I

INTRODUCTION

General Description of the 102-A System

The CRC 102-A is a general purpose automatic digital computer, capable of being internally programmed.

The ability to be "internally programmed" means the existence of a memory system, in which can be stored numerically coded instructions and data necessary for the solution of a given problem. "General Purpose" implies the property of flexibility; that is, a general purpose computer is capable of being programmed to solve a variety of problems, whereas a special purpose computer is designed to solve only a definite class of problems. The ability to perform arithmetic operations, alter its own computational routine, and make decisions which will govern the course of action is an important feature of the 102-A. "Automatic" signifies that after the start of computation the computer is capable of execution of all prestored instructions without the aid of an operator.

The 102-A, then, is capable of receiving information (instructions and data) from an operator and storing it in its memory, following the instructions automatically, and delivering the results to the operator. Most important is that this is all done at very high speeds.

The CRC 102-A system consists of a computer proper and the associated equipment necessary to provide all of the functions of a general purpose computer. They are specifically:

The CRC 102-A Computer

This machine is the outgrowth of the prototype model 102 developed and built by CRC. It is a binary, magnetic drum, serial computer housed in a single cabinet, complete with power supply, all logical elements and air conditioning equipment.

The CRC 102-A Console

This is a desk of a conventional design into which has been built the operator's console. This console consists of six push buttons, six toggle switches, and seven indicator lights. A Flexowriter, mounted on the desk, is electrically connected to the computer so that signals from its keyboard may be used to fill and control the computer.

The CRC 126 Magnetic Tape Unit

The computer may utilize up to seven of these tape units, which are connected to it through a common bus. Each tape unit contains logical circuitry which enables it to search for information stored on its tape independently of the computer.

The IBM Machines

The computer is capable of accepting data from, and transmitting data to, IBM cards and requires two specially modified IBM machines to provide this feature. These modifications are provided by IBM upon request when ordering the IBM machines. Two machines are used separately, one to read, and one to punch the cards.

General Machine Operation

Essential Computing Machine Terminology and Principal Units of the Computer Proper:

1. Memory - The "memory" is the computer's information-storage device. It is distinguished from other storage devices by the fact that it is automatically controlled by the computer, and any part of it is automatically accessible to the computer. Memory may be internal (Magnetic Drum), or external (Magnetic Tape). Memory may also be classified as "volatile" or "non-volatile", according to whether stored information will be lost or not, if the computer's power supply is turned off. Magnetic drum and magnetic tape are both non-volatile.

2. Cell - The allotted space in the memory for the basic unit piece of information in the computer (word) is referred to as a "cell".

3. Address - The physical position of a cell in the memory is identified numerically. The "address" of a cell is the number that designates a particular memory location.

4. Instruction - An "instruction" is a particular operation the computer is capable of obeying, that is, add, subtract, etc.

5. Word - The basic unit of information in the computer is referred to as a "word". A word may be a numerically coded command, a nine-digit decimal number with a plus or minus sign, or six characters of alphabetical information, occupying a one-cell storage space in the memory.

6. Command - A word which causes the computer to perform an operation is called a "command". A command word consists of a numerically coded instruction and the memory addresses of the operands.

ARITHMETIC COMMAND: Calls for one of the operations of arithmetic.
LOGICAL COMMAND: Used for certain operations which modify data for processing, and also in requiring the computer to make a decision in order to decide what course of action must be taken.
INPUT and OUTPUT COMMANDS: Direct the computer to accept information from, or deliver information to, external equipment.

7. Program - A "program" (often referred to as a routine or code) is the complete set of commands and data, expressed in computer language, and arranged in proper sequence, to direct the computer to perform a desired task.

8. Control and Arithmetic Unit - This unit of the computer is basically a single section. The control unit interprets the commands specified in the program, and informs the memory and the arithmetic unit how to communicate with one another. The arithmetic unit temporarily stores the operands and the result of a machine operation.

The computer is internally programmed by proper positioning of commands and numerical data in the memory. Computation is started in a "pushbutton" fashion; and because of the automatic feature, intervention by the operator is very rare. The following analogy will perhaps serve as a brief illustration of the general machine operation.

Consider the memory as a filing cabinet containing 1,000 numbered drawers, and the control section as a very moronic, but efficient, secretary who can count, and who can understand

and execute a very limited number of fundamental operations. Consider the arithmetic unit as her tools for executing these operations. She cannot think creatively, but will faithfully follow instructions; consequently, we load the filing cabinet with a pre-determined sequence of commands and numbers and tell her to start in drawer 0100, in which we have stored the first of the sequence of commands.

She immediately opens drawer 0100 and automatically interprets its contents as a command. For example, this drawer may contain the number 35 0127 0622 0736, but she is able to interpret it as saying "Take the number you find in drawer 0127 and add (code 35) it to the number in drawer 0622, putting the result in drawer 0736, after discarding whatever else was in drawer 0736." Also, she makes certain to retain the original values in drawer 0127 and 0622. Having done so, she proceeds to open drawer 0101, interprets and executes the command herein, and continues this procession sequentially through the cabinet drawers. However, her procession may be interrupted and she may be transferred elsewhere for her next command. For example, as she proceeds through drawers 0102, 0103, ..., 0136, she may come upon the number 34 0200 0165 0300 in drawer 0136, which she interprets as "Compare (code 34) the number in drawer 0200 against the number in drawer 0165 and if the number in drawer 0200 is larger, take your next instruction from drawer 0300, otherwise continue to drawer 0137. Assuming this test does send her to drawer 0300, she would then continue in the sequence 0300, 0301, ..., etc., until she would again be rerouted, or finally told to stop.

At this time we wish to place emphasis on the brevity of this analogy, and explain that it is only intended to help unveil any mystery that accompanies the automatic feature and "thinking" capacity of the machine. An attempt has been made to illustrate these powerful machine tools in use, and at the same time familiarize the programmer with some of the problems he will have to cope with in preparing a program. It should be clear now that the machine will only do what it is told to do, and optimum efficiency through programming will be achieved only when we are thoroughly familiar with the nature of the machine's operation and its functional capacity.

Consider, now, the skeleton program illustrated in the previous analogy, and how it would look when written on a code sheet by the programmer. First, of course, the programmer would determine the necessary logical and arithmetic operations. Next, he would decide on the most efficient memory storage for the commands and data and he would indicate this on a standard code sheet as shown below. The column headings I, m_1, m_2, m_3 represent the instruction digits, and the memory addresses of a command word.

Address (drawer number)	I	m_1	m_2	m_3	Remarks
0100	35	0127	0622	0736	Add contents of 0127 + contents of 0622 and store the sum in 0736.
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	
0136	34	0200	0165	0300	If contents of 0200 greater than contents of 0165 go to 0300
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	

In order that the programmer properly utilize a word to represent a command, he must be aware that the contents of the command word will be scrutinized by the control unit as three addresses and an instruction code. In most commands the m_1 and m_2 addresses name the operands, and m_3 is the put-away address of the result.

This illustration is not intended as a tutorial example in programming; however, it is felt that the general appearance of a code and a few comments relative to the command structure at this time will be beneficial to the reader.

II

NUMBER SYSTEMS

The subject of primary importance in connection with an electronic computer is the machine language. Communication with the machine is an essential prerequisite for programming.

When considering the subject of arithmetic and its associated operations, our early teachings and experience familiarize us, as a rule, only with the decimal number system and the digits 0, 1, . . . , 9. However, since a number system is fundamentally a scheme for counting, it is quite feasible for us arbitrarily to choose a number system to suit our needs.

The binary number system, which concerns itself only with the digits 0 and 1 (referred to as bits), was chosen with the design of the 102-A. Arithmetic operations in the binary system are extremely simple by virtue of the existence of only these two numerical characters, and consequently, lead to simple machine mechanization. For example, representation of the "0" or "1" bits can be made to correspond to the cut-off or conducting state of a vacuum tube, to the opposite states of polarity of an electrical charge, to the existence or non-existence of a pulse, or any "yes" or "no" form for differentiating between 0 and 1. Now, analogous representation of the 0, 1, . . . , 9 digits in the decimal system would be impossible in most cases; however, where it would be possible, it would perhaps require ten different voltage levels.

Number Representation

The method of representing numbers in the binary system becomes quite reasonable to us when we recall the fundamental principals of the familiar decimal system.

As an example, consider the number 3974.23. The following equality, $3974.23 = 3 \times 1000 + 9 \times 100 + 7 \times 10 + 4 \times 1 + 2 \times \frac{1}{10} + 3 \times \frac{1}{100}$, reminds us of the expression, "3 is the thousands digit; 9 is the hundreds digit; etc.," and illustrates the weight factor associated with each digit. We will find it more advantageous, however, if we write this equality with proper powers of ten, i.e.,

$$3974.23 = 3 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 4 \times 10^{0*} + 2 \times 10^{-1} + 3 \times 10^{-2},$$

where we consider 10 as the base of the number system. Thus, the true meaning of a number in any system is best illustrated when expanded as shown here.

In a similar fashion the number 11011.001 in the binary system (base 2) would be written as

$$11011.001 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 1 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.$$

Just as the decimal point separates the integral and fractional parts of a decimal number, so does the binary point.

Conversion from Binary to Decimal

Number representation in systems other than base 10 frequently conveys no information to us unless first converted to the equivalent decimal notation. Nevertheless, the corresponding decimal number can always be obtained from the expanded form of the binary representation if one would merely execute the indicated operations using decimal arithmetic. Let us then consider the previous example.

$$11011.001 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 + \frac{0}{2} + \frac{0}{4} + \frac{1}{8}$$

$$= 27.125 \text{ (decimal equivalent)}$$

*By definition, any number to the zero power equals 1.

Conversion from Decimal to Binary

Conversion in this direction must be considered independently for integral numbers and fractions. Mixed numbers, on the other hand, must be resolved into the sum of the whole plus the fractional parts. Consider the following example:

Example 1. Convert 937 to an equivalent binary number. Now, conversion in this direction implies the determination of the digits corresponding to the b's in the following expansion.

$$937 = b_n x 2^n + b_{n-1} x 2^{n-1} + \dots + b_1 x 2^1 + b_o x 1,$$

where n indicates the largest power of the new base necessary for representation of the given decimal number. A well known algorithm for obtaining these coefficients is successive divisions of the decimal number by the base (2) until the quotient is zero. The remainder at each step, beginning with the units position, is the required digit; i. e., dividing 937 successively by 2 we have

		Remainder	
2	937	1	units digit
2	468	0	
2	234	0	
2	117	1	
2	58	0	
2	29	1	
2	14	0	
2	7	1	
2	3	1	
2	1	1	
	0		

Hence, the equivalent binary number is 1110101001.

Example 2. Convert 0.8493 to an equivalent binary fraction. In this example the b's of the following series must be obtained.

$$0.8493 = \frac{b_{-1}}{2} + \frac{b_{-2}}{2^2} + \frac{b_{-3}}{2^3} + \ldots.,$$

where the negative subscripts correspond to the negative powers of the base, or fractional weight factors. It is well to note at this time that an exact fraction in one number system may become a repeating fraction when converted to another number system.

The algorithm used here is successive multiplication of the decimal number by the new base (2) and the integral part of the product is the required digit in the new number system, beginning with the most significant fractional digit. The fractional part of the product is the new multiplicand for the next successive multiplication by the base. The successive multiplication is continued until the fractional part of the product becomes zero; or the result is repetitive; or until a sufficient number of digits have been obtained. This process is illustrated below in tabular form.

CONVERSION TO BINARY

	Binary Digit		Fractional Part of Product
0.8493 x 2 =	1	.	6986
0.6986 x 2 =	1	.	3972
0.3972 x 2 =	0	.	7944
	1	.	5888
	1	.	1776
	0	.	3552
	0	.	7104
etc.	1	.	4208
	0	.	8416
	1	.	6832
	1	.	3664
	0	.	7328

The converted binary fraction, then, is 0.110110010110---.

Relationship between the Binary and Octal Number Systems

In the binary system a number of any size or a fraction of any precision requires a long string of zeros and ones. In practice this becomes very difficult to work with outside the computer, so a substitute system which incorporates the binary system exactly is usually substituted. One system which lends itself to this definition is the octal system. The octal system has as a base the number eight, which is equal to 2^3. Thus any combination of three binary digits gives a digit in the octal system (0, 1, . . . , 7). All the possible combinations of three binary digits and their octal equivalents are shown below in Table 1.

Binary Group	Octal Digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

TABLE 1

Rapid conversion between the binary and octal systems is trivial inasmuch as every triad of binary digits, marked off from either side of the binary point, correlates an octal digit. The following example of a binary number marked off in triads illustrates the simplicity by which conversion may take place in either direction.

$$\begin{array}{ccccccc} 11 & 111 & 100 & . & 110 & 111 \\ \updownarrow & \updownarrow & \updownarrow & & \updownarrow & \updownarrow \\ 3 & 7 & 4 & . & 6 & 7 \end{array}$$

In order to validate this operation, one could easily show that these two numbers are precisely equal in magnitude, perhaps convert each to a decimal number and compare the results.

In the light of the above example, emphasis must be placed on the impossibility of conversion, in a similar fashion, between the binary and decimal systems. The reason, of course, is a lack of correlation between a single decimal digit and a fixed set of binary digits; that is, the inability to express the base 10 as an integral power of 2 (in contrast to the binary and octal bases, where $2^3 = 8$). However, for the sake of proving a point, let us attempt such a conversion from a decimal to a binary number. This attempt might not appear invalid in as obvious a fashion as it would if we were to first attempt a conversion in a binary to decimal direction.

For example, we will "convert" the decimal number 89 in this manner and examine the result; that is,

$$\begin{array}{cc} 8 & 9 \\ \downarrow & \downarrow \\ 1000 & 1001 \end{array}$$

If we examine the result, 10001001, as a binary number, we have

$$10001001 = 1 \times 2^7 + 1 \times 2^3 + 1 \times 1 = 137,$$

which is not equivalent to 89. This falacy was not obvious, however, until we checked our result.

Now, let us attempt to convert in the opposite direction - binary to decimal. Choose the number, 11001011,

$$\begin{array}{cc} ? & ? \\ \uparrow & \uparrow \\ 1100 & 1011 \end{array}$$

and we are obviously "stopped in our tracks," because there are no decimal digits corresponding to 1100 and 1011. However, if we thought we could merely write the decimal equivalents to 1100 and 1011, we would have

$$\begin{tikzcd} 12 & 11 \\ 1100 \arrow[u] & 1011 \arrow[u] \end{tikzcd}$$

but

$$11001011 = 203 \text{ (decimal equivalent).}$$

Just how many bits in a set then, would we choose to represent a single decimal digit? Inasmuch as we know sets of three bits convert to the base 8, apparently four bits convert to the base 16 ($2^4 = 16$). Consequently, we just can't convert between decimal and binary numbers digit by digit, as we can between octal and binary numbers.

Arithmetic in the Binary and Octal Number System

Arithmetic operations in any number system are fundamentally the same; that is, they are simply variations in methods of counting. Thus, if one would commit to memory the following addition and multiplication tables of the binary and octal number systems, he could perform arithmetic operations in these systems with as much ease as he does in the decimal system.

BINARY

ADDITION

	0	1
0	0	1
1		10

MULTIPLICATION

	0	1
0	0	0
1		1

OCTAL

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1		2	3	4	5	6	7	10
2			4	5	6	7	10	11
3				6	7	10	11	12
4					10	11	12	13
5						12	13	14
6							14	15
7								16

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1		1	2	3	4	5	6	7
2			4	6	10	12	14	16
3				11	14	17	22	25
4					20	24	30	34
5						31	36	43
6							44	52
7								61

Examples of arithmetic operations in the binary and octal number systems:

ADDITION	BINARY	OCTAL
	1011011	306702
	1101110	531066
	11001001	1037770
MULTIPLICATION	110101	6753
	1011	506
	110101	51602
	110101	426270
	1101010	
	1001000111	4334502

DIVISION

```
             1011                506
110101/1001000111       6753/4334502
        110101               42627
         1001111               51602
          110101               51602
           110101
           110101
```

This, then, is an introduction to the binary and octal number systems as related to our familiar decimal system. The need for an understanding of these number systems will be further emphasized in the following sections.

III

MEMORY

The model 102-A is a magnetic-drum-memory machine. The drum is a cylinder coated with a magnetic material, and measures approximately twelve inches in diameter and six inches in length.

Once a small spot on this material is magnetized to either of two possible states, it will remain so indefinitely unless changed. Since we represent the values "0" and "1" by these two states, a series of magnetized spots corresponds to a sequence of binary digits. This is the way in which information is recorded on the surface of the drum. These spots are located on circular tracks (channels) around the circumference of the drum; each track is divided into 64 sectors (cells); each sector has a 42 binary digit storage capacity (i. e., one word, see Section IV).

The drum is enclosed in a cylindrical housing within the computer, and rotates continuously about its vertical axis at the rate of 40 rps. As the drum rotates, information is "read off" or "written on" the drum via a channel read-write head. The read-write heads are embedded in the drum housing at equal vertical intervals in order to fix the physical position of each channel.

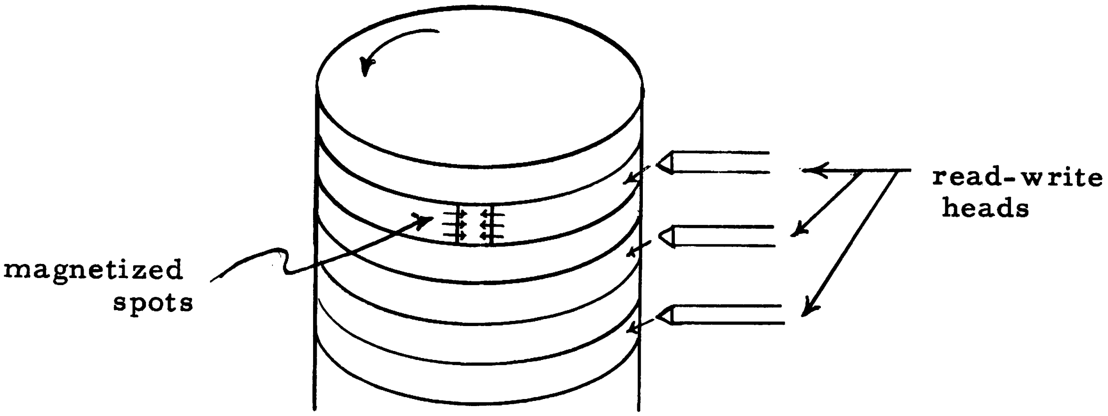


Figure 1

The permanent memory, word channel, buffer register, recirculating registers, and two clock channels comprise twenty-one memory channels.

The Permanent or Main Memory consists of 16 channels of 64 cells each, or a total of 1024 words of permanent storage. Permanent storage implies a non-volatile type of magnetic drum recording, where the magnetic pattern remains on the memory when the machine power is turned off.

In order to refer to these 1024 main memory cells in a convenient manner we shall assign numbers to each channel and each sector. However, we shall use numbers which are most convenient for the computer. Instead of numbering the sixteen channels in the normal manner, we shall use, for reasons which will become obvious later, the following sixteen numbers:*

00, 01, 02, 03, 04, 05, 06, 07, 10, 11, 12, 13, 14, 15, 16, 17

For the same reasons, the sixty-four sectors will be assigned the following numbers: *

00, 01, 02, 03, 04, 05, 06, 07, 10, 11, 12, ...
... 65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77

Each main memory cell is assigned a four-digit octal address. The two left octal digits represent the channel number and the two right octal digits specify the sector number. The addresses are, then,

0000, 0001, ... 0007, 0010, ... 0077, 0100,...
0177, 0200, ... etc. to 1777.

* The reader should recognize these numbers as "octal" numbers.

The single read-write head associated with each channel will write information on the drum as it rotates past and simultaneously erase what was previously written; but will read from the drum and not cause an erasure. It is obvious now that any given cell in the main memory is made available to the computer once during a drum revolution.

Since the drum revolves at the rate of 40 rps., one drum revolution requires .025 sec., or 25 ms. (1000 ms = 1 sec.). Hence, the time required for a sector to pass its respective channel read-write head - referred to as one word time - is 25/64 ms., or approximately .4 ms.

The Word Channel is a single track on the drum. The sole purpose of the word channel is to monitor the reading and writing of words between the control and arithmetic units and the memory.

During filling of the computer, during computation, and for output of information, the word channel serves to physically locate the sector position of a pre-selected channel. Each sector of the word channel has permanently recorded in it a sequence of binary digits equivalent to one of the 64 sector numbers to which an address may refer (i. e. - octal numbers 00 thru 77) - (Figure 2). By means of the word channel read-write head the computer is capable of locating any one of these 64 sector positions.

By means of the read-write heads the sector addresses of all channels are correlated to those of the word channel. This correlation is brought about by virtue of the fact that when a word-channel sector, containing the permanently recorded number, passes its read-write head the sector passing a read-write head of a main memory channel is automatically located. However, in

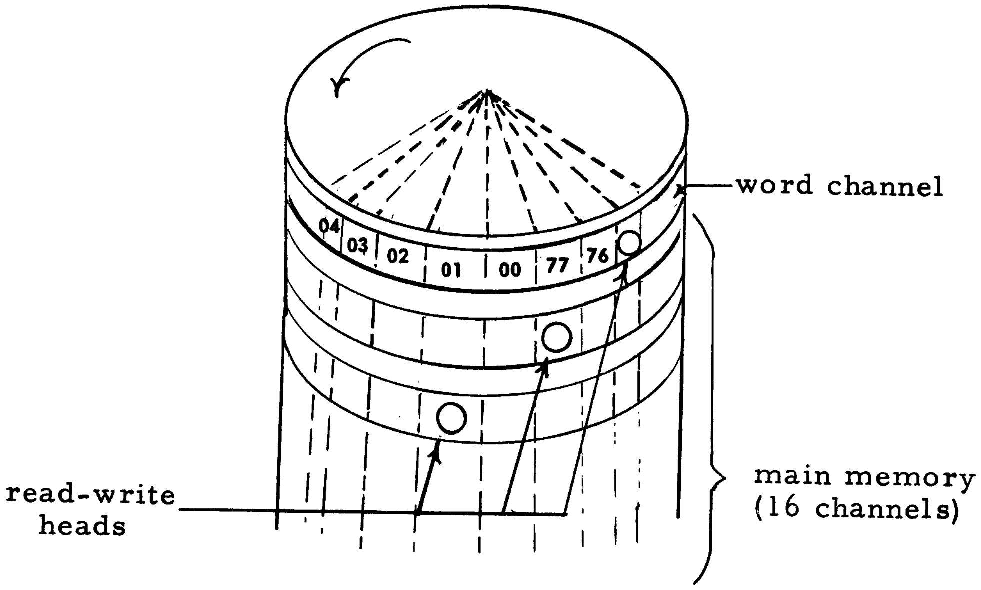


Figure 2

order to allow for the time necessary to recognize the sector adress recorded in the word channel, it will be the sector address of the next cell to pass under the main memory channel read-write head(see Figure 3).

There are four one-word Recirculating Registers associated with a single channel on the drum. They are referred to as the E, F, G, and H registers, and act as temporary one-word storage units for the arithmetic and control units of the computer. Distinct read and write heads are assigned to each one-word register, as indicated in the diagram (Figure 4) of this single track on the drum. Information held in these registers is constantly recirculating as the drum revolves.

Let us suppose a word is written on the drum between the read and write heads of a register. As the drum rotates (indicated

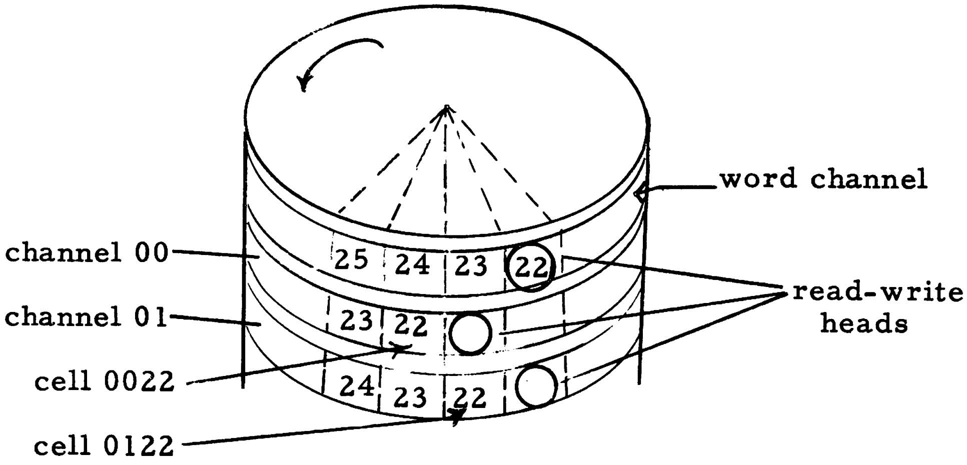


Figure 3

by arrow), the bits are read off, one by one, at the read head and recorded back on the drum through the write head. Current information in a register will continue to recirculate until it is written over by new information entering that register. This is a rather superficial explanation of the behavior of these registers, but adequate for the present time.

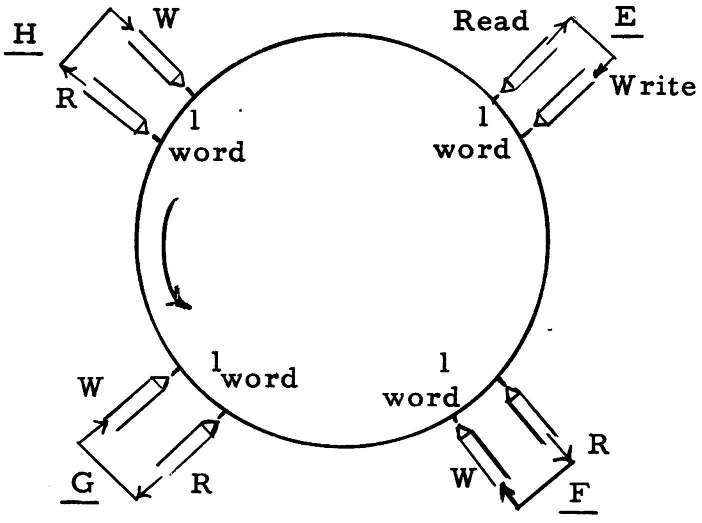


Figure 4

The function of these registers is as follows:

E - Arithmetic unit register. Also, intermediate storage for information during input and output.

F - Arithmetic unit register.

H - Control register.

G - "Hybrid" register; i. e., it participates either as a control register or as an arithmetic unit register, depending on the phase of machine operation.

Although these registers are, in effect, storage in the memory, they are volatile in the sense that the meaning of their information is lost when the machine is turned off.

The Buffer Register is an eight-word recirculating register physically located on a distinct channel of the memory (see Figure 5). This register offers an extension of eight words of temporary (volatile) storage to the main memory. These eight words are accessible to the programmer by the addresses 2000 - 2007.

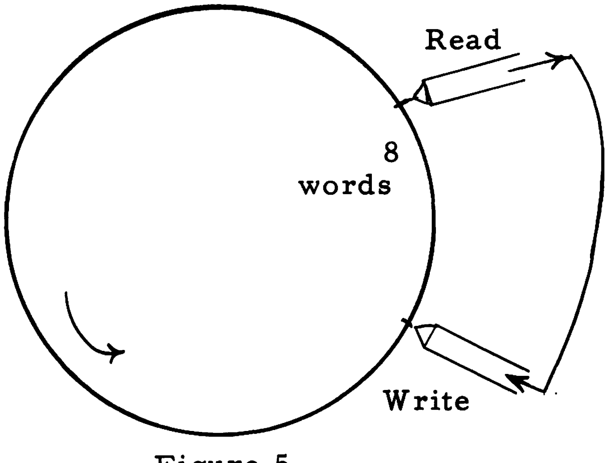


Figure 5

One of the outstanding features of the buffer register is the rapid access to its contents as compared with that of a main memory cell. This rapid access is achieved primarily by the fact that the buffer register recirculates and makes its contents available exactly eight times per drum revolution. Also, only the least significant octal digits of the word channel are inspected when a buffer cell address (channel 20) is recognized; that is, the least significant digits of the buffer register cells, 0 - 7, are synchronized with the least significant digits, 0 - 7, of the word channel eight times per drum revolution. General use of these cells for storage of often-used constants, or commands, in a routine will cut down computing time considerably.

Communication between the 102-A, punched card, and magnetic tape units is accomplished by means of the buffer register. These units are connected directly to the buffer, and any information entering or leaving the 102-A memory via punched card or magnetic tape must first be placed in the buffer register. The 102-A command list includes "buffer out" and "buffer load" commands, which permit the transfer of blocks of eight words between the main memory and the buffer register.

Two Clock Channels occupy two additional tracks on the magnetic drum memory. These clock channels contain the permanently recorded clock pulses that regulate the timing and synchronization of machine operation. There are forty-two pulses to a word, hence 64 x 42 or 2688 clock pulses over the entire channel. The circuitry of the machine enables clock pulses to be counted and recorded for determination of the unit time: one word.

Cell 2100. Although this is not a cell actually located on the drum, but a bit of logical circuitry in the machine, we will discuss it at this time since it is referred to by an address. This circuitry actually provides us with a cell of all zeros whenever needed. It is often called a "minimum access-time zero," and actually turns off the circuitry so that the machine reads nothing for one word time and the result is all zeros in the desired register. Effectively, it is absolutely nothing. The nature of cell 2100 will become more apparent after one becomes familiar with the control system for reading and writing of memory cells.

Cell 3000. This cell is in reality the G register previously mentioned. It is worthy of additional note now since it is also referred to by a memory address. Thus, the programmer is provided with the ability to remove information from the G register during a routine, and obtain knowledge which is very convenient for controlling the sequence of events of a program. Further clarification of the behavior of cell 3000 is given in the discussions of the control unit.

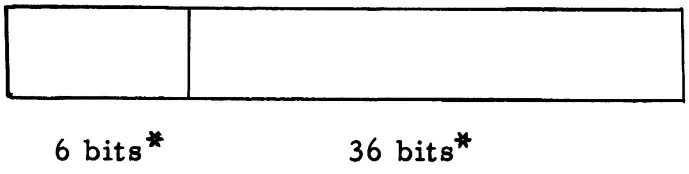
IV

WORD STRUCTURE

In the previous sections the "word" was depicted as being the unit piece of information in the machine - stored on the surface of the drum as a block of forty-two binary digits which can represent a command or a number.

The purpose of this section is to familiarize the reader with the function within the machine of the forty-two binary digits of a word, and the equivalent octal representation of the word outside the machine.

Basically all words are considered as split into the two major sections shown below. The following discussions will illustrate the word structure for commands and numbers with respect to these two sections.



Commands

All command words are comprised of an instruction code, and addresses referring to memory locations of the operands involved in the operation specified by the instruction code.

* "Bit" is the accepted contraction for "binary digit."

Instruction	Addresses		
I	m_1	m_2	m_3
6 bits	12 bits	12 bits	12 bits

The instruction section is that physical portion of the word structure allotted for the numerical code that designates the operation to be performed. These six instruction bits are represented by two octal digits according to the code in Table 4 , page VIII-1.

The thirty-six bit section of a command word is scrutinized by the machine as three independent twelve-bit sections; namely, m_1, m_2, and m_3. Each twelve-bit section, represented by four octal digits, is an address referring to a memory cell. In most commands the m_1 and m_2 sections are the addresses of the operands, and the m_3 section is the put-away address of the result (thus, the 102-A is classified as a three address machine).

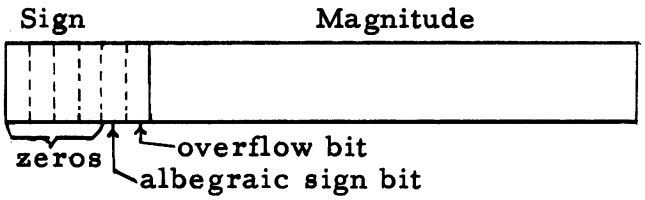
Emphasis must be placed on the fact that the octal numbers to be placed in the m_1, m_2, and m_3 sections of a command word are only addresses of memory cells and not the actual numerical values of the operands. Thus, a command, when stored as a sequence of binary digits in the machine, would have no meaning until interpreted sectionally by the control unit (Section VII).

Example of a command that instructs the machine to add the numbers in cells 1026 and 1037, and record the result in cell 0345.

I	m_1	m_2	m_3
35	1026	1037	0345

Numbers

The constituents necessary for complete numerical representation are the sign and magnitude. Consequently, in the 102-A all numbers are represented according to the following schematic.



The magnitude section stores any number that can be represented by a sequence of thirty-six binary digits. All arithmetic operations consider the entire magnitude section as a thirty-six bit number. It is noteworthy that the binary point does not occupy any physical space in the word; however, during computation its position is kept track of by programming.

The six-bit sign section consists of four zero bits, the sign bit, and the overflow bit (see above diagram).

Sign bit: A binary "0" is plus and a binary "1" is minus.

Overflow bit: During normal computer operation this bit will be "0". If the result of an arithmetic operation exceeds the capacity of the thirty-six bit magnitude section, a "1" will carry over into this bit position. Unless the machine is programmed to do otherwise, it will immediately halt when overflow occurs.

The following examples illustrate numbers as they would appear on the coder's sheet and their corresponding binary configurations within the machine.

Example 1: The negative octal number 0. 137670245206

Sign		Magnitude											
0	2*	1	3	7	6	7	0	2	4	5	2	0	6
000	010	001	011	111	110	111	000	010	100	101	010	000	110

Example 2: The positive octal number 375. 6672

Sign		Magnitude											
0	0	3	7	5	6	6	7	2	0	0	0	0	0
000	000	011	111	101	110	110	111	010	000	000	000	000	000

The above examples suggest that in order to use numerical data in the computer, we must first find the equivalent octal number representation, since undoubtedly numerical data relative to a given problem will usually be available in the decimal number system. Fortunately, decimal numbers may be entered directly into the 102-A, and manual conversion from decimal to octal is not necessary, since the computer can be programmed to do this.

Input in the decimal mode is accomplished by a simple preliminary setting of the proper control, and the computer is prepared to accept a group of four bits per decimal digit from the Flexowriter.

*For input, "-" ("NEG." Key) may also be used to enter a negative sign.

Example 3: The negative decimal number 928.75

Sign		Magnitude								
0	2	0	0	0	0	9	2	8	7	5
00	0010	0000	0000	0000	0000	1001	0010	1000	0111	0101

It should be apparent that this number, as it appears in the machine, can only be utilized as a method of keeping track of the decimal digits of the original number. If, for example, this binary configuration was mistakenly used in normal computer operation, the machine would consider all thirty-six bits as one number, and not scrutinize them four at a time. Since we can only consider these groups of four bits per decimal digit as a code, we refer to them as binary coded decimal digits in the machine. Consequently, prior to the use of decimal numbers in normal machine operation, it will be necessary to automatically convert them within the machine, by means of a conversion routine, to a true binary representation.

V

FUNDAMENTALS OF PROGRAMMING

When a problem involving numerical computation is delegated to a human operator for execution on a suitable desk type machine it is necessary that he be supplied with a list of instructions and data, or "program." Similarly, if we expect an automatic electronic computer to do our work for us, we must supply it with a "program."

However, when dealing with an electronic computer, the probelm of supplying a "program" becomes two-fold: (a) Programming, or preparation of the code; and (b) Proper insertion of the code into the machine. In this section we will discuss only the first of these problems.

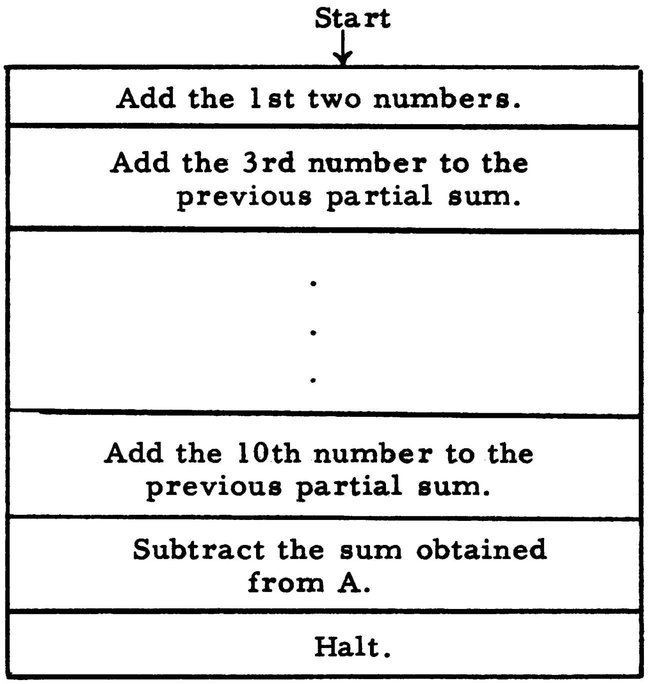
The first phase of programming is planning. Once a suitable plan is formulated, it is best illustrated by means of a flow-chart. A flow-chart is a written schematic of the logical sequence of operations necessary for the completion of the given problem. In the second phase, each operation noted in the flow-chart is represented by one or more 102-A commands, which involves choosing suitable storage space in the memory and translating the program into numerically coded form.

For the sake of simplicity, we will choose a trivial example to illustrate these essential phases of programming. Consider, then, the problem of obtaining the difference between a given number, A, and the sum of a given set of ten numbers, and storing the result ready for printing. The development of a program which will supply a satisfactory answer to this problem is explained in the following paragraphs.

The procedure of obtaining a flow-chart is simplified by the use of formulas and algebraic symbols, whenever possible. For example, if we represent the ten given numbers by c_1, c_2, . . . c_{10}, we can write the following equation:

$$\text{Answer} = A - (c_1 + c_2 + \ldots + c_{10})$$

Hence, the logical sequence of operations necessary to obtain the answer becomes more apparent to us. One of the many recommended forms of writing a flow-chart is the following:



Start

Add the 1st two numbers.

Add the 3rd number to the previous partial sum.

.
.
.

Add the 10th number to the previous partial sum.

Subtract the sum obtained from A.

Halt.

The program is obtained, now, by simply transforming the steps in the flow-chart into satisfactory 102-A commands. First, of course, the programmer must know the functional capabilities of the 102-A commands, and how they must be coded.

Rather than attempt to introduce the student to the entire command list at one time, we will familiarize him with command usage by means of tutorial examples such as this one. The entire command list, nevertheless, is explained in detail in Section VIII.

In this example we will need commands which will add and subtract numbers, and a command which will halt computation. The word structure for these commands is as follows:

ADD - "ad" - code 35

I	m_1	m_2	m_3
ad	address of augend	address of addend	address of sum

The "add" command will add the contents of memory cells m_1 and m_2 and store the sum with proper sign in memory cell m_3.

SUBTRACT - "su" - code 36

I	m_1	m_2	m_3
su	address of minuend	address of subtrahend	address of difference

The "subtract"command will subtract the contents of memory cell m_2 from the contents of memory cell m_1, and store the difference with proper sign in memory cell m_3.

HALT - "ht" - code 22

I	m_1	m_2	m_3
ht	xxxx	xxxx	xxxx

Then "halt" command will halt automatic computer operation and put the computer in a state of "rest." The m_1 and m_2

addresses are irrelevant, that is, they can be any four octal digit numbers. The m_3 address is irrelevant for our present purpose.

It is noteworthy, once again, that when the command word is actually coded it contains only the memory addresses locating these quantities -- not the quantities themselves.

Once we have decided on suitable storage space in the memory for the commands needed in this program and for the eleven given numbers, we will be ready to complete the code. Fortunately, we can begin by storing the commands anywhere in the memory, but we are restricted by the fact that after choosing stoaage for the first command subsequent commands must occupy successive memory cells, since normal computer operation is sequential. For this problem, assume the first command to be stored in cell 0100 and the eleven numerical constants, c_1, c_2, . . . , c_{10}, A, in cells 0600 -0612, respectively. Also, working storage space (memory cells to store intermediate results) must be allotted. In this example we will choose cell 0700 in which to accumulate the answer.

Referring to the flow-chart, we will replace the prescribed operations by the proper 102-A commands, coded with the selected memory addresses. The final code shown below is in the standard format used by the programmer. Included, then, on every code sheet will be the word structure of the commands and numerical data, their respective memory addresses, and a "Remarks" column. The "Address" and "Remarks" columns are indispensable for coding subsequent commands after the first, during code checking, and as a future reference. Also, the "Address" column is needed when the program is filled into the computer (Section VI).

Address	I	m_1	m_2	m_3	Remarks	
0100	35	0600	0601	0700	$c_1 + c_2 \rightarrow 0700$. (*)	
0101	35	0700	0602	0700	$(0700) + c_3 \rightarrow 0700$. (**)	
0102	35	0700	0603	0700	$(0700) + c_4 \rightarrow 0700$.	
0103	35	0700	0604	0700	$(0700) + c_5 \rightarrow 0700$.	
0104	35	0700	0605	0700	$(0700) + c_6 \rightarrow 0700$.	
0105	35	0700	0606	0700	$(0700) + c_7 \rightarrow 0700$.	
0106	35	0700	0607	0700	$(0700) + c_8 \rightarrow 0700$.	
0107	35	0700	0610	0700	$(0700) + c_9 \rightarrow 0700$.	
0110	35	0700	0611	0700	$(0700) + c_{10} \rightarrow 0700$.	
0111	36	0612	0700	0700	$A - (0700) \rightarrow 0700$.	
0112	22	0000	0000	0000	Halt computation.	
0600	02	0000	0021	6532	c_1	Octal Numbers.
0601	00	0007	5063	1005	c_2	
.	.	.	.	.	.	
0611	02	0000	0110	7742	c_{10}	
0612	00	0000	0060	4306	A	

Problems that warrant the use of an electronic computer are certainly much greater in magnitude and more profound than the previous example. The need for more subtle programming techniques is evident. For instance, if the previous problem had been one that involved many more additions, it would be impracticable to repeat the "add" command as we have done. The

(*) - The arrow is used to symbolize "stored in;" i. e. - "The sum $c_1 + c_2$ is stored in cell 0700.

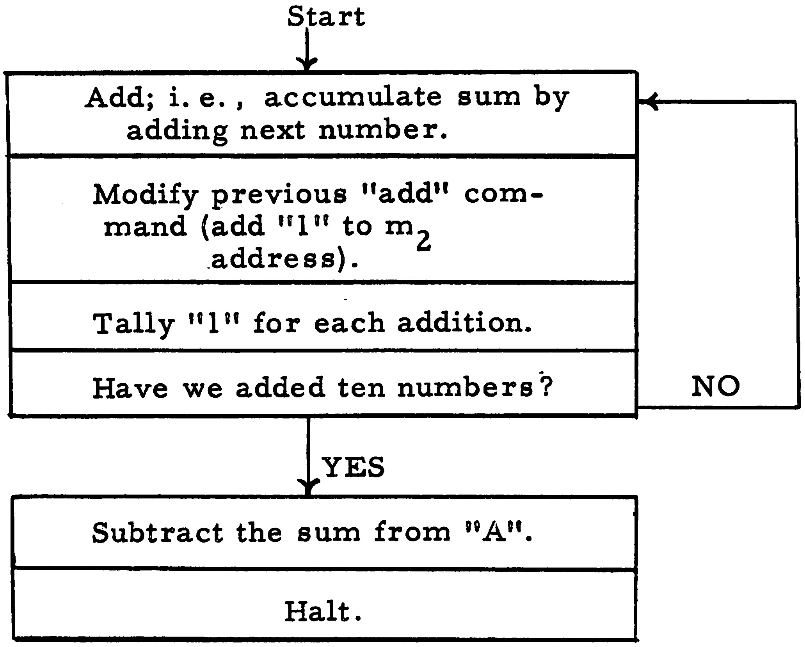
(**) - Parentheses used around the address of a cell indicates the contents of that cell. i. e. - "The sum c_3 + (Contents of cell 0700) is stored in cell 0700."

development of a more efficient code, and one which will be applicable regardless of the number of additions, is not difficult once we are more familiar with certain fundamental programming techniques and with the versatility of the command list.

Since our objective in seeking a more efficient program is the elimination of repetitive "add" commands, let us consider the code just developed and see what logical changes would have to be made in order to achieve this goal. Perhaps the computer will be capable of executing the logic we will prescribe.

Close examination of the nine successive "add" commands suggests that we possibly can use the same "add" command for all the additions if we can successively increase its m_2 address by "1", and then interrupt the machine's sequential mode of operation in order to repeat this modified "add" command. We must also control this cyclic procedure in some manner in order that it will terminate after the required number of additions have been made. To accomplish such a control, the machine must be made aware at the end of each cycle of the number of additions completed. Let us consider, then, the possibility of keeping a tally after each addition.

The essence of this procedure is more clearly illustrated by the following flow-chart:


Start
Add; i. e., accumulate sum by adding next number.
Modify previous "add" command (add "1" to m_2 address).
Tally "1" for each addition.
Have we added ten numbers?
NO
YES
Subtract the sum from "A".
Halt.

The ability of the machine to modify its own commands, and to make decisions is the crux of this procedure. In this example the prescribed logic can be fulfilled by means of the "add" and the "test magnitude" commands.

First, let us consider the modification of a command word, which is made possible by special use of the "add" command. A number can be added to a command word, and the instruction digits of the command word will be retained in the sum, if the address of the command word is named by m_1 of the "add" command. Consequently, we will write our code such that a word containing a "1" in its m_2 position will be added to the initial "add" command and recorded back in the same memory location. Thus, we will have modified the initial "add" command by augmenting its m_2 address by "1." Now, when the computer repeats this command (as indicated in the flow-chart) the next number will be added to the previous partial sum. Assuming the same memory storage for the program and the constants as in the previous example, the following assumptions and schematics illustrate this procedure:

Assume the initial "add" command to be

(0100) =	35	0700	0600	0700

and a "1" to be stored in the m_2 position of 0613

(0613) =	00	0000	0001	0000

Assume the second command in the program to be one which adds the two words above

(0101) =	35	0100	0613	0100

The result of this command will be the following addition:

	35	0700	0600	0700
	00	0000	0001	0000
Sum in (0100) =	35	0700	0601	0700

The reader is cautioned that this operation will be executed properly by the "add" command only if the <u>command word</u> is named by m_1 and the <u>number</u> by m_2. If these addresses are interchanged in the "add" command, the machine will add the two words as two algebraic numbers, and will not retain the instruction digits of the command word.

Secondly, we will consider how it is possible to cause the computer to interrupt the normal sequence of operation and repeat a set of commands. Referring to the flow-chart, we are reminded that our plan is to tally after each addition. Therefore, we will add "1" to a tally storage cell, and immediately ask the question: "Has the tally count reached 10?" This, then, is where the decision has to be made as to whether we will return to the "add" command, or continue in the program. If the answer is "no", we must cycle back and continue adding, but if the answer is "yes," we are ready to continue.

The 102-A "test magnitude" command, whose word structure is shown below, is designed to ask such a question and act accordingly.

TEST MAGNITUDE - "tm" - code 34

I	m_1	m_2	m_3
tm	address of magnitude$_1$	address of magnitude$_2$	address of alternate command

This command actually causes the computer to compare the magnitude sections of the two memory cells named by its m_1 and m_2 addresses, and make the following decision: If the magnitude portion of cell m_1 is greater than that of m_2, the address of the next command is automatically the m_3 address; otherwise, the computer will take the next command in the normal sequence.

In this problem a satisfactory decision will be made if we pre-store the octal equivalent of "10" in the memory and name its address as the m_1 portion of the "tm" command; also, we must name the address of the variable tally count as the m_2 portion; and m_3 must be the address of the initial "add" command.

The code for the solution of this problem follows.

Address		I	m_1	m_2	m_3	Remarks
0100	ad	35	0700	[0600]*	0700	Next number added to (0700) ((0700) initially zero).
0101	ad	35	0100	0613	0100	Modify (0100); i. e., add "1" to m_2 of 0100.
0102	ad	35	0614	0615	0615	Tally "1" in 0615.
0103	tm	34	0616	0615	0100	Has tally reached "9"? If not, next command 0100.
0104	su	36	0612	0700	0700	A - (0700)→0700.
0105	ht	22	0000	0000	0000	Halt computation.
0600		02	0000	0021	6532	c_1.
0601		00	0007	5063	1005	c_2.
.		.	.	.	.	.
0611		02	0000	0110	7742	c_{10}.
0612		00	0000	0060	4306	A.
0613		00	0000	0001	0000	Modifier for 0101.
0614		00	0000	0000	0001	"1" for tallying.
0615		00	0000	0000	0000	Tally count storage, initially zero.
0616		00	0000	0000	0012	Octal equivalent of "10" as gauge for 0103.
0700		00	0000	0000	0000	Working storage, initially zero.

This code could be used, then, to add as many numbers as we wish. We would merely have to change the contents of the word, 0616, which acts as the gauge in the "test magnitude" command.

* Brackets are used in a command word to indicate which address in the word structure will be modified in the course of machine computation.

The next example will introduce the reader to two more powerful machine tools, and a technique for using them.

Consider the problem of adding forty numbers which are stored in the memory in such a way that every storage word contains two of the given numbers. Assuming each number to be six octal digits or less, and the two numbers "packed" in the same storage word to be of the same sign, the following diagram illustrates how each number will occupy half of the magnitude section of the storage word.

00	0506 73	11 4752

In this case the two numbers are + 50673 and + 114752.

In addition to the commands previously discussed it will be necessary in this problem for the student to be familiar with the "shift magnitude" and "extract" commands. The word structure and function of these commands are as follows:

SHIFT MAGNITUDE - "sm" - code 30

I	m_1	m_2	m_3
sm	address of word to be shifted	address of shift-control word	address of shifted result

Under the influence of this command, the machine will shift the magnitude section of a word a specified number of bit positions to the right or left; zeros will appear at either end of the magnitude section to replace digits that are shifted off the other end. The sign portion will be unchanged. The m_1 address of the "sm" word structure identifies the word to be shifted; the m_3 address specifies the memory address in which to store the shifted result; the m_2 address designates the word whose contents specify the direction and number of shifts. The contents of m_2 must be formed by the programmer in the following manner:

(m_2) =	direction	number of shifts

The number of shifts is specified by the magnitude section as an octal number; the direction of shift is specified by the sign section, that is, a negative number will cause a right shift and a positive number will cause a left shift.

Also available, the "shift logically" command ("sl" - code 27) is identical to the "shift magnitude" command except that the entire word is shifted.

EXTRACT - "ex" - code 32

I	m_1	m_2	m_3
ex	address of word to be copied	address of extractor	address of cell altered by extract

The "extract" command permits selected bits from one word to be copied into the corresponding bit positions of another word. This operation functions in such a way that the binary digits of (m_1), which are in the same positions as the binary "ones" of (m_2), replace the corresponding binary digits of (m_3), and the bit positions of (m_3) corresponding to the binary "zeros" of (m_2) will not change. It may be helpful to think of the extractor (m_2) as a sieve, in which "1" bits represent the holes. Bits of (m_1) are dropped through this sieve into (m_3).

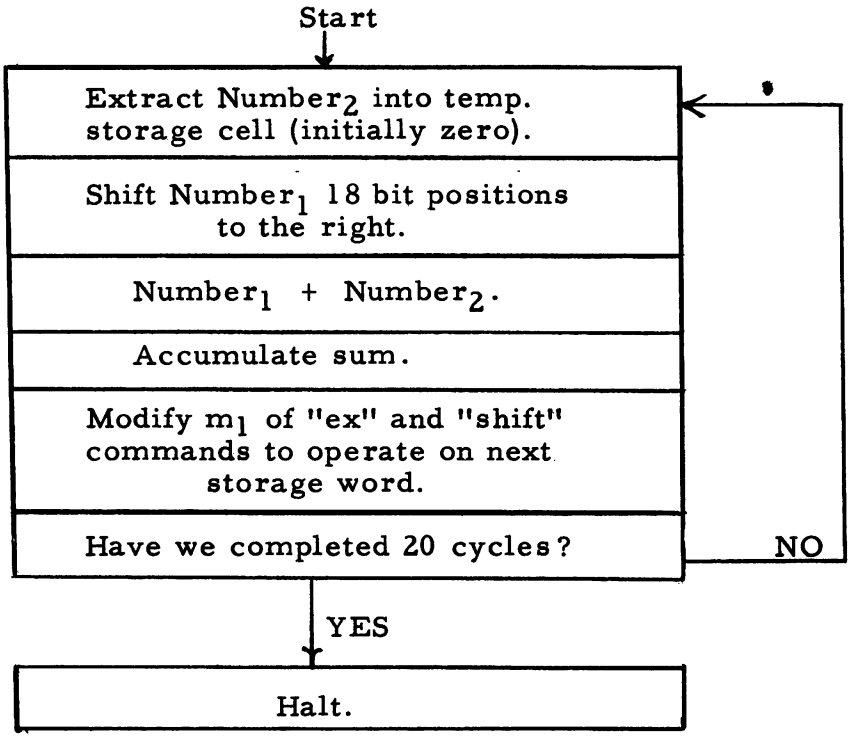
The availability of these two commands permits us to consider the following plan for the addition of the given forty numbers, and leaving the result stored in the memory. Consider, again, the plan of "packing" two numbers in a single word.

sign	Number$_1$	Number$_2$

and the steps necessary to "unpack" and add them:

1. Extract Number$_2$ into temporary storage.
2. Shift Number$_1$ completely to the right (18 bit positions).
3. Add the two numbers.
4. Accumulate sum.
5. Repeat this cycle until all forty numbers have been added (20 cycles).

Now incorporating these ideas in the form of a flow-chart, we have:


Start
Extract Number$_2$ into temp. storage cell (initially zero).
Shift Number$_1$ 18 bit positions to the right.
Number$_1$ + Number$_2$.
Accumulate sum.
Modify m$_1$ of "ex" and "shift" commands to operate on next storage word.
Have we completed 20 cycles?
NO
YES
Halt.

The corresponding code which we will now write will introduce the student to the use of the buffer register, and a more sophisticated way of making a decision than shown in the previous example. Assume the first command to be stored in 0200; the given numbers in the twenty cells, 0300-0323; program constants in cells 0324-0327.

Address	I	m_1	m_2	m_3	Remarks
0200	ex 32	[0300]	0324	2000	Number$_2$ → 2000.
0201	sm 30	[0300]	0325	2001	Number$_1$ shifted to right and → 2001.
0202	ad 35	2000	2001	2002	(Number$_1$ + Number$_2$) → 2002.
0203	ad 35	2002	2003	2003	Accumulated sum + (2002) → 2003.
0204	ad 35	0200	0326	0200	Modify (0200);i. e., add "1" to m_1 of 0200.
0205	ad 35	0201	0326	0201	Modify (0201);i. e. -add "1" to m_1 of 0201.
0206	tm 34	0327	0200	0200	Have we finished adding; i. e., is m_1 of 0200 = 0324 yet?
0207	ht 22	0000	0000	0000	Halt.
0300	00	0506	7311	4752	
0301	02	6362	4700	4430	
.	.	.	.	.	Given data.
0323	00	0004	0211	1252	
0324	02	0000	0077	7777	Extractor for 0200, extracts 6 octal digits and sign.
0325	02	0000	0000	0023	Shift constant for 0201; shift right, 19 bits.
0326	00	0001	0000	0000	Modifier for 0204, modifies (0200).
0327	00	0324	0324	2000	"Gauge" for test in 0205 .
2000	00	0000	0000	0000	Working storage, initially zero.

Use of buffer register cells in this program does not imply a new programming technique, but is intended to show the reader that they are used exactly like main memory cells. The advantage in using cells 2000 - 2007 is that they are accessible to the computer eight times as fast, on the average, as are main memory cells.

The method which we used in this example to cause the computer to make the decision relative to cycling back is worthy of additional note. This is the method, perhaps, that most programmers would use inasmuch as it eliminates tallying. The fact that we are augmenting the m_1 address of (0200) by "1" each cycle provides us with a tally. Since the last two numbers to be added are stored in 0323, and since (0200) is modified at the end of the cycle, the contents of command word 0200 at the end of the last cycle will be:

(0200) =	32	[0324]	0324	0200

Hence, if we are cognizant of this fact and store a word whose magnitude section is equal to that noted above, we will have a "gauge" word (or limiter) for the "test magnitude" command in 0205. The reader should recall now that the "test magnitude" command causes the computer to take its next command from the m_3 address only if the contents of m_1 is greater than the contents of m_2. Consequently we chose to store the following word in 0327:

(0327) =	00	0324	0324	2000

Now, when the computer executes the "tm" command in cell 0206

(0206) =	34	0327	0200	0200

it will recognize equality between (0327) and (0200), and take its next command from 0207, which is the "halt" command.

The next example is intended to instruct the student in the use of the "test for overflow marker" command as a means of transferring computer control away from the normal sequential mode of operation.

The word structure and function of this command are as follows:

TEST FOR OVERFLOW MARKER - "to" - code 27

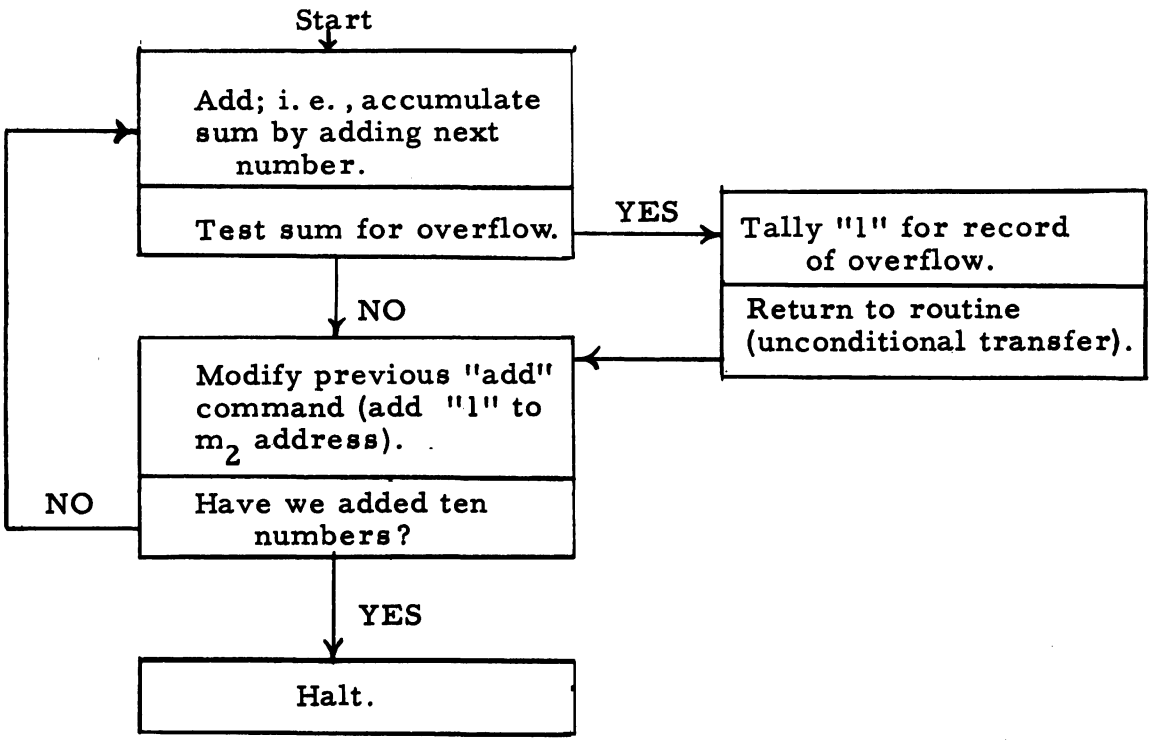
I	m_1	m_2	m_3
to	address of word to be tested	xxxx	address of alternate command

The "to" command causes the computer to test the contents of the word named by the m_1 address for the presence of a binary "one" in the overflow bit position. If a bit is present in this position the machine will automatically take its next command from the address named in m_3, otherwise it will continue in the normal sequence. The m_2 address is irrelevant. It is noteworthy that in this command the computer bases its decision merely on the presence of a "one" bit in the overflow position of the specified word, and does not concern itself with how it got there.

One of the principal uses of the "to" command is its use in conjunction with the "add", "subtract", or "divide" commands. The reader's attention is called to the automatic alarm signal built in the computer; that is, if an overflow bit is generated as a result of any of these commands the computer will automatically print the contents of the G register (cell 3000) on the Flexowriter and halt, unless the command in the program immediately following the command that caused the overflow is either "test for overflow marker" or "shift logically".

In many problems we know ahead of time that the magnitude of our data will not exceed the thirty-six bit capacity of a word, consequently,it is not necessary to incorporate any precautionary logic in the program. However, when it is not known ahead of time whether overflow will occur, it is necessary to include in the program logic which will satisfactorily handle the situation and also prevent the computer from halting. The following example illustrates a typical method of using the "to" command for this purpose.

Assuming the possibility of overflow now, the following flow-chart shows how we intend to handle this situation in a problem which requires the sum of ten given numbers:


Start
Add; i. e., accumulate sum by adding next number.
Test sum for overflow.
YES
Tally "1" for record of overflow.
Return to routine (unconditional transfer).
NO
Modify previous "add" command (add "1" to m_2 address).
NO
Have we added ten numbers?
YES
Halt.

Since we will follow the "add" command by a "test for overflow marker" command the computer will not halt if overflow occurs, but will take its next command from a cell which will instruct it to tally "1", and unconditionally return to the routine so that the problem may continue. At the conclusion of the problem, the cell containing the tally of overflows would be the most significant part of the sum.

Consider the ten numbers stored in cells 1100 - 1111, and the program beginning in 1200. The corresponding code is shown:

Address	I	m_1	m_2	m_3	Remarks
1200	ad 35	2000	[1100]	2000	Next number added to (2000), initially zero.
1201	to 37	2000	2100*	1205	Is an overflow bit present in (2000)?
1202	ad 35	1200	1207	1200	Modify (1200); i.e., add "1" to m_2 of 1200.
1203	tm 34	1210	1200	1200	Have we added ten numbers? No, return to 1200.
1204	ht 22	0000	0000	0000	Halt computation.
1205	ad 35	1211	2001	2001	"1" + (2001) ⟶ 2001 - (overflow tally), initially zero.
1206	tm 34	1210	2100	1202	Return to 1202.
1207	00	0000	0001	0000	Modifier for 1202.
1210	00	2000	1112	2000	"Gauge" for test in 1203.
1211	00	0000	0000	0001	"1" for tally in 1205.
1100	00	6502	3450	0000	Numerical Data.
1101	02	7300	2540	0300	
.	.	.	.	.	
1111	00	0470	6600	0000	
2000	00	0000	0000	0000	Working storage, initially zero.
2001	00	0000	0000	0000	Used for tally, initially zero.

*Since the m_2 address in the "to" command is irrelevant, use 2100.

Emphasis is placed on commands, 1205 - 1206, which keep track of the number of overflows and return computer control to the routine of adding. We referred to command 1206 as being as unconditional transfer command; it is coded in such a way that the test will always "work" and computer control will always transfer to 1202. The reader is reminded that address 2100 supplies us with a cell of all zeros, hence, any available non-zero program constant can be used as the m_1 reference in command 1206.

At the present time we will not attempt to familiarize the student with additional 102-A commands. The next two sections will deal with general internal computer operation, followed by the entire command list and additional tutorial examples.

VI

FILLING THE COMPUTER

At this point we will endeavor to bridge the gap in comprehension that so often occurs when passing from a general discussion of the machine to a discussion of the intrinsic nature of its operation, as presented in the following section on the control unit.

When the computer is being filled the control unit plays its simplest role; hence, the information contained in this section will serve as an excellent stepping stone for grasping knowledge of the essence of the 102-A. We will restrict the discussion, for the present time, to initial input from the Flexowriter unit -- the primary source of all input.

Initial filling, or proper internal positioning of a given routine, merely implies transcribing words from a code sheet to the designated memory cells. Simplicity of the fill process is further emphasized by the fact that the entire contents of a word are numerical characters, which correspond to the Flexowriter numerical keys. Also, the appearance of the Flexowriter unit (page V-1a) is similar to an ordinary typewriter, and when the proper key is struck binary information automatically enters the 102-A.

First it will be necessary for the student to learn the function of a few additional Flexowriter keys that control the positioning of information being filled into the computer, and some of the functions of the E and H registers.

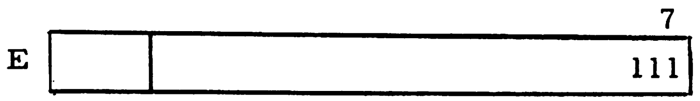
1. The E register, which is a one-word input register, is the only means by which information can enter the 102-A from the Flexowriter. When a Flexowriter key which is

significant to the computer is struck, the equivalent binary representation automatically enters the least significant end of the E register.

2. The Flexowriter "o" key prepares the computer for octal fill, and clears the E register (resets to zeros). Once the "o" key has been struck, any significant Flexowriter key will cause three bits per character to enter the E register according to Table 6, page VIII- 26 . It is noteworthy, however, that the computer is normally prepared for octal input.
3. The Flexowriter "d" key prepares the computer for decimal fill, and clears the E register. Once the "d" key has been struck, any significant Flexowriter key will cause four bits per character to enter the E register according to Table 5, page VIII-25.

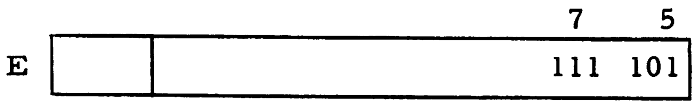
The following example illustrates the function of the E register during fill:

The result after striking the "o" key, then the "7" key:

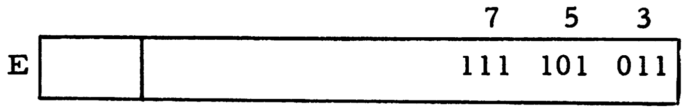


(Note that the "o" key has cleared the E register)

Then, after striking the "5" key:



Then, after striking the "3" key:



In this same example, if the "d" key had been struck instead of the "o" key, the result would have been:

		7	5	3
E		0111	0101	0011

This procedure of filling the most significant digit of a word first, and the automatic shifting of the word as the next digit enters, would continue until the one-word capacity of the E register is satisfactorily filled. If additional digits are entered after the E register is completely filled (14 octal or 10 decimal digits), the equivalent number of digits will be lost off the left end.

4. The H register is a one-word control register. During "fill" the function of the H register is to control the transferring of the word in E to the memory cell named by the address in m_3 of H.

5. The "-" (hyphen) key will transfer information from the E register to the H register, and also clear the E register. It is in this manner that we are able to place a monitoring address in H at any time. For example, if we wish to place the octal address, 1625, in m_3 of H we merely type it into E, and transfer its contents into H by use of the hyphen key. This procedure is illustrated below:

Type "1625":

E	00	0000	0000	1625	
H	xx	xxxx	xxxx	xxxx	(irrelevant)

Strike the "-" key and the result will be:

E	00	0000	0000	0000
H	00	0000	0000	1625

All addresses and command words must be filled in the octal mode.

6. The "TAB" key. The transfer of the contents of the E register to the memory cell named by the address in the m_3 portion of the H register is effected by striking the Flexowriter "TAB" key. Striking the "TAB" key also clears the E register, and augments the current address in m_3 of H by "1." The fact that m_3 of H is automatically increased by "1" when the "TAB" key is struck permits information to be filled into sequentially numbered memory cells when only the initial address is placed in H.

7. The Flexowriter "f" key is a special character that is used to fill the equivalent of four octal or four decimal zeros. Striking the "f" key enters one zero and signals the computer to circulate the contents of E left three additional digit positions (octal or decimal, depending on the mode of input). If, in the example of Item 1, the "f" key had been struck next, the E register would contain the following:

		7	5	3	0*	0	0	0
E		111	101	011	000	000	000	000

However, if E contained

E	0 2	1 2 5 0 7 7 3 2 5 0 3 5

*Indicates the "zero" entered because of the "f" key.

and the "f" key was struck, the result would be

E	5 0	7 7 3 2 5 0 3 5 0* 0 2 1

Consequently, the "f" key will enter four zeros only when the three left most digits of E are zeros.

8. The Flexowriter "s" key controls the start of machine computation. After the routine is completely filled, we insert the address of the first command to be executed in the $\underline{m_2 \text{ position of } H}$. The E and H registers would take on the following form during this procedure:

Type address of initial command: (abcd denotes an abritrary address)

		m_1	m_2	m_3
E	00	0000	0000	abcd

Strike the "f" key:

E	00	0000	abcd	0000

Strike the "-" key:

H	00	0000	abcd	0000

Now, when the "s" key is struck, computer operation will be initiated according to the address in m_2 of H.

As an illustrative example of the fill process, consider the sample code on Page V-5. In this example, we have indicated that we want to fill data in main memory cells 0600 - 0612 and the sequence of commands in cells 0100 - 0112. The sequence of events for filling this example are as follows:

*Indicates the "zero" entered because of the "f" key.

(1) Strike "o," which clears E and prepares computer to accept octal address 0100 and octal command words to follow.

(2) Type "0100."

E	00	0000	0000	0100

(3) Strike "-" (hyphen), which transfers E to H and clears the E register.

(4) Type 35060006010700.

E	35	0600	0601	0700

(5) Strike "TAB." Contents of E are transferred to main memory cell 0100; E is re-set to all zeros; H now contains:

H	00	0000	0000	0101

(6) Continue typing the command list, and tab after each command is properly filled in E.

(7) After the last command is filled m_3 of H contains 0113. In order to begin entering data in cell 0600, we must change m_3 of H; hence, type "0600."

E	00	0000	0000	0600

(8) Strike "-" (hyphen), which transfers E to H and clears the E register.

H	00	0000	0000	0600

(9) Type 2000000216532

E	*2	0000	0021	6532

(10) Strike "TAB." Contents of E are transferred to main memory cell 0600; E is reset to all zeros; H now contains:

H	00	0000	0000	0601

(11) Type remaining data words and tab after each number has been properly filled in E.

(12) Type "0100f," which places initial command address in m_2 of E.

E	00	0000	0100	0000

(13) Strike "-" (hyphen), which transfers E to H.

H	00	0000	0100	0000

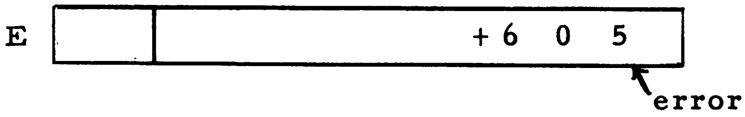
(14) Strike "s." Computer operation begins (see Section VII-Control Unit). Pressing the "compute" button on the operator's console has the same effect.

Since errors will often occur when typing on the Flexowriter, the following methods indicate how corrections can be made:

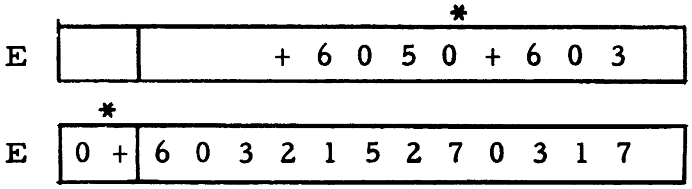
1. If an error is noted before the "TAB" key is struck, it can be corrected by merely typing in the full word again. The contents previously entered can actually

*It is not necessary to type zero in this position if the E register is already clear.

be ignored because the E register has only a one-word capacity and all previous information will have been, so to speak, pushed out. Consider the octal number, +603215270317; and the typographical error indicated below:



The following schematics illustrate the contents of the E register as the correction is made.

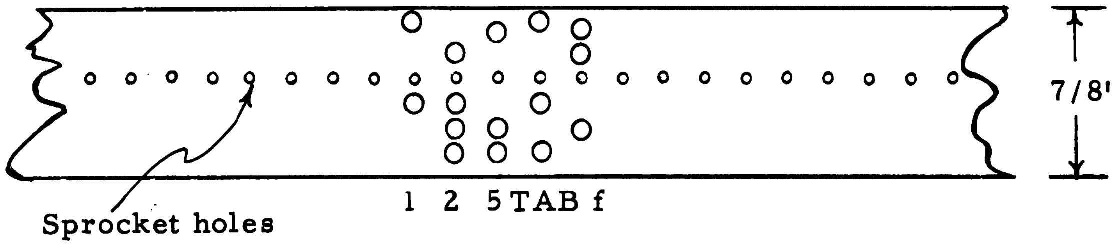


An alternate method for correcting an error before the "TAB" is struck would be to strike the "d" or "o" key (depending on the mode of fill at the time), and then retype the correct information. The "_d_" or "_o_" key _automatically clears the E register._

2. If an error is detected after the "TAB" key has been struck, it is necessary to re-enter the address in the H register before entering the correct information. In this way, we merely write over the incorrect data already entered in the memory.

* When a number is filled initially, this zero is usually omitted because the E register has previously been re-set to all zeros. However, if it were not entered while making this correction, the preceding 5" would not be pushed off and remain as incorrect data.

Flexowriter Paper Tape. Attached to the side of the Flexowriter unit is a paper tape "punch" and "read" unit. It is possible to simulate the complete fill process by punches on a paper tape. Preparation of the paper tape is identical to the manual filling process. Once the "PUNCH ON" control is depressed, striking any Flexowriter key will automatically punch a distinct configuration of holes in the paper tape as shown in the following diagram.



A "READ TAPE" Flexowriter control permits punched paper tape to be fed into the Flexowriter and communicate with the 102-A. The Flexowriter interprets the punched holes as if a human operator were striking a key. Paper tape feeding may be halted automatically by a "STOP CODE" punch on the tape. A punched paper tape, then, can fill the computer, start computation, and halt itself from further feeding.

The principal feature of using punched paper tape is that the tape can be prepared on a separate Flexowriter unit, and it can be proofread and corrected prior to its being fed into the computer. Hence, many new tapes can be prepared and checked while computer operation is in progress. Also, paper tape makes a permanent and accurate copy of a routine available for future use.

At the beginning of this section emphasis was placed on the simplicity of the filling process because only the Flexowriter o, d, -, f, s, TAB, and numerical keys are used. No comments were made relative to the results of striking any of the Flexowriter keys not mentioned above. However, it is noteworthy now that all other Flexowriter characters *are not significant* to the computer, *with the exception of the PERIOD key and SPACE bar*; Nothing will enter the 102-A,then, unless one of the above mentioned characters is struck.

A more technical discussion of the functioning of the E register in conjunction with the Flexowriter data and control keys during fill is given in appendix I.

VII

CONTROL UNIT

The control unit, consisting of the H and G registers, is in essence the automatic feature of the computer. Its function is to guide the complete operation of the machine under the influence of the coded program.

Section VI dealt with filling the computer and illustrated how the address of the initial command in the program was placed in m_2 of the H register. The next, and final, step listed in the sequence of events was to strike the "s" key on the Flexowriter or the "COMPUTE" button on the operator's console. From this point on the control unit, or automatic feature of the machine, takes over.

Sketches illustrating the contents of the H and G registers during the various phases of control are furnished to aid in the discussion. We will follow the example on page V-5 under influence of the control unit. We start, then, with the address of the initial command in the m_2 portion of H, and the "s" key having been struck. The H register appears as follows:

	I	m_1	m_2	m_3
H	00	0000	0100	0000

The initial address, 0100, is referred to as the control number.

First, the machine will automatically seek the contents of cell 0100; that is, the start of any computation is always channel and sector selection of the m_2 address of H. After cell 0100 is selected the following events take place simultaneously during the next unit time interval of machine operation (one word-time):

1; The control number (m_2) is augmented by one and the entire contents of H are transferred to G.

2. The contents of cell 0100 are read from the memory and written in H.

Thus, G and H contain:

	I	m_1	m_2	m_3
G	00	0000	0101	0000
H	35	0600	0601	0700

Next, the computer proceeds to locate the memory cells named by the m_1 and m_2 addresses in H. The contents of these cells are read from the memory and written in the E and F registers, respectively. Immediately after the contents of cell 0601 are found and written in F, the next control number in m_2 of G (0101) is copied into H. The appearance of H after execution of these phases of control is shown below:

	I	m_1	m_2	m_3
H	35	0600	0101	0700

Next, the operation named in the I section of H is performed. Since this is an "add" command, the E and F registers, which hold (0600) and (0601) are added and the sum is generated in E.

Finally, the result is written into the memory cell named by m_3 of H, namely, 0700. At the conclusion of this phase of control the machine will automatically return to the first phase of control operation

The procedure from now on is actually repetitive. As before, the cell named in the m_2 portion of H (new control number) identifies the next command to be obeyed. After having selected this cell the machine reads its contents from the memory and writes it in H, and at the same time augments 0101 by one, when transferring H to G. Hence, the contents of H and G would be:

	I	m_1	m_2	m_3
G	00*	0600	0102	0700
H	35	0700	0602	0700

This, then, is a brief illustration of how the computer would obey the first command of a given program, and proceed sequentially until the problem would be completed. The final command that definitely tells the computer to stop is the "halt" command in cell 0112,interpreted as such by the machine from the 22 code.

Since execution of the "halt" command becomes effective as soon as the control unit recognizes the 22 code, the m_1 and m_2 addresses are irrelevant -- they are merely selected and read from the memory because of the systematic procedure of control unit operation. "Rest" is the idle condition, in which the computer remains until either the operator's console "COMPUTER" button is pressed, or the Flexowriter "s" key is struck.

Control Unit Operation Procedure Briefly Summarized:

1. The computer selects the memory cell named by the address in m_2 of H (control number).
2. The contents of this cell are read from the memory and copied into H, and at the same time the contents then in H are transferred to G with "1" being added

*In the course of executing a command the computer reduces the I digits in H to 00; consequently, the I digits of G will be 00 when H is transferred to G.

to the m_2 address (m_2 of G now holds the next control number).

3. The cell named by m_1 of H is read from the memory and its contents are copied into the E register.
4. The cell named by m_2 of H is read from the memory and its contents are copied into the F register.
5. The m_2 address of G (next control number) is copied into H. Note that only m_2 of G is copied.
6. The operation called for by the instruction in the I portion of H is executed, and the I portion of H becomes zero.
7. The result, now in E, is written into the memory cell named by m_3 of H.
8. Return to 1 above, or to "rest."

The alternate code for adding ten numbers on page V-10 utilized the procedure of transferring control away from the sequential mode of operation by means of a decision command. What, then, will be the reaction of the control unit under the influence of a decision command?

The reader is reminded that during automatic control unit operation the address of the next command in numerical sequence is always in m_2 of H (step 5 above) just prior to the conclusion of execution of the current command. However, in a decision command, if the test works, the address of the next command is in m_3 of H. Therefore, since the address of the next command must always be in m_2 of H (step 1 above), the final phase of control operation in a decision command when the test works will automatically circulate H left four octal digit positions (m_3 of H ⟶ m_2 of H). However, in a decision command when the test fails, H will not be circulated left and the next command in numerical sequence will still be in m_2 of H. Hence, in either case (test fails, or works) when computer control is returned to step 1 at the conclusion of a decision command, the proper address will be in m_2 of H.

As an illustration, consider the sample code on page V-10. In the course of computer execution of this code, and during step 2 of computer control, when (0103) is read from the memory and written in the H register, G and H will contain the following:

	I	m_1	m_2	m_3
G	00	0614	0104	0615
H	34	0616	0615	0100

After (0616) and (0615) are read into E and F, and the control number is copied from m_2 of G into m_2 of H (thru phase 5 of control unit operation), the H register will contain

H	34	0616	0104	0100

Next, the instruction specified in this "test magnitude" command causes the computer to compare (0616) with (0615), which are held in E and F, respectively. If the magnitude section of E is larger than the magnitude section of F, the test works and H is circulated left four octal digit positions; that is, H will contain the following:

H	16	0104	0100	0000*

Hence, if the test works the next control number is 0100.

On the other hand, if the test fails, H remains unchanged and the address of the next command, 0104, remains in m_2 of H. In either case, computer operation is immediately returned to the first step of control unit operation.

An understanding of the functioning of the control registers will prove to be of considerable help to the programmer.

1. He may wish to use cell 3000 as an operand in a command. A knowledge of its contents would, of course, be essential.
2. Alarm checks exist that cause the computer to print the contents of the G register on the Flexowriter. This will occur whenever a number, rather than a command, is brought into the control unit, or whenever overflow occurs during addition, subtraction or division. Thus, recalling that G will always contain the next control number, we can scrutinize m_2 of G and determine the guilty memory cell.

The discussion of the control unit presented in this section illustrated the automatic operating features of the machine.

*The first of these four zeros is introduced by the computer when it circulates the H register. The next two zeros have been carried around from the I portion (previously reduced to 00). The last zero is the most significant digit from the m_1 address, also carried around by the circulation of the H register.

Emphasis is placed on the fact that once computation has been started with the address of the initial command it will automatically continue with commands located in numerically sequential cells, unless a decision command causes a transfer of control. When a transfer of control does take place, the computer will automatically resume its sequential mode of operation beginning with the command named by the m_3 address of the transfer command.

Channel and Sector Selection Operation for Reading and Writing

This systematic procedure by which the machine reads and writes information to and from the memory is a function of how an address appearing in the H register is interpreted with respect to channel and sector selection, which was treated as a routine step in the previous discussion.

For investigative purposes let us label the octal digits of any address present in either m_1, m_2, or m_3 of H as O_4, O_3, O_2, O_1, and the corresponding bits by the L's and B's indicated in Tables 2 and 3. First, the computer will scrutinize the L_1-L_5 bits and select the corresponding memory channel by bringing the proper circuitry into operation. Next, selection of the designated sector of this channel is accomplished by means of the word channel. In brief, the bits corresponding to the sector portion of the address (O_2 O_1) are compared with those of the permanently recorded word channel, as the word channel passes its read-write head. Only when these numbers coincide does the machine know that it has located the correct sector. Since a full word time is involved in comparing each sector of the word channel, the next cell to appear under the read-write head of the chosen channel will be the correct cell as named by the address in the H register.

It is noteworthy that the selection procedure of a main memory cell is the same whether reading or writing. The selection procedures for cells 2100, 3000 and the buffer register differ when reading or writing. The following tables will help emphasize this difference.

O_4			O_3			O_2			O_1			
L_6	L_5	L_4	L_3	L_2	L_1	B_6	B_5	B_4	B_3	B_2	B_1	Memory Address
-	1	1	-	-	-	-	-	-	-	-	-	3000
-	1	0	-	-	1	-	-	-	-	-	-	2100
-	1	0	-	-	0	-	-	-	0	0	0	2000
-	1	0	-	-	0	-	-	-	0	0	1	2001
-	1	0	-	-	0	-	-	-	0	1	0	2002
-	1	0	-	-	0	-	-	-	0	1	1	2003
-	1	0	-	-	0	-	-	-	1	0	0	2004
-	1	0	-	-	0	-	-	-	1	0	1	2005
-	1	0	-	-	0	-	-	-	1	1	0	2006
-	1	0	-	-	0	-	-	-	1	1	1	2007
-	0	X	X	X	X	X	X	X	X	X	X	Main Memory

Table 2. Channel and Sector Selection for Reading

O_4			O_3			O_2			O_1			
L_6	L_5	L_4	L_3	L_2	L_1	B_6	B_5	B_4	B_3	B_2	B_1	Memory Address
-	1	-	-	-	-	-	-	-	0	0	0	2000
-	1	-	-	-	-	-	-	-	0	0	1	2001
-	1	-	-	-	-	-	-	-	0	1	0	2002
-	1	-	-	-	-	-	-	-	0	1	1	2003
-	1	-	-	-	-	-	-	-	1	0	0	2004
-	1	-	-	-	-	-	-	-	1	0	1	2005
-	1	-	-	-	-	-	-	-	1	1	0	2006
-	1	-	-	-	-	-	-	-	1	1	1	2007
-	0	X	X	X	X	X	X	X	X	X	X	Main Memory

Table 3. Channel and Sector Selection for Writing

X: denotes the possibility of either 0 or 1.
Hyphen (-): denotes bits ignored by machine during selection.

Careful examination of these tables reveal the following important facts:

1. The differences in reading and writing procedures are based primarily on the differences in channel selection.

2. The L_6 bit (most significant bit of an address) is totally irrelevant.

3. The L_5 bit selects either 2100, 3000, or the buffer if equal to "1"; and the main memory if equal to "0".

4. The L_1, L_2, L_3, and L_4 bits are ignored when writing (if L_5 is "1") hence, it is impossible to write into cells 2100 and 3000. The buffer would be selected instead.

5. Sector selection for the buffer takes place only over the least significant octal digit O_1.
6. The machine ignores any four octal digit addresses referring to a non-existent memory cell, and will choose a cell according to the bits significant to the machine. For example, if told to read from cell 2106, the machine would read cell 2100, since only the underlined bits are relevant.

O_4	O_3	O_2	O_1
$0\underline{10}$	$00\underline{1}$	000	110

If told to write into this same cell, cell 2006 would be written into, since a different set of bits are relevant for writing.

O_4	O_3	O_2	O_1
$0\underline{1}0$	001	000	$\underline{110}$

It is essential that the programmer become familiar with the procedure used within the machine for interpreting an address when reading or writing. Many useful applications in problem programming will result from a skillful use of the facts illustrated in Tables 2 and 3.

VIII

COMMANDS

Before further study in programming is attempted, it is essential that the student become familiar with the entire list of 102-A commands, which are the programmer's tools for instructing the computer to efficiently carry out the solution of a given problem. Techniques of using the commands will be further illustrated by tutorial examples in the following sections.

Listed below are the twenty-five commands in the order of their octal codes.

Octal* Code	Abbreviation	Instruction
04	bo	Buffer out
05	bl	Buffer Load
06	rc	Read I. B. M. card
11	fl	Fill (from Flexowriter tape)
12	pd	Punch I. B. M. card - decimal
13	po	Punch I. B. M. card - octal
14	bs	Block Search
15	wt	Write Magnetic Tape
16	rt	Read Magnetic Tape
17	ts	Test Switch or Test Search
21	pr	Print on Flexowriter
22	ht	Halt computation
23	dr	Divide and Round-off
24	dd	Divide and Save Remainder
25	mr	Multiply and Round-Off
26	md	Multiply Double Length
27	sl	Shift Logically
30	sm	Shift Magnitude
31	sf	Scale Factor
32	ex	Extract
33	ta	Test Algebraically
34	tm	Test Magnitude
35	ad	Add
36	su	Subtract
37	to	Test for Overflow Marker

Table 4.

*Any octal code other than those listed, if used in a command word, would cause the computer to print the contents of the G register on the Flexowriter and automatically halt computation.

Emphasis must be placed on the fact that the contents of a word in the memory may be either a command or a number, and both commands and numbers can be referred to as operands by any command. The approach, then, when learning the commands is to concentrate on the basic function of each command, while keeping in mind that the operands could be either commands or numbers.

All commands are of the form

I	m_1	m_2	m_3

Hence, we will expound each command by qualifying its m_1, m_2, and m_3 addresses. The contents of a cell is distinguished from its address by parenthesizing that address -- (m_1), (m_2), or (m_3).

ADD - "ad" - code 35

This command operates in two distinct ways, as indicated below:

a. Addition of two numbers. The magnitude sections of (m_1) and (m_2) are added algebraically (signs considered), and the sum is recorded in m_3. If the sum exceeds the capacity of the thirty-six bit magnitude section, a "1" is recorded in the overflow bit position, whereupon the machine prints the contents of G and automatically halts computation. However, if the next command is "Test for Overflow Marker" or "Shift Logically," G will not be printed and computation will continue normally.

b. Addition of a number and a command. This alternate mode of addition is intended for modification of the address(es) of a command word; that is, the magnitude section of a number addressed by m_2 is added to the magnitude section of a command word addressed by m_1, and the instruction code of the command word is retained in the sum. Even though overflow might have occured in this mode of addition, the overflow bit will not be generated and the overflow alarm will not be set.

The reader is cautioned that this operation will be executed properly by the "ad" command only if the command word is named by m_1 and the number by m_2. If these addresses are interchanged, the machine will add the two words as two algebraic numbers, and not retain the instruction digits of the command word.

The above exposition of the "ad" command is certainly adequate for the average programmer's use; nevertheless, experience has taught us that the programming student has a strong desire to understand how the control and arithmetic unit functions during the execution of arithmetic operations. Consequently, we will simulate the operational procedure of the control and arithmetic unit during the execution of a sample "ad" operation. However, we will not treat all commands in this way, since an understanding of the mechanization of commands is not a prerequisite for programming.

Example: Consider the "ad" command,

		I	m_1	m_2	m_3
		35	0410	0411	0410,
where	(0410) =	36	0206	0220	0600
	(0411) =	00	0000	0001	0000

We intend this "ad" command to add "1" to the m_2 address of the command in 0410. Consider, then, that this command has been brought into the control unit and that the systematic operation of the control unit has automatically read and recorded (0410) and (0411) into the E and F registers, respectively. Consider, also, that the new control number has been copied from G to H.

Now, the add operation will be carried out. First, the proper mode of addition must be determined. The machine makes this decision by scrutinizing the I portion of E (the contents of F plays no role in this decision). The decision is: If E contains a command, then E and F are added accordingly, otherwise E and F will be added as if they are both numbers.

Next, the addition is performed according to the above decision, with the result appearing in E as follows:

> E contains a command. According to the sign bit of F, its magnitude section is added to or subtracted from the absolute value of the magnitude section of E. The sum is collected in E, and the instruction digits in E are retained. The generation of an overflow bit is suppressed in this mode of addition. From the example, we have, then:

E	36	0206	0220	0600
F	00	0000	0001	0000

Adding, the computer obtains

E + F → E =	36	0206	0221	0600

Finally, the sum in E is recorded in the memory cell named by m_3.

We are now in a position to consider what would happen if, in attempting to modify a command, we interchanged the m_1 and m_2 addresses. Of course, E would not contain a command, and the computer would decide to add E and F as if they both contain numbers. This mode of addition -- two numbers -- is executed by the machine in the following manner:

> The magnitude sections of E and F are added algebraically, as the computer considers only the single sign-bit of the I portion of each word, and the sum is collected in E. An overflow bit will be recorded in E, if necessary, and a warning to print G and halt computation will be issued to the computer in the event the next command entering the control unit is not a "test for overflow marker" or "shift logically" command.
>
> It is noteworthy that the overflow bit of either operand might, for various reasons, contain a "I" prior to addition. If this bit is present in E it will appear in the result; if this bit is present in F (but not in E) it will not appear in the result. However, in either case the overflow alarm will not be set unless actual arithmetic overflow occurs.
>
> Using the above example for this mode of addition, E and F would contain:

E	00	0000	0001	0000
F	36	0206	0220	0600

Since E contains a number, E and F are added algebraically: That is, F will be considered a negative number in the addition process because of the "1" in the sign bit position (36 = 011 1$\underline{1}$0). Hence, the result

E + F ⟶ E =				
	02	0206	0217	0600

would be meaningless on the basis of our intent to modify the given command.

Modification of a command, then, involves a conditional use of the "ad" command. Because of the automatic traits of the add operation it is imperative that the "ad" command be used properly-- m_1 must name the command word and m_2 must name the modifier.

SUBTRACT - "su" - code 36

This command is executed by the computer in the same manner as the "ad" command, to the extent that the computer will recognize a command named by the m_1 address and modify it by the number named by the m_2 address. For subtracting numbers, the word structure for the subtract command is:

(m_1) = minuend

(m_2) = subtrahend

m_3 = address of difference

Example:

su	m_1	m_2	m_3
36	0312	0345	0400

(0312) = 00 000000005642

(0345) = 00 000000000112

Result in (0400) = 00 000000005530

MULTIPLY DOUBLE LENGTH - "md" - code 26

Mechanized multiplication in the 102-A automatically multiplies the contents of two words as two numbers; that is, the magnitude sections of (m_1) and (m_2) are multiplied to form the product. Since each operand is thirty-six bits long and the automatic multiplication process considers all thirty-six bits (zeros, or not), the available product is seventy-two bits long, or two full words. The sign of the product, determined from the sign bits of (m_1) and (m_2), appears in each word of the product. Any other information present in the sign portion of the operand will not effect the product.

The "md" command allows for a two word product storage by recording the least significant half of the product in cell m_3 and the most significant half of the product in the next cell adjacent to m_3 on the surface of the drum.

In illustrating the "multiply" and "divide" commands, in order to enable the student to follow the arithmetic more conveniently, we have chosen decimal numbers and shown the results in decimal form. The student is cautioned again that, when using the arithmetic commands, decimal numbers must first be converted to octal form.

Consider the "md" command,

md	m_1	m_2	m_3
26	0210	0211	2000 *

*In these examples, we have written the command to store the result in the buffer register, since in the buffer, consecutively-numbered cells are physically adjacent. However, in the main memory, consecutively numbered cells are not physically adjacent. The programmer need not concern himself with this fact except in using the five multiple-put-away commands noted in this section. In the section dealing with minimum-access coding, we will discuss the numbering scheme used in the main memory.

Example 1:

```
   (0210) =             +000056933
   (0200) =             -000067083
  product = -000000003  -819236439
              (2001)      (2000)
              M. S. H.    L. S. H.
```

Example 2:

```
   (0210) =              +000000012
   (0200) =              +000000012
  product =  +000000000  +000000144
               (2001)      (2000)
```

Example 3:

```
   (0210) =             +120000000
   (0211) =             +120000000
  product =  +014400000 +000000000
              (2001)      (2000)
```

Example 4:

```
   (0210) =              -900000000
   (0211) =              -800000000
  product =  +720000000  +000000000
               (2001)      (2000)
```

MULTIPLY AND ROUND-OFF - "mr" - code 25

The "mr" command obtains a product in exactly the same way as the "md" command, except that only the most significant half of the product is retained by the machine; it is rounded if the least significant half is 1/2 or greater, and then recorded in cell m_3. If the "mr" command was used in the above examples, the rounded products would be as follows:

Example 1:

(2000) = -000000004

Example 2:

(2000) = +000000000

Example 3:

(2000) = +014400000

Example 4:

(2000) = +720000000

These examples clearly indicate that a product will never overflow; and emphasizes the importance of digit position in a word with respect to the position of the product, and with respect to the choice of the most satisfactory multiply command.

DIVIDE AND SAVE REMAINDER - "dd" - code 24

Mechanized division in the CRC 102-A divides the algebraic value of (m_1) by the algebraic value of (m_2) and produces a quotient with the proper algebraic sign; however, a valid quotient is solely dependent on the position of the operands within the word. The inflexibility of the mechanized procedure assumes the binary points of both divisor and divident to be located in the same position in the word, and automatically forms the fractional quotient as if its binary point is at the left end of the magnitude section. If the absolute quotient is less than two, the unit digit ("1" or "0") will appear in the overflow position and the fractional representation will appear in the magnitude section. On the other hand, if the quotient is two or greater, it will be recorded as a meaningless number. In either case, when an overflow bit is generated, the computer will cause an overflow alarm and react in the same manner as in the "ad" command.

The "dd" command assumes the following word structure:

(m_1) = dividend

(m_2) = divisor

Remainder is recorded in m_3.

Quotient is recorded in next cell adjacent to m_3 on the surface of the drum (See footnote to "multiply double" command, page VIII-7).

The remainder will appear in m_3 in proper position for a repeated division, if desired, and it will have the same sign as (m_1).

Consider the "dd" command,

dd	m_1	m_2	m_3
23	0210	0211	2000

and the following examples, using decimal numbers as before:

Example 1:

```
                (0210) = -200000000
                (0211) = +300000000
   Quotient in (2001) = -.666666666
  Remainder in (2000) = -200000000
```

Example 2:

```
                (0210) = +300000000
                (0211) = -200000000
   Quotient in (2001) = -1.500000000   (overflow alarm)
  Remainder in (2000) = +000000000
```

Example 3:

(0210) = +030000000

(0211) = -200000000

Quotient in (2001) = -.150000000

Remainder in (2000) = +000000000

Example 4:

(0210) = -000000020

(0211) = -000000030

Quotient in (2001) = +.666666666

Remainder in (2000) = -000000020

Example 5:

(0210) = +000020000

(0211) = +000003000

Quotient in (2001) = meaningless (overflow alarm)

Remainder in (2000) = meaningless

DIVIDE AND ROUND-OFF - "dr" - code 23

The "dr" command word structure is the same as "dd," except that the quotient is rounded according to the ratio of the remainder and the divisor, and recorded in cell m_3. If the "dr" command was used in the above examples, the rounded quotients would be as follows:

Example 1:

(2000) = -.666666667

Example 2:

(2000) = -1.50000000 (overflow alarm)

Example 3:

(2000) = -.150000000

Example 4:

(2000) = +.666666667

Example 5:

(2000) = meaningless (overflow alarm)

SHIFT LOGICALLY - "sl" - code 27

The "sl" command will shift the entire contents of a word a specified number of bit positions to the left or right. Digits shifted off one end of the word will be replaced by zeros on the opposite end.

Command structure:

(m_1) = word to be shifted

(m_2) = the direction and number of shifts. The number of shifts is specified by the magnitude section as an octal number; The direction of shift is specified by the sign section, that is, a negative number will cause a right shift and a positive number will cause a left shift.

m_3 = address of shifted word.

Example: Consider the problem of shifting the command word.

35 0100 0101 0222

in such a way that the I portion will occupy the m_1 position, and of storing the result elsewhere in the memory (shift 12 bit positions to the right). Assume the command word which will be shifted to be stored in cell 0300.

Solution: Write the "sl" command,

27 0300 0400 0320

where (0400) = 02 0000 0000 0014

The shifted result would be:

(0320) = 00 0035 0100 0101

SHIFT MAGNITUDE - "sm" - code 30

The function of the "sm" command is exactly the same as the "sl" command except that only the magnitude portion of the operand will be shifted. Zeros will appear at either end of the magnitude section to replace digits that are shifted off the other end; the sign portion will be unchanged. For instance, if in the previous example a "sm" command was used, the result would be

(0320) = 35 0000 0100 0101.

SCALE FACTOR - "sf" - code 31

The "sf" command will shift the magnitude portion of a word left until a binary "one" appears in the most significant binary position of the magnitude. The machine records the number of bit positions shifted by subtracting this number from (m_1). The command structure is as follows:

(m_1) = an arbitrary number (chosen by the programmer) from which the number of shifts is subtracted.

(m_2) = word which will be shifted.

(m_3) = (m_1) minus the number of bit positions shifted.

The shifted word is recorded in the next cell adjacent to m_3 on the surface of the drum (See footnote to "multiply double" command, page VIII-7)

Example: Consider the "sf" command,

31 2100 0100 2000

where,

(2100) = +0000 0000 0000

(0100) = -0146 7203 2457

Results of this command would be:

(2000) = -0000 0000 0005

(2001) = -6335 0152 2740

It is suggested that the reader verify this result by writing (0100) and (2001) in binary form.

In the event (m_2) is zero, the machine will record sixty-four (octal 100) shifts and terminate the "sf" operation, instead of recording a countless number of shifts as a result of its attempt to locate a "one" in the most significant bit position.

EXTRACT "ex" - code 32

The "ex" command permits selected bits from one word to be copied into the corresponding bit positions of another word. This operation functions in such a way that those binary digits of (m_1) which are in the same positions as the binary "ones" of (m_2) replace the corresponding binary digits of (m_3), and the bit positions of (m_3) corresponding to the binary "zeros" of (m_2) will not change.

For example, consider the "ex" command,

32 1100 1120 1130,

where, (1100) = 00 0000 0237 0000

(1120) = 00 0000 7777 0000 (extractor)

(1130) = 26 0404 0460 0406

Results of this "ex" command would be

(1130) = 26 0404 0237 0406

Since the binary "ones" of the extractor, 1120 occupy the entire m_2 position, the corresponding portion of the word in cell 1100 replaces the m_2 address of the command word in cell 1130. Hence the address of a command has been modified. It is noteworthy that the extract command can function over the full length of a word, including the Sign or Instruction digits.

TEST FOR OVERFLOW MARKER - "to" - code 37

The "to" command tests the contents of a word for the presence of a binary "one" in the overflow bit position. If an overflow marker is present in (m_1), the machine will automatically take the next command from cell m_3; otherwise it will continue in the normal sequence.

The m_2 address of the "to" command is irrelevent.

TEST MAGNITUDE - "tm" - code 34

The "tm" command compares the magnitude sections of the two words named by the m_1 and m_2 addresses. If the magnitude section of (m_1) is greater than the magnitude section of (m_2), the machine will automatically take the next command from cell m_3. Otherwise it will continue in the normal sequence of commands.

Example: Consider the "tm" command,

34 1102 1205 0112

where, (1102) = 00 1532 1001 0600
(1205) = 26 1500 1001 0600

Since the machine compares only the magnitude portions of (1102) and (1205), the sequence of digits, 1532 . . . . , in 1102 will be considered greater than the sequence, 1500 . . . . , in 1205; hence, the next command will be taken from cell 0112.

TEST ALGEBRAICALLY - "ta" - code 33

The "ta" command compares the algebraic contents of the two words named by the m_1 and m_2 addresses, that is, both words are treated as positive or negative numbers, according to their sign bits. If (m_1) is greater algebraically than (m_2), the machine will automatically take the next command from cell m_3. Otherwise it will continue in the normal sequence of commands.

Example: Consider the "ta" command,

```
33 1065 1074 0300
```

```
where,  (1065) = 02  1256  7362  0327
        (1074) = 02  2376  2370  0245
```

Since the machine compares (1065) and (1074) algebraically, and the negative number, 125. . . . . . , is greater than the negative number, 237. . . . , the next command will be taken from cell 0300.

TEST SWITCH - "ts" - code 17

The "ts" command is unique in that it is the only 102-A command which allows the operator to exercise manual control of machine operation. There are four toggle switches, labeled 2010, 2020, 2040 and 2100, on the operator's console. This command causes the computer to "examine" the switch designated by m_1 and, if that switch is in the "up" position, to take its next command from the address in m_3; if the switch is in the "down" position, the computer will take the next command in the normal sequence. The m_2 address of the "ts" command is irrelevant. Note that m_1 is not an address, but the actual number of the switch being tested.

Example: Consider the following "ts" command to be stored in 0212:

```
17  2020  2100  0400
```

If test switch 2020 is in the "up" position, the computer will take its next command from cell 0400; if test switch 2020 is in the "down" position, the next command will be taken from 0213.

HALT - "ht" - code 22

The "ht" command, immediately after being executed in the course of a routine, stops automatic computer operation and returns the computer to an idle condition. The manner in which the halt command is executed by the computer allows it to name the address of the first word of any information that might be filled from the Flexowriter.

Since the "ht" command word structure remains in the H register after the computer has returned to "rest," the computer is prepared to fill octal information from the Flexowriter into the memory cell named by the m_3 address of the "ht" command. The m_1 and m_2 addresses are irrelevant with respect to the "halt" operation; however, if it is intended that the "ht" command is also to be used for subsequent filling, m_1 should be 2100 to insure clearing of the E register . Furthermore, automatic control unit operation places the next control number (address of next command) in m_2 of H during the course of execution of the command. The halt command, then, may be used as a means of temporarily halting computation, while new data is entered into the computer. Computation will resume in the normal sequence when the "compute" button is depressed.

Example: Consider the "ht" command of a program as being stored in cell 0321.

(0321) = 22 2100 2100 0700,

Immediately upon execution of this command the computer would go to rest and the H register would contain the following:

	I	m_1	m_2	m_3
H	00	2100	0322	0700

This is the same configuration that an operator would fill into the m_2 and m_3 portions of the H register if he intended to fill the initial word from the Flexowriter into memory cell 0700; and start computation in memory cell 0322.

FILL (from Flexowriter Tape) - "fl" - code 11

The "fl" command is used to control the automatic filling of information from Flexowriter paper tape in the course of execution of a routine. The "fl" command functions in exactly the same way as the halt command, with respect to the halting of computation and the configuration that remains in the H register, but in addition, the computer automatically starts the Flexowriter paper tape reader.

If the "fl" command is to be used to temporarily halt computation and fill the information from the paper tape automatically, without manual intervention, it is necessary that the paper tape contain the necessary control characters. Such characters would normally be the "s" (start computation) and "STOP" (stop paper tape reader) codes.

BUFFER LOAD - "bl" - code 05

The "bl" command will copy any block of eight physically adjacent main memory cells into the buffer register. The m_3 address of the "bl" command names the first of the eight words to be copied from the main memory to the buffer -- m_1 and m_2 are irrelevant. The first of the eight buffer cells to be filled will be that one which has the same least significant digit as m_3.

Example: Consider the "bl" command,

```
05 2100 2100 1223
```

The following cell coincidence between the main memory and the buffer will take place:

Main Memory		Buffer
1223	→	2003
----		2004
----		2005
----		2006
----		2007
----		2000
----		2001
----	→	2002

We remind the reader at this point that consecutively numbered cells in the main memory are not physically adjacent.

In order to execute the "bl" command, the computer locates address m_3 by making the proper channel and sector selections. Immediately after locating the cell specified by m_3, the computer reads its contents and the contents of the next seven physically adjacent words continuously by means of the selected channel's read-write head and records them into the buffer register.

BUFFER OUT - "bo" - code 04

The "bo" command will copy the contents of the eight words of the buffer register into any block of eight physically adjacent main memory cells. The command word structure and cell coincidence is exactly the same as in the "bl" command -- merely reverse the arrows in the previous example.

PRINT - "pr" - code 21

The "pr" command will cause the contents of one or more words in the memory to be printed on the Flexowriter in the course of a given routine. The entire, or partial, contents of the word(s) can be printed in the octal, or decimal mode (with or without their respective cell addresses), or in the alphabetic mode.

The word structure of the "pr" command controls printing in the following manner:

m_1 = address of the first word to be printed.

m_2 = address of the print control word, whose contents specifies the print mode and the number of characters to be printed from the magnitude section of each word. The sign portion of each word is automatically printed in accordance with the mode. (see Table 5, page VIII-26.

m_3 = the total number of words to be printed by the "pr" command. This number must be written as an octal number in the command word structure. The programmer is cautioned not to use m_3 as the address of a word whose contents specify the number of words to be printed.

The contents of the word named by m requires considerable description in order to code the print command properly. Since the contents of m_2 must designate two distinct qualities of the word(s) to be printed, the sign and magnitude sections are coded independently to represent the mode and the number of characters, respectively.

PRINT CONTROL WORD -- Sign Section

The octal sign digits of the print control word determine the mode of printing, according to the following list:

Sign Digits	Mode of Printing
00	Octal - Type the sign and magnitude digits as an octal number according to Table 6.
02	Octal and address - Type the octal address of the cell being printed, a space, and the contents of the cell in the octal mode.
01	Decimal - Type the sign and magnitude digits as a decimal number according to Table 5.
03	Decimal and address - Type the octal address of the cell being printed, a space, and the contents of the cell in the decimal mode.
10	Alphabetic - Print the alphabetic or numerical characters which correspond to the two-octal-digit codes contained in the magnitude section of the word(s) which are to be printed. (See Table 7.) The sign section is not printed in the alphabetic mode.

Both the octal and decimal modes of printing automatically tab the Flexowriter carriage to the next preset tab stop after printing each word. However, there is no automatic tabbing after printing each word in the alphabetic mode. Therefore, any desired editorial characters (shifting up or down, color shifting, backspace, etc.) must be coded just as any other alphabetic characters.

PRINT CONTROL WORD -- Magnitude Section

The number of characters which will be printed from each word is controlled by coding a binary "1" in the proper position in the magnitude section of the print control word.

Consider the magnitude section of the print control word as being divided into groups of three, four, or six bits, according to whether printing is to be octal, decimal, or alphabetic. Each of these groups of bits will then correspond to a similar group, representing one character in the word to be printed. The first of these groups, within the print control word, which contains a binary "1" in the units position corresponds to the last character to be printed from the word.

For example, the print control word 00 0000 0001 0000 will cause printing to be in the octal mode without addresses, and will print the sign digits and the first eight octal digits of the magnitude of each word.

The print control word 03 0000 0020 0000 will cause printing to be in the decimal mode with addresses, and will print the decimal sign digit and the first five decimal digits of the magnitude of each word. This will be clearer if we write the magnitude section of this print control word in binary form, and mark off the bits in groups of four:

$$0000\,\vdots\,0000\,\vdots\,0000 \quad 0000\,\vdots\,\underline{0001}\,\vdots\,0000 \quad 0000\,\vdots\,0000\,\vdots\,0000$$

The group which has been underlined corresponds to the fifth decimal digit in the magnitude of the word being printed, and that digit will therefore be the last one in each word which will be printed.

The print control word 10 0000 0100 0000 will cause printing to be in the alphabetic mode, and will print the first three alphabetic characters of each word.

Example 1: Assume the contents of 1045 and 1046 to be as indicated below:

(1045) = 23 0201 0202 0201
(1046) = 35 0201 0304 0201

Consider the print command,

21 1045 2100 0002.

Since this command specifies printing two full octal words beginning with (1045), the result would be:

```
23020102020201     35020103040201
```

Example 2: Assume the contents of 0120 to be:

(0120) = 02 406270000000

Consider the print command

21 0120 0435 0001,

and

(0435) = 02 000010000000

Since this command specifies printing one word, (0120), according to (0435), the result would be:

```
0120  0240627 (Note: the first two digits
                are the sign digits)
```

Example 3: Assume the contents of 1734 to be:

(1734) = + 690890000 (decimal, four bits per digit).

Consider the print command,

21 1734 1600 0001,

and

(1600) = 01 000000200000 (binary "1" in units position of fifth decimal digit)

Result:

+69089

Example 4: Assume the contents of the cells 1100 - 1107 to be as indicated below:

(1100) = 00 327536744564
(1101) = 00 326636417560
(1102) = 00 776641716432
(1103) = 00 623641447464
(1104) = 00 326336457360
(1105) = 00 447545636432
(1106) = 00 623677767041
(1107) = 00 665427641212

Consider the print command,

21 1100 0200 0010

and

(0200) = 10 000000000000.

Since the word structure of this print command specifies the alphabetic printing of eight full words, beginning with 1100, the magnitude sections of 1100 - 1107 would be scrutinized six bits at a time and the result would be (See Table 7):

The National Cash Register Company.

DECIMAL MODE

	Flexowriter Print-Out		Flexowriter Fill
Binary Configuration	Printout from Sign Portion	Printout from Magnitude Portion	Key for Fill
0000	+	0	0, +
0001	p (Positive,) (overflow)	1	1
0010	-	2	2, NEG
0011	n (Negative,) (overflow)	3	3
0100	4	4	4
0101	8	5	5
0110	6	6	6
0111	5	7	7
1000	Ignore	8	8
1001	Ignore	9	9
1010	Ignore	Ignore	No Flex Key
1011	Ignore	Ignore	No Flex Key
1100	Space	Space	Space Bar
1101	Backspace	Backspace	No Flex Key
1110	Ignore	Ignore	No Flex Key
1111	Ignore	Period	Period

Table 5.

The reader will recall that the sign portion of a word consists of six binary bits, whereas only four bits have been shown in the above table as determining the configuration to be printed from the sign portion. In decimal printing, the computer ignores the two left-most bits of the sign portion, and these bits are therefore completely irrelevant to the printing.

It should be noted that the first four configurations may appear in the sign portion of a word as the result of normal computer operation. However, any configuration may be placed in the sign portion by the programmer for special purposes.

OCTAL MODE

Binary Configuration	Printout from Sign or Magnitude Portion	Flexowriter Key for Fill
000	0	0, +, 8
001	1	1, 9
010	2	2, NEG
011	3	3
100	4	4, Space Bar
101	5	5
110	6	6
111	7	7, Period

Table 6.

Flexowriter Printout in the Alphabetic Mode from Information contained in the Magnitude Section.

Flexowriter Character Printout		Octal Code	Flexowriter Character Printout		Octal Code
Upper -	Lower		Upper -	Lower	
A	a	41	Z	z	55
B	b	57	&	2	00
C	c	62	/	3	01
D	d	47	$	4	04
E	e	45	%	5	07
F	f	46	?	6	06
G	g	73	!	7	03
H	h	74	*	8	05
I	i	60	(	9	53
J	j	43	)	0	52
K	k	42	¢	Neg. (minus)	22
L	l	71	=	+	23
M	m	76	"	'	20
N	n	66	,	,	26
O	o	77	.	.	27
P	p	70	_	- (hyphen)	21
Q	q	50	:	;	24
R	r	63	shift down		32
S	s	44	shift up		36
T	t	75	carriage return		34
U	u	40	back space		15
V	v	72	space bar		64
W	w	51	color shift		67
X	x	56	tab		31
Y	y	54	code delete		12

Table 7.

The programmer is cautioned against using any combinations not listed in Table 7. Some of them will be ignored and others will cause the computer to "hang-up."

The complete list of twenty-five CRC 102-A commands is summarized into a tabular listing on the following page. Included in this list are the Punched Card Unit and Magnetic Tape Unit commands, although they will not be considered until these units are discussed in their respective sections. Hence, this list will serve as a unified rapid reference when programming.

COMMAND LIST FOR CRC 102-A

Operation	Instr & Code	m_1	m_2	m_3	Remarks
		---ADDRESSES OF---			
Add	ad - 35	Augend	Addend	Sum	If (m_1) is a command, the I portion is
Subtract	su - 36	Minuend	Subtrahend	Difference	retained and (m_2) is treated as a number
Multiply-Round	mr - 25	Multiplier	Multiplicand	Product	Product is rounded
Multiply-Double	md - 26	Multiplier	Multiplicand	LSH of Product	MSH of Product → Cell following m_3
Divide-Round	dr - 23	Dividend	Divisor	Quotient	Quotient is rounded
Divide-Double	dd - 24	Dividend	Divisor	Remainder	Unrounded quotient → Cell following m_3
Shift Logical	sl - 27	Operand	Shift control	Result	Magnitude of (m_2)=No. of binary shifts
Shift Magnitude	sm- 30	Operand	Shift control	Result	I of (m_2): + = Shift left − = Shift right
Scale Factor	sf - 31	Control	Operand	(m_1) - # shifts	Shifted operand → Cell following m_3
Extract	ex - 32	Source-word	Extractor	Cell Modified	Bits of $(m_1) \rightarrow (m_3)$ per "1" bits of (m_2)
Test Overflow	to - 37	Tested Word	2100 or 3000	Alternate Cmnd.	If overflow bit, execute alternate command
Test Magnitude	tm- 34	Magnitude$_1$	Magnitude$_2$	Alternate Cmnd.	If Mag$_1$ > Mag$_2$, execute alternate command
Test Algebraic	ta - 33	NUMBER ~~Word~~$_1$	NUMBER ~~Word~~$_2$	Alternate Cmnd.	If NUMBER ~~Word~~$_1$ > NUMBER ~~Word~~$_2$, execute alternate command
Test Switch	ts - 17	Switch No.	2100 or 3000	Alternate Cmnd.	If switch is up, execute alternate command
Test Search	ts - 17	2400	2100 or 3000	Alternate Cmnd.	If "bs" still in progress, execute alternate cmnd.
Print	pr - 21	First Word to be printed	Print control	No. of Words (not an address)	Bit in Magnitude of (m_2) determines number of ~~words~~ CHARACTERS printed
Halt	ht - 22	2100 ~~or 3000~~	2100 or 3000	Fill from Flex.	Computer to rest; wait for Flexowriter
Fill	fl - 11	2100 or 3000	2100 or 3000	Fill from Flex.	Computer to rest; fills from Flexowriter
Buffer Out	bo - 04	2100 or 3000	2100 or 3000	First of 8	Least significant digit of m_3 corresponds
Buffer Load	bl - 05	2100 or 3000	2100 or 3000	Memory Cells	to least significant digit of first buffer cell
Block Search	bs - 14	2100 or 3000	- T - A	A A A A	T: tape unit
Write Tape	wt - 15	2100 or 3000	- T - K	- - - O	A: block address (0 0000 thru 1 7777)
Read Tape	rt - 16	2100 or 3000	- T - K	- - - N	K: tape section; N: buffer cell
Read Card	rc - 06	Input	Addend	- - - N	N: first of the four buffer cells
Punch Decimal	pd - 12	Input	Addend	- - - N	involved in the command
Punch Octal	po - 13	Input	Addend	- - - N	

No Addr.	Addr.	I of (m_2) is mode
00	02	Octal
01	03	Decimal
10	-	Alphabetic

IX

TECHNIQUES OF PROGRAMMING

This section will be devoted to programs which will instruct the student in the use of commands and programming techniques that heretofore have not been illustrated. Generally, these programs will be of a practical nature and can perhaps be used efficiently in actual problems that the student will eventually run on the 102-A. Detailed explanations of the techniques used in each example will be given with the code.

Inasmuch as decimal numbers must be converted to binary numbers before they can be operated upon in the 102-A, consider as the first example, the program for the conversion of an integral decimal number to an equivalent binary number.

Example 1: Convert the decimal number, 89079, to a binary number. Assume, however, that the number is stored in a word in the computer as indicated:

0 0	0 0 0 0 8 9 0 7 9

The student is reminded that this number is recorded in the memory as a sequence of binary coded decimal digits (4 bits per decimal digits) as shown below.

00 0000	0000 0000 0000 0000 1000 1001 0000 0111 1001

The solution of the problem will be more apparent if we write the given number in the following form

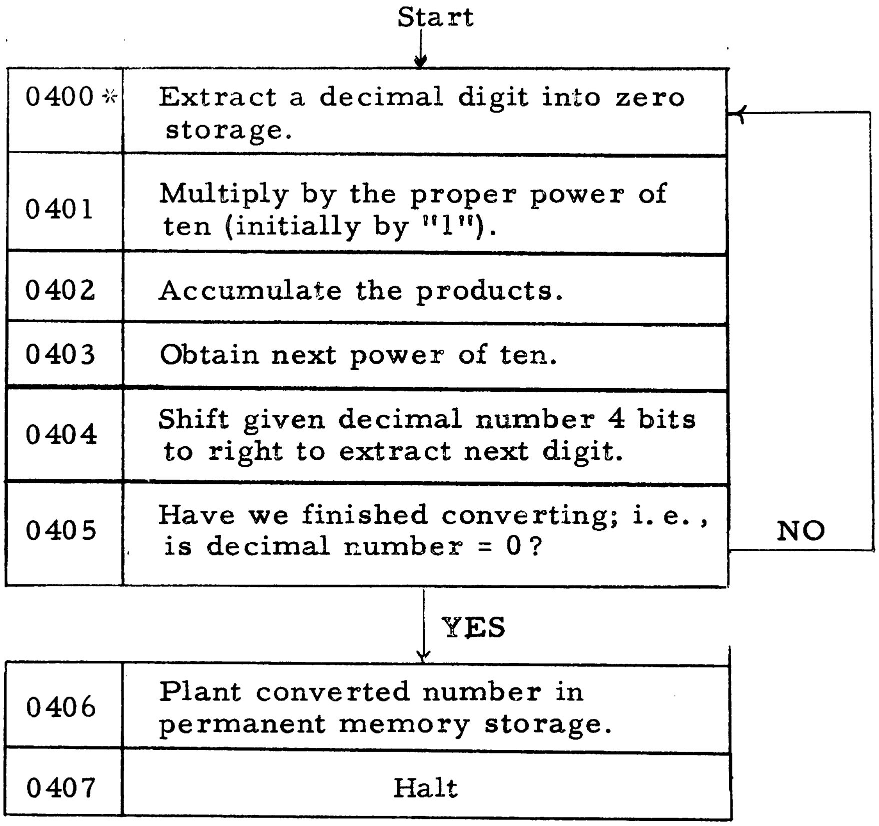
$$89079 = 8x10^4 + 9x10^3 + 0x10^2 + 7x10^1 \; 9x1 \; .$$

Since each decimal digit of the given number is represented within the machine by its equivalent four binary digits, it seems feasible that if we store the binary equivalents of "1" and "10" in a word we could then generate the binary equivalents of each term in the expanded form of the decimal number. Then, when these terms are added their sum would yield the equivalent binary number desired.

The basic plan, then, is outlined below.

1. Extract a binary coded decimal digit.
2. Multiply by the proper power of ten which is expressed in binary form.
3. Add the products obtained in step 2 to form the desired binary number.

The following flow chart incorporates the steps outlined above.



0400*	Extract a decimal digit into zero storage.
0401	Multiply by the proper power of ten (initially by "1").
0402	Accumulate the products.
0403	Obtain next power of ten.
0404	Shift given decimal number 4 bits to right to extract next digit.
0405	Have we finished converting; i. e., is decimal number = 0?
0406	Plant converted number in permanent memory storage.
0407	Halt

The corresponding code is shown on the standard 102-A coding sheet on page IX-4. Memory storage was chosen as follows:

* It is helpful to include addresses in the flow-chart once the code has been written.

Decimal number to be converted in 0200; commands and program constants in 0400 - 0413; converted number will be stored in 0300; working storage in the buffer register. The detailed description of the code is outlined below:

1. Command 0400: In this command the student is introduced to a technique which is equivalent to extracting bits from a selected word in the memory into a word of all zeros. This technique eliminates the necessity of first clearing the cell that the bits are to be copied into, but also requires that the cell being copied into be a buffer cell, which in most cases is an advantage. Command, 32 0200 0410 2101, accomplishes this feat because of the 2101 address in m_3. The reason is that the "extract" operation automatically requires the contents of m_3 to be read into the arithmetic unit, and after the "extract" operation is completed, the result is then automatically written into memory address m_3. The "ex" command is the only 102-A command that utilizes the m_3 address for both reading and writing processes during its execution. Since the computer interprets addresses such as 210X (X = 0-7) differently when reading and writing, 2101 will be interpreted as 2100 during the reading process; hence, a word of all zeros will be read into the arithmetic unit, into which the desired bits will be extracted. However, when writing the result of the extraction into the memory, 2101 will be interpreted by the computer as 2001 (See Tables 2 and 3, pages VII-8, 9). It is noteworthy that in this procedure the original contents of 2001 are irrelevant.

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 100 Date 1 - 3 - 55 Page 1 of 1

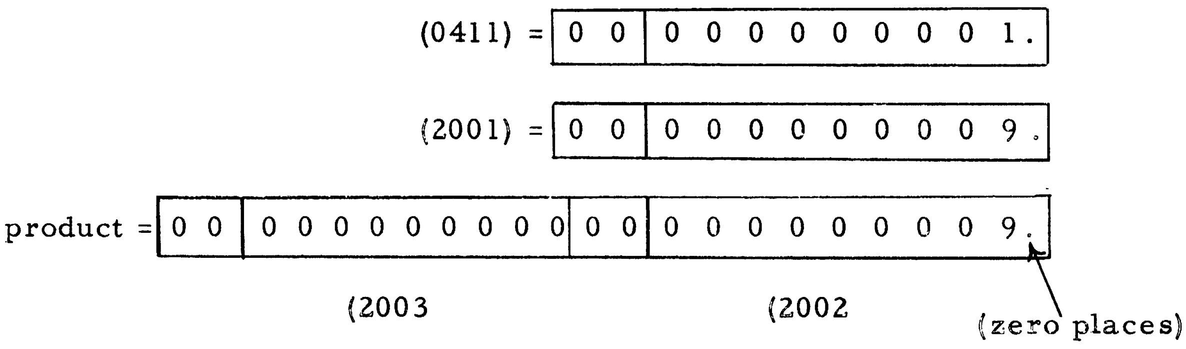
TITLE: Conversion of an integral decimal number to an equivalent binary number.

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
0400	ex-32	0200	0410	2101	Next decimal digit → (clear) 2001.
0401	md-26	0411	2001	2002	(Decimal digit) x (power of ten) → 2002.
0402	ad -35	2004	2002	2004	Accumulated products → 2004 (initially 0).
0403	md-26	0411	0412	0411	Form next higher power of ten → 0411.
0404	sm-30	0200	0413	0200	Shift decimal number 4 bits to the right.
0405	tm-34	0200	2100	0400	Is decimal number =0? No, return to 0400
0406	ad-35	2004	2100	0300	Plant converted number in 0300.
0407	ht -22	2100	2100	0000	Halt computation.
0410	02	0000	0000	0017	Decimal digit and sign extractor for 0400.
0411	00	0000	0000	0001	"1" for units digit for 0403.
0412	00	0000	0000	0012	"10" for forming powers in 0403.
0413	02	0000	0000	0004	Shifter for decimal number for 0404.
2004	00	0000	0000	0000	Temporary storage for generating converted number.

Note that the extractor, (0410), contains a "1" in its sign bit position. Thus, this routine could be used to convert negative numbers as well as positive numbers. Each time a decimal digit would be extracted it would contain its respective sign bit; consequently, the subsequent products and sums would carry the proper sign.

2. Command 0401: We will discuss at length the "multiply double length" command used here, since it has not been illustrated in a program until now. This "md" command will multiply the decimal digit previously extracted into 2001 by the proper power of "10" stored in cell 0411, and write the two-word product in cells 2002 and 2003. Even though a two-word capacity is not needed for the product in any of the multiplications in this problem, the "md" has been used because of the nature of the problem and the type of multiplication (integers). Since mechanization of the 102-A "md" command is similar to that of a desk calculator or similar to manual multiplication, it is rather simple for the programmer to decide on how to position the factors in their respective cells in order that the binary point appear in the proper position in the product. In locating the binary point in the product, the programmer merely adds the binary places to the right of the assumed binary point of each factor and this sum equals the number of binary places from the least significant end of the product. For example, during the first cycle of computation in this problem, the product of the contents of 0411 and 2001 is shown in the following schematics (for convenience, decimal numbers are used throughout this portion of the example):



The student is reminded that in "md" the m_3 address, 2002, is that of the least significant half of the two-word product; also, the most significant half of the product, which is automatically recorded in the cell physically adjacent to 2002, will be zero because of the size and position of the factors in 0411 and 2001. When the programmer intends to utilize only one call of the two-word "md" product he often forgets to allow storage for the second cell. The programmer must remember to make certain, then, that 2003 is not being used as temporary storage to preserve a number or command to be used later in the program.

3. Command 0402: This command is intended to accumulate the sum of the products of the decimal digits and their respective powers of ten. The product of the decimal digit and its respective powers of ten, stored each cycle in 2002, is added to the sum of the products accumulated thus far in the routine in 2004. Initially, 2004 is filled with all zeros.

4. Command 0403: This command is used to form the next higher power of ten needed when the next decimal digit

is to be operated upon. Initially (0411), named as m_1 in this command, is filled equal to "1"; after being used in command 0401 this command makes (0411) = "10"; then, during the next cycle it is made = "100"; etc. Since this command also requires a two-word put away for the result it seems, at first glance, that the most significant half of the product (zero) would be written into 0412 and wipe out the constant multiplier, "10". However, in the main memory numerically consecutive cells are not physically adjacent, as noted in Section VIII on the commands. In this case, the next cell physically adjacent to 0411 is 0452, which is not being used in the program anyway. The actual manner in which the sectors of physically adjacent main memory cells are numbered will be discussed in detail in Section X; however, for the present it should be sufficient for the student to know that physically adjacent sectors are addressed octal forty-one apart.

5. Command 0404: This command causes the decimal number stored in 0200 to be shifted four bit positions to the right ("-4" in 0413) and recorded back in 0200. As a result of this shift the next significant decimal digit in the given number will be in the units position - ready for extraction at the beginning of the next cycle.

6. Command 0405: This command introduces a very common programming technique for causing the computer to make a decision. The essence of this technique is to test the magnitude of the decimal number,after the previous decimal digit has been utilized and the decimal number has been shifted by command 0404, against a

word of all zeros. Until the decimal number becomes "zero" (completely shifted off the right end of 0200) the computer will recognize the magnitude of the decimal number, (0200), as being larger than zero, (2100), and control will be transferred back to command 0400. When the decimal number in 0200 does become "zero", then, of course, (0200) will no longer be larger than (2100), and computer control will continue with the command in 0406.

The principle advantage of using this technique is to generalize the routine for converting any number. The reader is reminded that the decision which this command causes the computer to make is based on whether the given decimal number is zero (all digits have been utilized), and not on the number of digits present in the given number.

7. Command 0406: The "ad" command has been used to illustrate the most obvious way to transfer the contents of a cell to another memory location. Examination of the word structure in this command reveals that we intend to add the converted number in 2003 to "zero"(2100), and record the result in 0300. Obviously, adding "zero" to (2003) will not change the contents of 2003.

8. Command 0407: Computation is halted and the computer is put in a state of "rest". Since m_1 and m_2 are irrelevant in the halt command, 2100 is used.

Example 2: Convert the fractional decimal number, .73965, to a binary fraction. Assume the given number to be stored in the memory as indicated

0 0	7 3 9 6 5 0 0 0 0

Writing the given number in its expanded form, we have

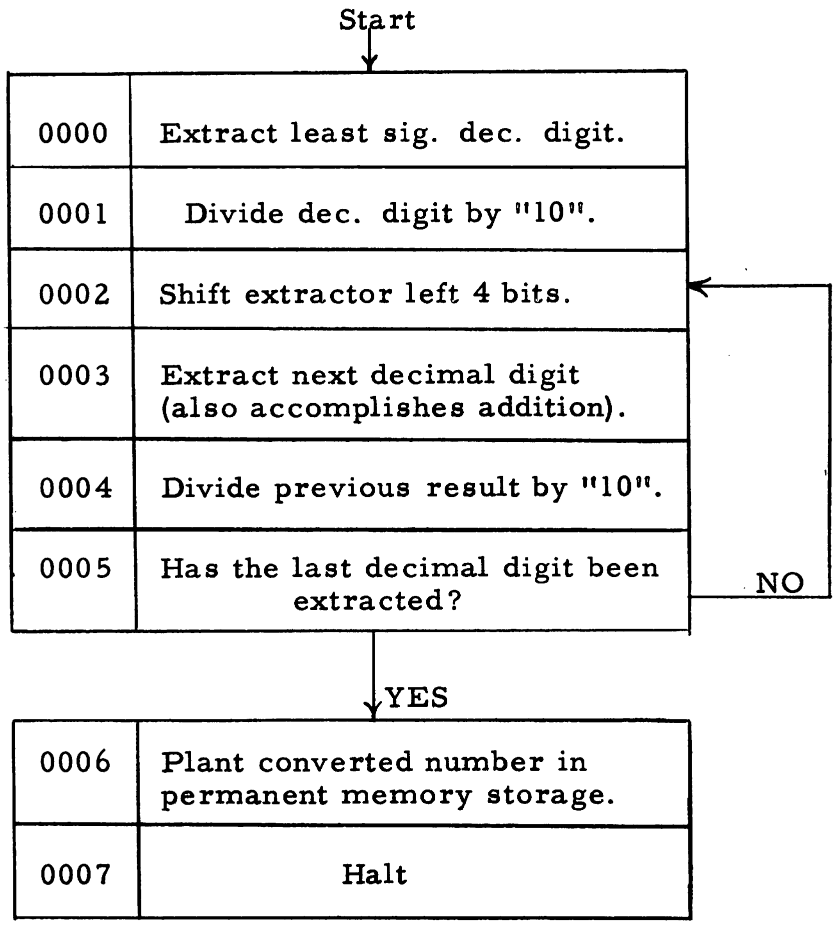
$$.73965 = \frac{7}{10} + \frac{3}{10^2} + \frac{9}{10^3} + \frac{6}{10^4} + \frac{5}{10^5}$$

An equivalent form would be

$$.73965 = \frac{1}{10}\left(7 + \frac{1}{10}\left(3 + \frac{1}{10}\left(9 + \frac{1}{10}\left(6 + \frac{5}{10}\right)\right)\right)\right) ,$$

which suggests a definite plan for the program. For example, consider that we have stored the binary equivalent of decimal "10" (binary 1010) in a memory cell. Now, if we extract the binary coded digit, 5, and divide by the "10" we have stored, we obtain the binary equivalent of . 5. Next, we extract and add the binary coded digit, 6, and obtain the binary equivalent of 6. 5. Dividing by "10" again, adding "9", dividing by "10", etc., we will finally obtain the binary equivalent of . 73965.

Let us now illustrate this plan in the form of a flow-chart.


Start
0000
Extract least sig. dec. digit.
0001
Divide dec. digit by "10".
0002
Shift extractor left 4 bits.
0003
Extract next decimal digit (also accomplishes addition).
0004
Divide previous result by "10".
0005
Has the last decimal digit been extracted?
NO
YES
0006
Plant converted number in permanent memory storage.
0007
Halt

The corresponding code and selected memory storage is shown on the following page. A detailed description of the code is outlined below.

1. Command 0000: Since the extractor in 0010 contains four binary ones in the fifth digit position, the least significant decimal digit will be extracted into 2000. Also, the binary one in the sign bit position

```
          0   2    0   0   0   0   0   3   6   0   . . .
(0010) = |000 010| 000 000 000 000 000 011 110 000  . . .
```

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE **CRC 102A**

Free Address Coding Sheet

Job No. 101 Date 1 - 3 - 55 Page 1 of 1

TITLE: Conversion of a fractional decimal number to an equivalent binary number.

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
0000	ex-32	0100	0010	2100	Lst. sig. decimal digit ⟶ clear 2000.
0001	dr-23	2000	0011	2001	(2000) divided by "10" ⟶ 2001.
0002	sm-30	0010	0012	0010	Shift (0010) (extractor) left 4 bits.
0003	ex-32	0100	0010	2001	Extract and add next dec. digit into 2001.
0004	dr-23	2001	0011	2002	Divide number being generated, (2001), by "10".
0005	tm-34	0013	0010	0002	Is extractor in most sig. dec. digit position?
0006	ad-35	2002	2100	0200	Plant converted number in 0200.
0007	ht - 22	0000	0000	0000	Halt computation.
0010	02	0000	0360	0000	Initial form of extractor; 36=four bin. 1's.
0011	00	5000	0000	0000	Divisor, 0001 & 0004; equiv. to bin. 1010. . .
0012	00	0000	0000	0004	Shifter for 0002.
0013	00	7400	0000	0000	Gauge for 0005.

of 0010 will extract the proper sign bit with the decimal digit. Once again we have used the technique of addressing m_3 as 210X to insure extraction into a cell of all zeros.

2. Command 0001: Consider the binary contents of the dividend and divisor immediately after the extraction of the decimal digit "5". Decimal points are indicated

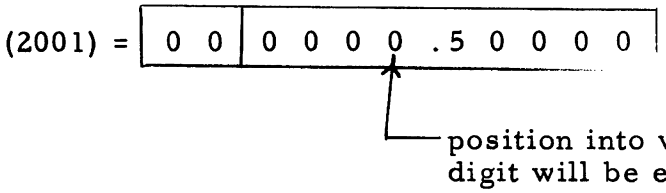
(2000) = Dividend	00 0000	0000 0000 0000 0000 0101.0000 0000 0000 0000

(0011) = Divisor	00 0000	1010. 0000 0000 0000 0000 0000 0000 0000 0000

in the word structure to aid in the discussion. The reader is reminded that the binary equivalent of "10" (divisor) is 1010, but indicated on the code sheet as an octal number, 00 5000 . . .

This command causes the binary equivalent of the decimal digit "5", extracted into 2000, to be divided by the binary equivalent of decimal digit "10", and the quotient to be rounded and stored in 2001. Emphasis must be placed on the choice of position of the divisor in this example. Actually we are treating the dividend and divisor as integers in our program, but placement of the divisor within a word depends on where we want to consider the decimal point in the quotient and how mechanized division in the 102-A functions.

The choice of placing the divisor at the extreme left end of the word was based on the fact that the next phase of computation required in this program is the addition of the next decimal digit. Since we have chosen the "extract" command as the means of adding the next decimal digit, we will want the decimal point in the quotient (. 5) to be located four decimal digit positions to the right of the most significant end of the word.



(2001) =	0 0	0 0 0 0 .5 0 0 0 0

position into which next digit will be extracted.

Since the 102-A automatically considers the binary point in the dividend and divisor to be located in the same position during division and forms the quotient in such a way that its point is assumed to be located between sign and magnitude sections, actual location of the point in the divisor to the right or left of that in the dividend is equivalent to shifting the point in the quotient an equal number of positions in the opposite direction. Hence, in this example, each time the next decimal digit is extracted into the previous quotient the next division by "10" will theoretically place the point four bit positions to the left.

For example, extract the first digit.

0 0	0 0 0 0 5. 0 0 0 0

Divide by "10".

0 0	0 0 0 0 .5 0 0 0 0

Extract next decimal digit.

0 0	0 0 0 6 .5 0 0 0 0

Divide by "10".

0 0	0 0 0 .6 5 0 0 0 0

Extract next decimal digit.

0 0	0 0 9 .6 5 0 0 0 0

Divide by "10".

0 0	0 0 .9 6 5 0 0 0 0

Continuing this process we will obtain the desired binary equivalent with the point as indicated.

0 0	.7 3 9 6 5 0 0 0 0

An alternate method for locating the binary point in a quotient is to keep track of it by powers of "2". Consider the binary point in any dividend and divisor as being located at the machine point (between sign and magnitude sections), and represent the number of positions the true point is to the right of the machine point as a negative power of "2". Manual division of the dividend and divisor represented in this manner will yield the position of the binary point in the quotient as the result of the division commands. For example, consider the first division in this problem:

$$\frac{5.}{10.} = \frac{5.\times 2^{-20}}{10.\times 2^{-4}} \begin{array}{l}\text{(5 decimal digits or 20 bit positions to the right)}\\ \text{(1 decimal digit or 4 bit positions to the right)}\end{array}$$

$$= .5 \times 2^{-16} \text{ (4 decimal digits or 16 bit positions to the right)}$$

3. Command 0002: This command causes the contents of 0010 (extractor) to be shifted four bit positions to the left and the result written back in 0010. This shift results from the " +4" code stored in 0012. As a result of this command, the "extractor" is in a position to extract the next decimal digit when needed.

4. Command 0003: This "extract" command, formed by the "sm" command in 0002, is used to extract all subsequent decimal digits, after the first, into the quotient in 2001 obtained from the previous cycle.

5. Command 0004: This "dr" command performs the subsequent divisions by "10", after the first, and stores the quotient in 2002.

6. Command 0005: This command enables the computer to decide if the conversion has been completed. The conversion will be completed, of course, when the last decimal digit has been extracted and the result divided by "10". The technique used here is to store a gauge word

which will be equivalent to the configuration of the magnitude section of the "extractor" when the last decimal digit has been extracted. Then, continually test this gauge word against the "extractor". When the "extractor" equals the gauge word the computer will discontinue cycling. For this reason we name the address of the gauge word, 0013, in m_1 and the variable "extractor", 0010, in m_2 in this "tm" command. Since this command only compares the magnitude sections of 0013 and 0010, we have not stored a "1" in the sign bit position of 0013 as we have done in 0010. Indicated below is the binary configuration of the gauge word

```
            0   0     7   4   0   0   0   0   0   0 . . .
(0013) = | 000 000 | 111 100 000 000 000 000 000 000 . . .
```

The initial configuration of the "extractor" is shown in item 1. of this discussion. After it is shifted left four bit positions (1 decimal digit) the first time by command 0002 it will take on the following form

```
(0010) = | 000 010 | 000 000 000 000 111 100 000 000 . . .
```

The magnitude section of (0013) is greater than the magnitude section of (0010) and the computer will cycle back to command 0002 in order to operate on the next decimal digit. Finally after 0010 has been shifted and used to extract the last decimal digit it will contain the following configuration.

```
(0010) = | 000 010 | 111 100 000 000 000 000 000 000 . . .
```

Now, when the computer makes the comparison it will find that the magnitude section of 0013 is not greater than that of 0010 (equality is not considered greater than) and will continue with the next command in numerical sequence (0006).

7. Command 0006: The converted number generated in 2002 is transferred to the permanent memory location, 0200; that is, its contents are added to zero and recorded in 0200.

8. Command 0007: The "ht" command stops automatic computer operation and puts it in a state of "rest."

In the previous examples we have outlined the entire program in detail. In the remaining examples in this section, detailed explanations will be given relative to the concepts and techniques which require emphasis. It is assumed that the reader will scrutinize the programs carefully and verify the coded commands which are used.

Example 3: Convert a fractional binary number to a fractional binary coded decimal number.

Assume the binary number to be stored in a word in the computer as follows:

0 $\pm$	$.b_1\ b_2\ b_3\ .\ .\ .\ b_{36}$

(The b_i's represent the binary digits of the given number.) Represent the desired decimal result as, $.d_1\ d_2\ d_3\ .\ .\ .\ d_9$, and we have the following equality

$$\frac{b_1}{2}+\frac{b_2}{2^2}+\frac{b_3}{2^3}+\ .\ .\ .\ +\frac{b_{36}}{2^{36}}=\frac{d_1}{10}+\frac{d_2}{10^2}+\frac{d_3}{10^3}+\ .\ .\ .\ +\frac{d_9}{10^9}\ .$$

The problem, of course, is to determine the binary coded decimal digits, d_1, d_2, etc., within a word in the computer ready for decimal print-out.

Consider the following plan: If both sides of the previous equation are multiplied by "ten" in their respective number systems, the result will be two equal mixed numbers. For example, the right hand side becomes

$$d_1 + \frac{d_2}{10} + \frac{d_3}{10^2} + \ldots + \frac{d_9}{10^8} .$$

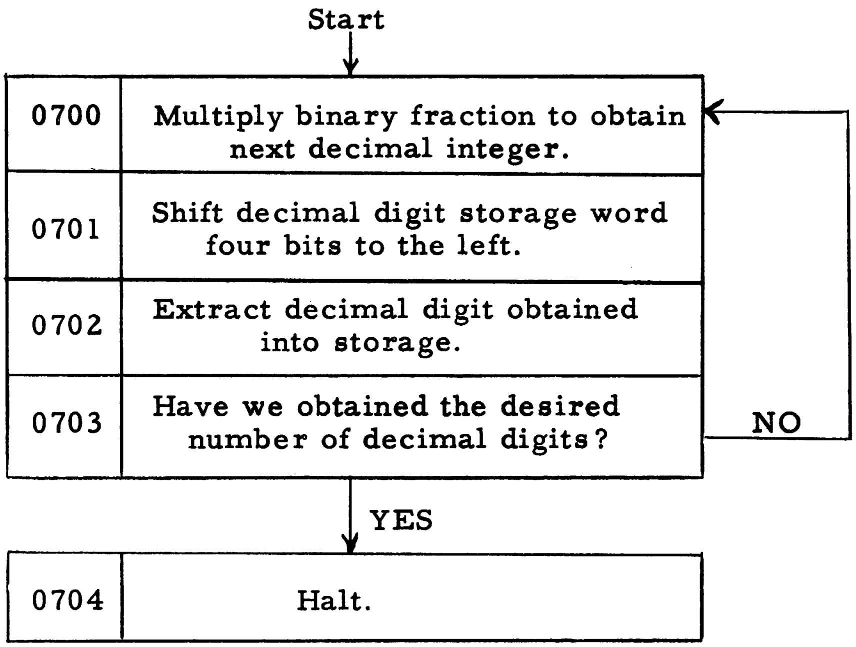
Since the whole parts and the fractional parts of two equal numbers must be equal respectively, the integer, d_1, will equal the integral part of the binary product on the left side. Consider, now, only the fractional parts of each side. Successive multiplication by "ten", then, of the fractional part of each product would yield the binary coded decimal digits, d_2, d_3, etc.

For the purpose of writing a satisfactory flow-chart, consider the following detailed plan:

1. Multiply the given binary fraction by the binary equivalent of "ten."
2. Capture the integral part of the product, d_1, and store in temporary storage.
3. Multiply the fractional part of the product by "ten" again.
4. Shift the word, which stores the first decimal integer, four bit positions to the left and store the second decimal digit obtained from the previous product.

5. Continue this process of multiplying, shifting the decimal storage word and storing the next decimal digit, until the desired accuracy is obtained or until the nine decimal digit capacity of the storage word is used.

The following flow-chart illustrated this procedure:


Start
0700
Multiply binary fraction to obtain next decimal integer.
0701
Shift decimal digit storage word four bits to the left.
0702
Extract decimal digit obtained into storage.
0703
Have we obtained the desired number of decimal digits?
NO
YES
0704
Halt.

The corresponding code is shown on the following page.

In addition to illustrating a method for converting a binary fraction to an equivalent decimal fraction, this example is intended to illustrate a subtle way of using the combination of the "sl" and "to" commands to tally and to make a decision. Use of these commands for this purpose in this problem is dependent on the initial configuration of the program constant stored in

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 103 Date 1-3-55 Page 1 of 1

TITLE: Conversion of a Fractional Binary Number to a Fractional Decimal Number.

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
0700	md-26	2000	0710	2000	Product:Integer→2001; fract. →2000.
0701	sl -27	0705	0706	0705	Storage word shifted 4 bits left.
0702	ex-32	2001	0707	0705	Extract decimal digit→0705.
0703	to-37	0705	2100	0700	If overflow bit, then→0700.
0704	ht-22	2100	2100	f	Halt.
0705	00	0421	0421	0420	Storage for decimal digits.
0706	00	f	f	0004	Shifter for 0701.
0707	02	f	f	0017	Extractor for 0702.
0710	00	f	f	0012	Constant multiplier, "10".

Note: "f" is conveniently used on the code sheet to indicate four octal zeros.

cell 0705. The technique will be more apparent if we write this octal constant, 00 0421 0421 0420, in binary form.

(0705) =	000000	0001 0001 0001 0001 0001 0001 0001 0001 0000

Now, when this entire word is shifted left four bit positions by the "shift logical" command 0701, a "1" will appear in the overflow bit position eight times in succession. Consequently, when command 0703 tests this word for overflow the computer will transfer control back to command 0700. However, the ninth time this word is shifted, and after the ninth decimal digit has been extracted into it, the overflow bit position will contain a "0" and the computer will take the next command in normal sequence, 0704.

Example 4. Obtain the square root of a fractional binary number. Assume the binary fraction to be stored in a memory cell in the following position:

0 +	$.b_1\ b_2\ b_3\ .\ .\ .\ b_{36}$

The desired square root can be obtained in the following manner:

Let

$$y = \sqrt{x} \ ; \quad 0 \leq x < 1.$$

Cons'der the least square linear approximation

$$y = ax + (1-a), \text{ for } \tfrac{1}{2} \leq x < 1 \ ,$$

where

$$a = 0.56380 \ 67877.$$

This linear equation will yield an initial approximation for $\sqrt{x}$, when $\frac{1}{2} \leq x < 1$. Now, when this initial approximation (call it y_0) is is iterated twice in the following well known formula for $\sqrt{x}$, the value obtained will have an error less than 5×10^{-9}.

$$y_{i+1} = \frac{1}{2}\left(y_i + \frac{x}{y_i}\right) \qquad (1)$$

Since the linear approximation y_0, restricts x such that, $\frac{1}{2} \leq x < 1$, we will first transform $0 \leq x < 1$ to $\frac{1}{2} \leq \bar{x} < 1$, obtain $\sqrt{\bar{x}}$, and then transform $\sqrt{\bar{x}}$, to $\sqrt{x}$. The mathematics involved in the transformation of x to $\bar{x}$ is conveniently carried out by use of the "scale factor" command.

The student is reminded that the "sf" command will shift a designated word left until a binary one appears in the most significant binary digit position of the magnitude section; also, the number of bit positions shifted will be recorded. Consequently, if we scale factor the given binary number, x, ($0 \leq x < 1$) it will automatically be transformed to $\bar{x}$ ($\frac{1}{2} \leq \bar{x} < 1$). When x is scaled in this fashion (shifted left n bit positions) the following equality is true

$$\bar{x} = 2^n x.$$

Hence,

$$\sqrt{\bar{x}} = \sqrt{2^n x}$$

or

$$\sqrt{x} = 2^{\frac{-n}{2}} \sqrt{\bar{x}}$$

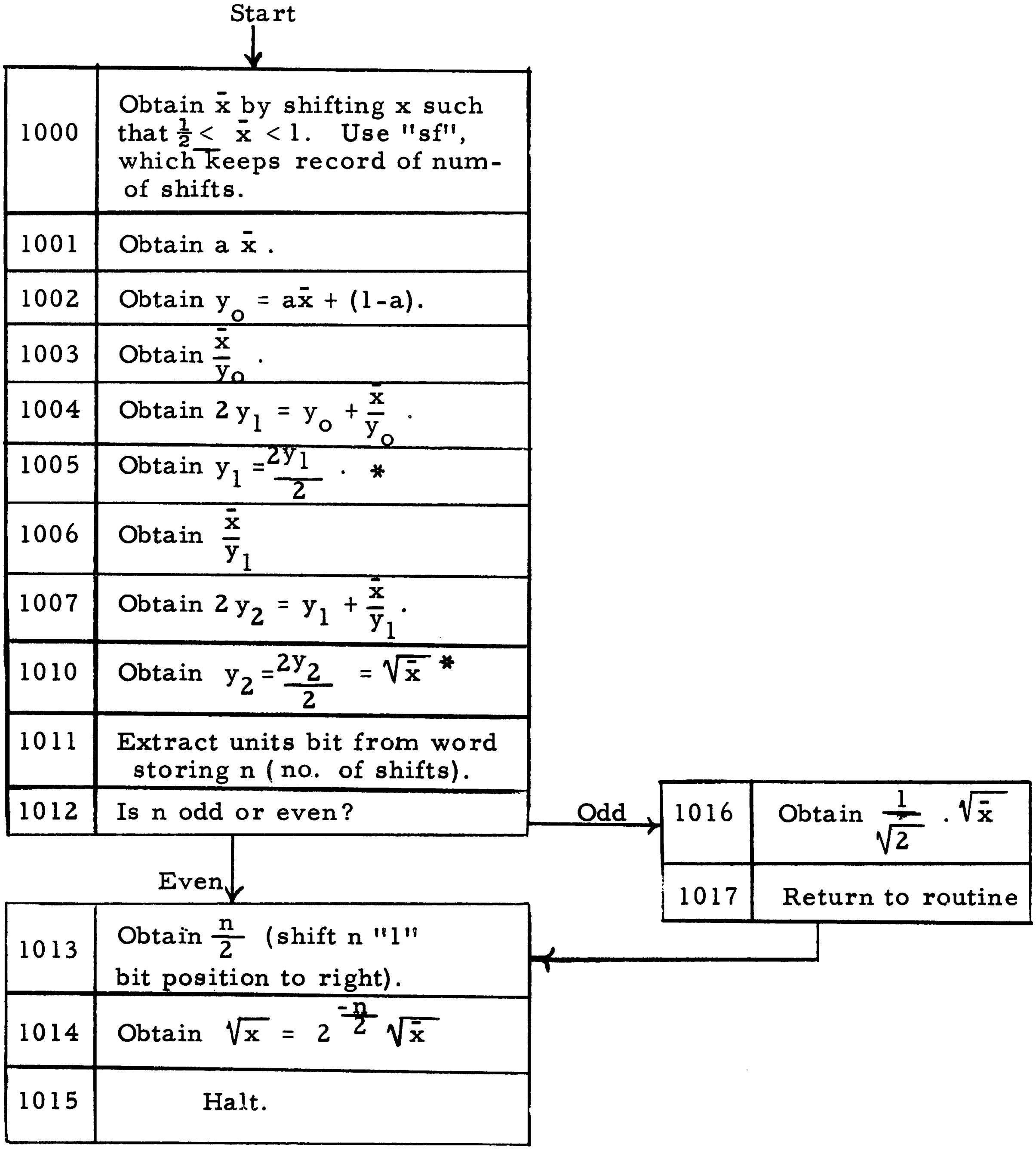
The essence of the routine we will write for $\sqrt{x}$ is outlined below:

1. Obtain $\bar{x}$ and n by use of the "sf" command.
2. Obtain $\sqrt{\bar{x}}$ by means of the linear approximation and formula (1).

3. Determine whether n is even or odd.

4. Obtain $\sqrt{x}$ by scaling $\sqrt{\bar{x}}$ by $2^{\frac{-n}{2}}$ (i.e., shift $\sqrt{\bar{x}}$ $\frac{n}{2}$ bit positions to the right. If n is odd, shift $\sqrt{\bar{x}}$ the greatest whole number contained in $\frac{n}{2}$ and multiply the result by $\frac{1}{\sqrt{2}}$).

The details of this procedure are shown in the following flowchart.


Start
1000
Obtain $\bar{x}$ by shifting x such that $\frac{1}{2} < \bar{x} < 1$. Use "sf", which keeps record of num- of shifts.
1001
Obtain a $\bar{x}$.
1002
Obtain $y_o = a\bar{x} + (1-a)$.
1003
Obtain $\frac{\bar{x}}{y_o}$.
1004
Obtain $2y_1 = y_o + \frac{\bar{x}}{y_o}$.
1005
Obtain $y_1 = \frac{2y_1}{2}$. *
1006
Obtain $\frac{\bar{x}}{y_1}$
1007
Obtain $2y_2 = y_1 + \frac{\bar{x}}{y_1}$.
1010
Obtain $y_2 = \frac{2y_2}{2} = \sqrt{\bar{x}}$ *
1011
Extract units bit from word storing n (no. of shifts).
1012
Is n odd or even?
Odd
1016
Obtain $\frac{1}{\sqrt{2}} . \sqrt{\bar{x}}$
1017
Return to routine
Even
1013
Obtain $\frac{n}{2}$ (shift n "1" bit position to right).
1014
Obtain $\sqrt{x} = 2^{\frac{-n}{2}} \sqrt{\bar{x}}$
1015
Halt.

*The "sl" command will be used to perform the division by "2" to prevent the computer from halting in the event overflow occurs as a result of the addition in 1004 and 1007.

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE **CRC 102A**

Free Address Coding Sheet

Job No. 104	Date 1-12-55	Page 1
TITLE: Routine for $\sqrt{x}$, where $0 \leq x < 1$.		of 1
x pre-stored in 0300		
		Coder M. H.

DDRESS	INST.	m_1	m_2	m_3	Remarks
1000	sf-31	2100	0300	2000	$-n \rightarrow 2000$; $\bar{x} \rightarrow 2001$.
1001	mr-25	2001	1020	2002	$a\bar{x} \rightarrow 2002$.
1002	ad-35	2002	1021	2003	$y_0 = a\bar{x} + (1-a) \rightarrow 2003$.
1003	dr-23	2001	2003	2002	$\frac{\bar{x}}{y_0} \rightarrow 2002$.
1004	ad-35	2003	2002	2003	$2y_1 \rightarrow 2003$.
1005	sl-27	2003	1022	2003	Shift right 1 bit ($y_1 \rightarrow 2y_1$ in 2003).
1006	dr-23	2001	2003	2002	$\frac{\bar{x}}{y_1} \rightarrow 2002$.
1007	ad-35	2003	2002	2003	$2y_2 \rightarrow 2003$.
1010	sl -27	2003	1022	2003	shift right 1 bit ($y_2 \rightarrow 2y_2$ in 2003).
1011	ex-32	2000	1022	2104	Units bit of n → zero storage in 2004.
1012	tm-34	2004	2100	1016	If n odd ("1" in 2004), → 1016.
1013	sm-30	2000	1022	2005	Shift right 1 bit ($\frac{n}{2} \rightarrow n$ in 2005).
1014	sm-30	2003	2005	2006	$\sqrt{x} = 2^{-\frac{n}{2}} \sqrt{\bar{x}} \rightarrow 2006$.
1015	ht -22	2100	2100	f	Halt.
1016	mr-25	2003	1023	2003	$\frac{1}{\sqrt{2}} \cdot \sqrt{x} \rightarrow 2003$.
*1017	tm-34	3000	2100	1013	Return to 1013.
1020	00	4405	2644	2047	a = 0.5638067877 (decimal).
1021	00	3372	5133	5731	1-a = 0.4361932123 (decimal).
1022	02	f	f	0001	Program constant for 1005, 1010, 1011, 1013.
1023	00	5520	2363	1503	$\frac{1}{\sqrt{2}} = 0.7071067812$ (decimal).

*Since 3000 (G register) is always greater than zero (2100), this word structure for the "tm" command is used as an unconditional transfer command.

Example 4. Compute sin θ for θ given in degrees; $-360^\circ < \theta < 360^\circ$.

We will use the following series as the basis of our computation:

$$\sin\left(\frac{\Pi}{2} X\right) = C_1\ X + C_3 X^3 + C_5 X^5 + C_7 X^7,$$

where

$$C_1 = 1.5707\ 949$$

$$C_3 = -0.6459\ 210$$

$$C_5 = 0.0794\ 877$$

$$C_7 = -0.0043\ 625$$

and

$$-1 \leq x \leq 1.$$

The error as a result of this series will be less than 1×10^{-6}.

This series gives $\sin\left(\frac{\Pi}{2} X\right)$, where X is that fraction of a quadrant represented by θ. For example, if θ is in the first quadrant, then X equals $\frac{\theta^\circ}{90^\circ}$. If θ is in one of the other three quadrants, $\frac{\theta^\circ}{90^\circ}$ will be greater than 1; the integral part indicates the quadrant; the fractional part (suitably complemented for the II and IV quadrants) equals X. Of course $\frac{\theta^\circ}{90^\circ}$ cannot be computed directly by the machine by a division command if $\frac{\theta^\circ}{90^\circ} \geq 1$ (i. e., overflow would occur and the quotient could be meaningless). Therefore, to obtain X, we will obtain $\frac{\theta^\circ}{360^\circ}$ and shift the quotient left 2 bit positions. This is most convenient because the indicator of the quadrant (integral part of $\frac{\theta^\circ}{90^\circ}$) will then be in the sign and overflow bit positions.

When $\frac{\theta^\circ}{90^\circ}$ is obtained by shifting the word containing $\frac{\theta^\circ}{360^\circ}$ left 2 bit positions, a "1" in the overflow bit positions indicates that θ is a II or IV quadrant angle ($\frac{\theta^\circ}{90^\circ}$ = "1" or "3"); a "0" or no overflow, indicates that θ is a I or III quadrant angle. Furthermore, the sign and magnitude section of the shifted word equals the sign and magnitude of X, respectively, in the I and III quadrant cases. In the II and IV quadrant cases, to obtain X the shifted word must first be properly complemented. Hence, in either case, the equivalent X for a given θ can easily be obtained from $\frac{\theta^{o}}{360^{o}}$, and sin θ can be computed from the series. However, computation of the series by the computer will be simplified if we scale the C_i's such that $\left| C_i \right| < 1$; that is, use $\overline{C}_i = \frac{C_i}{2}$. Hence, we will compute

$$\frac{1}{2} \sin\left(\frac{II}{2} X\right) = \overline{C}_1 X + \overline{C}_3 X^3 + \overline{C}_5 X^5 + \overline{C}_7 X^7 ,$$

and multiply the result by "2" to obtain $\sin\left(\frac{II}{2} X\right)$.

The computation will be carried out according to the following flow-chart and the corresponding code.

This routine emphasizes the following programming techniques:

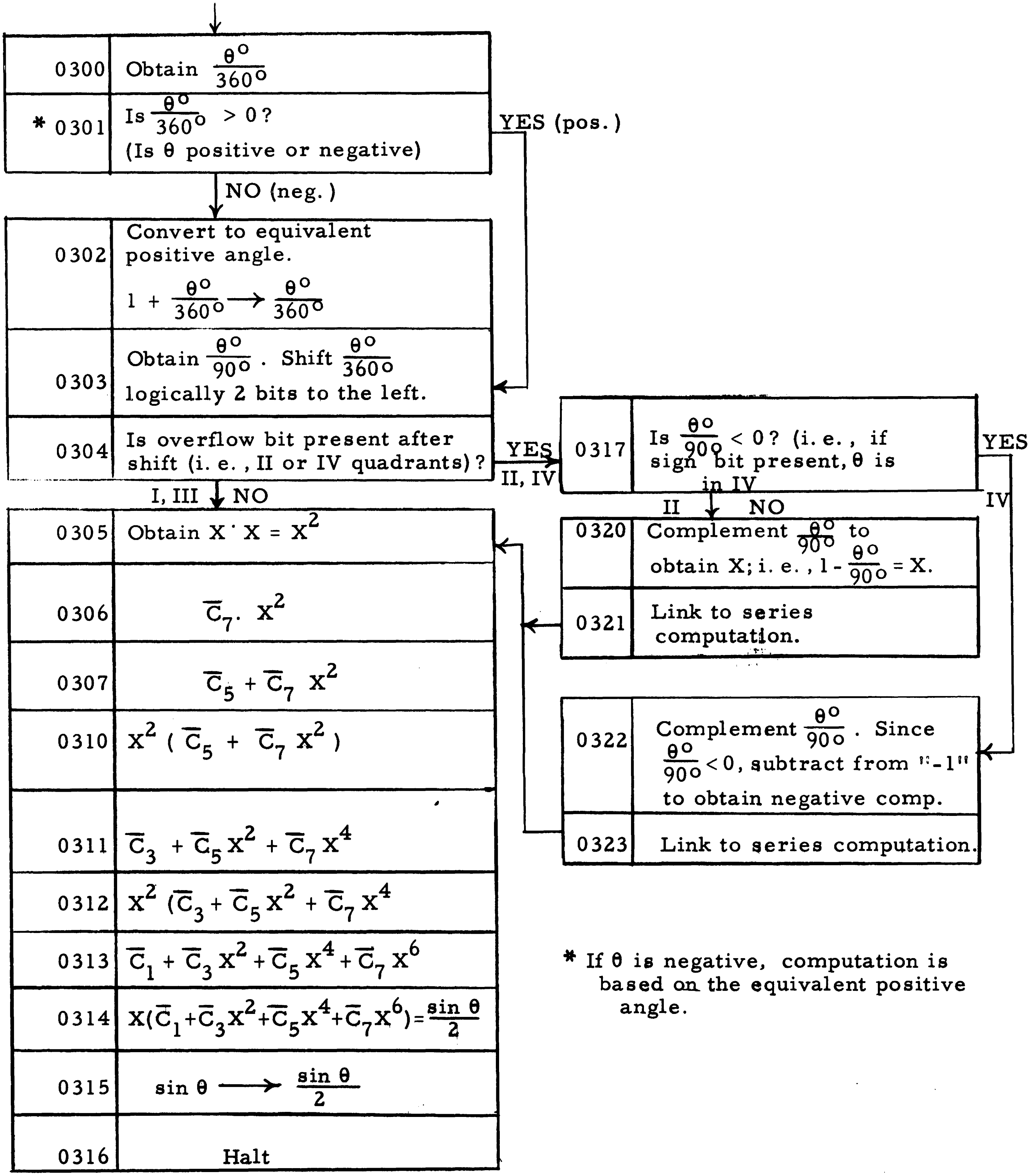
1. Scaling all quantities such that the binary point is located at the most significant end of a word (between sign and magnitude sections) during all computation.

2. Use of the "multiply and round-off" command when the operands are fractional numbers (binary point located as indicated in item 1 above). The binary point in the product is then located in the same position as the operands, which reduces computation to a fixed-point operation.

3. The "test algebraically" command has been used to determine whether a number is positive or negative, and choose an alternative set of commands

accordingly. This test is conveniently made by testing the number against "zero" (cell 2100).

4. The "test for overflow marker" command is used skillfully to make a logical decision, based on the presence, or absence, of an overflow bit. This example further illustrates the ability to use the "to" command to make a decision, which is not necessarily based on the presence of an overflow bit as the result of an "add", "subtract", or "division" operation.


0300 Obtain $\frac{\theta^\circ}{360^\circ}$
* 0301 Is $\frac{\theta^\circ}{360^\circ} > 0$?
(Is θ positive or negative)
YES (pos.)
NO (neg.)
0302 Convert to equivalent positive angle.
$1 + \frac{\theta^\circ}{360^\circ} \longrightarrow \frac{\theta^\circ}{360^\circ}$
0303 Obtain $\frac{\theta^\circ}{90^\circ}$. Shift $\frac{\theta^\circ}{360^\circ}$ logically 2 bits to the left.
0304 Is overflow bit present after shift (i. e. , II or IV quadrants)?
YES
II, IV
I, III
NO
0305 Obtain $X \cdot X = X^2$
0306 $\overline{C}_7 \cdot X^2$
0307 $\overline{C}_5 + \overline{C}_7 X^2$
0310 $X^2 (\overline{C}_5 + \overline{C}_7 X^2)$
0311 $\overline{C}_3 + \overline{C}_5 X^2 + \overline{C}_7 X^4$
0312 $X^2 (\overline{C}_3 + \overline{C}_5 X^2 + \overline{C}_7 X^4$
0313 $\overline{C}_1 + \overline{C}_3 X^2 + \overline{C}_5 X^4 + \overline{C}_7 X^6$
0314 $X(\overline{C}_1+\overline{C}_3X^2+\overline{C}_5X^4+\overline{C}_7X^6)=\frac{\sin\theta}{2}$
0315 $\sin\theta \longrightarrow \frac{\sin\theta}{2}$
0316 Halt
0317 Is $\frac{\theta^\circ}{90^\circ} < 0$? (i. e. , if sign bit present, θ is in IV
YES
IV
II
NO
0320 Complement $\frac{\theta^\circ}{90^\circ}$ to obtain X; i. e. , $1 - \frac{\theta^\circ}{90^\circ} = X$.
0321 Link to series computation.
0322 Complement $\frac{\theta^\circ}{90^\circ}$. Since $\frac{\theta^\circ}{90^\circ} < 0$, subtract from "-1" to obtain negative comp.
0323 Link to series computation.

* If θ is negative, computation is based on the equivalent positive angle.

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 105 Date 1-19-55 Page 1 of 1

TITLE: Computation of sin θ for θ given in degrees; $-360^\circ < \theta < 360^\circ$. Store θ in cell 0100 with octal point as indicated in cell 0324 (360°).

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
0300	dr-23	0100	0324	2000	$\theta^\circ/360^\circ \longrightarrow 2000$ (no overflow).
0301	ta-33	2000	2100	0303	If $\theta^\circ/360 > 0, \longrightarrow 0303$
0302	ad-35	2000	0325	2000	If $\theta^\circ/360^\circ \leq 0$, $1 + \theta^\circ/360^\circ \longrightarrow \theta^\circ/360^\circ$.
0303	sl-27	2000	0326	2000	$\theta^\circ/90^\circ \longrightarrow \theta^\circ/360^\circ$ in 2000.
0304	to-37	2000	2100	0317	If II or IV quadrants $\longrightarrow$ 0316.
0305	mr-25	2000	2000	2001	$X \cdot X = X^2 \longrightarrow 2001$
0306	mr-25	2001	0327	2002	$\bar{C}_7 X^2 \longrightarrow 2002$.
0307	ad-35	2002	0330	2002	$\bar{C}_5 + (\bar{C}_7 X^2) \longrightarrow 2002$.
0310	mr-25	2001	2002	2002	$X^2 (\bar{C}_5 + \bar{C}_7 X^2) \longrightarrow 2002$.
0311	ad-35	2002	0331	2002	$C_3 + (\bar{C}_5 X^2 + \bar{C}_7 X^4) \longrightarrow 2002$.
0312	mr-25	2001	2002	2002	$X^2 (\bar{C}_3 + \bar{C}_5 X^2 + \bar{C}_7 X^4) \longrightarrow 2002$.
0313	ad-35	2002	0332	2002	$\bar{C}_1 + (\bar{C}_3 + \bar{C}_5 X^2 + \bar{C}_7 X^4) \longrightarrow 2002$.
0314	mr-25	2000	2002	2002	$X(2002) = \frac{\sin \theta^\circ}{2} \longrightarrow 2002$.
0315	sl -27	2002	0333	2002	$\sin \theta^\circ \longrightarrow \frac{\sin \theta}{2}$ in 2002.
0316	ht-22	2100	2100	f	Halt
0317	ta-33	2100	2000	0322	If $\theta^\circ/90^\circ < 0$, then θ is in IV; $\longrightarrow$ 0322.
0320	su-36	0325	2000	2000	Complement $\theta^\circ/90^\circ$ for II, $1-\theta^\circ/90^\circ \longrightarrow 2000$
0321	tm-34	3000	2100	0305	Link to series computation.
0322	su-36	0334	2000	2000	Complement $\theta^\circ/90^\circ$ for IV $-1-\theta^\circ/90^\circ \longrightarrow 2000$
0323	tm-34	3000	2100	0305	Link to series computation.
0324	00	5500	f	f	360° = octal 550°
0325	00	7777	7777	7777	"1" for complementing in 0302 and 0320.
0326	00	f	f	0002	Shift control (left 2 bits) for 0303.
0327	02	0010	7363	0600	$\bar{C}_7$ = decimal -0.0021813.
0330	00	0242	6246	6600	$\bar{C}_5$ = decimal 0.0397439.
0331	02	2452	6611	7100	$\bar{C}_3$ = decimal -0.3229605
0332	00	6220	7716	2100	$\bar{C}_1$ = decimal 0.7853975
0333	00	f	f	0001	shift control (left 1 bit) for 0315.
0334	02	7777	7777	7777	"-1" for complementing in 0322.

Pre-Setting Commands and Program Constants.

Many of the programming techniques previously illustrated showed how command words or program constants in a rcutine were modified in the course of execution of the routine. Consequently, if that same routine were to be re-run, it would be necessary to re-set the words previously modified in the routine to their initial configuration. Furthermore, during code checking it is very likely that errors will occur in the initial code. When these errors are discovered and corrected, subsequent checks are necessary for final verification of the code. Each time the code is checked, then, those words which were modified during the previous check must first be re-set.

It is very important that pre-setting of words in a routine be done in the simplest and fastest way possible. Certainly it would be very expensive time-wise to make such changes manually each time the routine is run; hence, we will code the machine to do the job for us.

Pre-setting words in a routine can be accomplished by simply storing, as program constants, the initial configuration of those words which will be modified in the course of execution of the routine, and then code the initial commands in the routine such that they will "plant" these pre-stored words in their respective cells in the routine. For example, consider the sample code on page V-19 (for the convenience of the reader this code has been copied on the following page). As a tutorial example this code merely added ten numbers and took in consideration the possibility of overflow after each addition. However, it will serve as a good example for pre-setting program words.

If this routine had been used to add ten numbers in cells 1100 -1111, and then ten new numbers were filled into 1100 - 1111, this routine could not be used again until cells 1200, 2000 and 2001 were re-set to their initial configuration. Consequently, we will store these three words as program constants in cells 1212, 1213 and 1214:

(1212) = 35 2000 1100 2000
(1213) = 00 0000 0000 0000
(1214) = 00 0000 0000 0000

Now, when the following commands are affixed at the beginning of the routine, which previously started in cell 1200, the above named cells will be "planted" in 1200, 2000 and 2001, respectively:

Address	I	m_1	m_2	m_3	Remarks
1175	35	1212	2100	1200	(1212) + "0"→1200
1176	35	1213	2100	2000	(1213) + "0"→2000
1177	35	1214	2100	2001	(1214) + "0"→2001

The starting address of this routine, then, would be 1175; the complete code is shown on page IX-34.

Since the programmer must first write the basic code before he is cognizant of which cells will require pre-setting, it would be very helpful if he were to "flag" those cells in some way as the code is being written. This is one of the reasons why the programmer is advised, when the code is being written, to "bracket" those command words which will be modified.

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 106 Date 1-3155 Page 1 of 1

TITLE: Addition of ten numbers with possible overflow.

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
1200	ad-35	2000	[1100]	2000	Next number ± (2000);(2000)initially "0".
1201	to-37	2000	2100	1205	Is overflow bit present in (2000)?
1202	ad-35	1200	1207	1200	Modify (1200); add "1" to m_2 of 1200.
1203	tm-34	1210	1200	1200	Have we added 10 nos. ? No,⟶1200.
1204	ht -22	0000	0000	0000	Halt
1205	ad-35	1211	2001	2001	"1" + (2001)⟶2001; of.tally, init. "0".
1206	tm-34	1210	2100	1202	Return to 1202.
1207	00	0000	0001	0000	Modifier for 1202.
1210	00	2000	1112	2000	"Gauge" for test in 1203.
1211	00	0000	0000	0001	"1" for tally in 1205.
1100	00	6502	3450	0000	
1101	02	7300	2540	0300	Numerical
.	.	.	.	.	Data
1111	00	0470	6600	0000	
2000	[00	0000	0000	0000]	Working storage, initially zero.
2001	[00	0000	0000	0000]	Used for tally, initially zero.

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 107 Date 1-15-55 Page 1 of 1

TITLE: Addition of ten numbers with possible overflow. Modified cells are pre-set.

Coder M. H.

ADDRESS	INST.	m_1	m_2	m_3	Remarks
1175	ad-35	1212	2100	1200	(1212) + "0" ⟶ 1200.
1176	ad-35	1213	2100	2000	(1213) + "0" ⟶ 2000.
1177	ad-35	1214	2100	2001	(1214) + "0" ⟶ 2001.
1200	ad-35	2000	$\boxed{1100}$	2000	Next number + (2000); (2000) initially "0".
1201	to-37	2000	2100	1205	Is overflow bit present in (2000) ?
1202	ad-35	1200	1207	1200	Modify (1200); add "1" to m_2 of 1200.
1203	tm-34	1210	1200	1200	Have we added 10 nos. ? No, ⟶ 1200.
1204	ht-22	0000	0000	0000	Halt
1205	ad-35	1211	2001	2001	"1" + (2001) 2001; o. f. tally, initially "0".
1206	tm-34	1210	2100	1202	Return to 1202.
1207	00	0000	0001	0000	Modifier for 1202.
1210	00	2000	1112	2000	"Gauge" for test in 1203.
1211	00	0000	0000	0001	"1" for tally in 1205.
1212	35	2000	1100	2000	Initial configuration of 1200.
1213	00	0000	0000	0000	" " of 2000.
1214	00	0000	0000	0000	" " of 2001.
1100	00	6502	3450	0000	
1101	02	7300	2540	0300	Numerical
.					Data
1111	00	0470	6600	0000	

Sub-Routines

Certain specific calculations are common to a great many problems. Routines which will perform these specific calculations can be coded in advance and made available for future use in any problem in which they are applicable. Routines of such a specific nature are called sub-routines.

Since sub-routines are coded in advance of their need, they can be prepared on tape and carefully tested before being filed for future use. A collection of sub-routines (referred to as a "library" of sub-routines) would include, perhaps, such routines as conversion from decimal to binary, conversion from binary to decimal, square root, trigonometric functions, exponential functions, etc.

A library of sub-routines is often defined as an extension to the instruction code of the machine, since a programmer would only need to refer to a specific sub-routine in a flow-chart in exactly the same manner as he would refer to a standard command.

Certain conventions must be adhered to when sub-routines are prepared in order to make them convenient for general use. A sub-routine can be designed as a sequence of commands for inclusion in a main routine,or as a sequence of commands which will occupy an alternate part of the memory where they would be linked to and from by means of commands in a main routine. Since any sub-routine may be used many times in the same problem, it should be coded such that the operands will always be located in certain cells and the results will always be located in certain cells. Also, sub-routines should be coded with special emphasis on economy of memory space and on minimum access time coding techniques (Section X).

Considerable coding and computing time can be saved when programming a problem by judicious use of sub-routines. Since sub-routines will always have been carefully coded and checked before filing, duplication of work is avoided, and code checking of the entire problem is minimized.

The use of sub-routines in a program may be calssified into two categories:

1. A sub-routine will be used only once in the course of computation of a given problem.
2. A sub-routine will be used more than once in the course of computation of a given problem.

In the case of category 1, since any sub-routine would be used only once in a program for the solution of a given problem, it would perhaps be more profitable time-wise for the programmer to include the sequence of commands of the sub-routine as part of the main routine. For this reason, one of the ways to prepare sub-routines would be in a symbolic, or "free address" fashion, "free address" implies that the memory addresses locating the sub-routine commands and program constants, which are arbitrary, are represented symbolically, on a code sheet, but when the sub-routine is included as part of the code of the main routine, precise memory addresses will replace the free address symbols. As an example, the code on page IX-4 is written on the following page with the free addresses replaced by appropriate symbols. Note that the buffer addresses and address 2100 are not changed. The programmer must make certain, then, that these buffer cells are available at the time the sub-routine is entered.

When considering how to prepare sub-routines, which will occupy an alternate portion of the memory, so that they can be linked

The National Cash Register Company
ELECTRONICS DIVISION

SAMPLE

CRC 102A
Free Address Coding Sheet

Job No. 100

Date 1-3-55

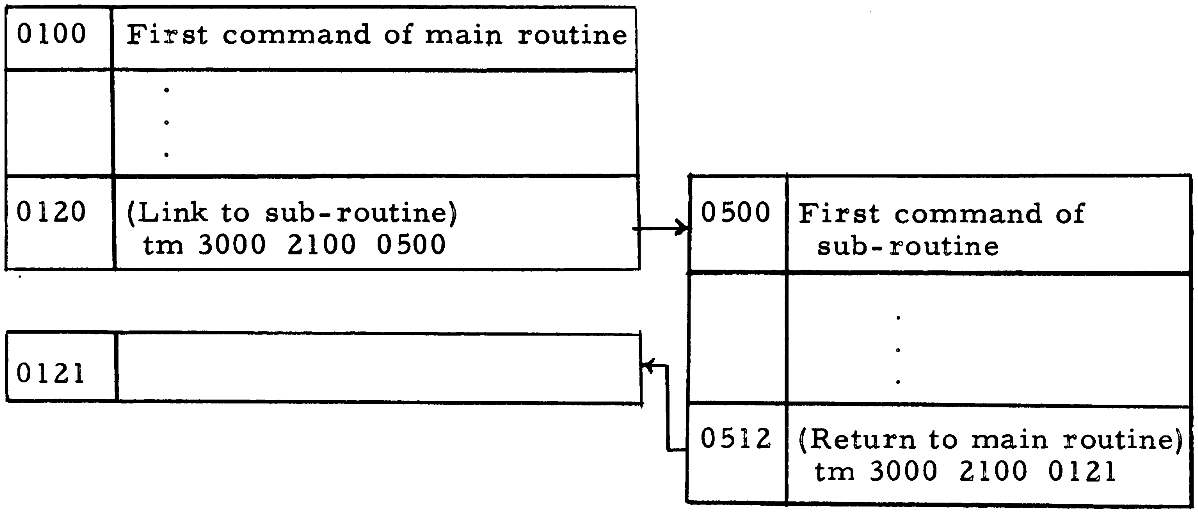
Page 1 of 1

TITLE: Free address code for conversion of an integral decimal number to an equivalent binary number.

Coder M. H.

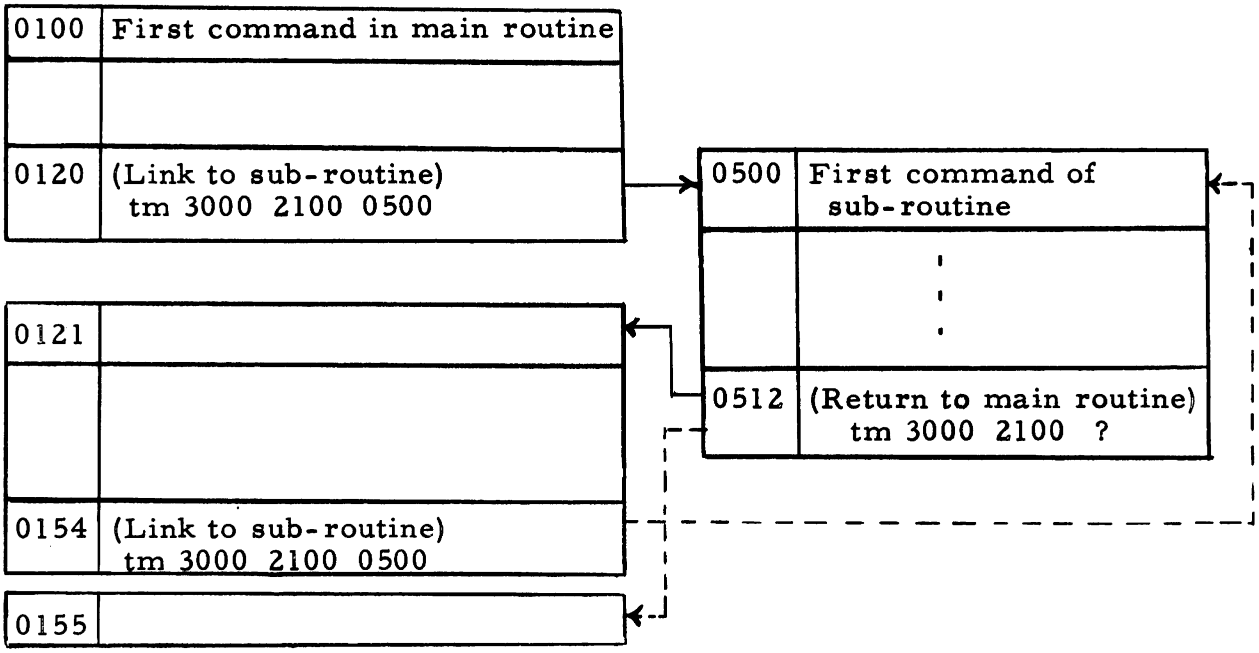
ADDRESS	INST.	m_1	m_2	m_3	Remarks
I_0	ex-32	No.	C_0	2101	Next decimal digit ⟶(clear) 2001.
I_1	md-26	C_1	2001	2002	(Decimal digit) x (power of ten)⟶ 2002.
I_2	ad-35	2004	2002	2004	Accumulated products→2004 (initially 0).
I_3	md-26	C_1	C_2	C_1	Form next higher power of ten⟶0411.
I_4	sm-30	No.	C_3	No.	Shift decimal no. 4 bits to right.
I_5	tm -34	No.	2100	I_o	Is decimal No. = 0 ? No, return to 0400.
I_6	ad-35	2004	2100	Ans.	Plant converted number in 0300.
C_0	02	f	f	0017	Dec. digit and sign extractor for 0400.
C_1	00	f	f	0001	"1" for units digit for 0403.
C_2	00	f	f	0012	"10" for forming powers in 0403.
C_3	02	f	f	0004	Shifter for decimal number for 0404.
2004	00	f	f	f	Temporary storage for generating converted number.

with a main routine, the emphasis will be placed on their use as indicated in category 2. The following skeleton flow-chart is only intended to illustrate how simple it would be to link between a main routine and a sub-routine by means of unconditional transfer commands ("tm"), if the sub-routine were to be entered only once from the main routine.



Apparently all that is required of the programmer when preparing such a linkage is that he know the initial address of the sub-routine and the return address in the main routine, which are the m_3 addresses of the unconditional transfer commands in the flow-chart.

However, when a subroutine is to be entered more than once from the main routine, the return link must permit a transfer to various places in the main routine. For example, consider the following flow-chart:



Obviously the unconditional transfer command in 0512 can be used as a link to the main routine only once, unless its m_3 address is changed each time it is used in conjunction with a different portion of the main routine. Fortunately, there is a way in which we can cause the computer to alter the m_3 address of the exit command each time the sub-routine is entered. The cunning manner by which we can pre-set this exit from the sub-routine will emphasize the importance of having access to the G register by means of address 3000.

Linking, or automatic setting up of an exit from a sub-routine, can be accomplished by coding the first two commands of a sub-routine to capture the control number of the next command in the main routine and plant it in m_3 of the sub-routine exit command. The reader is reminded that the control number or address of the next command is always held in m_2 of G during automatic computer operation, but at the conclusion of execution of a command the

the control number appears in m_2 of H (review section VII).

Consider, then, the given sub-routine to be located in 0500-0512; consider the first two commands to be necessary linking set-up commands (0500 and 0501); consider the last command to be an exit command, which will be pre-set by the two initial linking commands. Hence, the previous flow chart will serve as an example of how this sub-routine may be linked with a main routine.

Assuming that the main routine will transfer computer control to the sub-routine (0500) by means of an unconditional transfer command, the following two commands in 0500 and 0501 will properly set up the return link each time the sub-routine is entered:

Address	I	m_1	m_2	m_3	Remarks
0500	sm-30	3000	0513	2000	Shift control number in G 24 bits to right → 2000.
0501	ex-32	2000	0514	0512	Extract control number from 2000 → 0512.
0513	02	f	f	0030	Shift constant (24 to right).
0514	00	f	f	7777	Extractor (copies m_3).

The fact that these two commands will properly pre-set the return link will be verified by the following discussion:

1. At the time the transfer command, 0120, in the main routine enters the H register (control) for execution, G and H contain:

G	XX	XXXX 0121 XXXX
H	34	3000 2100 0500

(X denotes irrelevant digits. They would depend on the word structure of the previous command in the main routine, 0117.)

2. Just prior to that phase of control unit operation which interprets the instruction digits, the new control number, 0121, will be copied into m_2 of H. G and H will contain:

G	XX	XXXX 0121 XXXX
H	34	3000 0121 0500

3. When the computer identifies this command (0120) as a decision command which works, the H register is automatically circulated left four octal digit positions, thus, the transfer address, 0500, will be in the m_2 position of H. However, the control number, 0121, which referred to the next command in the main routine is now in m_1 of H. G and H now contain:

G	XX	XXXX 0121 XXXX
H	00	0121 0500 0003

4. The computer now sees that its next command is to be taken from cell 0500, which is the first command in the sub-routine. When this command is brought into H, H and G contain:

G	00	0121 0501 0003
H	30	3000 0513 2000

Consequently, the current command in 0500 (the first of the two linking set-up commands of the subroutine) is coded to shift the contents of G, which will have been transferred to E, right twenty-four bit positions and write the result in 2000. Cell 2000 would contain:

(2000) =	00	0000 0000 0121

5. The next command in the sub-routine, 0501, is an extract command which copies the control number we are seeking (0121) from m_3 of 2000 into m_3 of the exit command, 0512. Hence, 0512 would contain:

(0512) =	34	3000 2100 0121

Now, when the computer terminates the sub-routine with command 0512, computer control will be transferred back to 0121.

Referring to the flow chart once again, we see that the main routine will transfer control from 0154 to the sub-routine in 0500 (dotted lines on the flow chart). In like manner, commands 0500 and 0501 will capture the next control number in the main routine, which is 0155 in this case, and plant it in m_3 of 0512. Thus, the computer has been programmed to automatically set-up the return link to the main routine whenever we enter the sub-routine.

Consider this discussion as also being tutorial with respect to a technique for capturing information from the G register. The student will certainly find many other coding applications for such a technique.

APPENDIX I

This appendix is devoted to the general nature of the fill operation within the machine. The following table and E register schematics will perhaps help to clarify the procedure. Reference will be made to the flip-flops* labeled E_1 - E_5 and A_1 - A_6. The contents of TABLE 1a are the binary configurations set up in the A_1 - A_5 flip-flops when the corresponding Flexowriter character is struck.

Flexowriter Character	A_5	A_4	A_3	A_2	A_1
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
+	0	0	0	0	0
NEG (-)	0	0	0	1	0
Space Bar	0	1	1	0	0
Period	0	1	1	1	1
f	1	0	0	0	0
o	1	1	0	0	0
d	1	1	0	0	1
HYPHEN (-)	1	1	1	0	0
TAB	1	1	1	1	0
s	1	1	1	1	1

TABLE 1a

*A flip-flop (abbreviated f-f) is a vacuum tube circuit used for temporary storage of a bit. It is principally two tubes, generally housed in a common "bottle", with their circuitry arranged such that when one tube conducts current the other will not. That is, if tube A is conducting and tube B is non-conducting, the f-f is said to be in the "1" state; and conversely, if tube B is conducting, the f-f is said to be in the "0" state.

Examination of this table reveals the following facts:

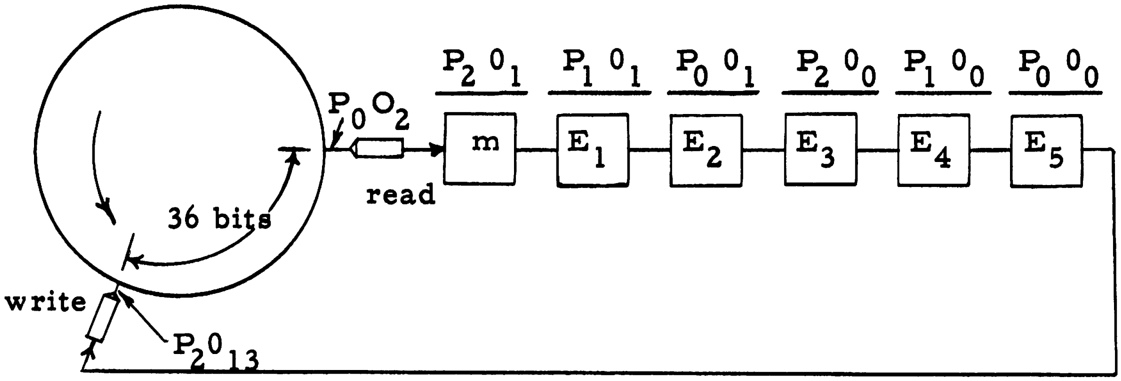
1. The state of the A_5 f-f, "0" or "1," informs the computer whether a data or control key, respectively, has been struck.
2. The actual binary equivalent of the numerals 0 - 9 is stored in the A_1 - A_4 f-f's (least significant bit in A_1).
3. The "+" and "-" characters transmit the same information to the 102-A, via the A_1 - A_5 f-f's, as do the "0" and "2" numerals, respectively.
4. The "f" character differs from the numeral zero by virtue of the state of the A_5 f-f. The "1" bit present here informs the computer that certain logical circuitry must be brought into "play" (see Item 4 - E register schematic on the following pages).
5. The o, d, HYPHEN, TAB and s characters are also control keys for the filling operation, and recognized as such by the computer since a "1" appears in A_5. Furthermore, the distinct function of each of these control keys is made known to the 102-A by the state of the A_1 - A_4 f-f's.
6. The space bar and period key are hybrid characters in the sense that they enter different information in the octal and decimal modes.

Although the E register has often been referred to as a recirculating register, no clarification of this principle has been offered as yet. The nature of such a register is quite simple if we first consider the 42 bits of a word numbered according to an octal digit and one of the three bit positions within it, i. e., P_0 O_0, . . . , P_2 O_{13}.

O_{13}	O_{12}	O_1	O_0
$P_2P_1P_0$	$P_2P_1P_0$	$P_2P_1P_0$	$P_2P_1P_0$

Secondly, consider these 42 bits as 36 magnetized spots on the drum and the remaining 6 bits held in flip-flops as shown in the following schematic:

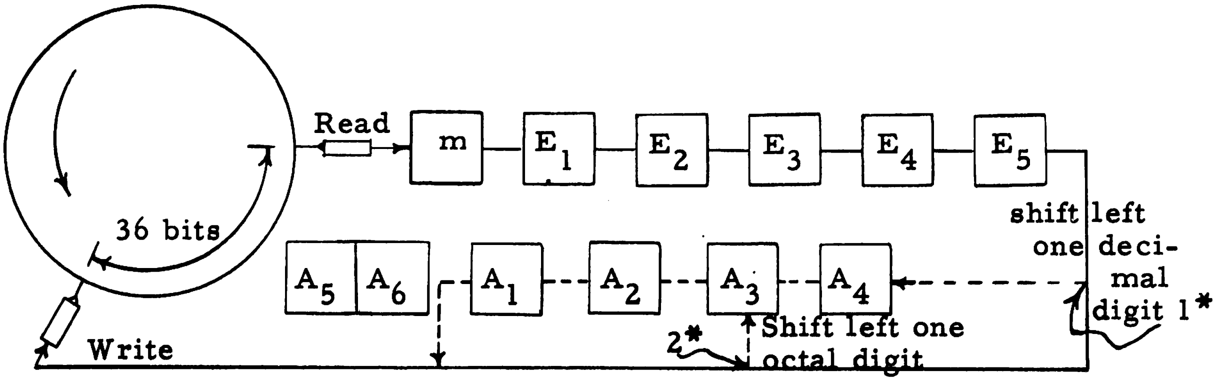
E register



Consider the drum to be rotating in the direction indicated by the arrow. At the beginning of a word time consider the 42 bits recorded as indicated. During the first pulse time P_0 O_0 is written on the drum, P_0 O_2 is read off, and all other bits shifted accordingly. At the end of one word time, or 42 pulse times, the word will be recirculated and P_0 O_0 will be located again in E_5, P_1 O_0 in E_4, etc.

The following schematic illustrates how the A flip-flops are injected into E during the fill process.

E Register



A_5 - Distinguishes between control information and data.

A_6 - Distinguishes between the decimal and octal information. When the "d" or "o" key is struck the nature of the A_1 - A_5 f-f set up will set A_6 to "0" or "1," for octal or decimal fill, respectively. In turn, the connections of the A_1 - A_4 f-f's into the E register would be as indicated.

The following facts illustrate the function of the A flip-flops for the fill process:

1. Striking a Flexowriter key will always set up A_1 - A_5 according to Table 1a. Prior to this, the machine is at rest; that is, the drum is rotating, all registers are recirculating, or, so to speak, the machine is just waiting for a Flexowriter key to be struck and begin the fill process.

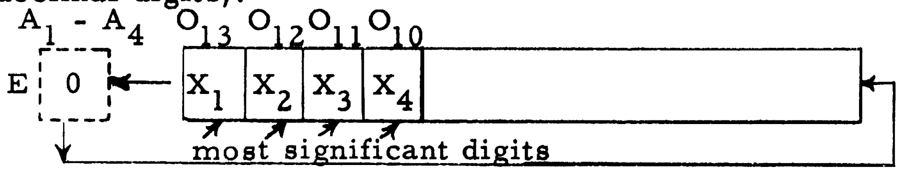
*1 - Connection into E register for filling decimal information.
*2 - Connection into E register for filling octal information.

2. The 102-A first inspects the A_5 f-f in order to determine the disposition of the binary configuration set up in A_1 - A_4. If A_5 is a "1," the various control operations defined by the A_1 - A_4 set-ups, indicated in Table 1a, will be executed (details of how the machine interprets these will not be included here). If A_5 is a "0," the computer treats the contents of A_1 - A_4 as a numeral assigned to enter the E register. However, in order to enter a digit in E it would have been necessary for the "d" or "o" key to have been struck first. That is, the A_6 would then be properly set, and as a result the proper connection of A_1 - A_4 into the E line would take place (by-pass A_4 for octal fill).

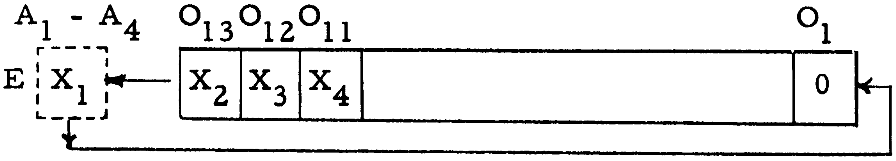
3. The injection of a digit into the E register will result when the E line circulates for one word time with the additional three or four bit positions (A_1 - A_3 for octal, or A_1 - A_4 for decimal). Referring to the above schematic, it is obvious that after 42 pulse times the bit in A_1 would have been written on the drum, read off the drum, and finally terminate in E_5; also, A_2 in E_4, etc. This procedure of extending the E register to 45 (or 46) bits will cause the most significant digit (octal or decimal) previously contained in E to terminate in A_1 - A_4. Hence, if another Flexowriter key is struck, A_1 - A_4 would be reset and the previous digit would be lost. It is in this manner that we fill the E register, one digit (octal or decimal) at a time, continually shifting the previously entered digit to the left, and losing the most significant digit of E. Thus, we are able to make corrections during the fill process by merely typing in the word again.

4. Emphasis will now be placed on the method whereby the machine interprets the "f" character, which supposedly enters either four octal or four decimal zeros in E. When the "f" key is struck (see Table 1a), the control bit in A_5 instructs the machine to circulate the E line, which now contains zeros in A_1 - A_4, four times. This process will enter four octal or four decimal zeros in the E register only if the three most significant octal or decimal digits in E are zeros initially. The following schematics will perhaps clarify this feature.

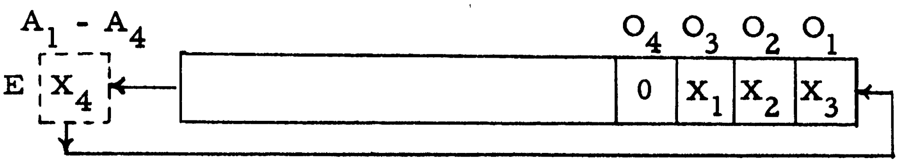
   Set-up when "f" is struck (X_i denotes arbitrary octal or decimal digits):



   Result after E circulates once:



   Result after E circulates four times.



   Also, it is apparent now that successive striking of the "f" key will not serve to clear the E register; it will merely recirculate its contents, injecting a zero every fourth digit position.

5. Reference is now made to the "8," "9," "+," "-," "space bar," and "period" Flexowriter characters in Table la. Since only A_1 - A_3 are incorporated into the E register when the octal mode is chosen, it is obvious that when these characters are struck the octal numbers 0, 1, 0, 2, 4, and 7, respectively, will fill E.

It is noteworthy that the computer must be in the "rest" state in order that the fill process take place. For this reason it is possible to strike a Flexowriter character while the machine is in the "computing" stage and not have it enter the memory. This is an automatic precaution against accidentally striking a Flexowriter character during computation and causing invalid results.