches of blank tape (zeros). These zeros act as a leader for the tape. To place the tape in the reader, the lid of the reader is raised. Then the tape is inserted so that the leader is over the drive cylinder, yet no pertinent characters are beyond the reading holes. The tape must be placed in the reader so that the space holes (fifth holes) on the tape are nearest the hinged side of the lid. A sample tape is attached to the reader to avoid mistakes of this type. After the tape is inserted, the lid of the reader is closed. One should make certain that the lid latches when it is closed to assure proper operation.

After the tape is inserted, the control counter is set to the desired initial address for loading. In loading, although it is only necessary to set the selector switches of the counter, it is recommended that the set counter switch be depressed so that one can check the counter setting by the display lights as well as the selector switches. When the desired address is set into the counter, the load switch is set to the "up" position and the loading commences. The words from the tape are transmitted into successive memory positions beginning at the address set into the control counter.

After the tape has been loaded into the memory, the load switch must be set to the "down" position. The computer will not operate if the load switch is not reset. The loading is terminated when two adjacent spaces on the tape being loaded are encountered by the reader; hence, any tape that is to be loaded into the memory must end with at least two adjacent spaces.

As the tape is loading into the memory, each word on the tape is transmitted into the R5 register, and from there into the memory. This fact allows a method of checking that the photoelectric reader circuitry is transmitting the information correctly from the tape. During the loading, a sum of the words from the tape is formed in R2. The first time

that a tape is loaded, the sum as shown in R2 should be recorded. It can be verified by immediately reloading the tape. Once a correct sum of the tape has been recorded, the sum given by all subsequent loadings must agree with the known correct sum. If it does not agree, there is a computer malfunction. The correct sum should be recorded on the box in which the tape is permanently stored. Remember, however, that a correct sum in R2 at the completion of the loading does not guarantee that the information is correct in the memory; it only says that the reader and its associated equipment operated properly. The contents of the memory are checked by a summing routine that must be incorporated in all problems. It is discussed later.

It is now worth noting several things that occur when the load switch is set to the "up" position; namely, the R6, R5, and R2 registers first clear to zeros. The R6 register remains zeros throughout the loading. At the completion of the loading, R2 contains the sum of the tape, R5 contains the last word loaded from the tape, and R6 is zeros. Note that the loading process does not affect the contents of the R4 register. At the completion of the loading, the control counter automatically resets to the original address.

The "manual-automatic" switch, the "start-next-order" switch, and the "perform-order" switch are those directly concerned with the running of the computer. We now discuss them.

The manual-automatic switch allows the computer to be operated so that it either stops upon the completion of each instruction or performs an entire instruction sequence without stopping. If the manual-automatic switch is in the "manual" position when the control performs an instruction, the computer stops upon the completion of the instruction. If the manual-automatic switch is in the "automatic" position when the control performs an instruction, upon the completion of the instruction the control proceeds to the next instruction in the sequence to perform it, and so on, through the entire code sequence.

The start-next-order switch is normally used to start the computer. Recall that if the computer is not running the function gate light indicates which side of the instruction pair is connected into the control circuitry. Depressing the start-next-order switch causes the next instruction in sequence to be performed. That is, if the start-next-order switch is depressed when the left-hand function-gate light

is on, the function gates are set for the right-hand instruction in R6; the function gate lights change and the right-hand instruction in R6 is performed by the control. If the start-next-order switch is depressed when the right-hand function-gate light is on, the control brings the instruction word located at the address specified in the control counter into R6. The function gates and lights have meanwhile switched to the left-hand side of R6 and then the left-hand instruction of the new word in R6 is performed by the control. The control counter is advanced by one.

The perform-order switch is somewhat similar to the start-next-order switch in that it causes the control to execute an instruction contained in R6. However, depressing the perform-order switch causes that instruction (indicated by the lighted function gate) connected into the control circuitry to be performed rather than causing the next instruction in sequence to be performed. The perform-order switch takes on added significance in connection with the breakpoint switches and is discussed further with them.

Returning to the manual-automatic switch, we see that the "manual-automatic" settings apply to either the start-next-order or perform-order switches. If on "manual", the start-next-order switch allows one to proceed through the code sequence an instruction at a time, while the perform-order switch allows one to repeat an instruction as many times as is desired. If on "automatic", depressing either the start-next-order switch or the perform-order switch allows the control to proceed automatically through the code sequence. The latter, however, causes the control to perform the instruction previously connected into the control circuitry before proceeding through the instruction sequence.

The breakpoint switches allow one to insert conditional stops into a code by setting either the first or fifth bigit of an order to zero. Since all orders are composed of letter pairs (AA, BA, DD, etc.) the first and fifth bigits are normally one. Setting the first bigit of an order to zero corresponds to the insertion of a red breakpoint and setting the fifth bigit to zero, a green breakpoint. The conditional stop arises from having a breakpoint switch in the "up" or "down" position. If either the red or green breakpoint switch is in the "up" (on) position and the control brings into R6 an instruction which contains

the corresponding breakpoint, the control stops the computer before the the instruction is performed. The breakpointed instruction is, however, connected into the control circuitry as indicated by the function-gate light setting. If either of the switches is in the "down" (off) position when the control brings into R6 an instruction with a breakpoint corresponding to the "down" switch, the control performs the instruction as though it contained no breakpoint.

The perform-order switch is used in conjunction with the breakpoints because depressing the perform-order switch causes the instruction connected into the control circuitry to be performed even though this instruction may contain breakpoints. If the control stops on a breakpointed instruction, it stops before the instruction is executed; hence the perform-order switch is the natural way of resuming operation. If the control is stopped at an instruction with a breakpoint and the start-next-order switch is depressed, the instruction containing the breakpoint is skipped (not performed) as the start-next-order switch executes the next instruction in sequence rather than the one already connected into the control circuitry.

With a knowledge of the operating switches at our disposal we now turn our attention to the code assembly and "debugging".

Recall that the absolute code is prepared in the computer by the assembly routine from the descriptive code tapes. These tapes are the problem and constant tape, and the subroutines tape or tapes. The assembly routine is an example of the category of codings called "helper-routines". A helper-routine is a routine, not incorporated directly as a part of the problem, which is used as an aid in the preparation, the running or the analyses of a problem on the computer. A library of helper-routines has been compiled much in the fashion of the subroutine library. Rather than give an elaborate discussion of these routines we refer the reader to the helper-routine library file, and we mention them only as their need arises in the ensuing discussion.

The first step in the assembly of a code is the loading of the code assembly helper-routine. (This routine is appropriately named "The Coder".) The tape and necessary explanations for the assembly routine are obtained from the library. The code is transmitted into the memory beginning at the desired address (specified by the explanatory

remarks) via the load process which is: the memory is cleared to zeros by depressing the memory-clear switch; the tape is set into the reader; the control counter selector switches are set to the desired starting address; and then the load switch is set to the "up" (load) position. After the assembly tape is loaded, the load switch is set to the "down" (off) position and the sum in R2 is checked against the sum as recorded on the assembly code tape box.

After the assembly routine is loaded and the sum is checked, the processing of the descriptive code tape begins. The descriptive code tape is placed in the photo-electric reader so that it is in position to be read into the computer by the assembly routine. The computer is started in operation by first setting the desired starting address into the control counter; second, setting the manual-automatic switch to the "automatic" position; and third, depressing the start-next-order switch to activate the control. The desired starting address is often contained in the control counter, since after loading the counter contains the initial load address.

After loading, to start the computer the right function-gate light must be on. Depressing the start-next-order switch then brings in to R6 the instruction word specified by the address in the control counter, and the control proceeds executing the instructions in sequence. If the left function-gate light is on, at the completion of the loading one may switch the function gates by depressing the start-next-order switch. R6 is cleared to zeros by the loading; hence the switching of the function gates does not cause any action as there is no instruction in R6.

The first group of instructions of the assembly code comprises a summing routine which forms a sum of the memory contents and checks this sum against the sum as left in R2 from the loading process. (Any problem which is to be run on the computer should contain such a summing routine.) If the sums do not agree, the computer stops at a programmed stop, since disagreement of the sums implies a computer malfunction. If the sums agree, the control proceeds automatically and the data from the descriptive tape is read and processed through the assembly routine. At the completion of the reading of the descriptive code tape, the control comes to a coded stop in order that the subroutines tape may be inserted into the reader. After this tape is inserted, depressing the

start-next-order switch causes the assembly of the absolute code to be carried to completion. During the processing of the code, a code listing (see Chapter V, pp. 232 ff.) is carried out. Upon the completion of the assembly, the absolute code may either be recorded onto magnetic tape or punched onto paper tape for subsequent use. The choice of the medium for recording the absolute code is made by selecting the appropriate assembly routine code tape, as there is one code which contains as a subroutine a magnetic tape recording, and another which contains a tape punch subroutine. However, in either situation the particular auxiliary equipment should be readied prior to the start of the assembly process.

After the assembly of the absolute code is completed with either the record on magnetic tape or a punched paper tape (for what follows we assume that the absolute code is on magnetic tape), "debugging" of the assembled code begins.

As a person gains experience in coding, he soon realizes that despite the great care exercised in the formulation and coding of a problem, errors are apt to occur. Before a problem can be run any existing errors must be detected and corrected. The process of eliminating errors from the mathematical formulation and the coding of a problem is called "debugging". As a person becomes familiar with coding and the computer, he will naturally develop his own "debugging" habits. The purpose here then, rather than to specify a rigid set of rules, is to discuss a general procedure that will assist a person in developing desirable debugging patterns.

In a problem of any complexity, the hunting for and detection of errors completely apart from the computer is a very difficult, if not impossible, task. In order to make the task of error hunting a tractable one, the computer is utilized.

Clearly, one approach for using the computer in debugging is to run the problem as though it contained no errors (this is often done with small problems). If there are no errors, this indeed is the fastest approach to debugging. However, if errors are present, the answers indicated upon the completion of the problem, if the control was even able to proceed to the end, would be incorrect; and one would have no idea where or why the errors occurred, so that such running time (which might be rather lengthy) would not be particularly useful in localizing any errors.

Another approach would be to perform each instruction in the code sequence on manual operation and to record the result of each operation so that it could be verified by hand methods. Such an approach would certainly find all existing errors, but the amount of computer time involved in such a debugging method is much greater than it need be.

The recommended approach combines the two extremes. The code of the problem is divided into several sections and the control performs each of these sections automatically, stopping upon the completion of each one. The division of the code of a problem into these sections is accomplished by inserting conditional stops into the code by means of breakpoints. These stops are inserted at locations in the code where the results of pertinent computation are available. Enough of these stops should be inserted so that sufficient data of the problem are recorded to allow one to perform a hand check if necessary. The control then performs automatically one of the short sections of code and stops at the designated instruction. The pertinent data from the preceding computation are recorded, and then the computer is restarted and the control performs the next code section, and so on, until all of the desired data are accumulated. This occurs when the control has proceeded through all of the code sequence at least once, or when it is observed that some of the data are in error. In either case, the problem is removed from the computer and the data are studied and verified.

If the accumulated data indicate the existence of errors, any particular error may be isolated to one of the short code sections by making a hand check of the results and observing in which section the error first appears. Once an error has been isolated to a section of code, that section of code is checked visually to see if the cause of the error may be easily located. If it cannot, that section of the code in which the error occurs is further subdivided and the problem is returned to the computer where the offending section is examined in greater detail in order to pin down the error. As soon as the error is located, it is corrected and then further debugging may proceed. This process is continued until all errors are removed from the coding, at which time the problem is ready to be run. We now discuss these matters in more detail.

After the absolute code is assembled and placed on magnetic tape, the problem is removed from the computer in preparation for debugging. This preparation involves a visual check of the code listing to detect any obvious errors, either from the coding or from the assembly process. After the listing is checked, the code is divided into sections for breakpoints. The breakpoints are to be inserted into orders of instructions where pertinent data are available in the arithmetic unit. The actual insertion of the breakpoints into the desired instructions in the assembled code may be accomplished by a Breakpoint Insertion helper-routine. One needs to specify to this helper-routine the address of the instruction receiving the breakpoint and whether the breakpoint to be inserted is "red" or "green". The details for accomplishing this are discussed in the helper-routine file.

There is an alternative method for inserting breakpoints which is perhaps more desirable than the one just outlined. This alternative is to decide upon the disposition of the breakpoints during the preparation of the descriptive code and to punch the orders on the descriptive code tape with the breakpoints inserted. The assembly routine accepts and modifies properly instructions whose orders contain breakpoints. As an example, an instruction m⟶Ac B.01, if it were to contain a "red" breakpoint would be punched as 2AB01 rather than AAB01. Similarly, ÷ D01 with a "green" breakpoint would be punched as D5D01 rather than DDD01.

If the breakpoints are included during the descriptive coding, they exist on the magnetic tape record of the absolute code. If they are inserted by the Breakpoint Insertion routine, the absolute code from the magnetic tape must be called into the computer; the breakpoints are then inserted by the Insertion helper-routine, and a subsequent record of the code with breakpoints is made onto the magnetic tape. The calling from and recording onto magnetic tape is accomplished by Magnetic Tape helper-routines, two of which were illustrated in Problem 12 of Chapter II. As soon as the breakpoints are inserted, one begins the debugging proper.

The most effective way of observing the data at the various breakpoints is to have the desired data printed. To do this, one again calls on a helper-routine. The particular routine used here is in a class of

interpretive helper-routines and is the so-called Breakpoint Monitor helper-routine.

An interpretive routine is any routine which interprets and causes to be performed any desired instruction sequence which is residing in the memory. Such routines act in a sense as a generalized control.

During the process of interpreting and performing an instruction sequence, an interpretive routine may perform many other functions, the extent of which is limited only by the capacity of the memory of the computer and the ingenuity of the person preparing such routines.

For the Breakpoint Monitor routine the desired interpretation is a very simple one, namely whether an instruction contains a breakpoint. For an order containing a breakpoint, the interpretive routine first causes the instruction to be performed and then the following data are printed as four words:

Word 1: The address at which the instruction containing the breakpoint resides, and the instruction itself.

Word 2: The contents of the R4 register

Word 3: The contents of the R2 register.

Word 4: The contents of the memory at the address specified in the instruction.

Words 2 and 3 give the contents after the instruction is performed and Word 4 gives the contents before the instruction is performed. Note, then, that the breakpoints are inserted into instructions which when performed give the desired data in the arithmetic unit. R2 or R4 contain the result from any arithmetic operation while the appropriate memory location contains one of the two operators entering into the operation.

There are many other interpretative routines similar to the Breakpoint Monitor (it was chosen only as a convenient example) and one should check the library file to ascertain which of these routines is best suited for his specific purpose.

Sometimes breakpoints are used to check that the control reaches a certain instruction in the problem and for this the numbers printed from the various registers may be unimportant for debugging; hence only the first word printed in the listing would have relevance.

In the Breakpoint Monitor routine, as in similar routines, one has the option of having the data printed as either decimal numbers or as hexadecimal numbers. The first word, i.e., the address and the instruction, is always printed as a hexadecimal number, since it would appear as nonsense as a decimal number.

To utilize the Breakpoint Monitor routine, one inserts the desired breakpoints into the absolute code. The absolute code and the Breakpoint Monitor routine are then loaded into the memory. Note that, since both routines are in the memory, the Breakpoint Monitor routine must be loaded into a set of addresses which are not relevant to the code being debugged. Breakpoint Monitor routines are coded beginning at a variety of addresses so that this is usually possible without undue red tape. If, however, one has an assembled code to be debugged which fills the memory, he has recourse to a generalized monitoring routine which utilizes the magnetic drum. It is not, however, discussed here.

The first step of the monitoring process is to specify the initial address of the code to the monitor routine (for details see the helper-routine library file). The control counter is set to the initial address of the monitor routine and then the computer is started. The data for the debugging is printed by the Synchroprinter, four words (discussed above) to a line.

As soon as one has collected a sufficient amount of data, the problem is removed from the computer and examined at leisure away from the computer.

It may happen that the breakpoints are not reached in the expected sequence, or even that the first one is not reached. We defer the discussion of the procedure to be followed when this happens.

So now, assuming that the breakpoints were reached, we have the data which is now examined to determine whether or not the numbers listed are the desired numbers. First, a cursory examination is made for any obvious errors. For example, a number known to be always positive may have been computed and printed as a negative number. Or perhaps the orders of magnitude of the numbers of the computaton are known and a visual check suffices to determine this.

If the cursory check does not indicate any troubles, a hand computation is made using the same data as for the listing. The hand check may often use shortcuts in that some of the numbers computed are known; e.g., the values for sin x, $\sqrt[3]{x}$, etc. may be found in tables. The comparison should agree except for truncation and round-off differences. Sometimes approximate values suffice for checking purposes. If no errors have occurred, the debugging of the portion of the code for which the data was obtained is complete. If an error is detected from the cursory examination one must set about isolating it to one of the sections of code between breakpoints. At first, one attempts to isolate the error by a visual check of the numbers leading to the error, and if this fails a hand check of the results in the region of suspect will isolate it.

Once an error is isolated to a particular section of code, the instructions in that section are examined in detail to see if the cause of the error may be observed. If it is found, that trouble is over. If it is not observed, one may divide the section of code by further breakpoints, so that the section may be monitored in greater detail upon returning to the computer. However, at this point, if the section of instructions is fairly short, as it should be, rather than doing further breakpoint monitoring one has recourse to another helper-routine for debugging, called the Auto-Monitor routine. It is discussed presently.

If the first error detected does not alter subsequent results too drastically, the programmer continues his checking process for other errors so that before returning to the computer as many errors as practicable are detected and corrected.

Since the absolute code of the problem exists only on magnetic tape one makes the actual corrections at the next session with the computer. However, prior to this a permanent written record is made of each error as it is detected. This record should contain at least the following:

1. The addresses of the incorrect words.
2. The incorrect words as they appear on the magnetic tape.
3. The number of the particular magnetic record on which they appear incorrectly (as will be seen later each record of an absolute code is on a numbered section of a spool of magnetic tape).
4. The correct words as they are to be inserted. And if any additional words are added, the addresses at which they are added.

Then after one has returned to the computer and made the corrections and recorded the corrected absolute code onto magnetic tape, the following information is added to the record.

5. The date on which the correction is made.
6. The number of the magnetic tape section on which the corted code resides.

In addition to the six items mentioned, any comments which the programmer feels are pertinent to the corrections should also be included.

There are, in general, two kinds of corrections that need to be made. The first is the easy kind which can be corrected by changing only those words in error without having to add additional coding. This kind of correction causes relatively few headaches. The second kind are those corrections where the number of words necessary to make the correction exceeds the number of words in error. In short, additional coding must be added. So we have found one of the ticklish parts of the debugging, and unfortunately many of the errors encountered are of this kind. For clarity we give an example of such a correction and indicate how it is carried out.

An error is found in the sequence of code words beginning, say, at address 050. The faulty coding is

050.	m⟶Q	271	X	272
051.	X	273	A⟶m	274
052.	---	---	-----	---

The sequence is supposed to form xyz and store it at

274: xyz

where x, y, and z reside in locations 271, 272, and 273, respectively. Now as the result of Instruction 50', the product xy is in R2. Instruction 51 is incorrectly a multiply instruction because the multiplier xy has not yet been placed in R4. Since all of the instructions in the sequence are needed, there is clearly no place to insert the necessary L(40) instruction to send the number xy from R2 to R4, or if it is not desirable to use L(40), two instructions A⟶m 275, m⟶Q 275 are needed where 275 is an available location at this time.

In order to make the correction one must have available somewhere in the memory 1 1/2 consecutive words. Assume that such space is available beginning at address 379. The corrections to be inserted are:

050.	m⟶Q	271	T	379
051.	X	273	A⟶m	274
052.				
379.	X	272	L(40)	028
37A.	T	051		

The right-hand side of Word 50 is changed to a transfer to 379. The first instruction of 379 performs the multiplication formerly done in 50'. 379' then shifts xy from the R2 register to the R4 register, so that it is in proper position as a multiplier. The next instruction then sends the control back to 051 where the multiplication by z is now correctly performed.

An alternative scheme of inserting the correction is to revert to the descriptive coding and actually recode in descriptive coding the operation box in which the error occurs. A corrected tape for this box is then punched. By making use of an 800XX symbol (a trivial change) incorporated in the "box number" code word on the descriptive tape, the assembly process may be stopped prior to the assembly of the code of any box and the code for new boxes or corrected boxes may be inserted. The entire problem is then reassembled by the assembly routine with the desired insertions of now or corrected boxes. At first glance the recoding of a box and the reassembly of the entire problem may seem rather a drastic way of eliminating an error; however, experience has shown that one of the most fruitful sources of errors in coding arises from the insertion of corrections for previous errors, and this recoding and reassembly virtually does away with such errors. One has only to examine and work with a highly complex problem to understand this. It should be mentioned that the reassembly process is quite easy and rapid.

When one returns to the computer to insert the corrections, he reassembles the code if the latter scheme is adopted. If the former is adopted, he has previously punched small tapes containing the desired corrections. Then after the absolute code is read into the memory, these corrections are loaded into the desired locations. Each sequential group of corrections consists of one tape; hence several such tapes are often needed. Several groups of corrections may, however, all

be placed on one tape with double spaces on the tape separating the various groups. For example, the correction of the example discussed above would consist of two groups. The first consists of Word 050 which is

EB271CA379

followed with a double space. Immediately following the double space the words beginning at 379 would be punched. They are:

DA272DE028
CA05100000

which is also followed by a double space. The correction tape is then loaded into the desired locations, namely addresses 050 and 379. When all of the corrections have been inserted, the problem is again recorded on magnetic tape so that an absolute code containing the corrections is available on tape. Note that all of these magnetic records discussed are distinct. That is, one should not destroy previous records of the problem when making a new one, and certainly not the immediately preceding record.

We are now ready to resume debugging, with the corrected code. We do this by first returning to our original breakpoint monitor scheme and printing the data for all of the breakpoints that had previously been listed. This is done to make certain that none of the changes and insertions in the code has molested any part of the code which was previously correct. In addition, the data pertaining to the corrections are printed. We have left from the previous debugging session those errors which were not found while off of the machine. If the method of inserting more breakpoints is used one has only to let the data be printed. However as previously mentioned, it is often advisable to resort to an Auto-Monitor helper-routine.

The Auto-Monitor routine is an interpretive routine which allows the results of each instruction to be printed. The data printed for each instruction are identical to those for the Breakpoint Monitor routine. When one comes to the section of code in which the error exists, he switches to the Auto-Monitor routine and lists the results of the computation for all of the instructions in that section. To switch from the Breakpoint Monitor to the Auto-Monitor routine one

loads the Auto-Monitor routine into the memory and specifies to it the desired starting address for monitoring. One should consult the library file for specific operating instructions. Upon the completion of the desired auto-monitoring, one may revert to the Breakpoint Monitor routine.

The Auto-Monitor routine is recommended to track down the error of the kind previously mentioned in which no breakpoints were ever reached or else reached in the wrong sequence, by the Breakpoint Monitor routine. One begins auto-monitoring at the start of the problem (or at the point of "no return"). This soon leads to the source of the trouble.

It is worth noting here that, since the Auto-Monitor and Breakpoint Monitor routines have a similar function, they may actually be incorporated as one routine where one need only make minor adjustments in order to switch from one to the other.

There are other helper-routines which one has as an aid to debugging other than the monitoring routines. We mention only a few of them in passing.

There is a Scaling Check routine which examines the results of all operations to see that numbers do not exceed the allowable range of $-1 \leq x < 1$.

There are various address and instruction search routines which scan the code and pick out all instructions containing any specified address, or pick out all instructions containing any specified order, or pick out any specified instruction.

Routines exist for comparing the contents of any magnetic tape record either against any other, or the contents of the memory, or the contents of the magnetic drum.

There are address altering routines which modify the addresses of any section of code in any manner desired.

Graph plotting routines are available for plotting data to see if they look reasonable.

There are routines which allow all operations on the computer to be done in duplicate in the event that one suspects a computer malfunction as the source of an error. Normally our standard test routines disclose

the garden variety of computer errors, but on rare occasions an infrequent intermittent may depend on particular numbers. In this instance there is some point to using these "duplicating" routines.

Many routines which cannot be used directly in debugging may still be of service. These are routines that can compute various functions and tabulate the results so that they can be compared with results in the problem being debugged.

The scope of helper-routines is too great to enumerate in detail here. However, it is suggested that, prior to the debugging of any code, the programmer should become familiar with helper-routines and their function as an aid to debugging.

We have thus completed the debugging of the absolute code. It should be mentioned, however, that the preceding discussion has not attempted to cover debugging in any detail, since such a discussion is not within the scope of a manual of this type, and apart from a general approach each code to be debugged presents new situations. Skill in debugging comes only through actual experience and a meticulous care on the part of the programmer at all stages of the problem preparation and the debugging. The next step then is naturally enough the actual running of the problem with the debugged code.

The procedure that one goes through in starting the problem should be somewhat familiar by now. The debugged code is called into the memory from the magnetic tape where it resides. After the code is in the memory, the control counter is set to the desired starting address, and the problem is started by depressing the "start-next-order" switch.

When at all possible, the code of a problem should be set up so that shortly after the computation begins, a few intermediate computation results, where the correct results are known, would be printed. In this way there is some assurance that the computer is starting its computation correctly.

Since many of the problems contemplated require anywhere from several hours to several days of computation time, it is necessary that intermediate records of the problem (code and numbers) be made so that in the event of computer malfunction it is not necessary to start the problem from the beginning. One has only to return to the last correct record of the problem and resume from there. Also in lengthy

computations the code should be constructed so that intermediate results of the problem are periodically printed, so that they may be examined in order to see if they are reasonable. This is a check on the formulation of the problem as well as on the computer.

The periodic records of the problem are made on magnetic tape. The entire contents of the memory are recorded onto the magnetic tape; hence in order to start a problem from any record one has only to call the magnetic tape section into the memory and then set the control counter to the address of the instruction immediately following the last instruction of the code performed before the record was made. This instruction is, of course, known for each record; and, in fact, it usually does not vary from one to another. Experience has shown that a magnetic record of the memory contents should be made about every 20-25 minutes to insure a maximum of effective computation time. It is desirable that some intermediate results be printed shortly after a record is made. Then, in the event that a problem has to be restarted there will soon after be some printed results which may be checked against those printed when the record was made. This insures that the computation is starting correctly.

The routines which perform these magnetic recordings exist as subroutines as well as helper-routines, so that if desired they may be directly incorporated as an integral part of a problem. A variety of print routines exist that are easily included in a problem to print the intermediate and final results. As suggested above, one of the reasons that the periodic magnetic tape records of the problem are made is in anticipation of any computer malfunction. A computer malfunction might manifest itself in any one of several ways. For example, a set of intermediate results that are printed might be in error. Such errors may be detected by inspection, by taking differences of the results, by the plotting of graphs, by programmed integral checks, etc. In addition to such manifestations, a malfunction may occur by a nonsense instruction being brought into R6, the control register, and the computer stops. Or yet another type of malfunction might manifest itself in that the control becomes stuck in an instruction loop. That is, the control is being cycled through a fixed section of the code rather than following the correct path. If the loop through which the

control is cycling has relatively few instructions, it can actually be observed on any one of the "memory monitor scopes". These are discussed later. If the cycle is relatively long, it may not be detected for some time, namely when one tries to print results.

In the event that a computer malfunction is detected, the following procedure is recommended:

If the trouble occurs very shortly after operation has begun, the first suspect for the error would be that it was a human error. That is, either in loading the code and any data that might be needed, or in making any alterations of data, or in the setting of addresses into the control counter, the operator may have made some sort of an error. Hence, one should try to restart the computation without making any other checks. If similar trouble seems to repeat, one then follows the same procedure as for malfunctions that appear after the computation has been underway for some time. It is:

If a malfunction appears that is evidently not from a human source, the problem being computed is removed from the computer and the basic computer test problems are run to see if they detect the malfunction. Every operator of the computer should become intimately familiar with these test problems so that he can run them and interpret properly any results which might indicate a malfunction.

We discuss these test problems only briefly here. There is a so-called "Inversion Test" which checks that the memory is operating properly. A "Vocabulary Test" is a general test of all of the orders to see if any of them are failing. This test will detect any consistent errors. For the more aggravating intermittent variety there are specific tests that attempt to test more exhaustively each kind of order with a wide variety of numbers. In any test in which a malfunction occurs, data are printed that indicate the nature of the malfunction. As soon as a malfunction is detected by a test routine, the engineering staff should be called to fix the trouble. In the event that the test problems do not indicate any errors but the troubles still persist in the problem, the engineers should be called. If the trouble is manifested by incorrect results which can be duplicated, and if the test problems do not indicate computer trouble, one should begin to suspect that there is some incorrect information on

the magnetic tape dump from which the problem was started, or an even more disastrous thing--one should begin to suspect that perhaps the code is not in reality debugged.

In computer malfunctions, the operator should be able to assist the engineers in localizing the source of trouble. To do this one certainly must completely understand the function of the various registers and the control counter. Such an understanding also helps one operate the computer more effectively at all times. We now discuss these matters where part of what follows is review and part is presented for the first time.

We discuss the registers first starting with R6, the control register. During the loading process, R6 contains zeros. During the operation of the computer, R6 contains the instruction-word that is being acted upon by the control. One may, in general, determine the address in the memory of any instruction-word contained in R6 by examining the control counter. The control counter contains the address of the next word to be brought into R6. This is one address greater than the word in R6 unless either the control has just executed a transfer instruction or the counter has been set manually. Whenever a "nonsense" word in R6 stops the computer, the address less one in the control counter always indicates the location of this nonsense word in the memory, and it should be so checked.

The R5 register has many functions,which we discuss in turn. During the loading process, words pass through R5 en route to the memory, and at the completion of any loading, R5 should contain the word on the tape immediately preceding the double space. Any word which is brought into the arithmetic unit passes through R5. Hence, at the completion of any such operation, R5 contains the word from the location specified by the address of the instruction. Orders 1-12, as shown in Table I, page 21, are of this kind. The following orders also affect R5. After a Q$\longrightarrow$m instruction, R5 should contain the same word as R4. After an a $\rightarrow$Ac or a$\longrightarrow$Ah instruction, R5 should contain in positions (0-11) the number which is equivalent to the address portion of the a$\longrightarrow$A instruction. Upon the completion of a read instruction, the word also resides in R5 as well as in the memory. Now upon completion of Instructions 19-22, of Table I, the substitution instructions, R5 contains the word into which the substitution is being made, as it appears before the substitution is effected. Note that an A$\longrightarrow$m instruction does not involve R5.

We discuss the registers R4 and R3 together since R3 is an auxiliary register for R4 (R4 $\equiv$ Q). Neither is affected by the loading process. When a number enters R4 via an m$\longrightarrow$Q instruction, R4 contains the number from the location specified by the address. The contents of R3, however, are irrelevant and may be anything depending upon past instructions. However, if a number enters R4 from any other source (viz., X, $\div$, L(n), or R(n) instructions), R3 contains the same information as R4 displaced one position left or right except perhaps for the sign position and the 2^{-39} position. In the X and R(n) operations the number in R3 is displaced to the left of the one in R4, while in $\div$ and L(n) operations the number in R3 is displaced to the right.

The magnetic tape instructions and the magnetic drum instructions use R4 and R3, and consequently upon completion of t$\longrightarrow$m or D$\longrightarrow$m, R4 contains the last reference word. R5 will also contain the same word. R5 contains the last reference word of m$\longrightarrow$t and m$\longrightarrow$D as well. On the instructions where R4 contains the last reference word, R3 contains the same word displaced once to the right except for sign.

R2 and R1 also work in conjunction; however, any time a word is in R2 from any instruction, the same word, except perhaps for sign position and 2^{-39} position, is in R1 displaced either one unit right or left.

Upon the completion of loading, R2 contains the sum of the contents of the tape. Upon the completion of a D$\longrightarrow$m instruction with address m+800, R2 contains the sum of the fifty words read from the drum to the memory.

Upon the completion of any of the add orders, a$\longrightarrow$Ac, a$\longrightarrow$Ah, X, R(n); R1 contains the same number as the R2 register displaced once to the left. Upon the completion of $\div$, L(n); R1 contains R2 displaced once to the right.

Upon the completion of a syncprint order (not considering the subroutine in which it is contained) R2 contains all ones. In this instance, and only in this instance, R1 may have completely foreign numbers to those of R2.

If a computer malfunction is suspected, the contents of the various registers should be closely observed, and if there is any deviation from the above-mentioned situations the discrepancies should be recorded, as they may aid in the detection of the malfunction.

As previously mentioned, the control counter is the mechanism used for the sequencing of instructions. The control counter always contains the address of the next word to be brought into R6, the control register. The control counter may be manually set to any desired address. While the computer is running, the control counter advances sequentially except when transfer or satisfied conditional transfer instructions are executed. These instructions set the control counter to the same address as that contained in the instruction. The control counter has several special functions which are:

In loading, the control counter is the sequencing mechanism. The control counter is first set to the desired initial address. Then the contents of the tape being loaded are sent to the memory into sequential addresses beginning with the initial one. Upon the completion of the loading, the control counter resets to the initial address.

In the drum instruction, the control counter indicates the fifty sequential memory addresses concerned with the instruction. At the outset of the instruction the counter is set to the memory address contained in the instruction. When the fiftieth word is transmitted, the counter contains the corresponding memory address. Since this is not, in general, the desired address for the next instruction, the drum instruction ends by setting the control counter to the address contained in bigits (28-39) of the drum instruction.

As with the registers, when a computer malfunction is suspected, the control counter should be observed to ascertain that its behavior corresponds to that given above.

We complete the chapter now with brief discussions of the "audio-monitor", the memory monitors, the magnetic tape, the Synchroprinter, the computer "turn-off" and emergency procedures, and a brief comment on the method of time scheduling for the computer.

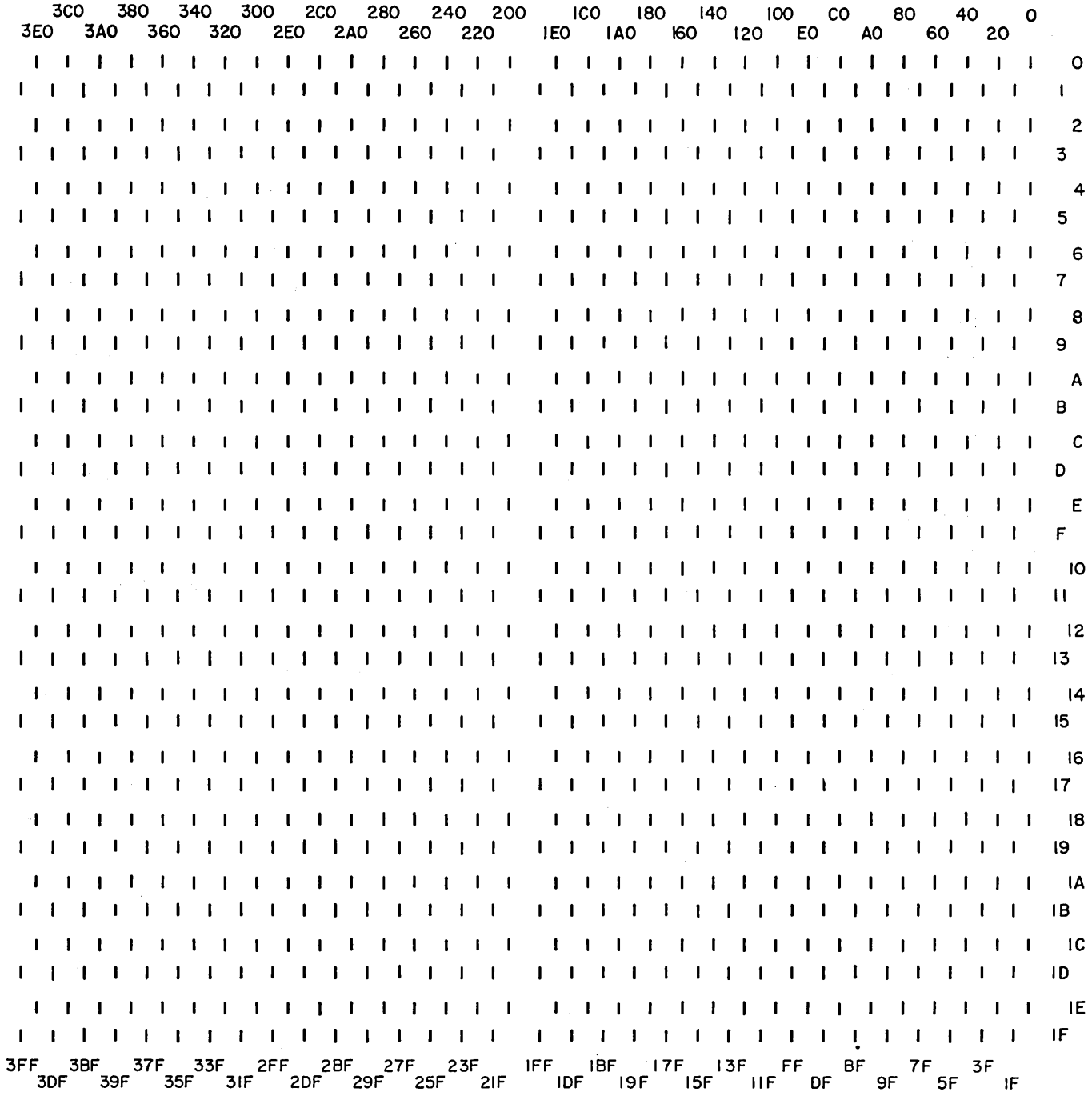
The "audio-monitor" is an amplifier and a loud-speaker that taps into the circuitry of the function gates. The frequency with which the function gates change (i.e., flip from left to right as successive instructions are performed) while the computer is running on automatic operation is in the audio-range. The amplifier merely amplifies and transmits these frequencies to the loud-speaker and hence into audible noise. The use for such a piece of equipment lies in the fact that in

many problems that are run on the computer the code patterns established by the various induction loops of the problem give rise to distinctive and easily detectable noise patterns. A person familiar with the noise patterns of a problem can often tell when there has been a computer malfunction if the malfunction manifests itself by the control altering its path through the code sequence. This circumstance causes a change in the noise pattern of the problem. A volume control switch allows one to control the audio-monitor and, if desired, the volume may be turned down.

The memory monitors consist of four three-inch cathode ray tubes. These tubes allow one to observe the contents of any of the forty memory tubes. The monitor tubes are mounted at each end of both banks of memory tubes as shown in Figures 2 and 3. There are six selector switches, four mounted directly under the central storage units of the front storage bank and two similarly mounted on the back side of the computer. The selector switches are eleven place switches, allowing an "off" position and the display of any of ten memory tubes by a monitor tube. Since there are four switches on the front, two connected to each of the front monitors, one can observe any of the forty memory units. The two left-hand switches select units (0-19) while the two right-hand switches select (20-39). However, each monitor tube may display only one unit at a time and care should be exercised that the two selector switches connected to a single monitor tube are not both set to a unit as this causes erroneous information to be stored into the memory units concerned. The two selector switches on the rear bank may only monitor that bank, the odd-numbered memory units as shown in Figure 3. The left-hand switch can monitor 21, 23, 25 ··· 39, and the right-hand switch can monitor 1, 3, 5 ··· 19.

The memory raster, as one views the monitor tube, is as shown in Figure 6. A bright spot at any position of the raster corresponds to a 1, while a faint spot corresponds to a 0.

As a problem is running, the code patterns due to induction loops often cause certain portions of the code to be performed more frequently than others. The memory locations concerned are then consulted more frequently, and these regions of higher consultation show a brighter intensity on the monitor tube than neighboring regions. One may then



MEMORY RASTER

FIG. 6

be able to determine, by observing a monitor tube, when certain sections of the code are being traversed. As with the "audio-monitor" and its noise patterns, the memory monitor often displays distinctive code patterns. If the computer malfunctions in a way that the display pattern is altered, this is often observable.

The magnetic tape unit has previously been discussed in Chapter II, Problem 12, and in Chapter IV; so that what is said here will pertain mostly to the operation of the unit.

Recall that the unit is a single channel serial system where the magnetic tape reels contain 1200 feet of 1/4 inch wide Scotch Sound Recording Tape. These reels of tape are, in general, pre-marked into sections, each of which will accommodate 1024 forty-bigit words. There are fifteen such sections to a reel and the markings dividing these sections are short lengths made transparent by removing the magnetizable material from the tape. A photo-cell in circuit with a fast forward and reverse mechanism affords the only searching facilities (manual). The tape may be advanced or reversed at a speed of roughly four seconds per block of 1024 words, and the photo-cell actuates a brake whenever a transparent section of tape, indicating a separation of the 1024 word blocks, passes through it.

In order to use the magnetic tape, one first threads the desired reel of tape onto the tape drive mechanism. Second, the tape is advanced to the start of the desired 1024 word block. Third, the tape unit switches are set so that the unit can then be operated by the control of the computer through the magnetic tape routines (cf. Chapter II, Problem 12) We now discuss these steps in detail.

The tape drive as it appears atop the console cabinet is shown in Figure 7. The different parts are clearly marked and need no explanation; hence with the aid of this diagram we turn to the tape threading procedure.

To thread tape

1. Remove the caps from both tape reel spindles.
2. Place the reel of tape on the left spindle. It is called the rewind reel. The tape feeds from this reel in the direction indicated by the diagram.
3. Set the forward-reverse-normal switch, hereafter called the setting switch, to the normal position.
4. Open the head housing door.
5. Unwind a length of tape and thread it as indicated in Figure 7.

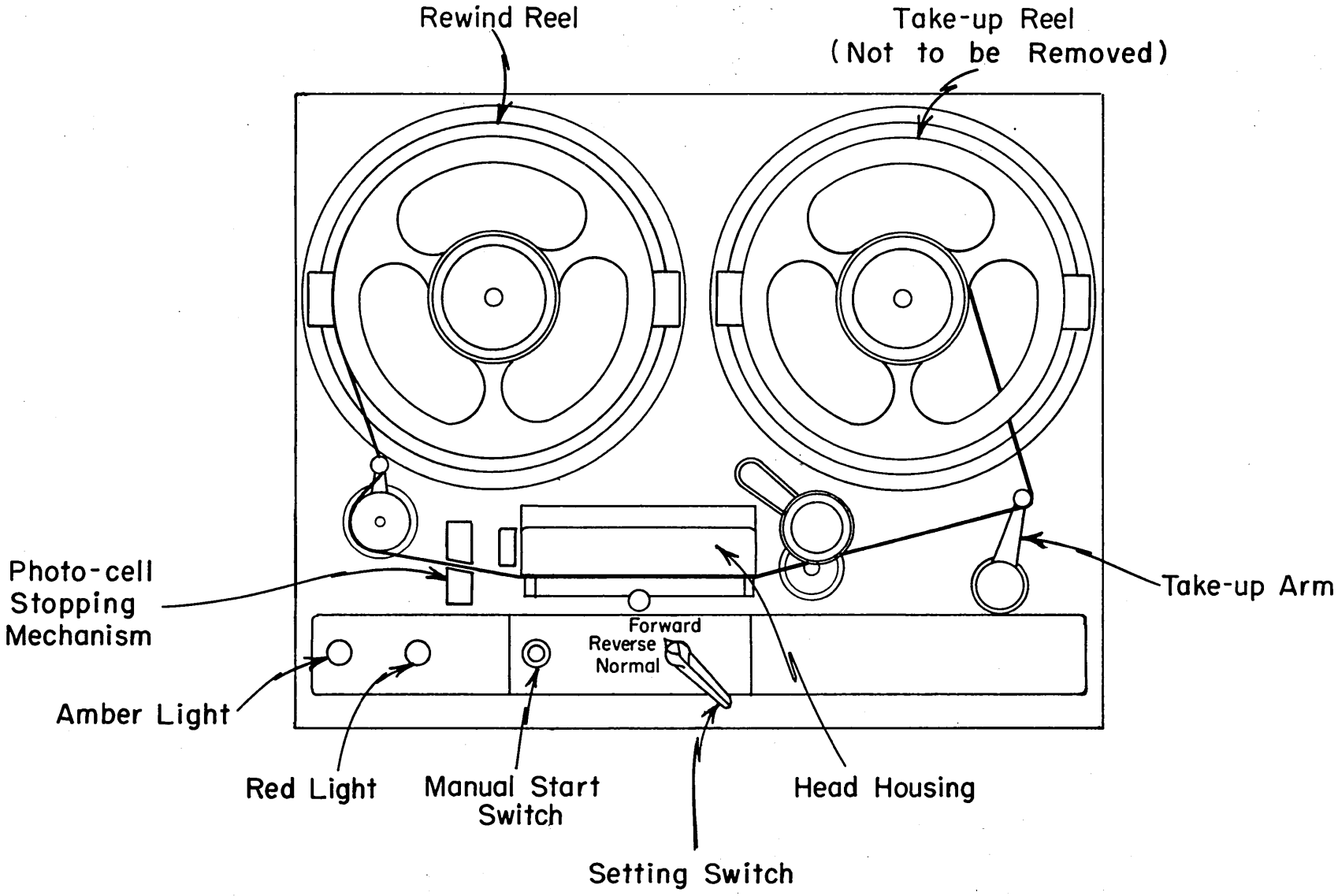

Rewind Reel
Take-up Reel
(Not to be Removed)
Photo-cell
Stopping
Mechanism
Take-up Arm
Forward
Reverse
Normal
Amber Light
Red Light
Manual Start
Switch
Head Housing
Setting Switch

FIG. 7

6. Wind several turns around the take-up reel. Wind the take-up reel until the take-up arm is in the position shown.
7. Replace the caps on the spindles. (Do not remove or replace caps while the tape unit is running.)
8. Close the head housing door. The tape is now ready to be advanced to the first transparent section, the starting position for the first block of information.

In order to have the tape in correct position to record or replay a block of storage, all that is necessary is that the transparent section of the tape identifying the start of the block must be visible in the region of tape between the two reels.

To advance or back up tape to start of desired tape section

1. Turn the setting switch to the desired direction of motion of the tape.
2. Open the head housing door; the tape advances in the desired direction. When a transparent section passes by the photo-cell, the tape stops. The braking is not instantaneous, and the transparent section may travel as much as 15 feet during the stopping process.
3. Turn setting switch to the opposite direction of the previous motion.
4. Depress the manual start button. This starts the tape moving in the direction shown by the setting switch. The transparent section of the tape again actuates the braking action when it passes through the photo-cell. This time the overshoot is less.
5. Repeat steps 3 and 4 until the transparent section lies in the region between the two reels. This is the desired starting position.
6. Turn the setting switch to the normal position and close the head housing door.
7. The tape is now ready to operate - either record or play back.

If it is desired to back up or advance the tape more than one block of words, at the end of step 2 press the manual start button without changing the setting switch. Repeat this until the desired block of information is reached. The procedure is then the same as previously noted starting at step 3.

To record or replay

1. The transparent section identifying the desired block must be in the region between the two reels.
2. The head housing door must be closed.
3. The setting switch must be in the normal position.
4. The take-up arm must be in the position indicated in Figure 7.
5. The red indicator light must be off.
6. When steps 1 through 5 are completed the tape is ready to be operated automatically upon instruction from the computer.

The indicator lights have the following significance:

i. The amber light indicates that the power is on. If this light is "off", call an engineer for assistance.

ii. The red light is "on" in any of the following circumstances.

a. The head housing door is open

b. The setting switch is in the reverse or forward position.

c. The take-up arm is not in correct position.

d. It is "on" while the tape is running during a recording or a play back.

If, in setting the tape to record or replay, the red light remains "on" after steps 1 through 4 have been completed correctly, call an engineer for assistance.

The Synchroprinter has previously been discussed in Chapter II, Problem 13, and in Chapter IV; so that, as with the magnetic tape, the remarks here pertain to operating procedures.

Recall that the Synchroprinter prints a line at a time; each line may contain 40 characters. The maximum speed of operation is 15 lines per second, or 36,000 characters per minute. This print order must be used in a routine (cf. Chapter II, Problem 13) which does the following: The four words to be printed are fanned into an array of eighteen words in the memory. During a print cycle, eighteen print orders are given. The first print order activates the printer and the remaining seventeen act in a timing capacity synchronizing the printer and the computer. Prior to each print order of the cycle, the appropriate word of the array is brought into R2.

In the discussion of the operation of the printer, we assume that the printer routine has been properly incorporated into the problem and discuss only the mechanics concerned with the printer unit.

Five switches are located on the front of the printer cabinet. These are:

i. the motor switch

ii. the filament switch

iii. the plate switch

iv. the thyratron switch

v. the paper advance switch

The filament switch and the plate switch are always to be in the "on" position. If this is not the case, one should not attempt to operate the printer, and an engineer should be called for assistance.

When the printer is to be used, the positions of the motor switch and thyratron switch should be checked. If they are in the "on" position the printer is ready to operate. If they are in the "off" position, the following is done: (The order is important.) First, the motor switch is turned to the "on" position and then the thyratron switch is turned to the "on" position.

The thyratron switch controls a bank of 40 thyratron tubes that are used for triggering the 40 print hammers. A thyratron tube is a gas discharge tube rather than a vacuum tube, and it permits the high current necessary for triggering the print hammers. Once a thyratron has been discharged, its plate voltage must be cut off in order to reset it to the non-conducting state. The triggering of the print hammer momentarily causes the plate voltage to be cut off so that the thyratron is reset. However, the circuitry is such that the triggering of any print hammer twice in a print cycle will cause its associated thyratron to stay in the discharge state, making any further triggering impossible. Attached to each thyratron is a neon bulb which is lighted whenever the thyratron is in the discharge state. These neons are visible through a glass panel immediately below the thyratron switch. Whenever a thyratron remains in its discharge state, as indicated by its lighted neon, it may be reset by turning the thyratron switch "off" momentarily and then turning it "on" again. If, in the "turn-on" procedure for the printer, some of the thyratrons discharge, as indicated by their associated neon being lighted, the above procedure is carried out for resetting them.

A thyratron should never be left in the discharge state, and as soon as such a condition is known the above reset procedure should be carried out.

During operation, the only times that a thyratron can be left in the discharge condition are:

i. when more than, or less than, the required 18 print orders are given in a print cycle

ii. when a print hammer has been triggered more than once per print cycle

iii. when there has been some computer malfunction effecting the printer

(i) and (ii) may be caused by an improperly coded print routine. If the computer is stopped during a print cycle, and if a print order is in R6, connected into the control circuitry, the computer cannot be restarted without danger of leaving some of the thyratrons in the discharged state. Restarting resumes the print orders, and with the control in the middle of the routine less than eighteen print orders will be executed by the control. If the computer is stopped during a print cycle, and if an order other than the print order is in R6, one should again check the thyratron neons, as there is danger that some thyratrons may be in the discharge state.

If any thyratrons are in the discharge state and an attempt is made to use the printer, the print hammers associated with the discharged thyratrons cannot be triggered; hence no characters will be printed in the corresponding columns.

The "paper advance switch" allows one to manually advance the paper so that printed material may be removed from the printer. Depressing the switch causes the paper to advance and it will continue to do so as long as the switch is held in the depressed state. Note that for manually advancing the paper, one should always use the paper advance switch, since advancing the paper by merely pulling it causes the printer ribbon to become misaligned.

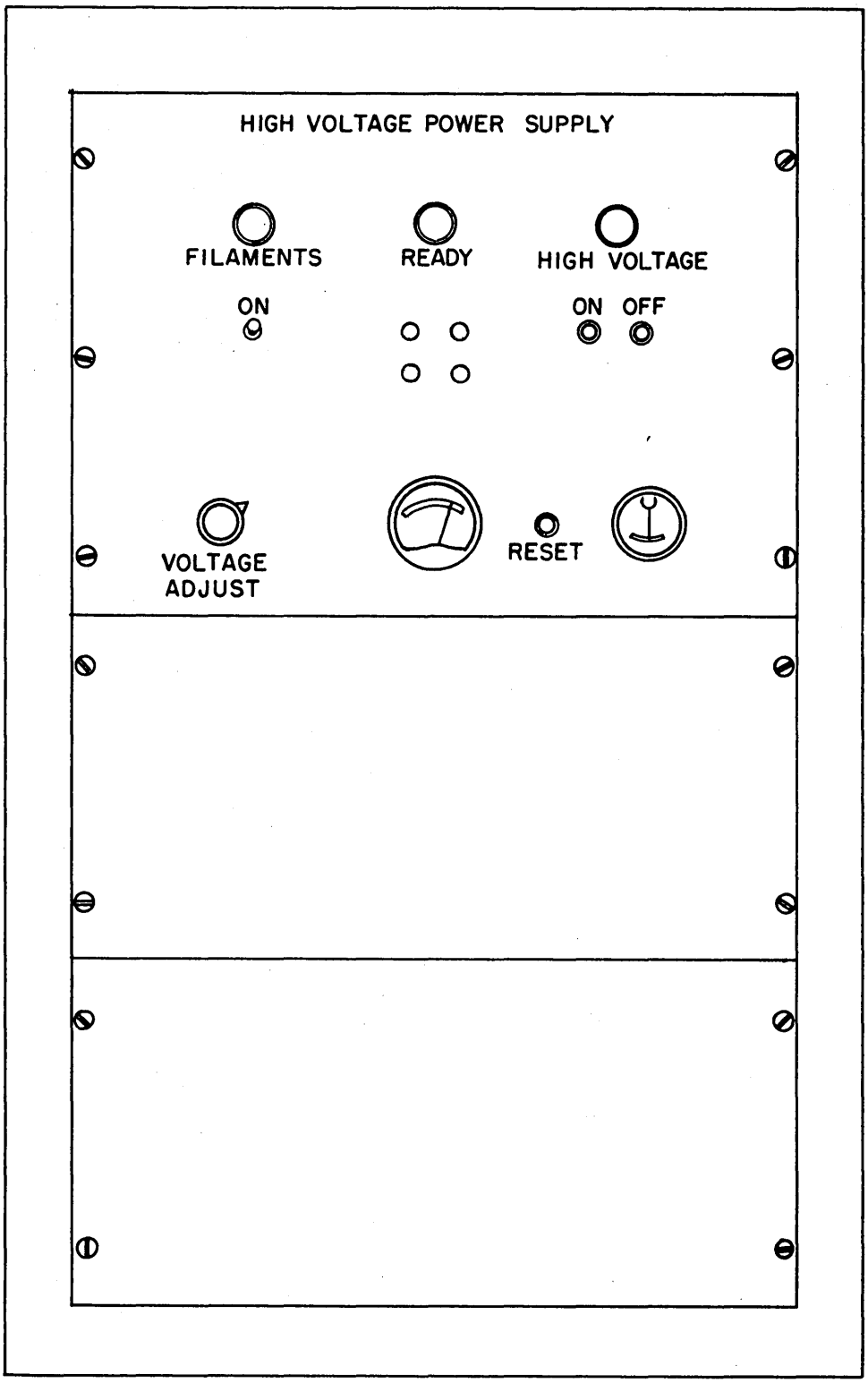
The "turn-on" and "turn-off" procedures for the computer naturally seem to be more in the domain of the engineers rather than that of the programmers; however, the turn-off procedure has been simplified to the extent that the programmers can do it.

In order to turn off the computer, one must set certain of the switches located on the Memory High Voltage Power Supply shown in Figure 8, the Switch Gear Panel shown in Figures 9 and 10, and the Magnetic Drum Control Panel shown in Figure 11. The relative position of these panels with respect to the computer proper is shown in Figure 1.

The "turn-off" procedure in its proper sequence is the following:

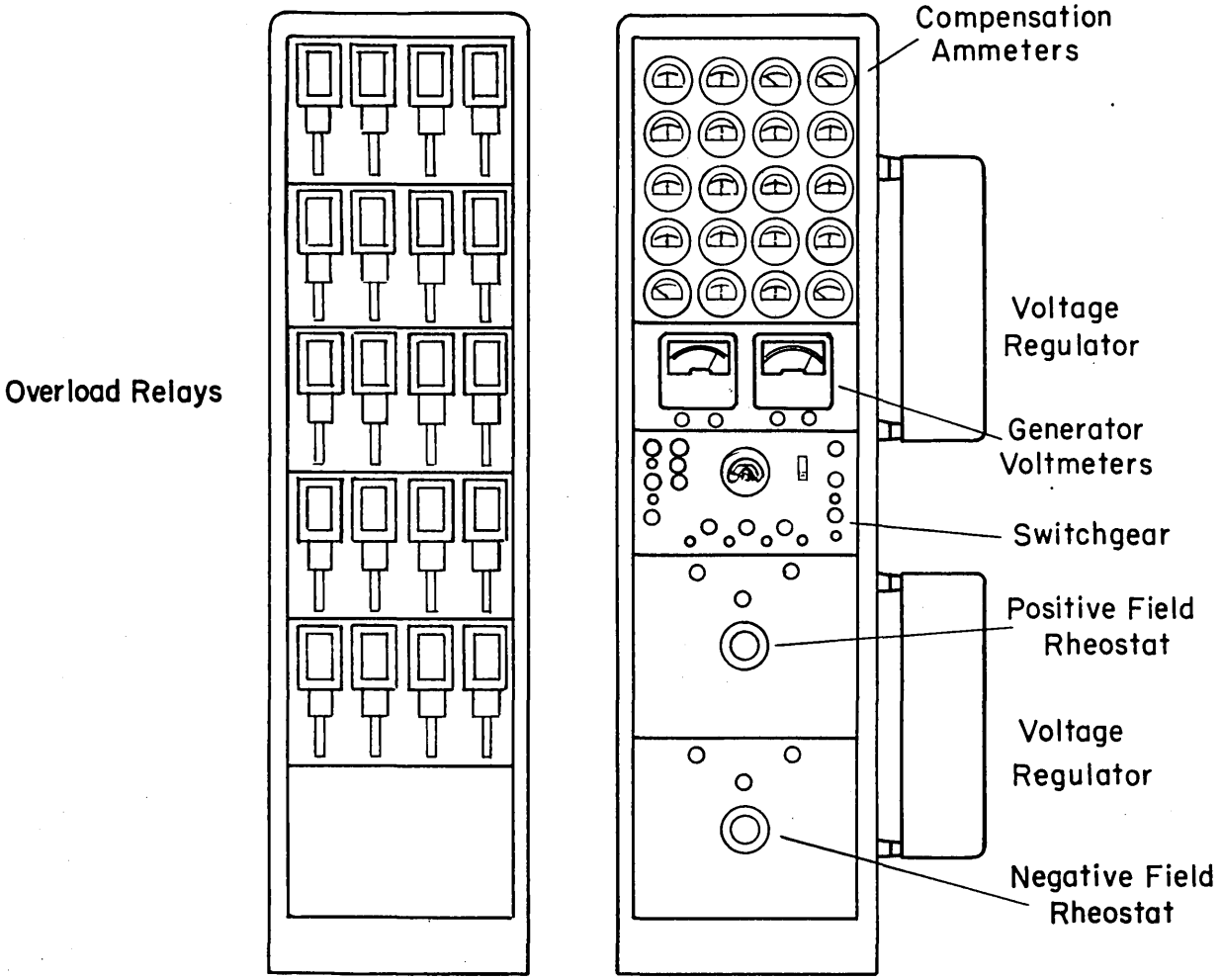
On the High Voltage Power Supply (Figure 8)

Depress "off" button. (Leave filament switch in "on" position, however.)



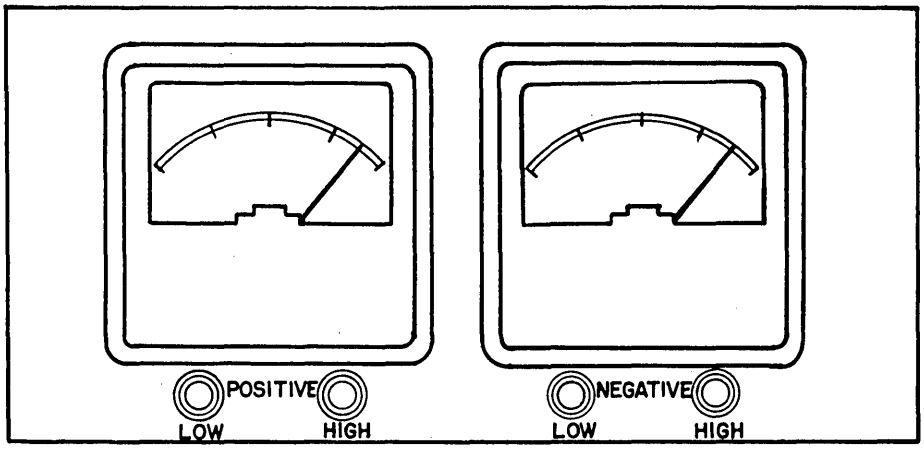
HIGH VOLTAGE POWER SUPPLY

FIG. 8

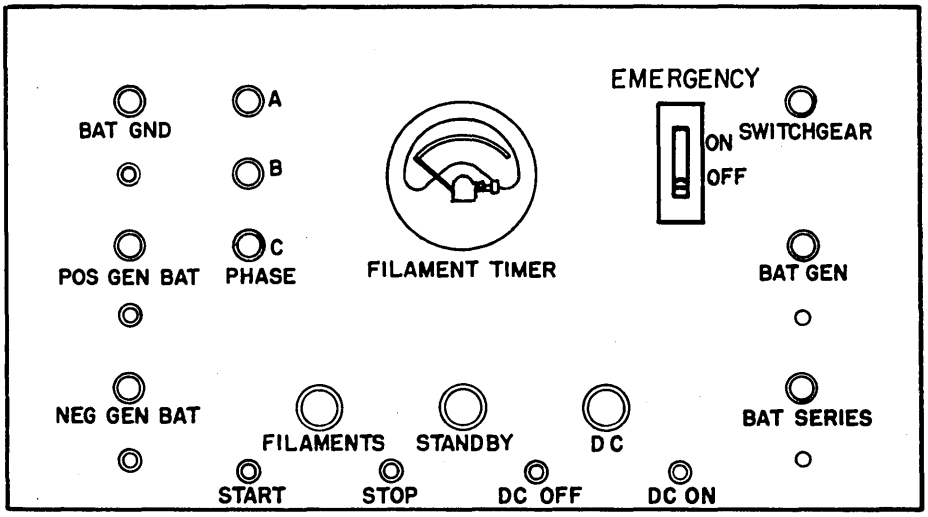


POWER SUPPLY CONTROL PANELS

FIG. 9

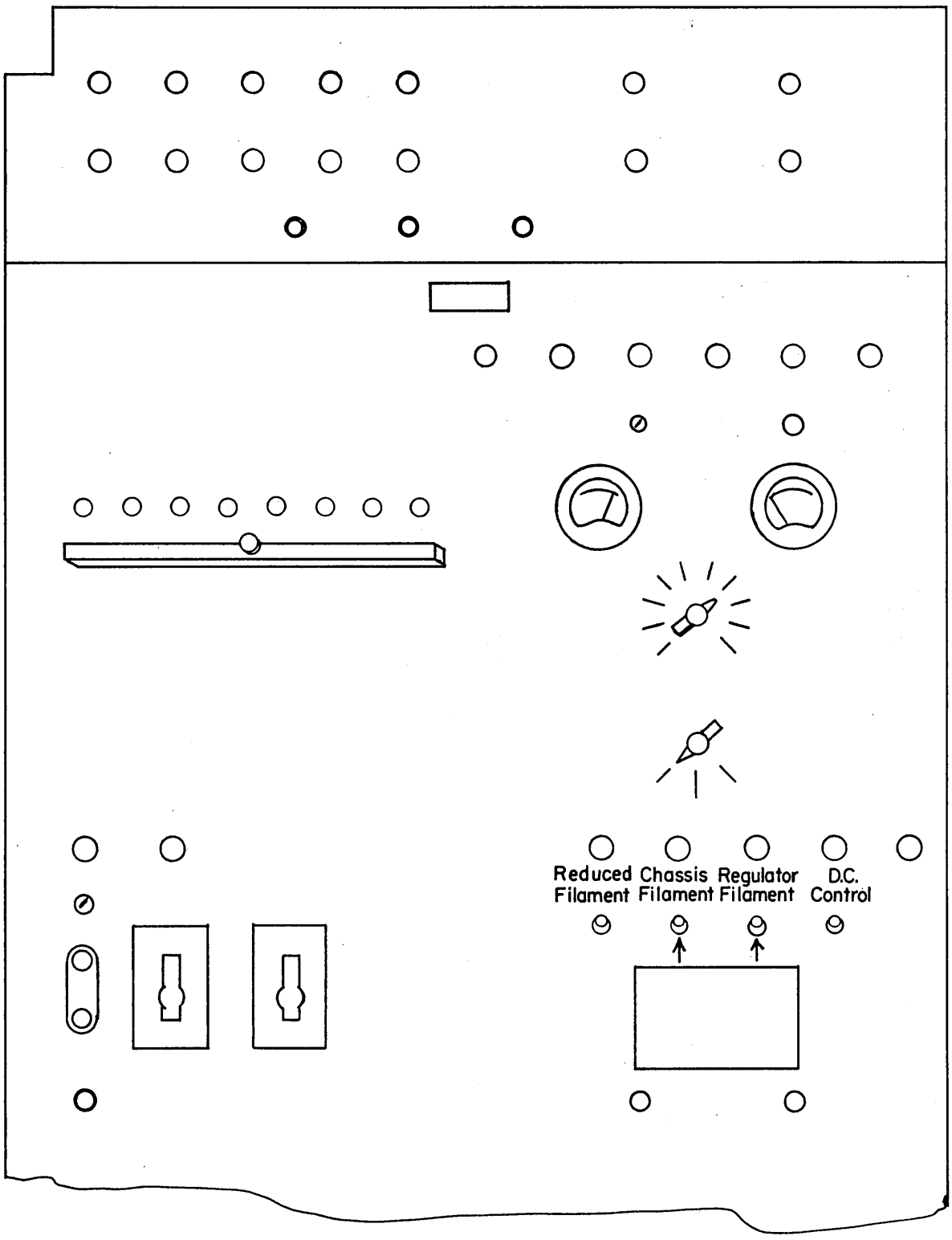


GENERATOR VOLTMETER PANEL – POWER SUPPLY



SWITCHGEAR PANEL – POWER SUPPLY

FIG. 10



DRUM CONTROL PANEL

FIG. 11

On the Switch Gear Panel (Figures 9 and 10)

1. Turn DC "off" by depressing DC off switch located between the DC (red) and standby (green) lights.
2. Set battery-generator switch into "down" position.
3. Set battery-series switch into "down" position.
4. Turn off generators by depressing the stop (red) switches for the positive and negative generators. These switches are each located immediately above its corresponding positive or negative field Rheostat.
5. Turn the filament variac down (turn the wheel counter clockwise as far as it will go). The variac is located between the memory high voltage power supply and the overload relay of the power supply control panels as shown in Figure 1.
6. Depress the stop switch located between the filament (white) and standby (green) lights.
7. Turn the Emergency switch to the "off" position.

On the Magnetic Drum Control Panel (Figure 11)

1. Set the "chassis filament" switch to "off" position.
2. Set the "regulator filament" switch to "off" position.

   Note: Do not set any drum switches other than the two indicated by 1. and 2.

In the event of an emergency, such as smoke or flame emitting from the computer, the emergency "turn-off" procedure is:

Emergency Turn Off

1. Set the emergency switch on the switch gear panel to the "off" position.
2. Immediately call an engineer.

In the discussion of"debugging" procedures, the emphasis was placed on using the computer effectively;when a reasonable amount of data has been obtained from the monitoring or as soon as an error has been detected during the monitoring, the problem whould be removed from the computer and the data studied away from the computer. This procedure naturally leads to the following questions: What is the length of time that one should spend with the computer per debugging session? And, how should the time on the computer be scheduled so that debugging sessions are coordinated in a way which utilizes the computer efficiently? At the present stage of the art there seems to be no clear cut answer to either of these questions. Our present attempt to answer them stems for experience gained during the past several years of operation.

It seems that a person will accomplish more in several short sessions than in a long session of the same total time, if the time between the short sessions allows him to study and digest the results. As a consequence, thirty minutes is the maximum time for any debugging period; however, shorter periods are recommended. Instead of arranging a schedule according to the clock, a programmer decides on each occasion when to terminate his debugging session.

Since a debugging session may range anywhere from about five to thirty minutes, and since the exact length of the period is left to the discretion of the programmer, this has brought about the following arrangement: Debugging periods on the computer are scheduled sequentially during the normal working hours. This is the time when most programmers are available. A debugging schedule is compiled; however, no specific time is allotted to any person. The list only serves to indicate the order in which the debugging periods are scheduled and, as mentioned above, the length of each period is determined by the programmer while he is debugging. It is the responsibility of those on the schedule to be available when their debugging period occurs.

As soon as the debugging periods are over, the running of problems is scheduled. Debugging time is not normally scheduled beyond the completion of the regular work day which is 5:00 PM. This means, then, that most of the problem running time is allotted in the hours between 5:00 PM and 8:00 AM the following day. Problem running time can, of course, be scheduled for fixed periods; hence there is no need, as in debugging, for all on the list to be available prior to their scheduled time.

APPENDIX I

SCALING OF NUMBERS

Numbers handled by the computer must be in the range

$$|x| < 1 \tag{1}$$

The numbers that occur in the course of a numerical computation are usually not so contained. As a result it is necessary in going to automatic computation to change some, if not all, of the fundamental set of units. The process of making these linear transformations is called scaling. Consider the following very simple example: Suppose one were interested in the distance in centimeters of free fall for times lasting to 100 seconds; i.e.,

$$S = 1/2\ g\ t^2 \tag{2}$$

where $\underline{S}$ is the distance, $\underline{g}$ = 980 cm/sec^2 is the gravitational acceleration, and $\underline{t}$ the time. In order to restrict the range of these quantities so that they satisfy Condition (1), one makes the following transformations

$$\begin{aligned} \tau &= 2^{-7}t \\ \gamma &= 2^{-10}g \end{aligned} \tag{3}$$

For convenience, one uses powers of two. Quite clearly τ, γ are contained in the proper range. Using (3) one finds

$$\begin{aligned} S &= 1/2(2^{10}\gamma)(2^{7}\tau)^2 \\ &= 1/2\ 2^{24}\gamma\tau^2 \end{aligned}$$

Hence, if the transformation

$$\sigma = 2^{-24}S$$

is made, one obtains

$$\sigma = 1/2\ \gamma\tau^2$$

where all the quantities as seen by the computer are now well contained.

The three transformations are not, of course, independent since only the dimensions of length and time are involved. An alternate way of expressing the above is to say that time is measured in units of 2^7sec. and length is units of 2^{24}cm. In reviewing a scaled number in a register, one may very easily unscale the number by imagining that

the binary point is shifted appropriately from its normal position (between 0 and 1 stages). In the above example, the unscaled time is found by considering the binary point moved 7 places to the right.

One chooses the minimum amount of change in units in order to have the maximum accuracy. Sometimes the variations in the quantities are so violent that it is necessary to make successive transformations in order to maintain sufficient accuracy. Nevertheless, this is usually much faster than appealing to floating point routines.

APPENDIX II.

VERTICAL BUSES

The vertical buses of the order gates, as discussed in Chapter IV, pages 202-204, Figure 11, have been modified and are shown below as Figures 1, 2, 3, and 4. Figure 11, of Chapter IV, illustrated the original arrangement of the vertical buses on the front and back section of the arithmetic unit control. As a result of several modifications across time, we now require the four figures, one for the front side of the control (Figure 1) and three for the back side (Figures 2, 3, and 4). The motivation was to simplify the control system. It was found desirable to incorporate a few new buses and, in order to do this, a more efficient distribution of buses was necessary. That is, although all of the buses as shown in Figure 11, of Chapter IV, are necessary, they were not all needed on both the front and back control section; e.g., COR4, COR1, R1R2L_1, etc., were not used for any order gates on the front section; and, similarly, (0-7)R2, (8-19)R2, (20-27)R2, etc., were not necessary on the back section.

VERTICAL ORDER BUSES

FRONT

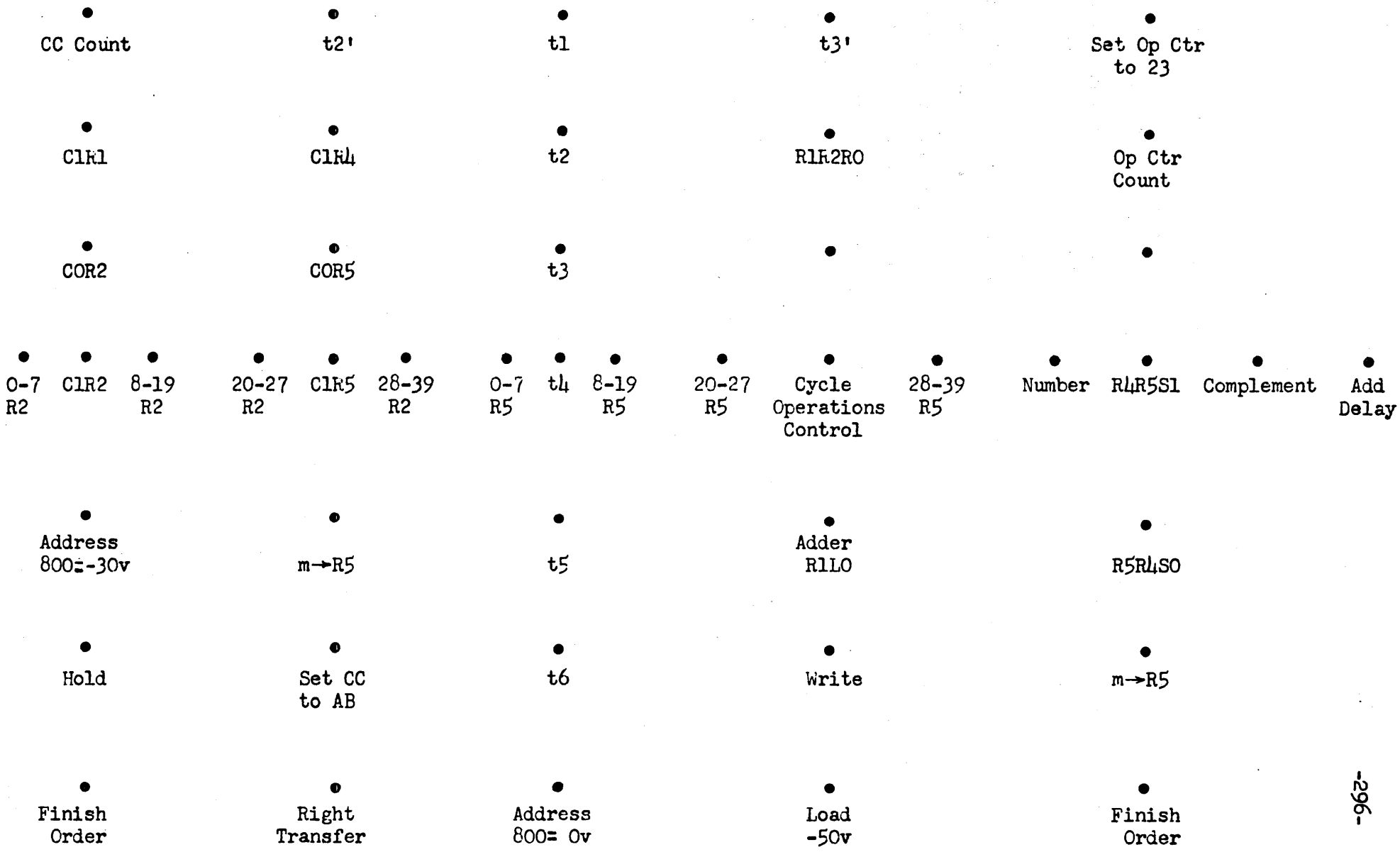


Figure 1.

VERTICAL ORDER BUSES

REAR

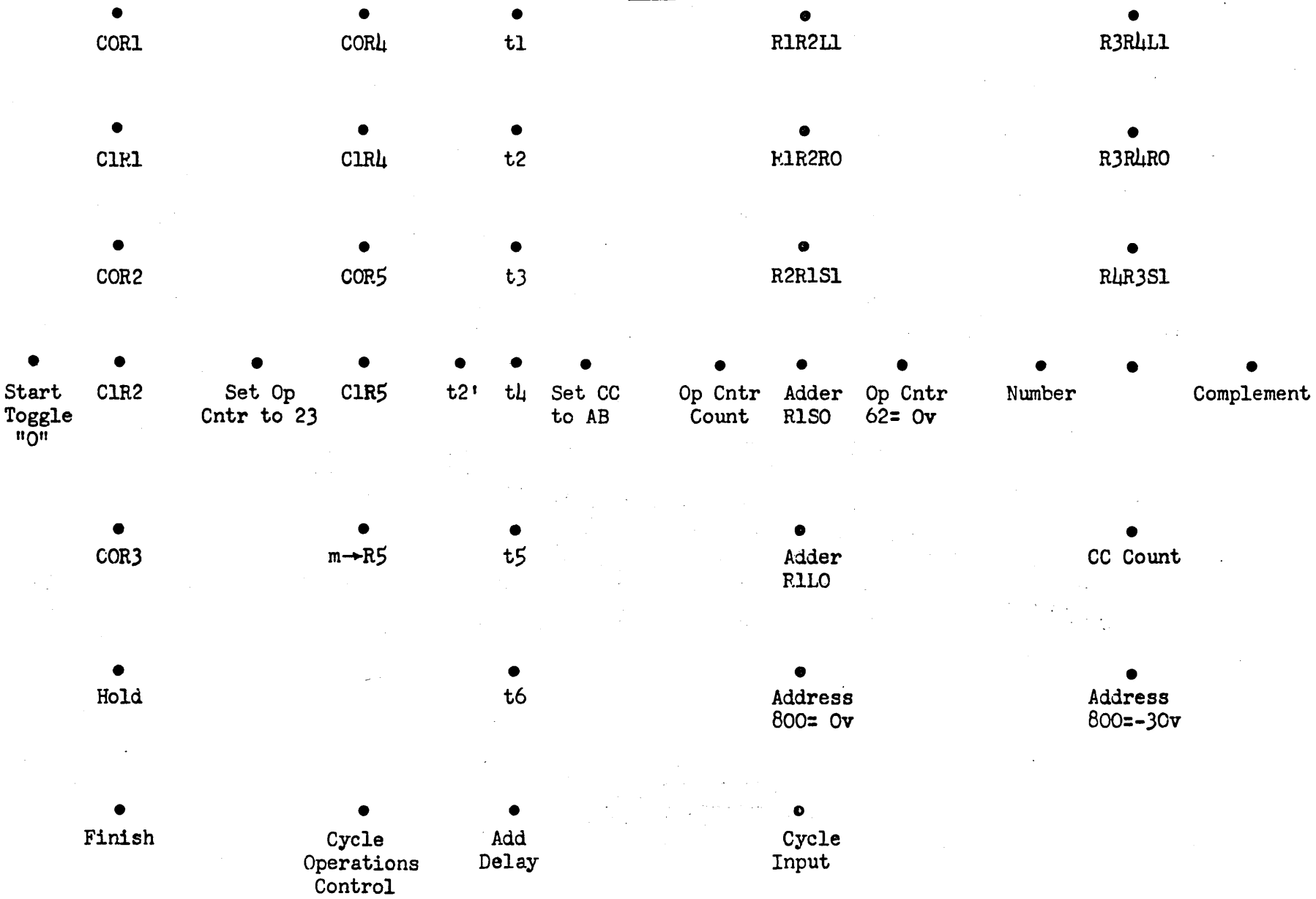


Figure 2.

VERTICAL ORDER BUSES
LOWER SECTION
REAR

• COR1	• C1R2	• COR4	• COR5	• t1	• t2	• R2R1S1	• R1R2L1	• R4R3S1	• R3R4L1
• C1R1	• COR3	• C1R4	• C1R5	• t3	• Set CC to AB	• Adder R1SO	• R1R2RO	•	• R3R4RO
• COR2	• Hold	• Set Op Cntr to 23	• Add Delay	• t2'	• t4	• Adder R1LO	• Address 800= -30v	• CC Count	• Complement
• Address R5 Gate	• Finish	• m→R5	• Cycle Operations Control	• t5	• t6	•	• Op Cntr Count	• Address 800= 0v	• Number

Figure 3.

HORIZONTAL ORDER BUSES
REAR

• Clear Drum Address	•	• Complement	• Set Op Cntr to 23	• Cycle Operations Control	• C1R6	• Address 800± 0v	• C1R4	• Address 800=-30v	• Control Counter Count	•
•	• m→R5	• Number	• Load= -50v	•	• t6	• t2'	• Finish	• C1R2	•	•
•	• R5R4S0	•	•	• R1R2R0	• t5	• C1R5	• Set CC to AB	• C0R2	•	•
• AB→R5 Gate	• R4R5S1	• Op Ctr Count	• Adder R1L0	•	• t4	• C0R5	• Hold	• C1R1	•	•

Figure 4.

APPENDIX III

SINGULAR ARITHMETIC OPERATIONS

In a division operation involving numerator $\underline{x}$ and denominator $\underline{y}$ there are certain combinations that violate the condition $|x| < |y|$, but nevertheless give rise to interesting and often useful results. We call such division operations singular operations. Some of the important results are:

i. $-1 \leq x < 1$, $y = 0$ then

$$Q = \frac{x}{0} \rightarrow 2 - x - 2^{-39}$$

ii. a special case of (i) is $x = y = 0$ then

$$Q = \frac{0}{0} \rightarrow 2 - 2^{-39} = 1.1111 \cdots 11$$

iii. $x = y > 0$ then

$$Q = \frac{x}{x} \rightarrow -(1 - 2^{-39}) = 1.0000 \cdots 01$$

iv. $x = y < 0$ then

$$Q = \frac{x}{x} \rightarrow 1 - 2^{-39} = 0.1111 \cdots 11$$

v. $x = -y > 0$ then

$$Q = \frac{x}{-x} \rightarrow 1 - 2^{-39} = 0.1111 \cdots 11$$

vi. $-x = y > 0$ then

$$Q = \frac{-y}{y} \rightarrow -(1 - 2^{-39}) = 1.0000 \cdots 01$$

Recall from the discussion of binary arithmetic in Chapter III that the allowed number range is $-1 \leq x < 1$. This implies that -1 (a $\underline{1}$ in the sign position followed by all $\underline{0}$'s) admits valid operations. In the addition process this is obviously the case. In division, if the numerator $x = -1$, the quotient is meaningless except for the cases (i and iv) where the donominator $y = 0$ and $y = -1$.

However, in division, if the denominator $y = -1$, one obtains the normally expected quotient; e.g.,

vii. $x > 0$, $y = -1$

$$Q = \frac{x}{-1} \rightarrow 2 - x - 2^{-39}$$

viii. $x < 0,\ y = -1$

$$Q = \frac{x}{-1} \quad x \ - 2^{-39}$$

ix. the special case for $x = 0,\ y = -1$

$$Q = \frac{0}{-1} \quad 2 - 2^{-39} = 1.1111 \cdots 11$$

For the multiplication operation -1 is admissable as one and only one of the factors, and

x. $x = -1,\ y \geq 0$

$p = xy \longrightarrow 2 - y$

xi. $x = -1,\ y < 0$

$p = xy \longrightarrow |y|$

The treatment of -1 is symmetric with respect to the multiplier and multiplicand. If

xii. $x = y = -1$

$$p = xy \quad 1 + 2^{-39} = 1.0000 \cdots 01$$

We see that the multiplication $p = xy$ where $x = y = -1$ does not give the correct product and hence is an exception to the rule admitting -1 as a legitimate number.

Returning to the division operation, there is one other fact worth noting; namely, if a division is exact with fewer than 39 quotient bigits, and if $x,y > 0$, and if

$$Q = \frac{x}{y} \text{ and } Q' = \frac{-x}{-y}$$

are formed, then

$$Q = Q' + 2^{-38}$$

Similarly, if $x,y > 0$, and if

$$Q = \frac{-x}{y} \text{ and } Q' = \frac{x}{-y}$$

are formed, then

$$|Q| = |Q'| \ - 2^{-38}$$

INDEX