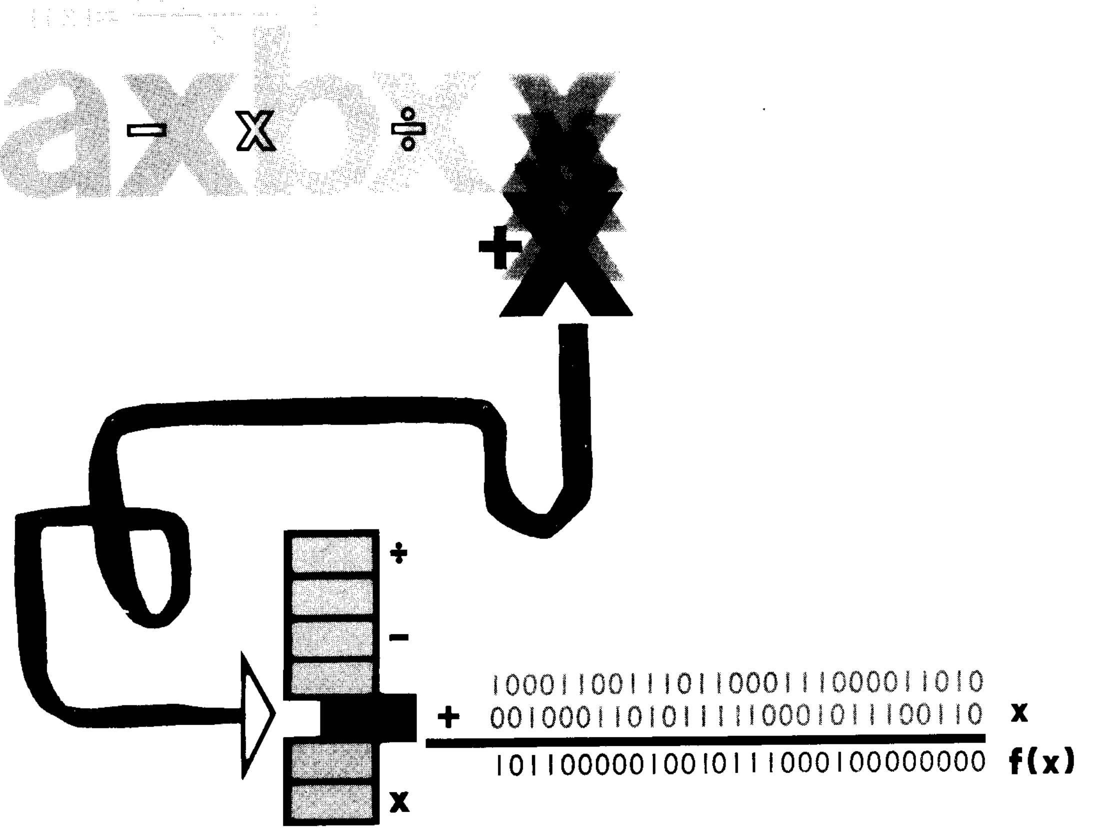
THE Bendix CORPORATION • COMPUTER DIVISION
5630 Arbor Vitae Street, Los Angeles 45, California

PROGRAMMING FOR THE G-15



FOREWORD

This manual is designed to teach the reader to program the G-15 computer in machine language. It was written in essentially two sections: the first, from the standpoint of an available assembly routine, discusses coding in a simplified form and operations on decimal numbers only; the second, from the standpoint of the machine itself, discusses coding and numbers in binary form.

It is assumed that the reader has previously read the introduction to the G-15, entitled "Bits of Meaning," and therefore is familiar with the decimal, binary and sexadecimal number systems.

Many of the topics covered in this manual are of an advanced nature. There are other complete programming systems available from Bendix which are not discussed in this manual. These systems are quickly learned and are easy to use.

Copyright 1960
Bendix Computer Division
The Bendix Corporation

MINOR REVISION
(July 1961)

This edition, APR-01601-1, July 1961, supersedes the January 1960 edition. The changes are:

Page	
14	First paragraph
17	The s key
45	Extract command, L T N C 31 D, first line
46	Other Extract command, L T N C 30 D, middle of page
52	AR format
57	Extract commands
97	First paragraph
130	New illustration
140	Typewriter Output, AR format
154	Location 88, c = (23.00)
158	Locations 11 through 50

TABLE OF CONTENTS

General Information	1
Problem Analysis	3
Method of Solution	3
Drum Memory and Addresses	5
Commands	13
Arithmetic Operations	20
Subroutines	27
Decimal Scaling	32
Multiply	35
Divide	39
Logical Operations	42
Shift	42
Extract	44
Repetitive Processing of Data - Loops	47
Input/Output System	51
Commands in Binary Form	60
Immediate vs. Deferred Commands	68
Sequencing of Commands	69
Command Lines	69
Special Commands	70
Multiplication and the Two-Word Registers	70
Division and the Two-Word Registers	76
Machine Form of a Number and Scaling	89
The Need to Automatically Check Computations	104
Test Commands	105
Test for Overflow	106
Test for Sign of AR (neg.)	108
Test for "Ready"	108
Test for Punch Switch On	109
Test for Non-0	109
To Subtract a Magnitude	110
Subroutines	113
Inputs/Outputs	127
Normal Inputs	128
Typewriter Inputs	128

TABLE OF CONTENTS (Cont'd.)

Enable Actions	133
Punched Tape Output and Output Format	134
Typewriter Output	140
Debugging	141
Break-Point	141
Single-Cycle	142
Input/Output Commands	142
Blank Leader	145
Loader Program	147
Program Preparation Routine (PPR)	150
Precession, As Used by PPR	163
Other PPR Operations Available	165
Decimal Number Inputs and Scaling	167
Extract, And Its Use in Number Conversion	177
Other Programming Techniques	186
Indexing	191
Floating-Point Operation	201
Miscellaneous	203
Index	209

PROGRAMMING THE G-15 IN MACHINE LANGUAGE

In the Introduction to the G-15, a separate booklet, we briefly traced the development of a growing need for computers to solve two types of problems:

1. problems whose solutions are essentially simple in nature, but which must be solved over and over again, each time for a different set of values; and

2. problems whose solutions are so complicated that men cannot spare the time and effort to solve them by the pencil-and-paper method.

In the first case, a program* can be written once to generate the solution, and then it can be operated again and again in the computer, each time with a different set of inputs, and each time yielding a new output. In this way literally hundreds or thousands of individual solutions to the same problem can be "cranked out" by the computer in the time required for a man with pencil and paper to generate a single solution. The cost of the computer is justified because it is less than the cost of the man-hours needed to do the same thing without the computer. In the second case, the programmer can decide what operations need to be performed, write a program for the computer, directing it to perform them in the proper sequence, and operate the program in the computer, feeding it the necessary original inputs. The speed at which the computer can perform these operations makes it possible to generate a solution in a reasonable period of time, whereas, with pencil and paper, so much time would be consumed in performing individual operations that the end solution would be years away. Just making a solution possible soon enough to be of some value justifies the computer's cost.

We saw that there are essentially two types of computers: analog and digital. The fact that precision is easier come by in a digital computer accounts for the increasing demand for them in certain applications where accuracy is of primary importance. The G-15 fills the common need for a medium-priced digital computer.

The G-15, like all other digital computers, is composed of five major sections:

1. input,

2. memory,

3. control,

4. arithmetic, and

5. output.

Numbers are stored in memory in binary form, and the computer works in binary. Each word of memory can contain, in its 29 bits, either a data number or an instruction in number form, referred to as a command. Whether a number is treated as data or as a command depends on the time during which it is inspected. There are three categories of machine-time:

1. read command time (RC),

2. execute time (EX), and

3. wait time (WT).

Because each command contains the address of the operand, there may be wait time before the operand is available. Similarly, because each command contains the address of the next command to be read and obeyed, there may be wait time before that command is available. Wait time may thus be further subdivided into two categories:

1. wait to execute (WTE), which will follow the reading of a command, and

2. wait to read (WTR), which will follow the execution of the previous command.

It is the programmer's duty, among other things, to minimize this wait time.

Commands contain the following basic parts:

1. a code for the desired operation or transfer,

2. address of operand,

3. address to which operand is to be transferred, and

4. address of the next command in the logical sequence of the program.

The reason an address is given, to which the operand is to be transferred, is that numbers may be moved about in memory under control of the program, without any arithmetic operations being performed on them. In many digital computers, this cannot be done directly; every number must go to the accumulator or from the accumulator to memory.

Words within the G-15 contain 29 bits. A data number is contained within one 29-bit word, having 28 bits of magnitude and a sign-bit.

We pointed out that a knowledge of the machine's operations on numbers as they appear to it will be important to programmers. Therefore, the binary number system was discussed, as was binary arithmetic. Methods for converting binary numbers to their decimal equivalents, and vice

versa, were pointed out. The possibility of overflow resulting from an addition in the computer was brought up. This is the condition that arises when an erroneous value results from an attempt to generate a value too great for the computer to hold. It was also mentioned that the computer must complement negative numbers prior to adding them to other numbers, and that, in such a case, the result must be recomplemented, if negative, in order to restore it to the normal form of a signed magnitude. In the addition of negative numbers, the end-around-carry feature was described. It was pointed out that subtraction is merely the addition of a number after changing its sign, and that the computer subtracts by doing exactly this.

The need for a "short-cut" number system was brought up, since bit-chasing is tedious when each number has 29 bits. The system adopted was the hex number system, because of the ease in converting back and forth between it and the binary system.

We then discussed briefly the duties of a programmer.

PROBLEM ANALYSIS

In order to use the computer to solve any problem, the programmer must devise a general, logical method of arriving at the solution. The first step in this is analysis of the problem itself. What is called for? Take, for example, the problem of finding the roots of a quadratic equation,

$$ax^2 + bx + c = 0.$$

Two values of x are called for, either of which, when substituted for x, will satisfy the equation. This is a very simple problem; usually the problem presented to a programmer will not be so clear-cut. He might be asked, for example, to choose the best route for a road through a mountain range, both from the standpoint of construction and from the standpoint of usage. In this case, defining exactly what it is that is called for is not so simple a task.

He will then have to find out as much as he can about the inputs for the problem: the number of them, the range in values, from great to small, the degree of accuracy that will be available. From this information he will have to deduce the best degree of accuracy to maintain throughout the solution of the problem. If there are too many inputs to be stored all at one time in memory, he will have to write his program to work in sections, calling for only a portion of the inputs at any one time. In the example of the quadratic equation, only three inputs are necessary: a, b, and c. Storage space will not be a problem in this case.

METHOD OF SOLUTION

When the programmer has adequately defined the problem and the data available, he must choose a method of solution. Usually there will be as many of these as there are programmers. Whether or not one

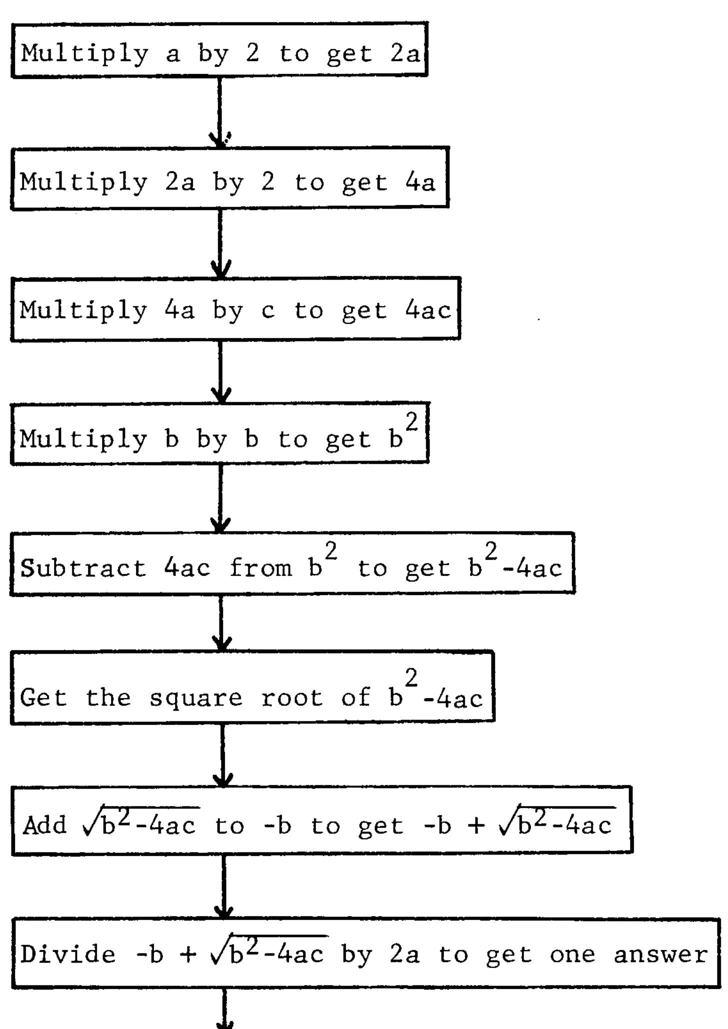
method is better than another depends on several factors. Is the time consumed in generating solutions an important factor? If they are to be used to control aircraft, it is. If they are to be used to file income-tax returns, it probably isn't. Time will probably be of little importance in the generation of solutions for the quadratic equation. If the program is going to be a long one, some choices of approach to the problem might significantly reduce the length of the program itself, saving the programmer work. The method of solution for the roots of a quadratic equation is pretty well standardized.

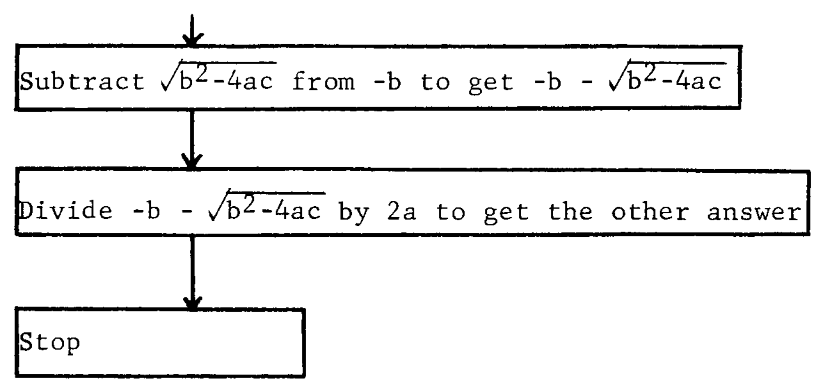
$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

FLOW DIAGRAM

When a method of solution has been determined, it must be outlined, one major step at a time, since this is the manner in which the computer operates. Each arithmetic process should be shown.

An outline of the logical pattern, or "flow", of the method to be used is called a "flow diagram". A flow diagram for the solution of the roots for the quadratic equation might be:



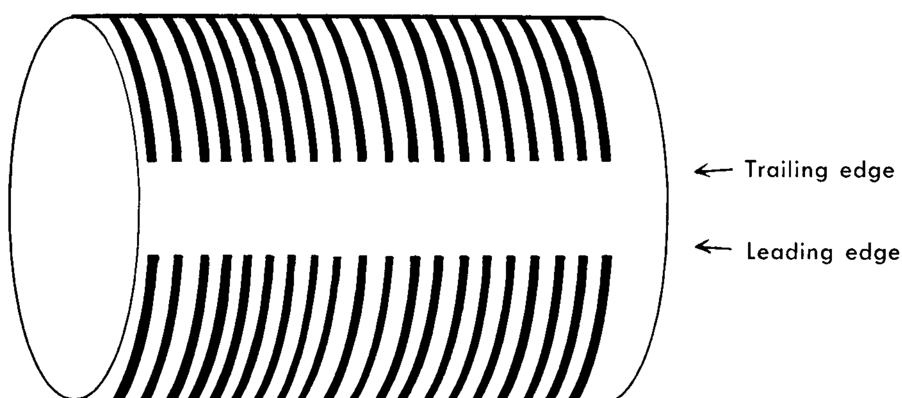


Remember, the above is merely a break-down, step by step, of the method of solution of the problem. Not shown in the method of solution, but nevertheless essential to the program itself, will be a provision for the input of the data (in this case, a, b, and c), and the output of the answers. You have noticed that this problem calls for additions, subtractions, multiplications, divisions, and taking the square root of a number.

Before we go any further into the development of the program we must explore the operation of the computer, with an eye toward making up the proper commands to achieve a desired result.

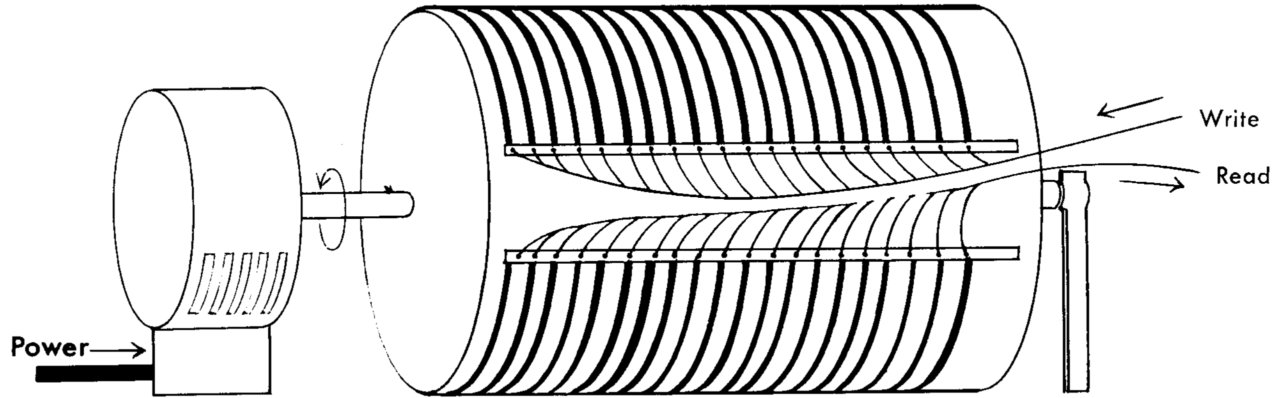
DRUM MEMORY AND ADDRESSES

It has been pointed out that each word in the memory of the G-15 is 29 bits in length. Now picture 108 words laid out in a long line, end-to-end, (29 x 108 =) 3132 bits in length. Find a cylinder (any old cylinder will do), with a circumference somewhat greater than the length of this long line, and wrap the line around it. Do this twenty times, so that the cylinder has twenty long lines around it. Make their leading and trailing edges line up. Leave some unused space for more long lines (for various special purposes).



You will now have something similar to that shown above. Notice the unused gap running the length of the cylinder on its circumference between the leading and trailing edges of the long lines (don't do anything with it yet, just notice it). Mount the cylinder on an axle, attach this to a motor, and supply some power. The cylinder will re-

volve. Mount a device length-wise over the cylinder barely raised away from its surface, and use this device to look at the bits in the long lines as they pass beneath it. Mount a similar device to write bits into the long lines.



Instead of using a paper cylinder and writing with a pencil, use a metal cylinder coated with a magnetic coating, and write with electrical pulses, magnetizing individual spots beneath the "write-heads" (to represent 1) and leaving other spots unmagnetized (to indicate 0). Read the magnetized spots with "read-heads", and generate the corresponding electrical pulses. Attach some complicated circuitry, including an input and an output system, put an attractive case around the whole works, and you have the Bendix G-15 digital computer.

We call the cylinder a "drum", and we speak of "drum memory". In the long lines of memory, there are (20 x 108 =) 2160 words. Each bit of each of these words is available at the read-heads once per drum revolution; at each bit-time, 20 bits are available, one out of each long line. The reason we use "T" numbers when numbering bits in a word, as shown in the drawing below, is that the numbering is in the order in which the bits will be inspected (T1 first, then T2 through T29, T standing for time). Thus, the sign of a word will be inspected before any of the magnitude bits. Similarly, each bit in each long line is available to be written into once per drum revolution; at each bit-time 20 bits are available, one out of each long line. The sign (T1) will be written first, then the magnitude bits in the order in which they are numbered (T2 through T29).

T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

The speed at which the drum revolves is 1800 rpm, or 30 rps. If the drum revolves once per 1/30th of a second, and, if there are 108 words per long line, the time required for the reading or writing of one complete word should be:

$$\frac{1}{108} \cdot \frac{1}{30} = \frac{1}{3240} \text{ sec.} = .00031 \text{ sec.}$$

Within the next few pages, you will see that actually a complete word can be read or written in slightly less time than this.

In computer operation, addresses of words in memory are of the utmost importance. Now that the 20 long lines of memory have been described, you can see that the location of a specific word is composed of two parts:

Part I: a designation of the line in which the word is located, and

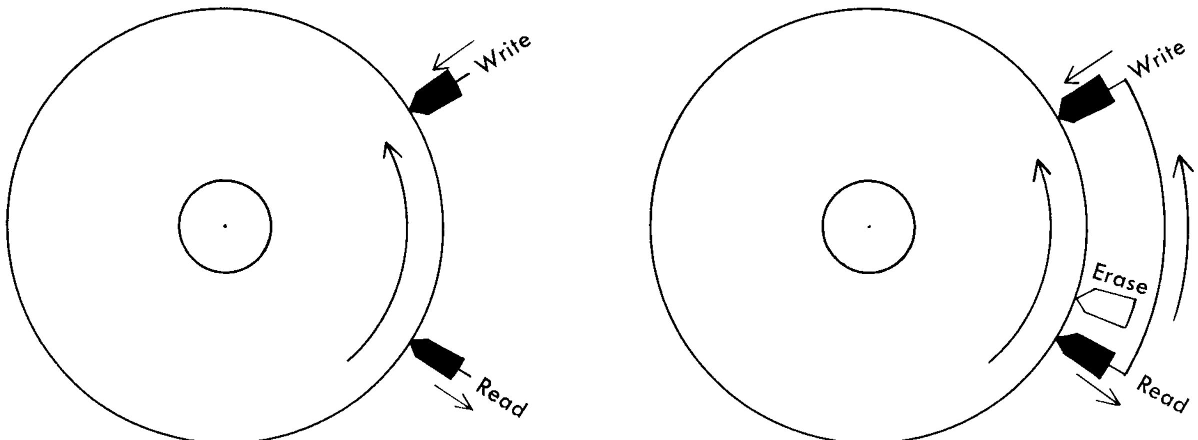
Part II: a designation of the location of the word within that line.

The 20 long lines are numbered 00 through 19, and the words in each one, starting at the leading edge, are numbered 00 through 99, u0 (100) through u7 (107). (The substitution of the hex digit u for the decimal digits 10 is for the purpose of holding the word-number to two digits.) The addresses of all words in long lines are:

line	:	word
00	:	00
	↓	
00	:	u7
01	:	00
	↓	
01	:	u7
	↓	
19	:	00
	↓	
19	:	u7

Before looking at addresses, as they appear in commands in the computer, we must fill out the rest of the picture of memory, since every location in "working" memory (that part of memory available to programmers, as opposed to the remaining part, which is essentially of engineering importance only) is addressable in the same manner as shown above.

The drawing to the left shows a cross-section of the drum, as already described. It is now time to add to the previous description. The drawing to the right shows an erase-head immediately following the

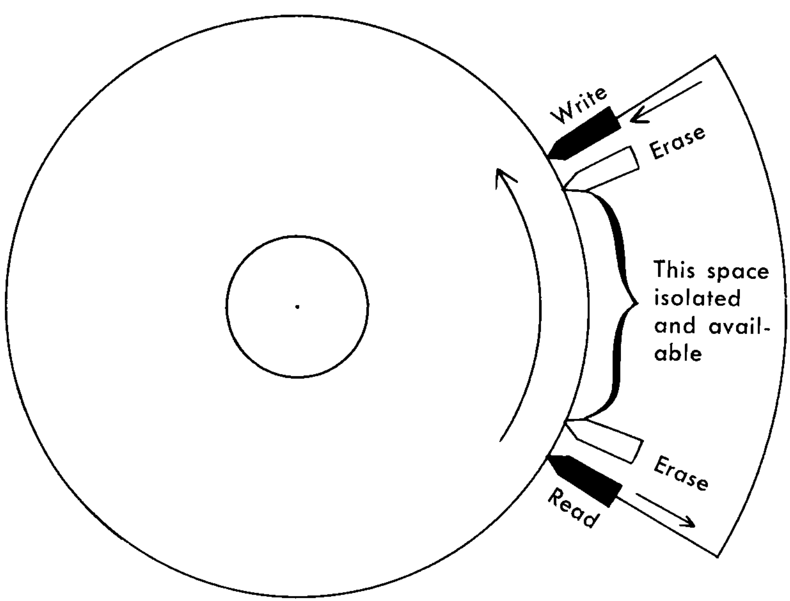


read-head; this is true for each long line in memory. The read-head feeds the write-head with electrical pulses mirroring the contents of the bit-positions as they pass under the read-head. Each bit in each line, as it is read, is moved ahead, along the circumference of the drum. Of course the trailing edge of each long line is moving along at the same speed as the leading edge, so each bit so written will be placed in a vacated bit-position on the drum. The "clear", or erased, state of the drum is 0's, so only 1's are written, when called for. The result is that 1's appear where they should, and all other bits are equal to 0. This is appropriately called a "recirculating" memory. Note that any specific bit in any specific word will not occupy the same physical position on the drum during each revolution. Because it is stepped ahead along the circumference, it actually will be available slightly more than once per drum revolution. An entire long line is inspected and recirculated in slightly less than one drum revolution. The length of time necessary for the complete inspection and recirculation of a long line is referred to as a "drum cycle".

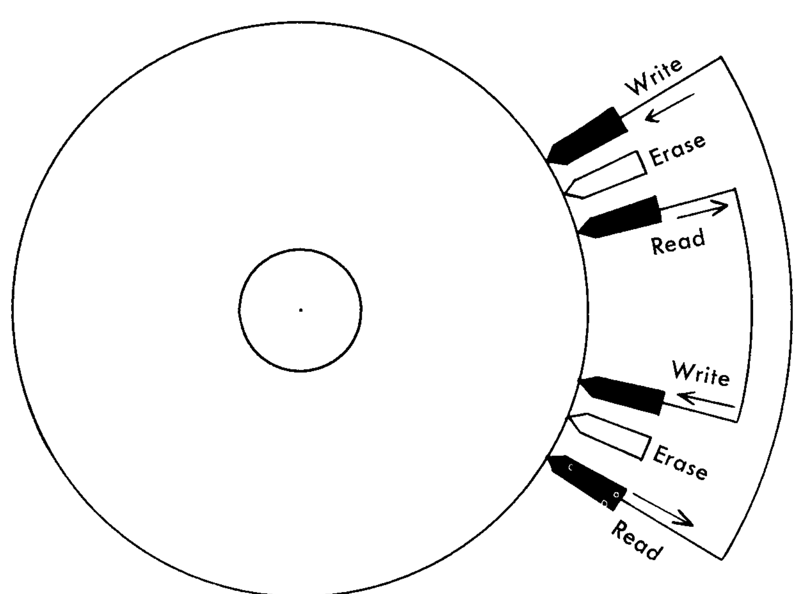
The spacing between the read- and the write-heads is such that there are approximately 2070 drum cycles per minute, or 34.5 per second. Each drum cycle requires .029 seconds. Each word is read or written in .00027 seconds. We refer to $\frac{1}{1000}$th's of a second as "milliseconds". Therefore,

1 word-time = .27 milliseconds (ms.),
1 drum-cycle = 29 milliseconds (ms.).

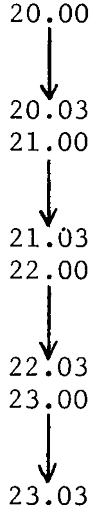
It seems there is an unused strip along the drum, between the long line erase- and write-heads. If a second erase-head is placed before the long line write-head, it will merely duplicate the effect of the other erase-head. But now, anything can go on between the two of them, and there will be no injurious carry-over into the long line; in it, 0's and 1's will be where they should be.



If, as shown in the drawing below, read- and write-heads are placed in the isolated gap on the circumference of the drum, a recirculating short line will be created. The shortness of the short line will be determined by the proximity of one head to the other. As far as storage capacity of the computer's memory is concerned, there is no advantage to this system; it would be more economical to increase the length of each long line. But remember that it was pointed out that any given bit, and therefore, any word, in a long line is available only once per drum cycle, or once per 108 word-times. If a short line contains four words, each bit, and therefore each word, will be available once per four word-times. Or, to put it another way, each word in a 4-word short line is available (108 ÷ 4 =) 27 times per drum cycle.



A consideration of timing within the computer will demonstrate the value of 4-word short lines. It has been stated previously that every word, whether it be data or a command, has a unique address. It can be seen that the word "address" as used here denotes more than mere location in space; it also denotes a location in time. At a given word-time (specified in its address), a given word will be available at the read-head. Suppose this word is a command, and the computer is in RC (read command) time. This command will be read and interpreted at the word-time specified in its address. It, in turn, calls for a data word, located at another address, and, therefore, at another word-time. If no care had been used originally in picking addresses for commands and data, this command would call for a data word which would be, on the average, 1/2 drum cycle away. The time a computer consumes in searching for a specified word is called "access-time". It is the programmer's job, among other things, to minimize this dead time by wisely selecting the addresses for commands and data when he is writing a program for the G-15. A well-written program, from this standpoint, and all other things being equal, will operate in the computer much more rapidly than will a poorly written one. The average access-time for any word in a 4-word short line is only two word-times. Availability of these short lines makes the programmer's job easier and provides for faster program operation than would otherwise be possible. The gaps between the leading and trailing edges of four of the long lines are used for short lines of this nature. These short lines are numbered 20 through 23, and the complete addresses of the words in them are:



Three more of the gaps are used for 2-word short lines, referred to as "2-word registers". These are also available for storage, although they have special circuitry associated with them which enables them to be used for certain operations of arithmetic, as well. These are numbered 24 through 26, and the complete addresses of the words they contain are:

24.00
24.01
25.00
25.01
26.00
26.01

Line 24 is called the "MQ" register; the two words in it are called "MQ_0" and "MQ_1". It derives this name from the fact that it holds the Multiplier prior to a multiplication and Quotient following a division.

Line 25 is called the "ID" register; the two words in it are called "ID_0" and "ID_1". It derives this name from the fact that it holds the multiplIcand prior to a multiplication and the Denominator prior to a division.

Line 26 is called the "PN" register; the two words in it are called "PN_0" and "PN_1". It derives this name from the fact that it holds the Product following a multiplication and the Numerator prior to a division.

It can be seen, then, that multiplication and division, when called for by the proper commands, will involve all three of the two-word registers.

Another gap is occupied by a 1-word short line, referred to as "AR". The number of this line may be either 28 or 29. This line has circuitry associated with it making it a 1-word accumulator; if it is referred to as line 28, this circuitry is not employed, and it behaves the same as any other word in memory (in such a capacity it is very convenient, of course, because it is available at every word-time); if it is referred to as line 29, the special circuitry is employed, and it will combine binary numbers, as discussed in the Introduction to the G-15.

For the programmer whose application of the computer requires more accuracy than can be carried in 29 bits, "double-precision" arithmetic is possible within the G-15. It is no harder to program using it than it is to program ordinary single-precision arithmetic. In double-precision operations, two computer words are used to express each data number. These two words must be contiguous in the same line, the first in an even location, the second in the following odd location. It has been pointed out (page 12 of the Introduction) that, in single-precision operation, each word starts with sign-time. This is true in double-precision, as well, except that sign-time occurs only during even-numbered word-times. The remaining 28 bits in the even-numbered word contain the least significant information in the number, and all 29 bits of the odd-numbered word are used to complete the magnitude. So a double-precision number actually has slightly more than double the precision of a single word, since it has 57 bits of magnitude, as opposed to 28. All operations which can be specified by commands to affect single words can be very easily modified to similarly affect double-precision numbers.

For the multiply and divide operations, the two-word registers which are used for single-precision arithmetic will also suffice for double-precision arithmetic. But in the cases of addition and subtraction, AR, the one-word line used for single-precision arithmetic, is obviously not capable of performing double-precision operations. PN (line 26) is used for this purpose: it is the double-precision accumulator, in addition to its other functions. If it is being used for this purpose, it is referred to as line 30.

Line numbers 27 and 31 are also legal, but are actually special codes, not referring to existing lines in the G-15 memory. They will be discussed later.

The remaining available gaps between the leading and trailing edges of the long lines along the length of the drum are used for engineering purposes, and are not available to the programmer.

The remaining available space on the circumference of the drum for additional long lines is used for timing and control information, mostly from an engineering standpoint. One of these long lines is of interest to programmers, however. It is called the "number track".

The number track is a long line, divided into bits and words, similar to any other long line. But each word in it contains, rather than data or a command, timing information which affixes a word-number, ranging from 00 through u7, to each similarly located word in each long line. This number track is recirculated in the same manner, about the circumference of the drum, as are the long lines. In word u7 of the number track, there is a special indicator which signifies that the next word is the beginning of the line, word 00. The pulse which is generated by this indicator, when it is read, is referred to as "TO". Thus the beginning, and each succeeding, word number in each long line is fixed and remains constant. This, of course, is essential for addressing words. The short lines are so situated that word 00 in each of them will occur simultaneously with word 00 in each long line. Notice that, for the 4-word short lines, word 00 will also arise concurrently with the following words in the long lines: 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88, 92, 96, u0, and u4. You might want to write for yourself a similar list of long line locations corresponding to words 01, 02, and 03 in the short lines. We knew that such would be the case as soon as we said that the short lines recirculate 27 times per each recirculation of the long lines. The 2-word registers recirculate 54 times per long line recirculation, and they are so situated that word 00 in them occurs at every even word-time of the long lines (00 is considered even), and word 01 in them occurs at every odd word-time of the long lines. AR, the one-word register, occurs at every word-time, since it only requires one word-time for its recirculation.

It is of the utmost importance to the programmer to know that the number track is correctly on the drum before he attempts to operate any program in the computer. If the number track is not correct, the addresses associated with the words in memory will not be correct or

constant. Therefore, a command is available to the programmer, which enables him to inspect the number track. This command will be discussed later.

COMMANDS

Although commands, as well as data, are in binary form when stored in the computer, we need not worry about the actual 29 bits that make up a command. A program was written by Bendix personnel which can accept commands in a simplified form and translate them into the binary language of the computer. No flexibility in the operation of the computer is lost in this translation. This Bendix program is called PPR (Program Preparation Routine), and is made available to every user of the G-15 computer.

A command for the G-15 must specify the following information:

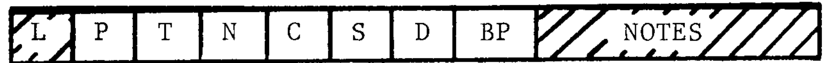
1. desired operation,

2. address of operand,

3. address to which operand is to be transferred, and

4. address of next command to be obeyed.

In addition to this information, a command may contain information relating to the duration of its execution.



The desired operation is specified by a decimal digit ranging from 0 through 7 in the C portion of the command.

0 - Calls for a straight transfer of a single-precision operand from one location to another. After this transfer has been performed, the operand, in its original form, will be in both locations in memory. This is sometimes called a "copy".

1 - Calls for use of "inverting gates" during the transfer of the operand from one location to another. The inverting gates will complement negative numbers passing through them in the manner described in the Introduction to the G-15.

2 - Depends, for its meaning, on the address of the operand and the receiving location.

If both of these addresses refer to memory lines whose numbers are less than 28, this C code calls for an exchange of AR, which is the single-precision accumulator, and memory, in the following way: the original contents of AR are copied into the specified receiving address, and the operand is copied into AR. If this exchange is called for at an even word-time, and if the receiving address is a two-word register, AR's original

contents will be blocked from entering the even half of the two-word register, and that half of the two-word register will be cleared to 0 instead. AR's original contents will be lost. The number in AR will be the absolute value of the operand, i.e., T1 = 0, and the sign of the operand will be held in a special flip-flop called "IP".

If AR is specified as either the operand or the receiving address, or if PN, as line 30, is specified as the receiving address, the absolute value of the operand (a positive number) will be transferred to the receiving address.

3 - Also depends on the specified address of the operand and the receiving location for its meaning.

If both of these addresses contain line numbers less than 28, an exchange of AR with memory, similar to that described above, is performed. In this case, however, the operand, on its way to AR, will pass through the inverting gates and be complemented if negative. If this exchange is called for at an even word-time, and if the receiving address is a two-word register, AR's original contents will be blocked from entering the even half of the two-word register, and that half of the two-word register will be cleared to 0 instead. AR's original contents will be lost.

If AR is specified, either as the operand or the receiving address, or if PN, as line 30, is specified as the receiving address, the sign of the operand will be changed during the transfer, and then the operand, with its new sign, will pass through the inverting gates. This is, in effect, a "subtract" command.

4 - Calls for a "copy" of a double-precision number from one memory location to another, being the double-precision equivalent of 0.

5 - Calls for use of inverting gates during the transfer of a double-precision operand, being the double-precision equivalent of 1.

6 - Depends on the specified address of the operand and the receiving location for its meaning.

If both of these addresses contain line numbers less than 28, this C code calls for an exchange of AR with memory for two word-times, each exchange being similar to that called for by a 2, under the same conditions. During the first word-time of execution (even), AR's original contents are copied to the first half of the receiving address, and the first half of the operand is transferred to AR. During the second word-time of execution (odd), AR's contents (now the first half of the double-precision operand) are transferred to the second half of the receiving address, and the second half of the operand is transferred to AR. If the receiving

address is a two-word register, during the first word-time of execution AR's original contents will be blocked from entering the even half of that two-word register, and that half of the two-word register will be cleared to 0 instead. AR's original contents will be lost.

If either the operand or the receiving address contains a line number greater than or equal to 28, the absolute value of the double-precision operand will be transferred.

7 - Depends on the specified address of the operand and the receiving location for its meaning.

If these both contain line numbers less than 28, a double-precision exchange will be performed, in the manner described above, for a C of 6, with the exception that all numbers entering AR will pass through the inverting gates and be complemented if necessary. If the receiving address is a two-word register, during the first word-time of execution AR's original contents will be blocked from entering the even half of the two-word register, and that half of the two-word register will be cleared to 0 instead. AR's original contents will be lost.

If either the operand or the receiving address refers to a line whose number is greater than or equal to 28, the sign of the double-precision operand will be changed during the transfer, and then the double-precision operand, with its new sign, will pass through the inverting gates and be complemented if necessary. This is, in effect, a double-precision "subtract".

The line in which the operand is located is called the "source", and, in the layout of a command, the two-digit decimal number of this line, ranging from 00 through 31, is referred to as "S". AR, as a source, must always be referred to as line 28. PN, as a source, must always be referred to as line 26.

The line which contains the receiving location is referred to as the "destination", and, in the layout of a command, the two-digit decimal number of this line, ranging from 00 through 31, is referred to as "D".

The address of the operand is completed by specification of a word-time. A two-digit number, ranging from 00 through u7, in the "T" portion of a command, specifies this word-time. This same T number is combined with D, in order to complete the receiving address. So we see that, if a word is copied from one long line to another, the word being transferred will occupy the same word-time in both lines.

The type of operation we have been discussing so far is referred to as "deferred" operation. No matter when (what word-time) the command itself is read and interpreted, the computer will wait, or defer action, until the word-time specified for the operation arises. There is

another type of operation called "immediate", in which the operation called for by the C code may be performed continuously for any number of word-times on S and D, up to 108 (a whole drum cycle). In this type of operation, the transfer called for will start immediately, in the word-time following that in which the command itself was read, and it will continue through continuous word-times, until a "flag" is reached. This flag will be a word-time specified as T in the command, and the execution will cease with the word-time immediately preceding the flag.

If immediate execution, rather than deferred, is desired, a one-digit prefix must be placed in the "P" portion of the command. This digit must be a "u". If no prefix is desired, this portion of the command should be left blank.

Each command contains within it the address of the next command to be obeyed, and this is why the computer can perform a sequence of commands of any length automatically, after once being told where to start. The word-time of the next command is entered as a two-digit number, ranging from 00 through u7, in the "N" portion of a command. This address contains no line number, because once the computer has started to obey a sequence of commands from one of the memory lines, it continues to look in the same line for the next command in the sequence.

The computer can follow a sequence of commands in either of two modes; continuous operation or break-point operation. Ordinarily the computer will be in the continuous mode, but the computer operator can, at any time, cause the computer to switch to the break-point mode through an external switch action. When in the continuous mode, the computer can only be stopped, under program control, through execution of a special command, called the halt command. When in the break-point mode, however, the computer can be stopped, under program control, after execution of a specially marked command, as well as by the halt command. Any command may be so marked, and this is done through insertion of a minus (-), in the "BP" portion of the command. If no break-point mark is desired in the command, this portion of the command should be left blank.

Shown in the layout of a command are two shaded portions: "L" and "NOTES". From experience, programmers of the G-15 have found it desirable to include, with each command, as it is written on a coding sheet, the word-time in which the command is located and some brief note explaining the function of the command. This information, although on the coding sheet, is not entered into the computer as part of the command.

If D = 31 in a command, the computer will treat this command as a "special" command, and interpret it in a special way. The S number will be treated as a special operation code, and the C number will usually be interpreted in the light of the special operation called for. Additions, subtractions, and copies of various types can be performed through any chosen combinations of the various portions of commands already discussed, but multiplications, divisions, and other

special operations are called for through use of special commands. We will discuss each special command as necessary, and they will be summarized on pages 56-59.

With a firm knowledge of:

1. the binary form of data within the machine, and

2. the format of machine commands,

we are ready to consider the various machine operations which can be combined to form a program.

Since many programs need data upon which to operate, usually one of the first things they do is to call for a computer input. The normal inputs to the G-15 computer are:

1. typewriter, and

2. punched paper tape.

The G-15 has an input/output system which only operates when commanded. There are two ways of commanding this system to operate:

1. special commands, under program control, and

2. special external switch actions, which the computer operator can take at will.

Initially, of course, when the computer is first turned on, there is no useful information in its memory. The question arises, therefore, how is a program initially entered into the memory of the computer, so that it can be operated later, calling in its own data upon which to operate? The answer is to make available some sort of external action for the computer operator to take, in order to activate the computer's input/output system. The external control console for the G-15 is an electric typewriter, connected by a cable to the computer. A picture of this typewriter is on page 130. Certain keys on the typewriter, namely q, r, t, i, p, a, s,* f, c, b, and m, can directly activate the computer in the ways indicated in the drawing, if the computer operator chooses to enable them to do so. He does this by moving the enable switch, mounted on the base of the typewriter, to the "ON" position. This switch should never be turned on until the compute switch, which controls the automatic operation of the computer in either of the two modes already discussed, is turned off. The "OFF" position for the compute switch is the center position.

The use of the keys already named, while the enable switch is on, is referred to as "enable action". Notice in the drawing that a "p" enable action will cause the computer's input/output system to read punched tape. From now on we will adopt the custom of underlining a letter in order to indicate the appropriate enable action; e.g., p.

* **The earlier model typewriter had no ⓢ key; throughout this manual, wherever the s key is indicated, use the ⓢ key.**

Given a punched paper tape containing the PPR program, you can mount this tape on the photo-reader of the computer, strike p, and you will see the photo-reader light turn on and the tape winding mechanism start to work, pulling the tape past the reader. One "block" of the tape will be read. A block of tape is a line's worth of information destined for the memory of the computer. When this initial block has been "read" into the computer, and the photo-reader light goes off, if you turn off the enable switch and turn the compute switch on to "GO", the commands now in the memory of the computer will be operated, and they have been written to call for the reading of four more blocks of punched tape. You will be able, of course, to see the photo-reader turn on again and four blocks of tape pass by it, at which point the basic portion of the PPR program will be in the memory of the computer. It will be occupying long lines 17, 16, 15, and 05. The initial block of tape, which was read in because of the p action, will no longer be in the memory of the computer.

With the compute switch still on "GO", PPR will operate. As a program operates, the neons on the front of the computer will flicker rapidly, as they reflect certain portions of each command being operated in the sequence of the program. A drawing of these neons is on page 208. Notice that there is a set of five neons for both S and D, and that each neon has a numerical value associated with it, the neons being arranged in the form of a binary number containing five bits. Through reading the lighted neons, you can determine the values for S and D of the command which has just been executed. Of course it will be impossible for you to read these neons as the program operates, because the computer is executing commands very rapidly. But when the computer stops, these neons will remain steady, showing the S and D of the last command executed. Below the S and D neons there is another set of five neons, which reflect the status of the input/output system. When no input or output is in progress, the "ready" neon, marked with an "R" will be on. If an input or output is in progress, this neon will be off, and some configuration of the other four will be lighted, showing the binary number associated with the input or output in progress. Each input and each output has a unique special number associated with it.

After PPR has been entered into the computer and is operating, the neons will eventually stop flickering, showing an S of 28, a D of 31, and an input or output called for with the unique number 12. 12 is the special number associated with a typewriter input. In PPR, a special command has been executed, and this command has told the computer to start a typewriter input. The special command for this is:

L L+2 N 0 12 31.

Notice that this is a special command (D = 31), and that the special operation code is 12, the number associated with the input called for. This is the case in all input/output commands. Special commands with D = 31 are always immediate. This can be overridden, and any special command can be made deferred through the insertion of a prefix, P = w. There is no such prefix in this command. It therefore will start

execution immediately, in L + 1, and this execution will continue up through the last word-time preceding the "flag" in T. This flag is L + 2; therefore the last word-time of execution will also be L + 1, and we have thus limited execution of this command to one word-time, L + 1. This is all that is necessary, since the input/output system can be properly activated in one word-time of execution.

When the input/output system has been activated through the execution of one of the appropriate special commands, the computer continues obeying commands in the normal sequence, taking the next command from location N. There is no interlock built into the G-15 to prevent computation during an input or output. If the program, in this case PPR, depends on the arrival of data in memory from the input called for, something must be done to prevent the computer from following the sequence of commands until the data has arrived. The programmer does this, when writing his program, through insertion, at a given point in the program, of a command designed to cause the computer to wait for the completion of the input before proceding to further commands in the sequence. This was done by the programmers who wrote PPR.

In order to understand how this can be done, you must first understand that the G-15, like most digital computers, can make simple decisions, based on the existence or non-existence of a given condition within the circuitry of the computer itself. The computer can be directed to interrogate any of several conditions through the use of certain special commands, called "test" commands. When the computer reads a command and finds that the command calls for a test, it performs that test during the specified word-time or word-times of execution. After the execution is complete, if the condition being tested for was not found to exist, in other words, the answer to the question asked was "no", the computer will take its next command from N. If on the other hand, the condition tested for was found to exist, in other words, the answer to the question asked was "yes", the computer will automatically take the next command from N + 1.

The special command which prevents the computer from continuing the sequence in a program until an input or an output is finished is a test command, called the "ready" test, which tests the input/output system for being ready. If the input/output system is ready, there is no input or output currently in progress. Thus, after an input or output has been called for, and the ready test is given, the test cannot be answered "yes" until the specified input or output is finished. In order to stop the computer from proceeding in the sequence of a program, the ready test is written in the following way:

L L L 0 28 31.

You can see, from inspection of this command, that, as long as the answer to the question is "no", the next command, being taken from N, will be the ready test itself. The only way this test can be prevented from repeating itself over and over again is for the input/output system to "go ready", making the answer to the question "yes", at which time

the next command will be taken from N + 1, which is in reality L + 1. The sequence of the program would resume at L + 1. Notice that it was said that the neons would remain steady at some point during the operation of PPR, with S = 28, D = 31, and the input/output neons indicating the special number 12.

At this point during the operation of PPR, the commands comprising any desired program can be typed in. PPR is also able to accept other inputs and operate on them at this time, but we will postpone a discussion of the various inputs to PPR until page 59, after you have some knowledge of the make-up of a program.

Only the computer operator will know when the typewriter input is finished because he will be doing the typing. When he is done, he strikes the "s" key in order to notify the computer that the input is finished. When he does this, the input/output neons will change, and only the "ready" neon will be lighted. The ready test in PPR will be answered "yes", and the program, in this case, PPR, will continue its normal sequence.

Now that we have some general knowledge (to be expanded later on) of the manner in which programs and data are entered into the computer, let's inspect the available methods for performing arithmetic and other operations on numbers under program control.

ARITHMETIC OPERATIONS

We will assume at this point that the numbers upon which we desire to perform these operations have already been entered into the proper memory locations of the computer.

Single-precision numbers are combined to form totals in the one-word short line called AR. If the destination during the transfer of a number is 28, the original contents of AR will be replaced with the number being transferred. If, however, the destination is 29, the number being transferred will be combined with the original contents of AR in whatever manner is prescribed by the C code in the command. Usually, when numbers contained in specified computer words are to be added or subtracted from each other in a program, we cannot predict, at the time we write the program, what the signs of these numbers will be. In such a case, it is, of course, necessary, in order to generate the proper sum or difference in AR, to transfer the numbers through the inverting gates on their way to AR. We would therefore use C codes of 1 for "add" and 3 for "subtract".

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	03	1	10	28		$10.02 \xrightarrow{+} AR_c$
03		04	05	1	10	29		$10.04 \xrightarrow{+} AR_+$
05		06	N	1	28	10		$AR \xrightarrow{+} 10.06$

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	03	1	10	28		$10.02 \xrightarrow{+} AR_c$
03		04	05	3	10	29		$10.04 \xrightarrow{-} AR_+$
05		06	N	1	28	10		$AR \xrightarrow{+} 10.06$

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	03	3	10	28		$10.02 \xrightarrow{-} AR_c$
03		04	05	1	10	29		$10.04 \xrightarrow{+} AR_+$
05		06	N	1	28	10		$AR \xrightarrow{+} 10.06$

If double-precision numbers are to be added or subtracted, the two-word short line, PN, which serves as a double-precision accumulator, is used as the destination for the transfer of the numbers. If D = 26, the original contents of PN are replaced by the number being transferred. If, however, D = 30, the number being transferred is combined with the original contents of PN in order to form the proper sum or difference, as called for by the C codes in the transfers. Here, of course, we would use the double-precision equivalents of 1 and 3 for C, 5 and 7 respectively. Notice that, although a single-precision number can be subtracted into a cleared accumulator, AR, with a C of 3 and a D of 28 (sometimes called "clear and subtract"), such is not the case with the double-precision accumulator, PN. In order to replace the original contents of PN with the number being transferred, D must equal 26. If D = 26 and the source line is any other line in memory (other than AR, of course), a C of 7 will be interpreted as calling for an exchange of AR with memory, because both S and D will be less than 28. Therefore in order to "clear and subtract" a double-precision number in PN, PN must first be cleared to zero and then the double-precision number subtracted using a C of 7. A special command is available, which will clear all of the two-word registers:

L   L+3   N   0   23   31.

Because D = 31, this command will be immediate. It will operate for two word-times, L + 1 and L + 2, during which it will cause 0's to be written into both halves of all three two-word registers.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	10	26		$10.02\text{-}03 \xrightarrow{+} PN_{0,1}$
04		06	08	5	10	30		$10.06\text{-}07 \xrightarrow{+} PN_{0,1+}$
08		10	N	5	26	10		$PN_{0,1} \xrightarrow{+} 10.10\text{-}11$

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	10	26		$10.02\text{-}03 \xrightarrow{+} PN_{0,1}$
04		06	08	7	10	30		$10.06\text{-}07 \xrightarrow{-} PN_{0,1+}$
08		10	N	5	26	10		$PN_{0,1} \xrightarrow{+} 10.10\text{-}11$

L	P	T or L_k	N	C	S	D	BP	NOTES
00		03	03	0	23	31		Clear 2-word registers
03		04	06	7	10	30		$10.04\text{-}05 \xrightarrow{-} PN_{0,1+}$
06		08	10	5	10	30		$10.08\text{-}09 \xrightarrow{+} PN_{0,1+}$
10		12	N	5	26	10		$PN_{0,1} \xrightarrow{+} 10.12\text{-}13$

The magnitude of a single-precision number can be added to a quantity in AR through the use of a C of 2, since AR, as the destination, will be line 29. The magnitude of a single-precision number can be placed in AR, replacing the original contents of AR, preparatory to adding something to it, through the use of the same C and a destination of 28. In either case the C of 2 will call for the transfer of the magnitude of the operand, because the destination is greater than or equal to 28. Similarly, the magnitude of an answer in AR can be transferred to some predetermined storage location in memory through the use of a C of 2, because the source in this command would be 28 (AR).

L	P	T or L_k	N	C	S	D	BP	N O T E S
00		01	02	1	10	28		$10.01 \xrightarrow{+} AR_c$
02		03	04	1	10	29		$10.03 \xrightarrow{+} AR_+$
04		05	N	2	28	10		$\lvert AR \rvert \longrightarrow 10.05$

L	P	T or L_k	N	C	S	D	BP	N O T E S
00		01	02	2	10	28		$\lvert 10.01 \rvert \longrightarrow AR_c$
02		03	04	1	10	29		$10.03 \xrightarrow{+} AR_+$
04		05	N	1	28	10		$AR \xrightarrow{+} 10.05$

L	P	T or L_k	N	C	S	D	BP	N O T E S
00		01	02	1	10	28		$10.01 \xrightarrow{+} AR_c$
02		03	04	2	10	29		$\lvert 10.03 \rvert \longrightarrow AR+$
04		05	N	1	28	10		$AR \xrightarrow{+} 10.05$

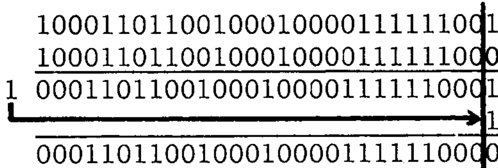
In order to generate a +0 in AR, we find the use of absolute values advantageous. You might guess that you could generate a +0 in AR by subtracting the contents of AR from AR, in a fashion similar to the one below:

L T N 3 28 29.

This method is fine if AR is originally positive, because the C of 3 will cause the sign of AR's contents to be changed, thus yielding a negative number, and then it will cause this negative number to pass through the inverting gates and be complemented. Because D = 29, this negative complement will be added to the original contents of AR, so that the sum generated in AR will be a positive number plus its negative complement. Any positive number plus its negative complement will yield +0 as a result.

```
  10001101100100010000111111000
  01110010011011101111000001001
1 00000000000000000000000000001
                              1
  00000000000000000000000000000
```

But, if AR originally contains a negative number, the C of 3 will cause the sign of this number to be changed to a positive sign, and thus the magnitude bits will pass through the inverting gates uncomplemented. The initial sum generated in AR, therefore will be negative, and its magnitude will be twice that of the original contents of AR; if, in the generation of the sum, an end-around-carry is generated, the final sign of the sum will be positive, but the magnitude, in most cases, will be unequal to 0.



Usually, when it is desired to generate a +0 in AR, you cannot predict, at the time you are writing your program, what the sign of the original contents of AR will be. Therefore, in order to insure a positive number in AR, you must precede the command shown above with another command whose purpose is to replace the original contents of AR with their absolute value. This command will be of the form:

L T N 2 28 28.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	2	28	28		$\lvert AR \rvert \longrightarrow AR_c$
02		03	N	3	28	29		$AR \xrightarrow{-} AR_+$

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	2	10	28		$\lvert 10.01 \rvert \longrightarrow AR_c$
02		03	04	3	28	28		$AR \xrightarrow{-} AR_c$
04		05	06	2	10	29		$\lvert 10.05 \rvert \longrightarrow AR_+$
06		07	N	1	28	10		$AR \xrightarrow{+} 10.07$

The absolute value of a double-precision number may be added to the original contents of the double-precision accumulator, PN, through the use of the double-precision equivalent of a C of 2: this would be a C of 6. The absolute value of the double-precision number will be transferred because D = 30. The command would be of the form:

L T N 6 S 30.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	10	26		10.02-03 $\xrightarrow{+}$ $PN_{0,1}$
04		06	08	6	10	30		$\lvert$10.06-07$\rvert$ $\longrightarrow$ $PN_{0,1+}$
08		10	N	5	26	10		$PN_{0,1}$ $\xrightarrow{+}$ 10.10-11

Notice that a C of 6 cannot be used to replace the original contents of PN with a double-precision absolute value, because D, in this case, would have to be 26, and therefore the rule that, for a C of 6 to call for the transfer of absolute value, either S or D must be greater than or equal to 28, would be violated. Similarly, the absolute value of a double-precision number in PN cannot be transferred to a predetermined storage location in memory by a C of 6, because PN, as a source, must always be referred to as line 26. The answer to this problem is to first clear the two-word registers, including PN, and then add the magnitude to PN, C = 6 and D = 30.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		03	03	0	23	31		Clear 2-word registers
03		04	06	6	10	30		$\lvert$10.04-05$\rvert$ $\longrightarrow$ $PN_{0,1+}$
06		08	10	6	10	30		$\lvert$10.08-09$\rvert$ $\longrightarrow$ $PN_{0,1+}$
10		12	N	5	26	10		$PN_{0,1}$ $\xrightarrow{+}$ 10.12-13

Any command which would normally affect only one data word, such as the deferred commands we have been discussing up to this point, can be made to affect a "block" of contiguous data words by being made immediate.

L	P	T or L_k	N	C	S	D	BP	NOTES
05		09	N	0	10	11		10.09 $\longrightarrow$ 11.09

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	09	N	0	10	11		10.01-08 $\longrightarrow$ 11.01-08

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	01	N	0	10	11		line 10 $\longrightarrow$ line 11

We thus can have "block adds", "block subtracts", "block copies", etc.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	01	1	10	28		10.01 $\xrightarrow{+}$ AR_c
01	u	01	02	1	10	29		10.02-00 $\xrightarrow{+}$ AR_+
02		03	N	1	28	09		AR $\xrightarrow{+}$ 09.03

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	2	28	28		$\lvert AR \rvert \longrightarrow AR_c$
02		03	04	3	28	29		AR $\xrightarrow{-}$ AR_+
04	u	05	05	0	28	18		AR $\longrightarrow$ 18.05-04
05		06	N	1	31	31		Number track to line 18

We have already mentioned the number track. There is a special command available, which will copy words from the number track into line 18, where they may be treated as data:

L T N 1 31 31.

PPR will make this command immediate, because D = 31. Any number of words may be copied, depending on the relationship of T to L. If the entire number track is desired in line 18, T should equal L + 1.

In this particular case only, the words arriving at line 18 will not replace the original contents of that line, but they will be logically added to the original contents, instead. Logical addition is an "either-or" proposition, in which a 1 will result in the sum if either of the numbers being added, or each of them, contains a 1; there is no "carry".

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

For this reason, line 18 should be cleared prior to receiving the contents of the number track.

So far we have been treating data words in the computer as if they were in binary form. Although this is correct, it is desirable to give a program decimal numbers as inputs and receive from the program decimal numbers as outputs. In such a case, the program itself will have to convert the decimal numbers it receives to their binary equivalents before performing operations on them, and it will have to convert its binary answers to their decimal equivalents before transmitting them as outputs. Fortunately each programmer who uses the G-15 does not have to write the necessary number-conversion routines in each program he develops, because this work has already been done for him by the Bendix Computer Division.

SUBROUTINES

The final effort of a programmer may go by any of various names, depending on the use for which it is intended. Some programmers write programs which are complete entities in themselves, in that they accept some raw data, perform all of the necessary operations on it, and yield valuable, final answers which are of use to the computer user. Other programmers write sequences of commands designed to accomplish some intermediate result which will be necessary during the manipulation of the raw data in other programs. The conversion of decimal numbers to binary is such a manipulation, and the binary numbers which result are intermediate to a final answer of a program. A sequence of commands designed to yield such useful intermediate results is called a subroutine, implying that it is designed to be a subordinate part of a longer routine, which might be called a program. Subroutines are written in such a way that they can be easily incorporated into longer routines in a manner prescribed by their author. These specifications always accompany a subroutine when it is distributed, or "issued", to computer users in general.

In order to understand how subroutines can be incorporated into your program, you must first be aware of the fact that it is possible, by a special command, to cause the computer to cease taking commands from the normal sequence, and start a new sequence at a prescribed location. Commands which can cause the computer to do this are referred to as "jump", "branch", or "transfer" commands; in programming the G-15, we refer to them as transfer commands, where we use the word transfer to mean "transfer control". The special command which will cause the G-15 to do this is:

L L+2 N C 21 31,

or,

L w T N C 21 31.

Normally the G-15 continues taking commands in a sequence from the same line in memory, where each command is found in that line at the word-time equal to N of the previous command. When this special transfer command is interpreted and executed, however, the computer will transfer program control to the line in memory specified by the C number in this command, and the first command in the new line will be found at the word-time equal to N of the transfer command. Notice that only eight lines can be specified as command lines, because C is a one digit number ranging from 0 through 7. This correctly implies that not all lines in the memory of the computer are capable of being read for commands. A memory line which has this capability is referred to as a "command line". Command line numbers are associated with memory lines according to the following table:

Command line number	Memory line number
0	00
1	01
2	02
3	03
4	04
5	05
6	19
7	23

The specifications for each available subroutine will contain the line number in which the subroutine must operate, the word-time in that line at which the first command of the subroutine is located, and the memory location in which the data upon which the subroutine is to operate is to be stored, along with other information.

Thus, if you have a data number which you want converted from decimal to binary, you must consult the specifications for the number-conversion subroutine for this information, and then incorporate into your program the necessary commands to:

1. place the decimal number in that memory location prescribed, and

2. transfer control to the prescribed command line, in which the subroutine is located, at the initial word-time in that line, which is also prescribed in the specifications.

When a subroutine has been entered, in the course of operation of a program, and the subroutine has done its work, there must be some provision for having the subroutine return program control to the main part of the program. The specifications for each subroutine will state where the output of the subroutine will be stored, and the main program can be written so that, upon re-entry from the subroutine, it will perform its operations using the intermediate result in this location. A question arises, for the programmer who is writing the subroutine, as to what line and what word-time within that line the subroutine is to transfer control to. Obviously, this will be different for each main program which uses the subroutine, and therefore a transfer command, coded in the form we have already discussed, will not help the programmer writing a subroutine.

The solution adopted for this problem is a second type of transfer command, called a "return" command. This is also a special command, and is coded in the following way:

L L+2 L+1 C 20 31.

When this command is executed, it will transfer control to command line C, at a predetermined word-time. The manner in which this word-time is predetermined is through the prior execution of a transfer command. When a transfer command is executed, in addition to transferring control to line C, word N, it "marks" a word-time, which is the first word-time of execution of the transfer command. This mark determines the word-time in line C to which the return command will return control; when a return command is executed, subsequent to the execution of a transfer command, control will be returned to line C (C in the return command) at the marked word-time. Because it generates this mark, the transfer command, whose special operation code is 21, is called a "Mark, Transfer" command.

Assume that the number-conversion subroutine is in line 02, and the main program is in line 00, and the following information is contained in the specifications for that subroutine:

Execution..........................Command line 02
Entry..............................Word-time 46
Exit...............................Word-time 47
Input..............................x (decimal) in ID_1
(7 digits and sign)
Return command in AR
Output.............................x (binary) in MQ_0

Assuming x (decimal), consisting of seven digits and a sign, is in 23.00, and that x (binary) is desired in 00.59, the following sequence of commands will satisfy the requirements:

L	P	T or L_k	N	C	S	D	BP	NOTES
00		04	06	6	23	25		x = (23.00) ⟶ ID_1
06		07	08	0	00	28		00.07 ⟶ AR
07		[49	48	0	20	31]		Return Command
08	w	50	46	2	21	31		Mark, transfer to 02.46
50		52	53	0	24	28		x = (MQ_0) ⟶ AR
53		59	N	0	28	00		x = (AR) ⟶ 00.59

Now we come to an interesting point, which, we hope, has been bothering you: if all words in the memory of the computer are of binary form, how is it that decimal numbers can be entered during an input? In order to answer this question, which now, at least, is bothering you, you must understand the input system for the G-15.

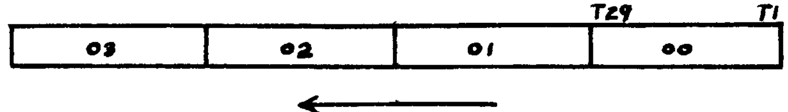
INPUTS

On the keyboard of the typewriter, as shown on page 130, the digit keys, certain letter keys, and the minus sign, tab, carriage return and "/" keys can all cause a direct effect in the way of data input. Any of the digit keys, 1 through 0, and the letter keys, u, v, w, x, y, and z, which are sufficient to complete the hex number system, will generate a 4-bit code during an input:

Key	4-bit code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
u	1010
v	1011
w	1100
x	1101
y	1110
z	1111

The minus key sets a "sign flip-flop" with a 1; if the minus key is not struck during the input of a number, this flip-flop will retain a 0. The tab and carriage return keys have identical effects on the input system; they each cause the sign flip-flop's contents to be placed in the sign-bit of a word.

Each time a 4-bit code is entered, during an input, it causes all of short line 23 (four words = 116 bits) to be shifted towards the high-order end of the line, in the direction shown by the arrow in the drawing below. The four vacated bit-positions in the low-order end of the line, bits T1 through T4 of 23.00, are cleared to 0000. These four bit-positions receive the incoming 4-bit code.



Thus, after seven digits have been entered, there are 28 bits properly set in 23.00, T1 through T28. If hex digits, representing a binary number, were entered, the 28-bit magnitude is exactly reproduced in these 28 bits. Unfortunately this magnitude is not properly positioned in the word, however, because we know that it should occupy bits T2 through T29, and a sign should occupy bit T1. Either the tab or the carriage return key will have the same effect on line 23: it will shift the line's contents toward the high-order end by one bit-position, vacating T1 of 23.00, and that bit will receive the present contents of the sign flip-flop. Four words, each consisting of seven hex digits and a sign, could be entered into line 23 by repeating this process. At any point, line 23's contents can be transferred, word-for-word, into the low-order four words of line 19, 19.00-19.03, by striking the "/" key, referred to, because of its effect, as the "reload" key. Line 19's contents will be shifted towards the high-order end of the line by four full word-times whenever this action is taken. Thus you can see that it is possible to enter an entire long line's contents (108 words) during one input operation.

Now, if the seven digits that are entered for any given number are decimal digits, rather than hex digits, we will have a 28-bit magnitude consisting of 4-bit codes, each of which ranges in value from 0000 to 1001. It is obvious that this binary number is not the binary equivalent of the decimal number entered.

Digits typed in: 9876543 (tab) s

23.00: 1001100001110110010101000011|0

binary integral value equivalent to $9876543._{(10)}$:

0000100101101011010000111111|0

We call a binary number in this form "binary-coded-decimal". It is a number of this form which we will obtain when typing in decimal digits and which we will supply as an input to the number-conversion subroutine.

DECIMAL SCALING

At this point, we must settle on some accepted system for discussing the quantities that are being handled by the computer. Since no decimal points are entered during the type-in of inputs, a system is necessary for interpreting the numbers entered. It is convenient, and therefore common, to consider all numbers in the computer as fractions. In other words, if we enter the decimal number, -9876543, we will assume that the computer handles it as the number, -.9876543. Now, this does not mean that the computer can only handle fractional quantities; it does mean that we must have some method for properly interpreting the numbers that are in the machine. If the quantity we are representing with this number is really -98.76543, we will say that the machine holds this number divided by 100, or multiplied by 10^{-2}. And we can, in general, say that the machine holds our number, A, multiplied by a factor of 10 raised to some power. This factor we call the "scale factor". If A* represents the machine form of the number A, and S represents the scale factor of A, then $A^* = S \cdot A$. This determination of scale factors for numbers is called "scaling". Usually the scale factor associated with a number in the computer which is being affected by a given command will be entered in the notes column for that command on the coding sheet.

As you know, numbers whose decimal points are not lined up cannot be added to, or subtracted from, each other, without first shifting either or both of them, in order to line up the decimal points.

$$\begin{array}{rcr} 1.654 & = & 1.654 \\ +\ \underline{398.7} & = & \underline{398.7} \\ & & 400.354 \end{array}$$

The decimal scale factor of a number in the computer merely fixes the decimal point in that number.

$$\begin{array}{rcl} \underline{A^*} & = & \underline{A} \cdot \underline{S} \\ .0001654 & = & 0001.654 \cdot 10^{-4} \\ .0003987 & = & 000398.7 \cdot 10^{-6} \end{array}$$

Therefore, numbers in the computer must have like scale factors before they can be properly added to, or subtracted from, each other. This can be accomplished by multiplying either or both of them by $1 \cdot 10^{n}$, where n equals the number of decimal places the number is to be shifted.

$$\begin{array}{ll} .0001654(=0001.654 \cdot 10^{-4}) & =.0001654(=0001.654 \cdot 10^{-4}) \\ .0003987(=000398.7 \cdot 10^{-6}) \cdot 100.000000(=1 \cdot 10^{2}) & \underline{=.0398700(=0398.700 \cdot 10^{-4})} \\ & .0400354(=0400.354 \cdot 10^{-4}) \end{array}$$

We have already said that it is convenient to consider all numbers in the machine as fractions. This would eliminate the multiplier, 10.0000000, in the above example. An obvious solution to this dilemma would be to rescale .0001654 rather than .0003987, in the following manner:

$$.0001654(=0001.654 \cdot 10^{-4}) \cdot .0100000(=01.00000 \cdot 10^{-2}) = .0000016(=000001.6 \cdot 10^{-6})$$

$$\begin{array}{ll} .0003987(=000398.7 \cdot 10^{-6}) & \underline{=.0003987(=000398.7 \cdot 10^{-6})} \\ & .0004003(=000400.3 \cdot 10^{-6}) \end{array}$$

Unfortunately, this method of rescaling, although it properly aligns the decimal points for the addition of the two numbers, causes a loss of accuracy in one of them, and therefore, in the result. It is desirable to rescale .0003987, by shifting it to the left, because no significance will be lost in that number, as you saw above, and yet, no accuracy will be lost, either. If we cannot have the number, 10.0000000, in the computer, we must find a substitute for it. A substitute for multiplication by any number is division by its reciprocal. The reciprocal of $1 \cdot 10^2$ is $1 \cdot 10^{-2}$. Therefore, instead of multiplying .0003987 by $10.0000000(=1 \cdot 10^2)$, we can divide .0003987 by $.0100000(=01.00000 \cdot 10^{-2})$:

$$\begin{array}{ll} .0001654(=0001.654 \cdot 10^{-4}) & =.0001654(=0001.654 \cdot 10^{-4}) \\ .0003987(=000398.7 \cdot 10^{-6}) \div .0100000(=01.00000 \cdot 10^{-2}) & \underline{=.0398700(=0398.700 \cdot 10^{-4})} \\ & .0400354(=0400.354 \cdot 10^{-4}) \end{array}$$

When two numbers are multiplied together, the scale factor of the product equals the product of the scale factors:

$$a \cdot 10^n \cdot b \cdot 10^m = a \cdot b \cdot 10^{n+m}$$

When a number is divided by another, the scale factor of the quotient is the quotient of their respective scale factors:

$$\frac{a \cdot 10^n}{b \cdot 10^m} = \frac{a}{b} \cdot 10^{n-m}$$

The method of scaling we have just discussed is called "fixed-point" scaling, because it is a means of interpreting numbers in the machine in relation to a fixed machine-point, which immediately precedes the most significant bit of a number, making the number a fraction, as it appears in the machine.

Because scaling is merely a means of interpreting values in the machine, however, any method of scaling is permissable, as long as it is consistent and dependable. Another method in common usage is "floating-point" scaling. For a discussion of this method, see page 201.

BACK TO ARITHMETIC

Because the G-15 has a limit of 28 magnitude bits for a single-precision number, it is possible to attempt to generate a sum in AR which cannot be contained within 28 bits, and therefore the sum which is generated is erroneous. The condition that arises in such a case is called "overflow", and the machine is equipped to detect this, although it will do nothing about it automatically. However, an overflow test command is available for inclusion in programs, and the programmer can take whatever action he deems necessary in that sequence of commands which starts with the

"yes" answer for any overflow test. The overflow test command is:

L L+2 N 0 29 31.

In addition to testing for the presence of the overflow condition, this command also turns off the overflow indicator. Furthermore, that indicator can only be turned off by the overflow test command. The corollary to this, naturally, is that, once the overflow indicator has been turned on, through the generation of an overflow, it will remain on until it is tested.

It is essential, when checking for the generation of overflow by a certain command or sequence of commands, to be sure that the indicator is off when that command or sequence of commands is entered. Turning off the overflow indicator through use of the overflow test command is the only way to insure this, but you must remember that, even though you are only using the test command for this purpose, nevertheless it is a test command, and, depending on the original setting of the indicator, the next command may be taken from either N or N + 1. One solution to this is, of course, to place the same command at N and N + 1, so that, regardless of the answer to the test, the same sequence of commands will follow.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	02	0	29	31		Turn off overflow
02		04	05	1	21	28		$21.00 \xrightarrow{+} AR_c$ 10^{-5}
03		04	05	1	21	28		$21.00 \xrightarrow{+} AR_c$ 10^{-5}
05		07	08	1	21	29		$21.03 \xrightarrow{+} AR_+$ 10^{-5}
08		10	10	0	29	31		Overflow?
10		12	N	1	28	22		No $AR \xrightarrow{+} 22.00$ 10^{-5}
11		13	00	0	16	31		Yes Halt

Another solution is to write the overflow test command in the following way:

L L+2 L-1 0 29 31.

If the answer is "no", the program will continue at **L** - 1. If the answer is "yes", the next command will be taken from N (= **L** -1) + 1 = **L**, and the test will be repeated. Of course it will be answered "no" the second time, because the indicator was turned off by the test the first time.

L	P	T or L_k	N	C	S	D	BP	NOTES
05		07	04	0	29	31		Turn off overflow
04		08	09	1	21	28		21.00 $\xrightarrow{+}$ AR_c 10^{-5}
09		11	12	1	21	29		21.03 $\xrightarrow{+}$ AR_+ 10^{-5}
12		14	14	0	29	31		Overflow?
14		16	N	1	28	22		No AR $\xrightarrow{+}$ 22.00 10^{-5}
15		17	00	0	16	31		Yes Halt

The same overflow test is also effective for double-precision arithmetic in PN.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	02	0	29	31		Turn off overflow
02		04	07	5	21	30		21.00-01 $\xrightarrow{+}$ $PN_{0,1}$ 10^{-8}
03		04	07	5	21	30		21.00-01 $\xrightarrow{+}$ $PN_{0,1}$ 10^{-8}
07		10	12	7	21	30		21.02-03 $\xrightarrow{-}$ PN_+ 10^{-8}
12		14	14	0	29	31		Overflow?
14		16	N	5	26	22		No PN $\xrightarrow{+}$ 22.00-01 10^{-8}
15		17	00	0	16	31		Yes Halt

Two arithmetic operations remain undiscussed: they are multiplication and division. These cannot be performed by normal commands which call for the transfer of words. They require special commands.

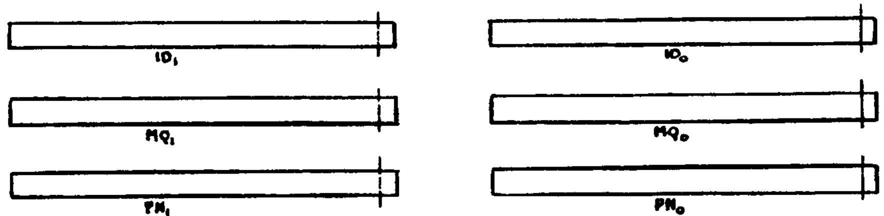
MULTIPLY

The special command for multiply is:

L T N 0 24 31.

When this command is given, the computer will automatically multiply two numbers in specific locations of its memory: the multiplicand will be in the two-word register ID, and the multiplier will be in the two-word register MQ. The product will be generated in the two-word register PN.

Multiplication is essentially a double-precision process, and both halves of ID and MQ enter into it, a double-precision product being generated in all of PN. We know that the most significant half of a double-precision number is in an odd word-time, and the least significant half of the same number is in the immediately preceding even word-time. Therefore, if we draw the two-word registers as shown below, the most significant bit in each number involved will be the left-most bit, as is customary in the writing of numbers in any number system.



But single-precision multiplication can also be performed in the two-word registers, the only difference being that the single-precision multiplicand will occupy only the most significant, or odd, half of ID, and the multiplier will occupy the same position, respectively, in MQ.

A single-precision multiplication will, nevertheless, yield a double-precision product in PN, due to the fact that multiplication, in the machine, is a double-precision process. The product, in PN, is generated through a series of successive additions of ID into PN (see pages 70 - 75 for an explanation of multiplication). For this reason, if a single-precision multiplicand is loaded into the odd half of ID, the even half of ID must be cleared to 0. After a multiplication of two single-precision numbers has been performed, and the product is in PN, if a single-precision product is desired, it will be available in the odd half of PN; if a double-precision product is desired, it will be available in all of PN.

The T number in the multiply command is a "relative timing number" indicating the number of word-times for which the multiplication is to be performed. Two word-times are necessary for each bit in the multiplier which is to enter into the multiplication. If two single-precision numbers are to be multiplied, obviously the multiplier will contain 28 magnitude bits which are to enter into the operation; therefore, the T number in the multiply command should be 56. If the multiplier is a double-precision number, 57 bits of magnitude are to enter into the operation; therefore, the T number in the multiply command should be v4 (= 114).

The location of the multiply command should always be an odd word-time, because the operation is essentially double-precision in nature, and immediate. It has already been pointed out that double-precision operations must begin in even word-times. You will find this situation applying in the cases of other commands, as well.

If the multiply command automatically multiplies the two numbers in ID and MQ, it stands to reason that, before the multiply command is given, the proper numbers must be in those two registers. They can be there only if your program places them there prior to calling for a multiplication. But copying words into the two-word registers is a bit more complicated than copying words into any other memory locations.

There is a flip-flop called "IP", which is associated with the two-word registers. Under certain conditions, the sign of a number will be divorced from the magnitude bits and sent to IP, when the destination is a two-word register; the magnitude bits will always be transmitted to the addressed register, however. Whenever a sign of a number is divorced and sent to IP, the bit in the two-word register which would normally have received the sign is cleared to 0. Similarly, under certain conditions, the sign of a number being transferred from a two-word register to some other memory location may be taken from IP, rather than from the two-word register source; the magnitude bits of the number being transferred will always come from the two-word register source, however.

It should be noted here that the clear two-word registers command also clears the IP flip-flop.

The following rules apply to transfers of information to and from the two-word registers.

1. If the destination is a two-word register (24, 25, 26) and C is even (0, 2, 4, 6), the sign of the number will be sent to IP.

    a. If ID is the destination, IP will be cleared prior to receiving the sign of the number.

    b. If either MQ or PN (26) is the destination, IP will not be cleared prior to receiving the sign of the number, but the sign will be added to the present contents of IP. Since IP can retain only one bit, it will contain, in this case, the least significant bit of the sum, and any carry generated by the sum will be lost.

2. If the source is a two-word register (24, 25, 26) and C is even, the sign accompanying the number will be taken from IP, rather than from the normal sign bit in the two-word register source.

3. If ID is the destination and C is even, for every bit set in ID, the corresponding bit in PN will be cleared.

4. If an exchange of AR with memory is called for (C = 2, 3, 6, 7) and the destination is a two-word register, during any even word-time of execution, AR's contents will be blocked from

entering the two-word register, and twenty-nine 0's will be transferred instead; AR's contents will be lost.

5. As an exception to rules 1 - 4, if the transfer called for is from one two-word register to another, IP will remain unaffected.

6. As an exception to rules 1 - 5, if the source is PN (26), the destination is PN (26), and C = 0 or 4, the sign bit in IP will be combined with the magnitude bits from PN, this number will pass through the inverting gates, and the resultant number will be placed in PN, the sign remaining with the magnitude bits.

L	P	T or L_k	N	C	S	D	BP	NOTES	
00		03	03	0	23	31		Clear 2-word registers	
03		05	06	0	10	25		10.05 ⟶ ID_1	10^{-5}
06		07	09	0	20	24		20.03 ⟶ MQ_1	10^{-5}
09		56	66	0	24	31		Multiply	10^{-10}
66		68	N	4	26	21		$PN_{0,1}$ ⟶ 21.00-01	10^{-10}

L	P	T or L_k	N	C	S	D	BP	NOTES	
00		02	04	4	10	25		10.02-03 ⟶ $ID_{0,1}$	10^{-5}
04		08	11	4	21	24		21.00-01 ⟶ $MQ_{0,1}$	10^{-5}
11		v4	18	0	24	31		Multiply	10^{-10}
18		70	N	4	26	10		$PN_{0,1}$ ⟶ 10.70-71	10^{-10}

L	P	T or L_k	N	C	S	D	BP	NOTES	
00		02	04	6	10	25		10.02 ⟶ ID_1	10^{-5}
04		06	09	6	20	24		20.02 ⟶ MQ_1	10^{-5}
09		56	66	0	24	31		Multiply	10^{-10}
66		67	N	0	26	11		PN_1 ⟶ 11.67	10^{-10}

DIVIDE

The special command for divide is:

L T N (1 or 5) 25 31.

There is no difference in the effect of the divide command between a C of 1 and a C of 5. When this command is given, the computer will automatically divide the numerator, in PN, by the denominator, in ID, generating a quotient in MQ.

Division is also essentially a double-precision process, both halves of ID and PN entering into it, but the precision of the quotient generated in MQ is determined by the number of word-times for which the division is carried out. The T number in the divide command is also a relative timing number. If a single-precision quotient is desired, let T = 57; the single-precision quotient will be generated in the even half of MQ. If a double-precision quotient is desired, let T = v6 (= 116); the double-precision quotient will occupy both halves of MQ. A more complete description of the division process is contained in pages 76 - 84.

The location of the divide command should always be an odd word-time.

L	P	T or L_k	N	C	S	D	BP	NOTES	
00		03	03	0	23	31		Clear two-word registers	
03		05	06	0	21	25		21.01 $\longrightarrow$ ID_1	10^{-5}
06		07	09	0	10	26		10.07 $\longrightarrow$ PN_1	10^{-7}
09		57	67	5	25	31		Divide	10^{-2}
67		68	N	0	24	28		$MQ_0 \longrightarrow AR_c$	10^{-2}

L	P	T or L_k	N	C	S	D	BP	NOTES	
00		02	04	6	10	25		10.02 $\longrightarrow$ ID_1	10^{-5}
04		05	07	0	20	26		20.01 $\longrightarrow$ PN_1	10^{-7}
07		57	65	1	25	31		Divide	10^{-2}
65		66	N	0	24	21		$MQ_0 \longrightarrow$ 21.02	10^{-2}

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	4	10	25		10.02-03 ⟶ $ID_{0,1}$ 10^{-6}
04		06	09	4	21	26		21.02-03 ⟶ $PN_{0,1}$ 10^{-8}
09		v6	18	5	25	31		Divide 10^{-2}
18		22	N	4	24	20		$MQ_{0,1}$ ⟶ 20.02-03 10^{-2}

Some problems require more mathematical processes than we have discussed up to this point, for instance, the generation of a square root or a trigonometric function. Any of these more exotic processes can be performed through a series of arithmetic operations which approximate the desired value. Subroutines have been written by the Bendix Computer Division for various mathematical processes, and each of these subroutines can be incorporated into a main program in the same manner in which the number-conversion subroutine was, in the previous example.

Suppose the following information is included in the specifications for the square root subroutine, and you desire to store in 19.u6-u7 the double-precision square root of a double-precision number in 21.00-01:

Execution..........................From command line 01
Entry..............................At word-time 94
Exit...............................Return command from 01.98
Input..............................N $\xrightarrow{+}$ PN0,1
Return command ⟶ AR
Output.............................$\sqrt{N}$ double-precision = $PN_{0,1}$
$\sqrt{N}$ single-precision = 20.03
N = 21.00-01

A sequence of commands starting at word-time 56 in the main program, which could be in any command line other than 01 of course, in this case, line 00, to accomplish the above purpose,might be:

L	P	T or L_k	N	C	S	D	BP	NOTES
56		57	58	0	00	28		00.57 ⟶ AR_c
57	[	u0	99	0	20	31]		Return command
58		60	62	5	21	26		21.00-01 $\xrightarrow{+}$ $PN_{0,1}$
62		64	94	1	21	31		Go to square root subroutine
63		u6	N	5	26	19		$PN_{0,1}$ $\xrightarrow{+}$ 19.u6-u7

Mention of taking the square root of a number gives rise to the discussion of two more test commands which are available, and which operate in the same fashion as the other test commands which have already been discussed. These two test commands are:

L T N C S 27:

the contents of the operand will be tested for non-zero. If all of the bits tested equal 0, the answer to this question will be "no"; the next command will be taken from N. If any of the bits tested equals 1, the answer will be "yes"; the next command will be taken from N + 1. Any C may be used in this test command, and the test will be performed on bits in a predictable manner, depending on the C used.

L L+2 N 0 22 31:

the sign of AR will be tested for negative. If the sign is negative, the answer will be "yes".

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	1	20	28		$20.01 \xrightarrow{+} AR_c$
02		03	04	3	10	29		$10.03 \xrightarrow{-} AR_+$
04		06	06	0	28	27		Test AR ≠ 0
06		08	00	0	16	31		= 0 Halt
07		09	N	1	28	21		≠ 0 $AR \xrightarrow{+} 21.01$

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	05	05	0	20	27		Test line 20 ≠ 0
05	u	10	06	0	21	20		= 0 Line 21 ⟶ Line 20
06		07	11	1	20	28		≠ 0 $20.03 \xrightarrow{+} AR_c$
11	u	15	15	1	20	29		$20.00\text{-}02 \xrightarrow{+} AR_+$
15		17	17	0	22	31		Test AR negative
17		20	N	1	28	10		+ $AR \xrightarrow{+} 10.20$
18		20	N	1	28	11		- $AR \xrightarrow{+} 11.20$

LOGICAL OPERATIONS

So far the only operations upon data which we have discussed are arithmetic, and we have assumed that the data numbers represent quantities. But we spoke earlier of another meaning for numbers: code. Programs can be written to process data which is in code form, where each bit, or group of bits, in a data word represents some information other than a quantity; it might, for instance, represent the answer to a question, or the sex or marital status of a person who is being treated by the program as a statistic. There are operations available in the computer which are essentially logical, rather than arithmetic, in nature. The bits in a word may be shifted to the right or to the left, or individual bits in a word may be isolated from the rest, for independent treatment.

SHIFT

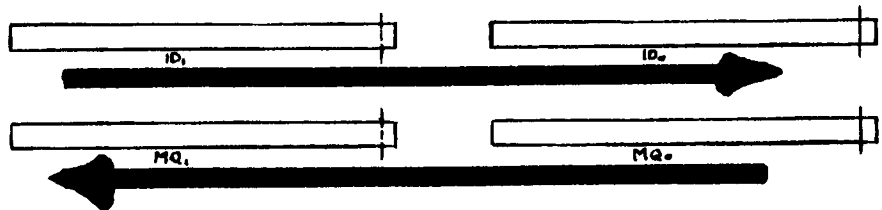
The shifting process can be performed by either of two commands:

L T N 1 26 31

or

L T N 0 26 31.

In either case, the words shifted, and the directions in which they are shifted, are the same: ID shifts right, and MQ shifts left, concurrently.



The data word which is to be shifted must be placed in either of these two-word registers, depending on the desired direction of the shift, prior to giving the shift command. All of ID will shift to the right, with the exception of bit T1 of ID_0, the sign bit, which will not be involved in the shift. All 58 bits of MQ will shift left.

The number of shifts that will be performed is determined by the number of word-times of execution allowed; each shift will move all bits in ID to the right one bit-position and all bits in MQ to the left one bit-position. T, in each of the shift commands, is a relative timing number, indicating the number of word-times of execution; two word-times are necessary for each shift. Therefore, T should equal 2 times the number of shifts desired: T will always be an even number.

The shift command, L T N 1 26 31, will operate on ID and MQ in the above manner for the number of shifts called for by the T number. The other shift command, L T N 0 26 31, will also operate in the above manner, but the duration of its execution may be determined either by the T number in the command, or by the contents of AR. For every shift performed by this command, a 1 will be added to AR, in the least significant magnitude bit-position, bit T2, and the generation of an end-around-carry in AR will terminate the shifting process with the shift

causing it. If the number of word-times of execution called for by T in the command is fulfilled before the occurrence of this end-around-carry in AR, the shift will also be terminated. Therefore, T, in this command, sets a limit upon the number of shifts that will be performed, but the number of shifts might be less, depending on the contents of AR. The location of a shift command should always be an odd word-time.

Before Execution

ID_1 1010110101100001011111100110|0 ID_0 1101111010101001001111010111|0

MQ_1 0110001111000111111100011011|1|0 MQ_0 1101001111000001111111010100|1

AR 0011110111110000000000000110|0

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	20	25		20.02-03 $\xrightarrow{+}$ $ID_{0,1}$
04		08	11	5	20	24		20.00-01 $\xrightarrow{+}$ $MQ_{0,1}$
11		40	N	1	26	31		Shift

After Execution

ID_1 0000000000000000000101011010|0 ID_0 1100001011111100110011011110|0

MQ_1 0011011101101001111000001111|1 MQ_0 1101010010000000000000000000|0

AR 0011110111110000000000000110|0

Before Execution

ID_1 1010110101100001011111100110|0 ID_0 1101111010101001001111010111|0

MQ_1 0110001111000111111100011011|1|0 MQ_0 1101001111000001111111010100|1

AR 0000000000000000000000000000|0

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	20	25		20.02-03 $\xrightarrow{+}$ $ID_{0,1}$
04		08	11	5	20	24		20.00-01 $\xrightarrow{+}$ $MQ_{0,1}$
11		40	N	0	26	31		Shift

After Execution

ID_1 00000000000000000000101011010 ID_0 110000101111110011001101111100

MQ_1 001101110110100111100000111111 MQ_0 110101001000000000000000000000

AR 000000000000000000000000101000

Before Execution

ID_1 101011010110000101111110011000 ID_0 110111101010100100111101011110

MQ_1 011000111100011111100011011110 MQ_0 110100111100000111111010100 01

AR 111111111111111111111111110001

L	P	T or L_k	N	C	S	D	BP	NOTES
00		02	04	5	20	25		20.02-03 $\xrightarrow{+}$ $ID_{0,1}$
04		08	11	5	20	24		20.00-01 $\xrightarrow{+}$ $MQ_{0,1}$
11		40	N	0	26	31		Shift

After Execution

ID_1 000000001010110101100001011111 ID_0 110011001101111010101001001110

MQ_1 110001111110001101110110100111 MQ_0 110000011111110101001000000000

AR 000000000000000000000000000000

EXTRACT

The isolation of certain bits in a word, so that they may be treated independently of the other bits in the word, is accomplished through a logical operation called, logically enough, "extraction". There are several "extract" commands, of which we will discuss only two here. When you call for an extract operation, you must, of course, specify the bits to be extracted; you do this by using a "mask" during the extraction, which is a word in which you have set 1's in those bit-positions corresponding to the bits in the data word you want to save and 0's in all the rest. For example, we want bits T29 - T22 and T1 only; the following mask should be used:

111111110000000000000000000001
T29 T1

One of the extract commands we will discuss now is:

L T N C 31 D.

The source of 31 marks this as a special command, but it will be deferred unless a prefix of u is inserted in the command. During each word-time of execution, the contents of the appropriate word in short line 21 will be compared with a mask in the corresponding word in short line 20, and the bits marked for saving by the mask will be saved. The resulting word, containing bits duplicating those in the data word in line 21, and having 0's in the bit positions not "covered" by the mask, will be transmitted to the appropriate word in the destination. All of this happens within a single word-time. It may be repeated for as many contiguous word-times as called for, if the command is immediate.

Before Execution

10.06 0011100000111000011001110100000

10.07 011011100011010111111000010101

10.08 0001110000111001110111110000010

10.09 1001110000010011110011110000 1

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	05	05	0	00	20		00.01-04 ⟶ 20.01-00
01								zzz0000+
02								z000000+
03								0000zzz-
04								z000zzz+
05	u	10	10	0	10	21		10.06-09 ⟶ 21.02-01
10	u	15	N	0	31	22		20·21 ⟶ 22.03-02

After Execution

22.02 0011000000000000000000000000000

22.03 000000000000000001111000010101

22.00 0001000000000000011011111000010

22.01 1001110000010000000000000000000

Before Execution

10.05   101101111101000100110111110011

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	0	00	20		00.01 ⟶ 20.01
01								zz00000-
02		05	06	0	10	21		10.05 ⟶ 21.01
06		09	11	0	31	28		20·21 ⟶ AR_C
11		05	N	0	28	10		AR ⟶ 10.05

After Execution

10.05   10110111000000000000000000001

The other extract command to be discussed here is:

L   T   N   C   30   D.

The use of a mask in line 20 and a data word in line 21 is the same as for the previous command, and the resulting word will be transmitted, as in the other case, to the destination, at the appropriate word-time. But the effect of the mask is reversed. Those bits in the word in line 21 corresponding to 0's in the mask in line 20 will be saved, and 0's will be transmitted in all those bit-positions corresponding to 1's in the mask. This is called "not mask" extraction.

Before Execution

10.05   101101111101000100110111110011

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	0	00	20		00.01 ⟶ 20.01
01								zz00000-
02		05	06	0	10	21		10.05 ⟶ 21.01
06		09	11	0	30	28		$\overline{20}$·21 ⟶ AR_C
11		05	N	0	28	10		AR ⟶ 10.05

After Execution

10.05 00000000110100010011011110010

Before Execution

10.06 00111000011100001100111010000

10.07 0110111000110101111110000101 01

10.08 00011100001110011101111000010

10.09 10011100000100111100111100001

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	05	05	0	00	20		00.01-04 ⟶ 20.01-00
01								zzz0000+
02								z000000+
03								0000zzz-
04								z000zzz+
05	u	10	10	0	10	21		10.06-09 ⟶ 21.02-01
10	u	15	N	0	30	22		$\overline{20}\cdot 21$ ⟶ 22.03-02

After Execution

22.02 00001000011100001100111010000

22.03 01101110001101010000000000000

22.00 00001100001110010000000000000

22.01 00000000000000111100111100001

Neither of these extract commands will alter the data word in line 21, or the mask in line 20. The extraction performed by the first command is expressed logically as: $20\cdot 21$. It is read as "20 and 21". The other extraction is expressed logically as $\overline{20}\cdot 21$. It is read as "not 20 and 21".

REPETITIVE PROCESSING OF DATA - LOOPS

Very often a programmer is faced with a necessity of writing a program designed to perform the same process, be it mathematically or logically,

on each of a sequence of stored data words. For instance, all of line 19 might be filled with 108 single-precision data numbers, each of which is to be processed in the same manner.

To write a program which contains 108 sequences of commands, so that the same process can be performed on each of these words, will be, in many cases, impractical, because the program will be too long to be stored in memory along with the data upon which it will operate. For this reason, programmers have adopted a method of repeating a given sequence of commands for any desired number of times. In the sequence, certain key commands, usually those which call for a data word and those which store a result, will be "modified" during each pass through the sequence, so that, each time they are interpreted, they will call for the same operation on a different word.

Because the computer's only method of determining whether a word is data or a command is based on the time during which it is read, RC or EX, a command can be treated as data by another command. This enables us to incorporate into a program one command which can transfer another command into AR, where a constant can be added to it, causing a predictable change in it. We can then transfer the new form of this command from AR to the command's original location in our program. Thus, the next time this command is read and interpreted, during the flow of our program, it will call for something different.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		00	01	1	10	28		10.00 $\xrightarrow{+}$ AR_c ←
01		02	03	3	11	29		11.02 $\xrightarrow{-}$ AR_+
03		00	04	1	28	10		AR $\xrightarrow{+}$ 10.00
04		00	06	0	00	28		00.00 ⟶ AR_c
06		07	08	0	00	29		00.07 ⟶ AR_+
07	[u	01	00	0	00	00]		
08		00	10	0	28	00		AR ⟶ 00.00
10		03	05	0	00	28		00.03 ⟶ AR_c
05		07	09	0	00	29		00.07 ⟶ AR_+
09		03	00	0	28	00		AR ⟶ 00.03 ┘

The preceding example is fine for the hypothetical case previously mentioned, with the one exception that the "loop" which we have generated will be unending. There must be some provision, within the loop, for exiting from it. We can do this through use of a "counter", as shown in the example below, where we test this counter for reaching a certain limit, at which point we exit from the loop.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		00	01	1	10	28		$10.00 \xrightarrow{+} AR_c$ ←
01		02	03	3	11	29		$11.02 \xrightarrow{-} AR_+$
03		00	04	1	28	10		$AR \xrightarrow{+} 10.00$
04		14	15	0	00	28		$00.14 \longrightarrow AR_c$
14								0000000+ Counter
15		16	17	3	00	29		$00.16 \xrightarrow{-} AR_+$
16								00000u7+ Limit
17		18	19	0	28	27		Test $AR \neq 0$
19		21	00	0	16	31		= 0 Halt
20		16	21	0	00	29		$\neq 0$ $00.16 \longrightarrow AR_+$
21		22	23	0	00	29		$00.22 \longrightarrow AR_+$
22								0000001+
23		14	26	0	28	00		$AR \longrightarrow 00.14$
26		00	06	0	00	28		$00.00 \longrightarrow AR_c$
06		07	08	0	00	29		$00.07 \longrightarrow AR_+$
07	[u	01	00	0	00	00]		
08		00	10	0	28	00		$AR \longrightarrow 00.00$
10		03	05	0	00	28		$00.03 \longrightarrow AR_c$
05		07	09	0	00	29		$00.07 \longrightarrow AR_+$
09		03	00	0	28	00		$AR \longrightarrow 00.03$

Another, more complex, method of looping and providing for an exit from the loop, is to set up a "base" command, as was done in the previous examples, a "difference" dummy command, by which we modify the base, as was also done in the previous examples, and a "limit", which equals the base command plus a predetermined number of increments. This method is shown in the example below.

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	02	0	00	28		Base I $\longrightarrow AR_c$
01	u	w7	15	1	10	28		Base I
02		03	04	0	00	29		Difference $\longrightarrow AR_+$
03	u	01	00	0	00	00		Difference
04		05	06	3	00	29		Limit $\xrightarrow{-} AR_+$
05		u8	15	1	10	28		Limit
06		07	08	0	28	27		Test AR $\neq$ 0
08		10	00	0	16	31		= 0 Halt
09		05	10	0	00	29		$\neq$ 0 Limit $\longrightarrow AR_+$
10		01	12	0	28	00		Reset Base I
12		14	20	0	31	31		Next command from AR
20		00	15	1	10	28		
15		02	07	3	11	29		11.02 $\xrightarrow{-} AR_+$
07	u	12	13	1	28	20		AR $\xrightarrow{+}$ 20.00-03
13		14	16	0	00	28		Base II $\longrightarrow$ AR
14	u	w7	00	0	20	10		Base II
16		03	11	0	00	29		Difference $\longrightarrow AR_+$
11		14	17	0	28	00		Reset Base II
17		19	19	0	31	31		Next command from AR
19		00	00	0	20	10		

Also shown in the example is the use of another special command, as yet undiscussed. This new command is:

L T N 0 31 31.

It directs the computer to take the next command, at word time N, from AR. As you can see, it would be possible to have the modified form of the base command in AR when the computer is given the "next command from AR" command. This will usually save the machine time, and is therefore preferable to the previous method of looping.

THE INPUT/OUTPUT SYSTEM

When all the arithmetic and logical operations necessary have been performed on the data by a program, the only remaining work for the program to do is to communicate the answers to the outside world. This necessitates use of the input/output system again.

The input/output system can perform only one operation at a time, and care must be taken, when programming this system, to prevent giving it a second operation, either input or output, to perform while it is still engaged in a previous one. The result of such a mistake will be that the system will attempt to start an operation called for by the "logical" sum of the two special codes. For instance, if you desired to type out the answers, and chose the command, L L+2 N 0 09 31, which does call for a "type-out", and the input/output system was still performing a "type-in", whose special code, as we know, is 12, the system would attempt to start an operation called for by the logical sum of these two special codes. In logical addition, a 1 will result in a bit-position if either of the numbers being added has a 1, or if they both have a 1, and there will be no "carry" from one bit-position to the next:

$$0 + 0 = 0,\ 0 + 1 = 1,\ 1 + 0 = 1,\ 1 + 1 = 1.$$

$$\begin{aligned} 12 &= 1100 \\ 09 &= \underline{1001} \\ &\ 1101 = 13. \end{aligned}$$

In this case, an input or output whose special code is 13 would be called for, and this, of course, would be erroneous.

The way to prevent this from happening is to precede each input and each output command with a ready test, previously discussed, so that the program cannot continue to the new input/output command until the input/output system is ready.

OUTPUTS

There are two normal outputs of the G-15:

1. typewriter, and

2. punched paper tape.

Line 19 is the only source of information for the tape punch; either line 19 or AR may be the source of information for the typewriter. The three output commands are:

1. L L+2 N 0 08 31 : type AR's contents,

2. L L+2 N 0 09 31 : type line 19's contents, and

3. L L+2 N 0 10 31 : punch line 19's contents.

The computer is very flexible in its choice of forms for an output; the form of an output may be determined by the programmer. As a matter-of-fact, the programmer must tell the computer what form the output is to be in. He does this by supplying the computer with a "format" for the output. This format must be placed in a specific location in memory prior to calling for the output. When the output is called for, the computer will automatically, in this order:

1. copy the format from its location in memory into a special buffer, called the MZ buffer, where it can control the output, and

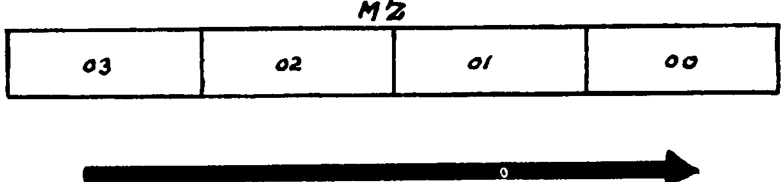
2. start the output.

The location for this format depends on the source of the information:

line 19 format: 02.00 - 02.03

AR format: 03.02 - 03.03.

When an output has been called for, and the proper format has been loaded automatically into the MZ buffer, inspection of the format will begin, and it will proceed in the direction shown by the arrow in the drawing below.



Inspection of the format will be from the high-order end of word 03 toward the low-order end of word 00. Each group of three bits will be inspected and interpreted as calling for some character of output, according to the following table:

Character	Code
digit	000
end	001
carriage return	010
period (point)	011
sign	100
reload	101
tab	110
wait	111

Example 1:

SDDPDDDWWTE

100 000 000 011 000 000 000 111 111 110 001

02.03 803007z-
02.02 1000000+

Example 2:

SDDDDDDDTSDDDDDDDTSDDDDDDDTSDDDDDDDCE

100 000 000 000 000 000 000 000 110 100 000 000 000 000 000 000 000

110 100 000 000 000 000 000 000 000 110 100 000 000 000 000 000 000

000 010 001

02.03 800000x
02.02 0000034
02.01 00000x0
02.00 0000110

If the output information being called for by the format is coming from line 19, the following rules apply to the inspection of the output format:

1. each sign code will cause the sign-bit (T1) of 19.u7 to be inspected, and the appropriate sign will be transmitted;

2. each digit code will cause the inspection of bits T26-T29 of 19.u7, and the proper hex digit will be transmitted, after which the entire contents of line 19 will be shifted toward the high-order end of the line by four bit-positions;

3. each carriage return code will cause a carriage return to be transmitted, after which the entire contents of line 19 will be shifted toward the high-order end of the line by one bit-position;

4. each tab code will cause a tab to be transmitted, after which the entire contents of line 19 will be shifted toward the high-order end of the line by one bit-position;

5. each wait code will cause a blank to be transmitted, after which the entire contents of line 19 will be shifted toward the high-order end of the line by four bit-positions;

6. each period (point) code will cause a period to be transmitted;

7. each reload code will cause the transmission, if tape is being punched, of a reload character, but, if the typewriter is being activated, nothing will be transmitted, after which the format will be reloaded and the inspection of the format will be resumed with the first code;

8. each end code will cause the output to cease, and the input/output system to go ready, as well as causing the transmission of a "stop code", if tape is being punched; but, before an end code is interpreted as an end code by the output system, that system will cause a check of all of line 19 for at least one non-zero bit: if no 1 is found, the end code will be allowed to operate as an end code, but, if a 1 is found, anywhere in the line, the end code will be interpreted as a reload code, as described above (7).

If the output information being called for by the format is coming from AR, the rules applying to outputs from line 19 apply, with the following exceptions:

1. all references to line 19 must be changed to refer to AR;

2. the end code will be interpreted as an end code, regardless of the current contents of AR.

You might wonder about the desirability of punching tape as an output, rather than typing the outputs. Punched tape output is useful for two purposes:

1. to keep a permanent, easily reproducible set of outputs, which can be reproduced, without using the computer, through use of a relatively cheap tape interpreter; and

2. as interim storage of results, to be used as inputs by the same program or another later on. The command to read tape is:

L L+2 N 0 15 31.

A tape input is the same as a typewriter input, but it is ended by the "stop code" already mentioned, rather than by activation of the "s" key.

Just as we converted decimal inputs to binary numbers for computer operation, so we must now convert the binary answers to their decimal equivalents, through another conversion subroutine, so that the outputs will be in decimal form.

If the specifications for the binary-to-decimal number conversion subroutine contain the following information, and if you have a binary answer ($x \cdot 10^{n}(2)$) in AR, the following sequence of commands, starting at word-time 10 in the main program in line 00, is designed to convert this answer to decimal form, and store the converted answer, ready for output, in 19.u7.

Execution.........................Command line 02
Entry.............................Word-time 61
Exit..............................Word-time 63
Input.............................x (binary) $\longrightarrow$ ID_1
Return command $\longrightarrow$ AR
Output............................$\pm|x|$ (decimal) in AR
(7 digits and sign)

L	P	T or L_k	N	C	S	D	BP	NOTES
10		11	12	0	28	25		AR $\longrightarrow$ ID_1
12		13	14	0	00	28		00.13 $\longrightarrow$ AR_c
13	[	65	64	0	20	31	]	Return command
14	w	70	61	2	21	31		Go to number conversion subroutine
70		u7	N	0	28	19		AR $\longrightarrow$ 19.u7

The output of this subroutine will be a decimal fraction, and, if we know the scale factor for the answer, we can properly position the decimal point (called for by a period code in the format) in the type-out of the answer.

In the above example, assume that the answer is $x \cdot 10^{-2}$. An output format for properly positioning the decimal point during the type-out would be:

S D D P D D D D D C E

100 000 000 011 000 000 000 000 000 010 001

When a program has reached the end of all that it is to do, and the only thing left is to stop, this can be accomplished by a special

command, called the "halt" command:

L L+2 N C 16 31.

The C in this command has no effect upon its interpretation or execution. A halt command may be given at any time, but, if it is given while an input or an output is in progress, the input/output system will continue to operate until the end of the process.

After the computer has been stopped by either a halt command or a break-pointed command, it will continue to operate under program control, with the next command taken from the location called for by the N of the command which halted the computer, if the compute switch is moved to "OFF", and back to either "GO" or "BP".

At this point we have covered the bulk of special commands which are available, although some remain unmentioned.

L	P	T	N	C	S	D	
L		T	N	0	23	31	Clear two-word registers. T must equal at least L + 3, since at least two word-times of execution are necessary. All 58 bits of ID, MQ, and PN, and the IP flip-flop will be cleared to 0.
L		56	N	0	24	31	Multiply, single-precision. The single-precision number in ID_1 will be multiplied by the single-precision number in MQ_1, and the product will occupy PN, while the sign of the product will occupy the IP flip-flop. If a single-precision product is desired, it is in PN_1. If a double-precision product is desired, it is in all of PN.
L		v4	N	0	24	31	Multiply, double-precision. The double-precision number in ID will be multiplied by the double-precision number in MQ, and the double-precision product will be in PN, while the sign of the product will be in the IP flip-flop.
L or L		57 57	N N	1 5	25 25	31 31	Divide, single-precision. The number in PN (if single-precision, in PN_1) will be divided by the number in ID (if single-precision, in ID_1), and the single-precision quotient will be in MQ_0, while the sign of the quotient will be in the IP flip-flop.

L	v6	N	1	25	31	Divide, double-precision. The double-precision number in PN will be divided by the double-precision number in ID, and the double-precision quotient will be in MQ, while the sign of the quotient will be in the IP flip-flop.
or						
L	v6	N	5	25	31	
L	T	N	1	26	31	Shift. ID will shift right, and MQ will shift left, for the indicated number of shifts. T = 2 times the number of shifts to be performed.
L	T	N	0	26	31	Shift under control of AR. ID will shift right, and MQ will shift left, for the indicated number of shifts. Shifting will cease at the end of execution time or after an end-around-carry has been generated in AR, whichever occurs earlier. 1 will be added to AR for each shift performed. Usually T will equal 54, allowing 27 shifts, which is the maximum that can be performed without shifting all bits of a word out of the word.
L	T	N	0	31	D	Extract 20·21. The bits from word T in 21 called for by the mask in word T in 20 will be transferred to word T in the destination. All other bits in word T in the destination will equal 0.
L	T	N	C	31	D	Extract $\overline{20}\cdot 21$. The bits from word T in 21 called for by the reverse of the mask in word T in 20 will be transferred to word T in the destination. All other bits in word T in the destination will equal 0.
L	T	N	C	30	D	Test word T in line S for non-0. The bits tested will depend on the C in this command. If none of the bits tested contain a 1, the next command will be taken from N. If any one of the bits tested does contain a 1, the next command will be taken from N + 1.
L	T	N	0	22	31	Test the sign bit of AR for negative. Only one word-time of execution is necessary, and the flag in T may be L + 2. If the sign of AR is positive, the next command will be taken from N. If the sign of AR is negative, the next command will be taken from N + 1.
L	T	N	0	28	31	Ready test. If the input/output system is not ready, the next command will be taken from N. If the input/output system is ready,

						the next command will be taken from N + 1. If it is desired to use this command in order to "hold up" a program from proceeding until the input/output system goes ready, both T and N should be set equal to L.
L		T	N	0	29 31	Test for overflow. Only one word-time of execution is necessary, so the flag in T may be set equal to L + 2. If the overflow flip-flop has not been set, the next command will be taken from N. If the overflow flip-flop has been set, the next command will be taken from N + 1. Execution of this command automatically resets the overflow flip-flop to the "off" condition.
L		T	N	C	16 31	Halt. This command needs only one word-time of execution, so the flag in T may be set equal to L + 2. The C in this command will have no affect on its operation. The computer will start a new sequence of commands at N, if, after it has halted, the compute switch is moved to the "off" position and then to either "GO" or "BP".
L or L	 w	T T	N N	C C	21 31 21 31	Mark, transfer control. Only one word-time of execution is necessary for this command. Program control will be transferred to line C, word N. The last word-time of execution of this command will be "marked", for use by a subsequent return command.
L		L+2	L+1	C	20 31	Return command. Program control will be transferred to line C at the marked word-time.
L		T	N	0	12 31	"Gate Type-in". Only one word-time of execution is necessary for this command, so the flag in T may be set equal to L + 2. The typewriter will be activated for input to the computer.
L		T	N	0	15 31	Read punched tape. Only one word-time of execution is necessary for this command, so the flag in T may be set equal to L + 2. One block of tape will be read into the computer.
L		T	N	0	06 31	Reverse punched tape. Only one word-time of execution is necessary for this command, so the flag in T may be set equal to L + 2.

						The tape will automatically be reversed, and positioned for the read-in of the last block previously read into the computer.
L	T	N	0	08	31	Type AR's contents. This command needs only one word-time of execution, so the flag in T may be set equal to L + 2. The type-out will be under control of the format contained in words 03.00 - 03.03.
L	T	N	0	09	31	Type line 19's contents. This command needs only one word-time of execution, so the flag in T may be set equal to L + 2. The type-out will be under control of the format contained in words 02.00 - 02.03.
L	T	N	0	10	31	Punch line 19's contents on tape. This command needs only one word-time of execution, so the flag in T may be set equal to L + 2. The punch-out will be under control of a format contained in words 02.00 - 02.03.
L	T	N	1	31	31	Copy number track into line 18. Any words may be copied, depending on L and T of this immediate command. To copy the entire number track into line 18, T should equal L + 1. Line 18 should be cleared prior to giving this command.
L	T	N	0	31	31	Take next command from AR. The next command will be read from AR at word-time N. Program control will return to the same line in which this command is located, for the succeeding command.

The commands we have discussed can be combined to constitute a program, and when this is done, PPR is used to enter the program into the computer. PPR takes each command and converts it to the binary form needed by the machine and places it in its proper location. PPR can also punch a block of tape containing the information in any long line in memory. It can accept hex constants and place them in their proper locations, and it can accept decimal constants, convert them to binary, and place them in their proper locations. It can read tape, accept corrections to the information from the tape, and produce a new, corrected tape.

Through certain auxiliary routines associated with it, PPR can help the programmer in checking out his program, it can automatically prepare output formats, and it can list, in decimal command form, all the commands in a program, either in the order in which they would be operated, or in the numeric order in which they are located.

In addition, PPR has other capabilities. The various tasks it can perform, and the way in which it is told to perform each of them, are listed and discussed in detail in the G-15 operating manual.

Although PPR can be used to enter a program into the memory of the computer, there must be a way of doing this that does not require PPR to already be in the memory of the computer. We know this, because PPR, itself, can be loaded into the computer when there is no useful information already in memory. It stands to reason that the same method by which PPR is originally loaded, could also be used to originally load any other program, under similar conditions.

This method is a "loader" program which operates in the manner described on pages 147 - 149.

Once we have used PPR to initially make up a program, we can give PPR an instruction to punch a block of tape containing this program, and we can precede this block of tape with another block, containing such a loader program. In this manner, any program prepared by PPR can be made self-sufficient, and PPR will no longer be necessary either to load the program into the memory of the computer, or to operate it.

Several times, in the preceding pages, reference has been made to information in the following portion of this manual. An attempt has been made, up to this point, to present a basic, yet fairly complete, picture of the G-15 and the methods to follow in programming it. The following pages present the same picture, in much greater detail, complete with some of the more exotic possibilities in utilizing the full powers of the computer.

COMMANDS IN BINARY FORM

Remember it has been pointed out several times that there is no difference in appearance between data numbers and commands in the computer; each form of computer word occupies 29 bits. We have already had a brief look at data words, both single- and double-precision; it is now time to consider the contents of commands. Commands occupy only single words; there is no such thing, in the G-15, as a double-precision form of a command, occupying 58 bits, although you will see the term "double-precision command" used. This term is used to refer to a command calling for an operation on a double-precision data number. In the 29 available bits, the following information must be specified:

1. operation,

2. address of operand,

3. address to which operand is to be transferred, and

4. address of next command to be obeyed.

29 28	22	21 20	14	13 12	11 07	06 02	01
ID	T	BP	N	CH	S	D	S/D

Specification of the operation requires three bits: 01, 12, and 13. Bit 01 indicates whether single- or double-precision operation is required; if it contains 0, single-precision is indicated; if it contains 1, double-precision is indicated. Bits 12 and 13 contain a two-bit code for the operation itself; this code is called the "characteristic (CH)".

00 - calls for a straight transfer of the operand from one location into another. After this operation has been performed, the operand, in its original form, will be in both locations in memory.

01 - calls for use of "inverting gates" during the transfer of the operand from one location to another. Inverting gates perform the complementation which has been previously described. The sign of the number is the first bit to be transferred. The inverting gates inspect it to determine whether or not it is a 1: if it is not 1, they allow the following magnitude bits to pass through unchanged; if it is a 1, they complement the following magnitude bits.

10 - calls for an exchange of numbers between memory and the one-word register, AR. After execution of a command calling for this, the specified receiving address will contain the original contents of AR, as the result of a straight transfer; AR will contain the specified operand, also as the result of a straight transfer. It obviously makes no sense to exchange AR with itself in this manner. Therefore, this characteristic has an entirely different meaning if AR is specified in the command as either the operand or the receiving address. Any other memory locations may be specified, but if PN is specified as either of the two addresses in the command, the line number 26 should be used, rather than the number 30, each of which, you remember, may refer to PN. In this way we have a rule governing the meaning of this characteristic: 10 calls for an exchange between memory and AR if neither the address of the operand nor the receiving address contained in the command is equal to, or exceeds, 28. Do not worry about line number 31; it is a special address, and it will be better to cover it later. We see, then, that the contents of AR and a memory location may be exchanged (the address of the operand = the receiving address), or the contents of AR may be transferred to one place in memory and AR may receive the contents of another, entirely different, word.

If AR is specified as the operand (AR is always referred to as 28 when it is the operand) in a command whose operation code is 10, the absolute value of the contents of AR is

transferred to the receiving address (28 bits of magnitude, less sign-bit). The result of the transfer will always be a positive number. Similarly, if AR is specified as the receiving address, it will receive only the absolute value of the operand. If, in such a case, AR is referred to as line 28, this absolute value will appear in AR, of course always positive. If AR is referred to as line 29, special circuitry which turns AR into an accumulator is called into action, and the absolute value of the operand (a positive number) will be added to the present contents of AR. The result may or may not be positive, depending on the previous contents of AR.

If line 30 (PN, as the double-precision accumulator) is specified as the receiving address, the magnitude of the operand will be added to PN. This will normally be a double-precision command. If it is not (it is, therefore, a single-precision command), 28 bits of the operand will be transferred to the receiving half (odd or even, depending on the address of the operand) of PN, and added to the present contents of that half of PN.

11 - if neither the specified address of the operand nor the specified receiving address is equal to, or exceeds, 28 (AR), an exchange of AR and memory similar to that called for by 10, discussed above, is performed. There is one important difference, however. Note that this characteristic is a combination of 10 and 01. If you remember that 01 called for use of the inverting gates (to complement negative numbers), you could make an informed guess that they are involved in this exchange of AR with memory. You would be right. The contents of the operand, on its way to AR, pass through the inverting gates, and the operand will be complemented if negative. The inverting gates will not be used for that part of the exchange that transfers the original contents of AR to memory. So, upon execution of this command, the original contents of AR will appear in memory, as the result of a straight transfer, and the contents of the operand, complemented if negative, will appear in AR.

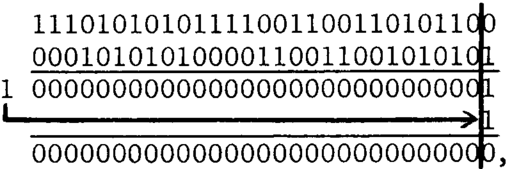
Again, there is a special meaning for this characteristic if AR is specified as either the sending or the receiving address, or if PN, referred to as line 30, is specified as the receiving address. In either of these cases, one covering single-precision operation, the other, double-precision operation, this operation will cause a subtraction, which is, in the terms of the computer, as already mentioned, a combination of changing the sign of the operand and then complementing the operand on its way to the receiving address, if necessary. If AR is specified as line 28 and as the receiving address, the operand, with changed sign and complemented if necessary will be transferred into AR. This we could call a "clear and subtract". If AR is specified as line 29 and as the receiving address,

the special circuitry which activates AR as an accumulator will be called into action, and the operand, so modified, will be added to the original contents of AR, which, in effect, is a subtraction. If PN is referred to as line 30 and as the receiving address, the contents of a double-precision operand will be subtracted from the original contents of PN. Notice there is no "clear and subtract" possible with PN; if this is desired, two commands will be necessary, one to clear it to 0, and another to subtract the desired operand from 0. If AR is referred to as the operand in a subtraction, the sign of the operand will be changed, and then the operand will be complemented if necessary on its way to the receiving address.

Notice here that one way to clear either of the accumulators (AR or PN) would seem to be to subtract its contents from itself. A - A = 0. Since they behave in similar fashion, we will consider here only AR, thus limiting the examples to 29 bits, rather than 58. If the number contained in AR is positive,

11101010101111001100110101100,

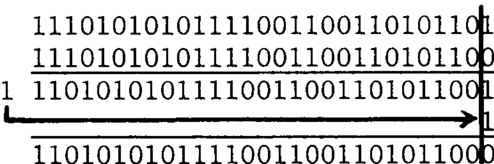
and we subtract it from itself (change the sign, complement if the new sign is negative, and add),



we're in good shape; we get what we expect. AR is cleared to 0. But, if AR is originally negative,

11101010101111001100110101101,

and we subtract it from itself (change the sign, complement if the new sign is negative, and add),



we're in terrible shape. We expected 0, and didn't get it.

You see, there was a basic assumption underlying the suggestion that subtracting a number from itself in AR would clear AR to 0. That assumption was that, when a 28-bit magnitude is added to its 28-bit complement, 28 0's must result with an end-around-carry of 1 into the sign position. Since the

addition of positive and negative sign yields 1 in the sign position, when the end-around-carry of 1 is added to the result, a positive sign (0) is obtained. Thus, remembering that there is no carry from the sign position into the least significant magnitude position, in such a case, 29 0's (+ 0) must result. The fallacy in our original suggestion was that, because two magnitudes were being added together, one with a negative sign and the other with a positive sign, we assumed that a magnitude and its complement magnitude would be added. We never did get the complement, however, if our original number was negative. When its sign was changed, during the subtraction process, a positive sign resulted, and the inverting gates allowed the number to pass through, unmodified.

Of course we can always make sure that AR contains a positive number to begin with, by transferring the contents of AR into itself with a characteristic of 10, calling for absolute value. Now AR can be cleared by subtracting its contents from itself.

We have now covered all of the possible combinations that can be squeezed out of the three bits in a G-15 command that specify the operation. For the sake of ease in remembering these, we'll assign a corresponding decimal number, called a "C" code, to each one, as shown below.

"C"	S/D	CH	Meaning
0	0	00	Straight single-precision transfer.
4	1	00	Straight double-precision transfer.
1	0	01	Single-precision transfer via the inverting gates.
5	1	01	Double-precision transfer via the inverting gates.
2	0	10	If 28, 29, 30 or 31 not specified, transfer contents of AR to receiving address, operand to AR.*
2	0	10	If 28, 29 or 30 is specified, transfer absolute value of operand to receiving address.
6	1	10	If 28, 29, 30 or 31 not specified, transfer contents of AR to first word of specified double-precision receiving address, first word of double-precision operand to AR. Then transfer the present contents of AR to second word of specified double-precision receiving address, and the second word of double-precision operand to AR. *

"C"	S/D	CH	Meaning
6	1	10	If 30 is specified as the receiving address, transfer a double-precision absolute value (57 bits).
6	1	10	If 28 is specified as operand, transfer absolute value from AR to first half of double-precision address, then transfer all 29 bits from AR, treated this time as most significant half of a double-precision magnitude, to the second word of the double-precision receiving address.
6	1	10	If 28 or 29 is specified as receiving address, transfer absolute value of least significant half of double-precision number to AR, then transfer the most significant half of the double-precision magnitude (all 29 bits) to AR.
3	0	11	If 28, 29, 30 or 31 not specified, transfer contents of AR to receiving address, and operand via inverting gates to AR.*
3	0	11	If 28, 29 or 30 is specified, change sign of operand, then transfer operand with new sign, via inverting gates, to receiving address.
7	1	11	If 28, 29, 30 or 31 not specified, perform same transfers as for operation code 6 under these conditions, except that all numbers transferred to AR are transferred via the inverting gates. *
7	1	11	If 30 is specified as the receiving address, change sign of double-precision operand and transfer it to the receiving address, via the inverting gates.
7	1	11	If 28 is specified as the operand, transfer the number in AR with its sign changed, and complemented if necessary, to the first word in the double-precision receiving address. Then transfer all 29 bits from AR, treated as the second 29 bits of a double-precision number, and comple-

"C"	S/D	CH	Meaning
			mented, if that was called for originally, to the second word of the double-precision receiving address.
7	1	11	If 28 or 29 is specified as receiving address, change sign of first word of double-precision operand and transfer it via the inverting gates to AR. Then transfer the second word, all 29 bits, via the inverting gates also, to AR.

* Note: <u>If this table is being used for reference</u>, see page 82.

Notice here that the operations listed so far do not constitute the complete list of operations available in the computer; to name two very important ones which were not included, <u>multiply</u> and <u>divide</u>. We will see later how these operations, and many others, are called for, but first we must complete the description of the basic parts of a command.

So far all we have discussed are operations. Another part of each command is the address of the operand: the computer must be told on which number in its memory the operation is to be performed. Remember that it has been pointed out that an exact address in memory consists of a line number plus a word number, called a word-time. The line in which the operand is located is called the "source", and, in the layout of a command on page 61, this number is referred to as "S". If the line containing the operand is called a source, the obvious name for the line which will receive the transferred word is "destination". This is referred to in the layout of a command as "D". S and D can each range from 00 through 31, although the meaning of 31 has not yet been explained. In binary (all words in memory are in binary), 5 bits are required to represent the decimal number 31.

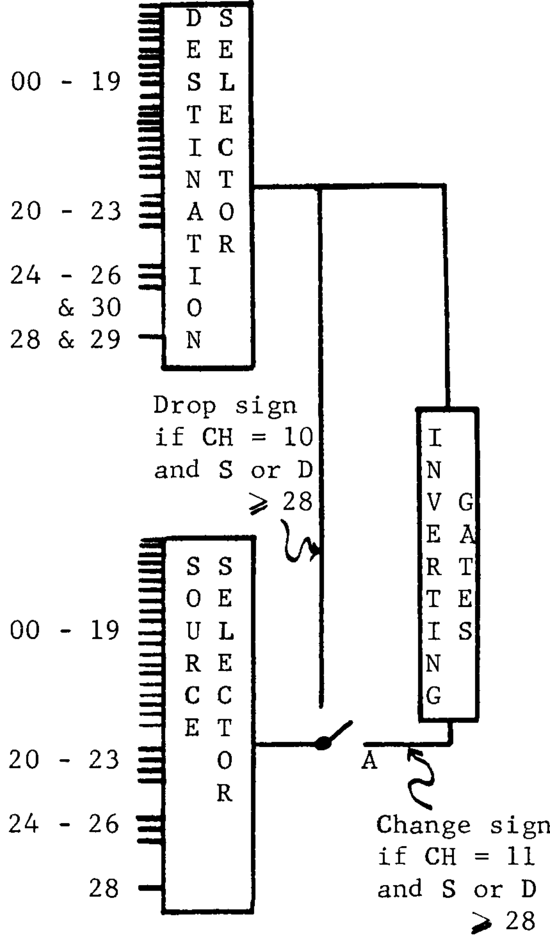
$$31_{(10)} = 11111_{(2)}$$

Notice that five bits have been allotted to both S and D. Since no address is complete without a word-time, there must be an allocation for this in a command. There is; it is "T". There are 108 word-times per long-line. No other line in memory requires more; in fact, no other line in memory requires nearly as many, so the range 00 - 107 (u7) will be sufficient.

$$107_{(10)} = 6v_{(16)} = 1101011_{(2)},$$

so seven bits will be sufficient for T; notice that seven have been allowed. Now, is T combined with S to form an address, or with D? The answer is: with both! So we see that, if the straight transfer

of a single word from one long line to another is called for, the word being transferred will, upon execution of the command, occupy the same word-time in the new line as it does in the old. The drawing below will clarify this.



As long as no command is being executed, the read-heads for each line in memory (there are 28 of them plus the number track) are connected to the corresponding write-heads, and bits, words, and whole lines are recirculating merrily along. This is all going on behind the drawing to the left. But then, when a command is exeuted, things begin to happen. If no commands were ever executed, it would be an easy matter to understand what's going on in the computer, but computers, as a rule, don't do much of anything useful if no commands are being executed. When a command is executed, it has a source (S), a destination (D), a word-time (T) during which it is to be executed, and an operation (CH and S/D), and other things we won't mention here. During word-time T a word is entering the source selector from each line in memory, as shown in the drawing. Notice that only one line number for PN is shown (26, not 30); this is because PN can be a source of data in only one capacity (as a storage location).

In the case of AR, 29 is an illegal line number for AR as a source. AR must be referred to as line 28 when being treated as a source. The source selector now has a word from each line, and it must pick out the correct one and ignore the rest. Of course it can do this, because the command has informed it of the line specified by S. The word from memory whose address is S.T leaves the selector, on its way to be processed.

Processing actually consists of being transferred back to memory again, over the proper circuit, which, in some cases, will perform an addition of bits. It also may consist of complementing a negative number. Part of the processing will be called for by the CH in the command. Depending on the value of CH, switch A in the drawing may be in one position or the other, causing the word being transferred to pass around or through the inverting gates. The word then arrives at the destination selector. The destination selector will usually disconnect the read-head for the line chosen as D from the corresponding write-head, pre-

venting recirculation of word T in that line. It will allow all other read-write connections to remain intact, so that word T in all other lines is being recirculated. It will then take the word it has received from the source selector and feed it to the proper line. Thus, what was originally in the destination line at word-time T is lost and replaced with word T from the source line. The source line has recirculated and still contains word T. If the destination is either 29 or 30, the original contents of the destination line at word time T is not lost, but is added to word T arriving from the destination selector.

In order to hold down the size of this book to a single volume, we will leave it to the reader to trace through this procedure for each operation code listed in the preceding table.

IMMEDIATE vs. DEFERRED COMMANDS

A series of commands could be written to perform any of these operations on a sequance of words; S, D, and the operation could be the same in each of the series of commands, with T being increased by 1, in the case of single-precision operation, or by 2, in the case of double-precision operation, in each succeeding command. As an example, a straight single-precision transfer of 04.10 to 05.10 would be coded with a "C" code of 0, a source of 04, a destination of 05, and T = 10. This could be followed by another command with the same "C" code, the same source, the same destination, but with T = 11. Then T could be increased by 1 again, and so on. Up to a whole long line could be transferred, one word at a time, in this way. By this method, it would require 108 commands to transfer 108 words. There is a way of accomplishing this with one command: it is to code the command in such a way that its execution will cover any desired number of contiguous word-times. In the layout of a command, as shown on page 61, bit 29 of the command is a one-bit indicator called "I/D". The "I" stands for "immediate". An immediate command is one which will be executed immediately after it is read and interpreted. Its execution will continue until the computer is told to stop the execution. The T number in such a command serves as a "flag", telling the computer when to stop the execution of the command. The execution will be stopped before T, but after the immediately preceding word-time. To indicate immediate execution of a command, bit 29 of the command is set with 0. In the above example, if it was desired to transfer words 10 - 15 from line 04 to line 05, a command with a "C" code of 0, S = 4, D = 5, T = 16, and I/D = 0, could be located at word 09 of some line out of which it would be read. Which line of memory the command would be in is as yet an open question. The command would be read at word-time 09 and executed immediately, meaning that its execution would start in the very next word-time, 10. It would continue operating through 10, 11, 12, 13, 14, and 15. The T number of 16 would serve as a flag, stopping the operation after word-time 15 and before word-time 16.

An important point to note in the discussion of immediate commands is that an immediate command must execute for at least one word-time before the flag can be effective. If, in the previous example, the immediate command located at word 09 had a T of 10, the flag could not

stop the operation until a complete drum cycle had elapsed, and word 10 was coming up for the second time. This, then, would be the way in which one command could cause the transfer of one whole long line to another: let T of the command be 1 greater than the location of the command itself, and let the command be an immediate command calling for the straight transfer of words.

Any of the previously discussed commands, either single- or double-precision, can be made immediate by setting bit 29 of the command equal to 0.

If an immediate command is not desired, or, to put it another way, it is desired that operation be deferred until some particular word-time, and then be performed for that word-time only, bit 29 of the command must be set to 1, indicating deferred (D) operation. Of course a deferred double-precision command will still obey the rules for double-precision operations: namely, the operation will continue until the next sign-time, which will be two word-times later. In other words, making a double-precision command deferred does not alter the fact that two contiguous words will be operated upon; it merely pinpoints the two words. An immediate double-precision command will operate on contiguous two-word numbers until stopped by a flag.

The immediate commands which can be made from each of the operation codes discussed so far are often referred to as "block" operations, since they operate on blocks of numbers.

SEQUENCING OF COMMANDS

As shown in the layout of a G-15 command on page 61, each command contains in itself the address of the next command (N) to be obeyed, and the word-time portion of this address is in bits 20 - 14. Notice that seven bits are allotted and are sufficient to express any word-time in memory. Commands, like data, may occupy any word. An address consists of more than a specified word-time, however; a line number must also be included. In the case of N, in a command, the line-number is implied to be the same as the number of the line in which the current command is located. In other words, the G-15 will continue looking for commands in the same line, once it has started with a command in that line. Since this is the case, it is only necessary to specify the word-time at which the next command is located in the same line.

COMMAND LINES

Not all lines in memory are connected to the special circuits which interpret commands. Any line which is so connected is called a "command line", and commands located in it can be read and executed. The command lines are 00, 01, 02, 03, 04, 05, 19, and 23. A command can also be executed out of AR, but this special action by the computer must, in turn, be called for by a special command, which will be discussed later. In order to preserve numerical continuity in all references to command lines, line 19 is referred to as command line 06, and line 23 is referred to as command line 07. AR, because of its special nature

in this regard, is not referred to as a command line. Once a command line has been chosen, the computer will continue to obey commands in that line, but how does a command line get chosen originally? What happens when a program must occupy more than one line? These are logical questions, and we will look into their answers just as soon as we complete the discussion of commands, themselves.

The only bit in a command word which remains unmentioned at this point is bit 21. You may now consider it mentioned, although this would be the wrong time in the discussion to describe its function. For our purposes at present, always assume it contains 0.

So far, although many computer operations have been discussed, they do not include all of the operations we will need for the solution for the quadratic equation. Multiplication and division are just two of the operations not supplied through the normal operation codes. It has been pointed out that, although there is no line 31 in the memory of the G-15, this number may be placed in a command as either the source (line) or the destination (line). If 31 is specified as either S or D in a command, the computer will know that no ordinary transfer is being called for.

SPECIAL COMMANDS

Upon discovery of D = 31 in a command, the computer will treat this command as a "special" command, and interpret it in a special way. The S number will be treated as a special operation code, and the three bits which normally specify the operation will usually be interpreted in the light of the special operation called for.

In the example of the quadratic equation, all additions and subtractions can be performed by using normal operations, but the other operations necessary, of which multiplication and division are two, will require special commands.

MULTIPLICATION AND THE TWO-WORD REGISTERS

The multiply command contains: D = 31, S = 24, and "C" code = 0. Before this command is executed, however, the proper numbers to be multiplied together must be in the two-word registers ID and MQ, as mentioned before. Therefore, the multiply command must be preceded by two other commands in the program, which load these two registers. The product, after multiplication, will appear in PN. The programming method for performing a multiplication can be derived from a further study of the two-word registers and how they operate.

Any two-word register can be loaded with either a single-precision number (via a single-precision transfer) or a double-precision number (via a double-precision transfer), but the two-word registers will always word in double-precision when a multiplication is called for. Two 57-bit magnitudes will be multiplied together. If a single-precision multiplication is really desired, it can be achieved by only loading the most significant bits of ID and MQ, making sure that the remaining, least-significant bits are cleared to 0. A 56-bit product (to be expected

when two 28-bit numbers are multiplied together) will appear in PN in double-precision form. If a single-precision product is desired, it will be in the most significant word of PN. So, in the case of a single-precision multiplication, the two-word registers must be cleared to 0 before they are loaded with the multiplier and multiplicand. Of course the product will be the same, regardless of which of the two numbers is treated as the multiplier and which as the multiplicand.

The G-15 is internally wired in such a way that each bit (of the 58 bits) in PN may be cleared as the corresponding bit in ID is set.*
Therefore, if all 58 bits of ID are set, regardless of how they're set, prior to a multiplication, all 58 bits of PN will automatically be cleared, and PN will be ready to receive the product. The setting of MQ will affect no other register, nor will it be affected by the setting of any other register.

In a multiplication, although the magnitudes of the two numbers are to be multiplied, we know that the signs must be added, if the laws of signs are to be obeyed. A product is usually worthless if it contains the wrong sign. The G-15 knows this, too. Therefore, when the two-word registers are being loaded, via a normal operation (transfer),

if the "C" code is even, (0, 2, 4, 6), the sign of the number is divorced from the magnitude and sent to a special "flip-flop" associated with the two-word registers, called IP. A flip-flop is a two-state device, one state equalling 0, the other equalling 1, and it can remember which state it is in. It can also be read, or "sensed", to determine which state it is currently in. The bit in the two-word register which would normally receive the sign will not; it will be set to 0. When ID is loaded, IP will be set with the sign of the number going into ID. When MQ or PN is loaded, the sign of the number being transferred will be added to the present value of IP, and the result will remain in IP. Similarly, when a number is transferred, via a normal operation, out of a two-word register, and the "C" code is even, the magnitude will come from the register specified as S, but the sign will come from IP. This function of IP is automatic. The only special precaution the programmer must take in order to insure its operation is to transfer numbers to and from the two-word registers with even "C" codes. So, in the setting of ID and MQ prior to a multiplication, the program will have to set ID first, then set MQ, thus insuring the correct sign of the product in IP. Then the multiply command may be given.

* Note: this feature is automatic if ID is set with any even C code (0, 2, 4, 6).

When the computer is commanded to multiply, the following will be the state of affairs in the two-word registers:

ID - Multiplicand

MQ - Multiplier

PN - cleared to 0 and ready for product

IP - correct sign (0 or 1) of product

It has been stated that the two-word registers will multiply in double-precision fashion, regardless of whether or not double-precision operation is really desired. Remember that, in double-precision numbers the most significant bits are in the odd-numbered word (in the case of the two-word registers, we refer to these as ID_1, MQ_1, and PN_1). All 29 bits are magnitude bits. 28 of the bits in the even-numbered word (ID_0, MQ_0, and PN_0) are the least significant bits of the magnitude, and the sign-bit of this word is the sign of the number, or 0, if the sign went to IP.

In the case of double-precision multiplication, then, we would want the initial conditions to be as follows, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0
PN:	00000000000000000000000000000	00000000000000000000000000000
IP:	0 or 1, whichever is the correct sign of the product.	

To transfer the double-precision multiplicand from its resting place in memory to $ID_{0,1}$, we would use a straight double-precision transfer (C = 4), with D in the command equal to 25 (ID). Because the C code is even, the sign will be disengaged from the magnitude, and sent to IP. Because ID is the destination, IP will be loaded with this sign. Then, to load $MQ_{0,1}$, we would transfer the double-precision multiplier, also with a C = 4, with D in the command equal to 24 (MQ). Because the C code is even, the sign will be disengaged and sent to IP. Because MQ is the destination, IP will add this sign to its present contents, and the result, which will appear in IP, will be the correct sign of the product. When the signs are disengaged, the bits in the two-word registers which would normally have received them are cleared to 0, as shown above. When ID is loaded (each of the 58 bits is set with some value, replacing what was originally there), each corresponding bit (and therefore, all 58 bits) of PN is cleared to 0. Thus the desired initial conditions will be achieved through the execution of two commands, the first of which loads ID, the second, MQ.

In the case of single-precision multiplication, we would want the initial conditions to be as follows, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

To transfer the single-precision multiplicand from its resting place in memory to ID_1 (the most significant half of the two-word register), we would use a straight single-precision transfer (C = 0), with D in the command equal to 25 (ID). Because the C code is even, the sign will be disengaged from the magnitude, and sent to IP. Because ID is the destination, IP will be loaded with this sign. Then, to load MQ_1, (again, the most significant half of the two-word register), we would transfer the single-precision multiplier, also with a C = 0, with D in the command equal to 24 (MQ). Because the C code is even, the sign will be disengaged and sent to IP. Because MQ is the destination, IP will add this sign to its present contents, and the result, which will appear in IP, will be the correct sign of the product. When the signs are disengaged, the bits in the two-word registers which would normally have received them are cleared to 0, as shown above. These are bit 1 in both ID_1 and MQ_1 (remember, this is a single-precision transfer from an odd-numbered location in memory, thus congruent to 1 mod 2, to word 01 in ID and MQ). When ID is loaded (the first 29 bits are set with some value replacing what was originally there), each corresponding bit (and therefore, the 29 bits of PN_1) of PN is cleared to 0. The desired initial conditions will not be totally achieved unless something further is done to clear the remaining bits of ID, MQ, and PN. It will do no good to follow the two commands which load ID_1 and MQ_1 with two more which load ID_0 and MQ_0, because this will cause the resetting of IP, and destroy the sign of the product. But the two-word registers could be cleared first, and then ID_1 and MQ_1 set with the desired numbers. This is the procedure to be followed, and there is a special command which clears all bits in the two-word registers and also clears IP to 0. Since it is a special command, it has D = 31. Since it is going to operate on two-word registers, it must operate for two word-times. Its source (S) = 23. The C code = 0. Since the C code = 0, it will operate for only one word-time, unless it is made immediate by setting the I/D bit in the command = 0. The T number in this command, as in any other immediate command, then becomes a flag, telling the computer when to stop execution of the command. Remember that an immediate command will be executed starting in the word-time after that in which the command was read, and it will continue to be executed during each succeding word-time up through the one immediately preceding the flag. It will not be executed during the word-time whose number equals the flag. If we refer to the location of a command as L, then, in the case of the

command which clears the two-word registers, T = L + 3. In order to simplify the writing of flags for T numbers, we drop the plus sign, and use the desired number to be added to L as a subscript for L. In the case of the command we are presently considering, then, T = L_3.

Three commands, then, are necessary to establish the desired initial conditions for what we might call a single-precision multiply, although that really is a misnomer. The first will clear the two-word registers and IP, the second will load ID_1, and the third will load MQ_1.

The special circuitry associated with the two-word registers does essentially two things. We have already seen that it enables PN to act as an accumulator. The other feature accomplished through this special circuitry is a "shifting" process. A shift is the movement of bits toward the high-order or the low-order position within a register. In the G-15 it is accomplished one bit-position at a time. ID shifts toward the low-order (T1) position (this is usually referred to as shifting to the right). MQ shifts toward the high-order (T29) position (this is usually referred to as shifting to the left).

Multiplication involves both the shifting and the additive features of the two-word registers, in the following way. The contents of ID are shifted right by one bit-position, moving all 57 magnitude bits to the right one place. The right-most bit (T2 of ID_0) is lost. The left-most bit-position (T29 of ID_1) is filled-in with a 0. Simultaneously MQ is shifted left by one bit-position, moving all bits to the left one place. The left-most bit enters an inspection station, where it is inspected for 1 (it will, of course, be either 1 or 0). The right-most bit-position is filled-in with a 0. After such a simultaneous shift, during a single-precision multiplication, ID and MQ would contain:

	Word 1	Word 0
ID:	0xxxxxxxxxxxxxxxxxxxxxxxxxxxx	00000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxx00	00000000000000000000000000000

Compare these with the initial conditions shown on page 73.

If the bit from MQ which is inspected is a 1, the new contents of ID are added to PN; if it is a 0, the addition is not performed. The first addition in PN will, of course, be to 0, since PN was initially cleared. This process requires two word-times; because it is essentially a double-precision process, it must begin with an even word-time. It can be repeated as often as desired (28 times for a full single-precision multiplication). The multiply command must be immediate, and it will perform the process over and over again, for the indicated number of word-times. T in the command is a "relative timing number". It will be set equal to the desired number of word-times of execution of the command; this should be an even number, and the execution should begin at an even word-time, requiring the immediate multiply command to be located at an odd word-time. If the

process is allowed to continue for 28 times (T = 56), two full single-precision words can be multiplied together, and their product, a series of sums, will appear in PN. Notice that at least one shift is performed prior to the first addition, and the product will actually occupy the 56 most significant bit-positions in PN. In any number system, if two 28-digit numbers are multiplied together, a 56-digit product, counting any leading 0's, will result. If the initial shift in the computer were not performed, it would be possible, in the case of large numbers, to generate an overflow and an erroneous result.

After a single-precision multiplication, the most significant bits of the answer will appear in PN_1, bits 29 - 2. Bit 1 of PN_1 and bits 29 - 3 of PN_0 will contain the least significant bits of the product. Assuming that a single-precision product is all that is required, the least significant bits are merely excess accuracy, and can usually be ignored.

In ordinary pencil-and-paper multiplication, if you were to multiply two 28-bit numbers, you would inspect the multiplier from right to

```
                            1111111111111111111111111111
                            1100101010000000001111000001
                            ----------------------------
                            1111111111111111111111111111
                      1111111111111111111111111111
                     1111111111111111111111111111
                    1111111111111111111111111111
                   1111111111111111111111111111
          1111111111111111111111111111
        1111111111111111111111111111
      1111111111111111111111111111
   1111111111111111111111111111
  1111111111111111111111111111
--------------------------------------------------------
11001010100000000011110000000011010101111111110000111111
```

left, one bit at a time. If you found a 1, you would add the multiplicand to what you already had in the way of a partial sum. If you found a 0, you would not add the multiplicand. You would then shift the multiplicand to the left one place, and inspect the next bit in the multiplier. You would do this 28 times, once for each bit in the multiplier, and you would generate, as a result, a sum, which represents a product of the two original numbers. The computer does the same thing, in reverse. It starts with the high-order end of the multiplier and inspects toward the low-order end. The shifts are exactly the reverse, therefore; you shifted to the left, but the computer shifts to the right. A double-precision multiply command causes exactly the same sequence of events, but the relative timing number (T) in the command is set to allow the process to continue for 57 times (T = 114). Notice that the multiply command may be allowed to operate for any number of times, merely by setting T = 2k, where k = the number of times desired. The resultant product will always be predictable.

DIVISION AND THE TWO-WORD REGISTERS

Division is somewhat similar to multiplication, in that it also utilizes the shifting and additive features of the two-word registers in order to reach a result.

The divide command contains: D = 31, S = 25, and C = 1 or 5 (these are interchangeable: the setting of the S/D in the command has no bearing on the operation). As in the case of multiplication, the numbers to be divided must be set up in the two-word registers. The rules governing the initial set-up of the two-word registers apply here as well as in the case of multiply, except that the denominator will be loaded into ID, the numerator into PN, and the quotient will appear in MQ. Because PN is cleared as ID is set, the denominator must be loaded first, then the numerator. In order to clear MQ, preparatory to receiving the quotient, the clear command will have to be given. The proper sign of the quotient will be generated in the same manner as it is for a product.

When the computer is commanded to divide, the following will be the state of affairs in the two-word registers:

ID - Denominator

PN - Numerator

MQ - cleared to 0 and ready for quotient

IP - correct sign (0 or 1) of quotient

In division, as in multiplication, the two-word registers will operate in double-precision fashion. The most significant word is the odd-numbered word (ID_1, PN_1, and MQ_1). If single-precision division is required, the single-precision denominator usually will be in ID_1,* bits 29 - 2 (remember its sign will be in IP), followed by insignificant 0's in bit 1 of ID_1 and all 29 bits of ID_0. Wherever the denominator is in ID*, the single-precision numerator should be similarly positioned in PN.

In the case of single-precision division, usually we want the initial conditions to be as follows *, where x's represent significant bits of magnitude.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the quotient.

* The reason for making this indefinite statement will follow a description of the machine's divide process.

In order to get MQ cleared, the first command in the set-up for a division would be the clear command. This would be followed by two straight single-precision transfers from memory to ID_1 and PN_1, in that order. The signs of these two numbers will be disengaged and sent to IP, where the proper resultant sign will be generated. 0's will occur in the bits in ID (bit 1 of ID_1) and PN (bit 1 of PN_1) which would normally have received the signs.

When the desired initial conditions have been established, the divide command may be given. In order to understand the computer's division process, we must first inspect the pencil-and-paper method, to determine what process is involved there; we're so used to doing it, that we usually don't consciously analyze the process as we do it, but there is an underlying, reasonable pattern to the process of "long division". Consider the following long division in binary arithmetic.

```
              0101000
       101000/011001000000
              000000
              0110010
               101000
               0010100
                000000
                 101000
                 101000
                 0000000
                  000000
                  0000000
                   000000
                   0000000
                    000000
                    000000
```

In division we are attempting to find the ratio of one number to another. We call one of these numbers the denominator, and the other, the numerator. The resultant ratio, which we call a quotient, is the ratio of the numerator to the denominator: N/D.

The first step is to subtract the denominator, in its present form ($D \cdot 2^0$), from the numerator. We find that this yields a negative result. (In the decimal system we would inspect each possible multiple of D, starting with 9·D, but, in the binary system, the only possible multiples of D are 1·D and 0·D). We therefore discard the coefficient of 1, and say that N contains 0·D plus a remainder, R_1. We now shift D to the right one place (in the binary system, this yields $D \cdot 2^{-1}$), and attempt to subtract it from this remainder. This remainder, of course, equals N, since $N - 0 = R_1$. In effect, what we are doing, knowing that N does not contain D, is attempting to discover whether or not N contains D/2.

$$N - 1 \cdot D \cdot 2^0 = -r_1$$
$$N - 0 \cdot D \cdot 2^0 = +R_1$$

$$R_1 - 1 \cdot D \cdot 2^{-1} = +R_2$$

$$R_2 - 1 \cdot D \cdot 2^{-2} = -r_3$$
$$R_2 - 0 \cdot D \cdot 2^{-2} = +R_3$$

$$R_3 - 1 \cdot D \cdot 2^{-3} = +R_4$$

It does, and we know that because we get a positive result after the subtraction. R_1 contains $1 \cdot D \cdot 2^{-1}$ plus a remainder, R2. We continue this shifting and subtracting process until we arrive at a remainder of 0 or until we achieve the desired accuracy in the resultant quotient.

$$N = 0 \cdot D \cdot 2^{0} + R_1$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + R_2$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + 0 \cdot D \cdot 2^{-2} + R_3$$

$$N = 0 \cdot D \cdot 2^{0} + 1 \cdot D \cdot 2^{-1} + 0 \cdot D \cdot 2^{-2} + 1 \cdot D \cdot 2^{-3}$$

The reason we have taken a close look at the way you divide is that, contrary to popular belief, the designers of digital computers are "just plain folks"; they think the way you do, and when they were faced with the problem of designing a division operation for the Bendix G-15, they followed the same reasoning we have followed here. They noted one important exception to it, however, from the standpoint of the computer: the computer cannot "inspect" prior to a subtraction; it must subtract, and then inspect the result. Since the numerator is in PN, and since the subtraction will also be performed in PN, it is obvious that, after the subtraction, the original numerator will be lost in any event, and either a positive or a negative remainder will be in PN. The computer will have to be able to determine its future course on the basis of the sign of the remainder in PN. The bit that goes in the quotient is easily determined: if the sign of the remainder is negative, a 0 goes in the quotient; if the sign of the remainder is positive, a 1 goes in the quotient. If the remainder is positive, there is no problem: the denominator must be shifted right one more place and a new trial subtraction performed. But, in the case of a negative remainder, the problem is a bit more difficult. We know that the division yielded a 0 at this point, and the remainder actually indicates the quantity by which the denominator exceeded the numerator (or previous remainder, if this is not the first subtraction). If we shift the denominator right one more time, obtaining 1/2 its previous value, and add this to the negative remainder, we will know whether or not an original subtraction of D/2, rather than D, from N would have yielded a positive result. ($N - D + D/2 = N - D/2$). In short, we can devise the following rule: subtract D from N; if the result (R_1) is positive, place a 1 in the quotient, and subtract D/2 from R_1, continuing the process. If the result is negative, place a 0 in the quotient, and add D/2 to R_1, continuing the process.

The designers worried about one other point: the necessity for carrying many insignificant trailing 0's along with N, in order to perform the long division process. They realized that, after subtracting D from N, and arriving at a remainder R_1, the ratio of R_1 to D/2 is the same as the ratio of $2 \cdot R_1$ to D ($R_1:D/2::2 \cdot R_1:D$). Therefore, rather than shift D right to obtain D/2, they decided to shift R_1 left to obtain $2 \cdot R_1$, and do this successively, with each remainder, always adding (in the case of a negative R) or subtracting D (in the case of a positive R) from the new value. Although it seems that overflow

might be caused by shifting a remainder to the left (if the remainder has a 1 in the most significant bit position prior to the shift), this will not cause overflow, because of the manner in which the numbers are treated by the circuitry employed during a divide operation. Any temporary overflow condition will right itself in the next step of the continuing process. Such a temporary overflow will not set the overflow indicator. The algorithm upon which this division process is based is that N will never equal or be greater than 2·D. The long division in binary, as shown on page 77, will look like the following, as it is performed by the computer.

```
                0101000
 101000/011001  N
     sub 101000  D
   0) -  001111  R1
      -  011110  2·R1
     add 101000  D
   1)    001010  R2
         010100  2·R2
     sub 101000  D
   0) -  010100  R3
      -  101000  2·R3
     add 101000  D
   1)    000000  R4
         000000  2·R4
     sub 101000  D
   0) -  101000  R5
       -1010000  2·R5
     add 101000  D
   0) -  101000  R6
       -1010000  2·R6
```

Notice that the overflow caused by $2 \cdot R_5$ is corrected by the next addition. This is a temporary overflow.

At the beginning of the division process, MQ is shifted left one bit-position, while D is subtracted from N. MQ, then, if it were not cleared prior to the division, would look like this, where Y's represent the original contents of MQ.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY0

In the case of division, although MQ shifts left, ID does not shift right, so $D \cdot 2^0$ remains in ID. R_1 is inspected: if it is positive, a 1 is placed in T2 of MQ_0; if it is negative, a 0 is placed in the same bit. PN, containing R_1, is shifted left one bit-position, so that it now contains $2 \cdot R_1$. The sign of R_1 is used to control the inverting gates during the next transfer of D to PN for addition or subtraction. (Notice that D will pass through the inverting gates because the C code of the divide command contains a 1.) If the sign of R_1 is positive, it will be reversed and combined with D from ID on the next pass, so that, as D passes through the inverting gates on its way to PN, the effect will

be to subtract D from $2 \cdot R_1$. If the sign of R_1 is negative, it will be reversed, combined with D from ID, and cause the addition of D to $2 \cdot R_1$. A complete step such as the one described above will require two word-times, since division is essentially a double-precision operation, even though single-precision numbers may actually be involved. The next step in the process will begin with the shifting of MQ left by one bit-position again, so that the first bit in the quotient will occupy T3 of MQ_0, and T2 will be ready to receive the next bit. During the second step, $2 \cdot R_1$ will be in PN, and D will be added to, or subtracted from it. This process will continue for as many word-times of execution as are allowed by the divide command. The command will be immediate, and the relative timing number in T will be set to allow 57 word-times of execution (T = 57) for a single-precision divide.

After 56 word-times of execution, at 2 per step in the division process, 28 bits of quotient will be generated in MQ_0, and MQ will look like this, where Y's represent original bits, and x's represent quotient bits.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0

If only 56 word-times of execution are allotted, the first x in the drawing above (in T29 of MQ_0) will represent $x \cdot D \cdot 2^0$, while the remaining bits in MQ_0 will represent a fractional quotient. If a 57th word-time of execution is called for, during that word-time MQ_0 will be shifted left one bit-position, and a new bit will be placed in T2 of MQ_0, so that MQ will look like this.

	Word 1	Word 0
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0

Notice that MQ_1 did not move, while MQ_0 did. The first bit in MQ_0 $(x \cdot D \cdot 2^0)$ is shifted into a flip-flop which detects overflow. The bit now in T29 of MQ_0 represents $x \cdot D \cdot 2^{-1}$, and the whole word is a fractional quotient. This is the normal form for a ratio, and it is the form most desirable when programming the G-15. Overflow will be indicated if the quotient actually equals or exceeds 1, since, in that case, a 1 will reach the overflow flip-flop during this last shift. If T = 56 in the divide command, the overflow indicator may be erroneously set, and should never be depended upon.

The rule, then, for a single-precision division is:

1. Never divide an N which is greater than, or equal to D.

2. Use a T = 57 in the divide command.

3. As in the case of multiply, the divide command must be located at an odd word-time.

You can see that there are exceptions to this rule, but that a thorough knowledge of computer logic and much experience are required. In no case will division work if N is greater than or equal to $2 \cdot D$.

Double-precision division involves exactly the same operations as does its single-precision counterpart. Of course, the execution time of the command must be greater. With 57 bits of quotient to be generated, 114 word-times would be necessary. If this is the time allotted, the first bit of the quotient (T29 of MQ_1) will represent $x \cdot D \cdot 2^0$. In the case of double-precision, two more word-times are necessary to shift the entire quotient one bit-position to the left. If T = 116, all 57 bits of the quotient will be fractional, T29 of MQ_1 representing $x \cdot D \cdot 2^{-1}$, and T2 of MQ_0 representing $x \cdot D \cdot 2^{-57}$. In this case, the overflow flip-flop will be set with $x \cdot D \cdot 2^0$. If the quotient equals or exceeds 1, x will equal 1, and overflow will be indicated. If T = 114, erroneous overflow may be indicated.

The rule, then, for double-precision division is:

1. Never divide an N which is greater than, or equal to D.

2. Use a T = 116 (v6) in the divide command.

Again, exceptions are possible, but thorough knowledge of computer logic and much experience are required. In no case will division work if N is greater than or equal to $2 \cdot D$.

Remember that a quotient is nothing more than a ratio of one number to another. It stands to reason that, if 2/17 = 4/34, etc.,

0000000000000000000000000000 0000000000000000000000000100

divided by

0000000000000000000000000000 0000000000000000000000100010,

equals

0000000000000000000000000000 0000000000000000000000001000

divided by

0000000000000000000000000000 0000000000000000000001000100,

which, in turn, equals

0000000000000000000000000000 0100000000000000000000000000

divided by

0000000000000000000000000010 0010000000000000000000000000,

and any of these quotients equals the quotient derived by dividing

00010000000000000000000000000 00000000000000000000000000000

by

10001000000000000000000000000 00000000000000000000000000000.

So long as the denominator and the numerator occupy corresponding bit-positions in ID and PN, respectively, which bit-positions they occupy will have no effect on the quotient. For each denominator and numerator, regardless of their positions in these two registers, provided the above requirement is met, there will be one correct quotient appear in MQ_0 for single-precision division, and one correct double-precision quotient, if that is called for, in all of MQ.

It would be desirable to round off a quotient to the nearest figure in the last digit carried, but this is not possible, because there is no way of examining the remainder in the G-15. Truncating the quotient, i.e., ignoring the value of digits beyond the one carried, will result in the quotient reported being somewhere between correct and one unit low, and averaging a half unit low. This would give a systematic error which could seriously affect accuracy after a large number of operations. One way of avoiding this is to add an average of one half to all quotients. This is simply done by making the last bit always 1, i.e., adding one unit to half of the uotients: those originally ending in zero. This is called the "Princeton Round-off". A truncated quotient is between zero and one unit low, while a Princeton-rounded quotient is between one unit low and one unit high.

There is no way to program the elimination of the Princeton round-off; it will always occur. If you desire a single-precision quotient without this round-off, you can carry the division out as a double-precision operation, making T = 116 (v6) in the divide command, in which case the single-precision quotient will be in bits 29 - 2 of MQ_1, and the rest of the bits in MQ will contain the remainder. The Princeton round-off will occur at T2 of MQ_0.

Notice that, in either single-precision multiplication or single-precision division, four commands are necessary to set up for, and perform, the operation. One command clears the two-word registers, two others load each of the two-word registers containing the numbers to be operated on, and the fourth command actually calls for the desired operation. The clear command can sometimes be eliminated by utilizing special circuitry connecting AR and the two-word registers.

In the table of normal operations (pages 64 - 66), it was pointed out that, if both S and D are less than 28, codes 2 and 6 call for exchanges of AR with memory. So do codes 3 and 7, under similar circumstances, except that, in these two cases, complementation of negative numbers is involved. If any one of these four commands is used to load ID (25), MQ (24), or PN (26) from some other location in memory (S less than 28), during all even word-times AR cannot load any of these three registers.

In other words, the contents of AR will be blocked off from ID_0, MQ_0, and PN_0. In the place of the contents of AR, the even half (word 00) of the specified two-word register will receive 29 0's. If the C code = 6, during the following odd word-time, the contents of AR will be transferred to the odd half of the specified two-word register. Because the C code is even, the sign of the double-precision number will go to IP, according to the rules discussed earlier. Notice that during the even word-time of execution, the original contents of AR attempt to reach the even half of the specified two-word register, but are blocked off, and 0's are transferred instead. During the same word-time, the even-numbered word from memory goes to AR. During the following odd-numbered word-time, the contents of AR (originally an even-numbered word from memory) goes to the odd half of the specified two-word register. The fact that this word was delayed one word-time because of its transfer via AR does not alter the fact that it is the first word of a double-precision number. Therefore, even though it reaches the two-word register at an odd word-time, its sign will be divorced, and sent to IP, in accordance with the rules already mentioned.

Consider, then, a single-precision multiplication: A·B, where A is stored in an even-numbered word. If a transfer of A to ID via the AR register is called for (C = 6), during the first word-time (an even numbered word-time), the original contents of AR attempts to reach ID_0, but is blocked off, and all 29 bits of ID_0 are cleared to 0. During the same time, A is transferred to AR. During the next word-time (an odd word-time), A is transferred from AR to ID_1, but, since A is the first half of what the computer believes to be a legitimate double-precision number, its sign, being treated as the sign of the number, is disengaged from the magnitude bits, and it is transferred to IP. T1 of ID_1, which normally would have received this sign, is cleared to 0. Since every bit in ID has been set during this operation, every bit of PN has been cleared. The only initial condition remaining to be satisfied is the placing of the multiplier, B, in MQ_1. If B is in an odd word in memory, a straight single-precision transfer to MQ_1 will accomplish this. Since the C code for this is 0 (therefore, it is even), the sign will be disengaged from the magnitude portion of B, and it will be sent to IP, to be combined with IP's present contents. Notice that the initial conditions in this case will be:

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY
PN:	00000000000000000000000000000	00000000000000000000000000000
IP:	0 or 1, whichever is the correct sign of the product.	

In the above layout of the two-word registers, the Y's in MQ_0 represent the original contents of that word, remaining after MQ_1 has been set. Since, during a multiplication, MQ is shifted to the left one bit-position

at a time, each succeeding bit being inspected to determine whether or not the contents of ID should be added to the contents of PN, and since, if a single-precision multiplication has been called for, only 28 bits from MQ will be inspected, the remaining "garbage" in MQ will have no effect on the multiplication. There is no need to clear MQ_0 prior to a single-precision multiplication. However, if the multiplier were in an even-numbered word in memory, it would be perfectly permissable to use a transfer via AR (C = 6) to get it into the odd half of MQ. In this case, of course, MQ_0 would be cleared.

In the case of a single-precision division (N/D), if D is stored in an even word in memory, and transferred into ID_1 via AR, this will succeed in properly preparing ID for the division and setting up IP for the addition of signs. But setting ID clears PN, and not MQ. PN could be set with the proper value (N), either by a straight single-precision transfer, or by a transfer via the accumulator, and, in either case, it would also be set up properly. If PN_1 were set by a straight single-precision transfer (C = 0), PN_0 would still have been cleared because each bit in ID_0 was set (to 0). MQ will remain unaffected, containing its original contents.

	Word 1	Word 0
ID:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
PN:	xxxxxxxxxxxxxxxxxxxxxxxxxxxx0	00000000000000000000000000000
MQ:	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY

IP: 0 or 1, whichever is the proper sign of the quotient.

The Y's in the above layout of MQ represent its original contents, remaining after both ID and PN have been set. This is perfectly all right, however, since MQ is shifted to the left one bit-position at a time, as each bit of the quotient is placed in T2 of MQ_0. T1 of MQ_0 is cleared by the initial shift, preparatory to the placement of the first quotient bit in MQ_0. If a full 28-bit quotient is generated, all the Y's shown above in MQ_0 will be shifted out to the left, and all 28 magnitude bits of that word will contain quotient bits.

REVIEW

At this point, we pause to review what has been covered. We first pointed out that, in order to effectively use the computer, the programmer must analyse the problem:

1. determine the formula(s) by which a solution can be reached, or in some way define exactly what is called for;

2. discover the form, magnitude, and ranges in values, of the data which will be available as input for the program;

3. choose an appropriate method of solution;

4. outline, very briefly, the logical path to be followed in this method, such an outline being called a flow diagram.

We then analysed a sample problem, that of solving for the roots of a quadratic equation of the form, $a \cdot x^2 + b \cdot x + c$. After we developed a flow diagram for its solution, we saw that we needed more thorough knowledge of available computer operations, especially the arithmetic operations.

This lead us to a discussion of commands as they appear, in binary, in the computer. The first part of a command we studied was the C code, consisting of a two-bit characteristic and a one-bit indicator for either single- or double-precision operation. We saw that the various possible C codes in three bits run the gamut of 0 - 7, where 4, 5, 6, and 7 are essentially the double-precision counterparts of the single-precision codes 0, 1, 2, and 3, respectively.

Each of these operations is a transfer of some type, and it may call for the use of special circuitry during the transfer, such as inverting gates, or circuitry which can change a sign, drop a sign, or add. The various types of transfers that were seen to be available were:

1. straight single- or double-precision transfer from one place in memory, including either the single- or double-precision accumulator, to another, also including the accumulators; C = 0, 4;

2. single- or double-precision transfer via the inverting gates (to accomplish complementation of negative numbers, preparatory for addition) from one place in memory, including either of the accumulators, to another, also including either of the accumulators; C = 1, 5;

3. exchange of memory with AR, both single-precision words, and the transfer in each direction, a straight single-precision transfer; C = 2;

4. transfer of a double-precision number via AR, in which, during the even word-time, the even word of the double-precision operand goes to AR while the original contents of AR go to the even word of the double-precision destination; during the odd word-time, the new contents of AR go to the odd word of the destination, while the odd word of the operand goes to AR; C = 6;

5. transfer either a single- or a double-precision magnitude to or from the appropriate accumulator, the sign being dropped during the transfer; C = 2, 6;

6. exchange memory with AR, both single-precision words, the transfer from AR to memory being a straight single-precision

transfer, but the transfer from memory to AR being via the inverting gates; C = 3;

7. transfer of a double-precision number via AR, as in (4), above, except that each word of the double-precision operand as it enters AR, enters via the inverting gates; C = 7;

8. transfer either a single- or a double-precision number to or from the appropriate accumulator, but with a change of sign, and subsequent passage through the inverting gates, for complementation, if necessary, preparatory to addition (changing the sign of a number and adding it accomplishes the same end result as subtracting it); C = 3, 7.

After discussing the normal operations (each one actually a different type of transfer of words available in the G-15), we examined the various addresses contained in a command.

One of these is the address of the operand, the word(s) to be transferred. All addresses in the memory of the computer are composed of a line number and a word number, or word-time, within that line. An address is denoted in the following manner, where LL stands for line number, and TT, for word-time: LL.TT. The line containing the number to be transferred is called the Source, and the address of the operand is SS.TT, usually written S.T. The address of the word(s) receiving the number to be transferred is also composed of a line number and a word-time, and is written D.T, where D stands for Destination. In a command, the word-time (T) involved in both these addresses is the same, and is given only once. Therefore, a transfer of a word(s) from one line to another will place the number being transferred in the Destination at the same word-time it occupies in the Source, or (in the case of transfers between lines of different lengths) in a word-time congruent to the word-time it occupies in the Source.

The functions of S and D were described. They control selectors which, in turn, modify the normal recirculation of memory at the proper word-time in the proper line.

We then discovered that a series of individual commands, each with the same S, D, and C, but with successively increasing T's, can be replaced by one immediate command, in which the T number is a flag, telling the computer when to stop the operation. In such a case, the operation commences in the very next word-time after the command has been read, so the location of an immediate command helps to determine how many, and which words will be transferred. We called these immediate commands "block" commands, since they work on blocks of congruent words in given lines.

If it is not desired that a command be immediate, it can be made deferred, in which case it will operate only on the word (or two words, in the case of double-precision) indicated by the T number. A bit indicating whether the command is immediate or deferred is included in the command, itself. It is the I/D bit, (T29).

It was pointed out that, when we say a G-15 command contains within itself the address of the next command to be obeyed, and thus the program sequence is determined by the programmer when he makes up the individual commands in his program, we are only partly correct. Each command contains the word-time at which the next command is to be read, but the line number in which that next command is located is not contained within the current command. The reason it is not, is that, once a sequence of commands is started in any "command" line, the line will remain the same, and thus, need not be specified from command to command. Only word-times need be specified. It was also pointed out that not all lines in the memory of the G-15 are "command" lines. Commands can only be read out of lines 00, 01, 02, 03, 04, 05, 19, and 23. These are called "command" lines 0, 1, 2, 3, 4, 5, 6, and 7, respectively.

Two points remained open, although they were discussed:

1. how a command line is initially chosen, and how a program can switch from one line to another, should that be necessary; and

2. the meaning of the BP bit in a G-15 command, this being the only bit not defined.

After the discussion of the various parts of a command, the concept of special commands was introduced. It was pointed out that not all of the operations necessary for the solution of the quadratic equation were, as yet, described. The two most apparent of these omitted operations were multiply and divide.

If D is set equal to 31 in a command, since there is no line referred to by that number, the G-15 treats this command as a special command. In this case, the S number in the command will become a special operation code, and the C code will usually be treated in the light of the special operation called for. Having thus defined special commands, we proceeded to discuss two of them, multiplication and division.

We saw that the command calling for a multiplication contains D = 31, S = 24, C = 0, and T = a relative timing number, which indicates for how many word-times the execution of the command is to be carried out, where two word-times are necessary for each bit in the product. The duration of operation of this command can be of any length, provided T is a multiple of 2, and, in any case, the results will be predictable. This command must be an immediate command, and, because its operation is always double-precision in nature, it must be located at an odd word-time, so that the first word-time of execution will be an even word-time.

We also saw that it is necessary to place the multiplicand and the multiplier in the two-word registers, ID and MQ, respectively, prior to giving the multiply command. Certain clearing of the two-word registers is also necessary. The rules for setting up these registers, and how they operate during the multiplication were discussed, but suffice it to say here that the product will appear in PN: if a full single-

precision multiplication is called for (T = 56), the product will be in PN_1; if a full double-precision multiplication is called for (T = 114), the product will be in $PN_{0,1}$. In any case, the correct sign of the product will be generated in IP. In this regard, we saw that, if a number, either single- or double-precision, is transferred to any two-word register with an even C code (0 is treated as even), the sign will be divorced from the magnitude, the sign going to IP, and the magnitude going to the magnitude bits of the appropriate word(s) in the appropriate two-word register. Similarly, when a number is transferred out of any two-word register, if the C code is even, the magnitude bits of the number will be picked up from the register itself, while the accompanying sign will be picked up from IP. Although this makes ordinary use of the two-word registers for storage slightly confusing, it is necessary for proper operation during multiplication and division.

After multiplication, we discussed division, and saw that it, too, utilizes the two-word registers and IP. The divide command contains D = 31, S = 25, C = 1 or 5 (the operation will be exactly the same, regardless of which is used), and T = a relative timing number. For a single-precision divide, T must equal 57; for a double-precision divide, T must equal 116. Exceptions to this rule are possible but require thorough knowledge of the internal logic involved and extreme care in treatment of the quotient. This command must be immediate, and, because its operation is always double-precision in nature, it must be located at an odd word-time, so that the first word-time of execution will be an even word-time.

The denominator and the numerator are placed in ID and PN, respectively, prior to giving the divide command. Certain clearing of the two-word registers is also necessary. The quotient will appear in MQ; a single-precision quotient will appear in MQ_0; a double-precision quotient will appear in $MQ_{0,1}$. The correct sign of the quotient will be generated in IP. The least significant bit in a quotient will always be 1; this is called the Princeton round-off.

A quotient represents the ratio of one number to another. In the G-15, this ratio should be in the form of a proper fraction, less than 1. If a quotient less than 1 is to be obtained, care must be taken to insure that, prior to the division, the numerator, as it appears in the machine, is less than the denominator, as it appears in the machine. Since a ratio is desired in a division, the location of the numbers to be divided, in ID and PN, is immaterial, provided they occupy corresponding bit-positions in those two registers.

Because of a unique circuit connecting AR and the two-word registers, use of a C code equal to 6 in the transfer of a single-precision number from an even location in memory (S less than 28), via AR, and into the odd half of ID (ID_1), will accomplish all the clearing necessary for a single-precision multiplication or division, eliminating the necessity for a clear command. This same circuit will cause the same clearing to occur whenever S is less than 28, D equals 24, 25, or 26, and the C code equals 2, 3, 6, or 7.

MACHINE FORM OF A NUMBER AND SCALING

Several times reference has been made to the machine form of a number, and, in the Introduction to the G-15, it was implied that the following numbers in machine form are equal to the decimal numbers shown below:

Binary number in machine	Decimal equivalent
0000000000000000000000000001|0	1.
0000000000000000000000000010|0	2.
0000000000000000000000000011|0	3.
0000000000000000000000000011|1	-3.

This implies that the binary point in a machine number is usually between the least significant magnitude bit and the sign bit.

$$0000000000000000000000000001.+_{(2)} = +1._{(10)}$$

This would mean that usually a machine number is entirely integral, and has no fractional bits. But if this were the case, the result of a multiplication or of a division is most disconcerting, for a multiplication will result in a product smaller than either the multiplicand or the multiplier, and a division will result in a quotient larger than the numerator, and, in some cases, also larger than the denominator. These statements are verified by the fact that multiplication of even the very largest possible numbers cannot cause overflow, and division of a number by another can cause overflow.

In short, it would seem, from inspection of these results, that numbers in the computer are actually fractional rather than integral. If two fractions are multiplied together, the resultant product will be smaller than either of the original numbers, and if a fraction is divided by another fraction, the result will in all cases exceed the value of the numerator, and may exceed the value of the denominator.

Actually, the computer treats every binary number in its memory as a 28-bit fraction with a sign. The binary point in the machine, sometimes referred to as the machine-point, precedes the most significant bit of a number. If this is the case, then:

$$.0000000000000000000000000001|0_{(2)} = 1\cdot 2^{0}\cdot 2^{-28}_{(10)} = 1\cdot 2^{-28}_{(10)}$$

$$.1000000000000000000000000000|0_{(2)} = 1\cdot 2^{27}\cdot 2^{-28}_{(10)} = 1\cdot 2^{-1}_{(10)}$$

We can interpret any 28-bit binary value in the machine in any way we want; that is to say, we can understand the true binary point for our purposes to follow T2 if the machine holds $1/2^{28}$th of what we intended. The maximum value we can express in 28 bits is:

$$1111111111111111111111111111._{(2)},$$

which equals

$$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + \ldots + 1 \cdot 2^{26} + 1 \cdot 2^{27},$$

and this is 1 less than $1 \cdot 2^{28}$. Therefore, the machine will always be able to make a fraction out of whatever we give it, not by changing any bits, but merely by assuming the binary point to always precede T29 of the word.

Of course addition and subtraction will yield the same, familiar results, as we interpret them, since these operations will be performed on fractions all of which have the same denominator.

If a + b = c, then $\frac{a}{2^{28}} + \frac{b}{2^{28}} = \frac{c}{2^{28}}$.

Regardless of the fact that we might be considering a and b as integers, and the computer is considering them as fractions, the fraction which results in the computer must represent the integer c, as we interpret it. We will be able to convert the bits of the result as a binary integer and arrive at the decimal integer equivalent.

```
a   0000000000000000000000000001|0
b   0000000000000000000000000010|0
-----------------------------------
c   0000000000000000000000000011|0
```

If a = 1, b = 2, c will equal 3. We can so place a in a word in memory that it will be of the form shown above. We interpret this as $a \cdot 2^{-28}$. We can similarly place b in another word, knowing that it will be similarly interpreted. We can then transfer a to AR (D = 28), and b to AR (D = 29), and they will be added in AR. The sum in AR will appear as shown above. We know it is really $c \cdot 2^0 \cdot 2^{-28}$, and we interpret it as $c \cdot 2^0$, or 3.

Suppose, in the example above, we wanted to add numbers containing fractional as well as integral bits. Let a = 1.5 and b = 2.75. The sum of these, c, must equal 4.25. We must first determine how many bits will be required to express the longest fraction in the binary equivalents of these numbers. $.5_{(10)} = .1_{(2)}$, which is 1/2. $.75_{(10)} = .11_{(2)}$, which is 1/2 + 1/4 = 3/4. $.25_{(10)} = .01_{(2)}$, which is 1/4. The most bits required by any fraction involved are two. Therefore, if two fractional bits are reserved, each number involved can be fully expressed. Note that not all decimal fractions will convert evenly to binary fractions. Sometimes a programmer must decide what accuracy is needed, and let this determine the number of fractional bits needed.

When we speak of fractional bits, as in the paragraph above, we are referring to bits which we will interpret as fractional, as opposed to those which we will interpret as integral. When we say that the computer holds $a \cdot 2^{-28}$, we are saying that we know that a, as it appears in the machine, is in the form, $a \cdot 2^0 \cdot 2^{-28}$, meaning that the true binary point, for us, is 28 bits to the right of the machine binary point.

This means there are no bits left to express any fractional value for a. If we were to place a in a word in memory in such a way that two fractional bits followed the true binary point, and do the same for b and c, then the conditions of the problem in the above paragraph would be met. This means that a, as it appears in the machine, must be in the form, $a \cdot 2^2 \cdot 2^{-28}$. For a = 1, the value in the machine would appear as

00000000000000000000000000100,

where the true binary point would be known to be at the point shown by the arrow. In such a case, we say that the machine holds $a \cdot 2^{-26}$ (note that this equals $a \cdot 2^2 \cdot 2^{-28}$), or, that the true binary point is 26 places to the right of the machine binary point.

Now, using the same process for placing b, and for interpreting c, let's add, using our new values, and check the result.

00000000000000000000000001100 = $a \cdot 2^{-26}$

00000000000000000000000010110 = $b \cdot 2^{-26}$

00000000000000000000000100010 = $c \cdot 2^{-26}$ = $100.01_{(2)} = 4.25_{(10)}$, which is correct.

If we let A* be the machine representation of any binary number A, then

$$A^* = 2^n \cdot A,$$

where n may range from 0 through -28, depending on the fractional accuracy being carried with A.

The entire subject we have just discussed is called "scaling". In the above expression, 2^n is called the "scale factor" of A. Every number is "scaled" somehow when it is placed in the computer. A is said to be scaled 2^n in the machine. From now on, we won't have to make long, cumbersome references to the true binary point in a number as it appears in the machine; we can simply say the number is scaled in a certain manner, and you will immediately know where the true binary point is.

As you can see, the concept of scaling is really quite simple and straightforward; scaling is a short-hand method of interpreting the numbers that appear in a digital computer.

In ordinary decimal arithmetic, you know that you cannot add

100.962
6.75401,

because the decimal points are not lined up correctly. Rather, you can add them, but you shouldn't; the result will be meaningless. Similarly, in ordinary binary arithmetic, you cannot, or at least you sould not, add

```
110.111
1.00011,
```

because the binary points are not lined up correctly. In the computer, you should not add two numbers which are scaled differently, for the same reason. You can, and occasionally programmers have, but the result is meaningless, as they have found out, much to their chagrin.

Suppose we are to add a + b, where a is scaled 2^{-15}, and b is scaled 2^{-13}. It's obvious that one or the other of these numbers will have to be moved, in order to line up the true binary points prior to the addition. You already know how numbers are moved back and forth within a word in the computer: they're shifted in one direction or the other. In the case of pencil-and-paper arithmetic, the job of lining up the base points of two numbers, in order to add them, is simple: we rewrite the numbers. In the previous binary addition, we would rewrite the numbers as

```
110.111
  1.00011,
```

and proceed to add them. Unfortunately, as we shift numbers in a computer, we must lose bits. 29 bits are allotted to each single-precision number; after it is shifted, there will still be only 29 bits allotted to any single-precision number. Thus, if the number is shifted to the left, bits will be lost from the most significant end; if the number is shifted to the right, bits will be lost from the least significant end. In the case under consideration, a can be shifted left two places, increasing it by a factor of 2^2, and thus rescaling it from 2^{-15} to 2^{-13}, and making it compatible with b. Or b can be shifted to the right two places, decreasing it by a factor of 2^{-2}, and thus rescaling it from 2^{-13} to 2^{-15}, making it compatible with a.

Which would be the better scheme can be determined from consideration of a number of factors:

1. the desired scaling of the answer, if any particular scaling is desired;

2. the number of integral bits that must be allowed to insure that overflow will not occur when the numbers are added (this can be determined by considering the largest possible sum of a and b, and in all events, this number of bits must be allowed, regardless of what shifting is necessary to insure it; otherwise, the answer will be erroneous);

3. the fractional accuracy desired in the sum.

From these considerations and perhaps others, unique to a given problem, you will determine the shifting that is required prior to the addition. It may be that both numbers will have to be shifted. In any event, once you have decided that shifting is necessary prior to an addition in your program, you will, of course, need a command which will direct the computer to do it.

The shift command is another special command, with D = 31, S = 26, C = 1 (or any other non-zero number), and T = a relative timing number (similar in function to T for a multiply or divide command). The command will be immediate, and, like multiplication and division, it is double-precision in nature. Two word-times are required for each shift of one bit-position. Therefore, two times the number of bit-positions desired in the shift = T. If you wished to shift a number 10 bit-positions in either direction, T of the shift command would equal 20. Because this operation is immediate and double-precision in nature, it must be located at an odd word-time.

You have already seen that shifting can take place in the two-word registers, and this is where the shifting caused by this command will occur. When this command is executed, ID will shift to the right the indicated number of bit-positions, and, simultaneously, MQ will shift to the left the same number of bit-positions. If you have one number you want to shift, prior to giving the shift command you must place that number in the appropriate two-word register. Either half of the register will do for a single-precision number. You might have a number in each of these registers, one moving to the right, the other to the left.

Notice, if you have a single-precision number you wish to shift to the right, and you load that number in ID_1, then execute the shift command, the number will move to the right, and the vacated bits will be filled in with 0's, which, of course, would be fine. But, under the same conditions, if you loaded that number in ID_0, the vacated bit-positions in ID_0 would be filled in with bits from ID_1, and unless ID had been previously cleared, the single-precision word containing your number would receive "garbage", which could very well contain 1's. Of course this would change your number, making it erroneous. A similar situation, but in reverse, holds true for the shifting of a single-precision number to the left in MQ.

You are probably wondering why any non-zero C is permissible in the command discussed above, and why a C of zero is not permissible. The only function of a C in this command is to distinguish it from a similar command, with D = 31, S = 26, and C = 0. The latter is also a shift command, calling for the exact operation described above, but if C = 0, a tally of the shifts performed will be kept in AR. For each complete shift of the registers by one place, $1 \cdot 2^{-28}$ will be added to the present contents of AR.

Of course the operation called for by this second shift command will cease at the end of the indicated number of word-times, just like any other command. But it will also cease if an end-around-carry is generated in AR, regardless of whether or not the indicated number of word-times have been consumed. In other words, this shift is performed under control of AR.

An example of the usefulness of such a command might be the following: rescale the binary number x, in the computer, by a factor of $2^{(a-b)}$, where a and b will also be available in the computer. Assume all numbers are single-precision. When you originally write your program, you won't know how many shifts to call for to be performed on x. As a matter of fact you won't even know in which direction x is to be shifted. All this depends on the current values of a and b.

You could subtract b from a in AR, and, depending on the sign (+ or -) of the answer, you could load x into the proper half of the proper two-word register: ID_1 if x is to be shifted to the right, because the sign of (a - b) is negative; MQ_0 if x is to be shifted to the left, because the sign of (a - b) is positive. There is an implication here that some provision is available to programmers to cause their programs to automatically determine which of two alternate logical paths to follow, based on inspection of a given condition in the computer. This is correct, and the method available for doing this will be discussed shortly, in pages 105 - 109. For the moment, you may assume that such a decision has been made, and x is in the proper two-word register.

The problem now is to use the number in AR to control the shifting process. We know that the shift command we want has D = 31, S = 26, C = 0. It's operation will cease either when an end-around-carry has been generated in AR or when the number of word-times called for by T has been consumed, whichever occurs earlier. We will set T with some maximum number, so that, unless (a - b) is useless (due to the fact that it calls for so many shifts that all of x will be lost), (a - b) will effectively control the process. Assuming that we want only a single-precision answer, $x \cdot 2^{(a-b)}$, from either ID_1 or MQ_0, the maximum number of shifts that can be performed in either direction, without losing all significance, will be twenty-seven. On the twenty-eighth shift in either direction, all twenty-eight magnitude bits of the original x will be lost. We will therefore set T = 54 (= 2·27). And thus we have the shift command that will be included in our program.

The problem now is to so set AR that, after (a - b) shifts have been performed, and $(a - b) \cdot 1 \cdot 2^{-28}$ has been added to AR, an end-around-carry will be generated. Any positive number plus its negative complement will yield +0 in the computer. Therefore, if we start with the negative complement of $|(a - b)|$ in AR, and if we add $1 \cdot 2^{-28}$ to it $|(a - b)|$ times, we will have generated, in AR, the quantity,

$$- \left|(a - b)\right| \cdot 2^{-28} + \left|(a - b)\right| \cdot 2^{-28},$$

and this must be +0. Because we cannot know that (a - b) will always be positive, we use its absolute value, which of course will be positive. After the last shift, there will be an end-around-carry, and the sign will be changed to +. The end-around-carry will halt the shifting process.

Of course the same scaling rules that apply to addition of single-precision numbers apply as well to the subtraction of single-precision numbers.

Multiplication is a slightly different case. If a•b is desired, a is scaled 2^{-15}, and b is scaled 2^{-13}, ab will be scaled 2^{-28}, in accordance with the rule of exponents.

$$a \cdot 2^{-15} \cdot b \cdot 2^{-13} = a \cdot b \cdot 2^{-28}.$$

A very simple rule governing the scaling of a product in PN is, following the multiplication of one number by another, the product will be scaled by a factor equal to the product of the scale factors of the two numbers.

Consider now a multiplication of $1 \cdot 2^{-28}$ by $4 \cdot 2^{-28}$. The product will be 4, scaled 2^{-56}. The initial condition of the two-word registers would be:

	Word 1	Word 0
ID:	00000000000000000000000000010	00000000000000000000000000000
MQ:	00000000000000000000000001000	YYYYYYYYYYYYYYYYYYYYYYYYYYYYY
PN:	00000000000000000000000000000	00000000000000000000000000000

IP: 0 or 1, whichever is the correct sign of the product.

Each bit in MQ, starting with T29 of MQ_1, will be checked for 0 or 1. If it is 0, nothing will be added to PN; if it is 1, the present contents of ID will be added to PN. The T number of the command will be 56, allowing the inspection of 28 bits from MQ. Thus the Y's in MQ_0, shown above, representing the original contents of MQ_0 before the multiplier was loaded, will have no bearing on the process. The bits in MQ are made available for inspection by being shifted out of MQ, to the left, to an inspection station. A shift is performed before the first bit from MQ_1 can be inspected. For each shift to the left of MQ, ID is shifted to the right. 25 0's will be inspected before the 1 in MQ_1 is sensed at the inspection station. Then the 1 will be shifted into the inspection station, making a total of 26 shifts to the left, before ID is to be added to PN for the first time. This means that ID will have been shifted to the right 26 times before it is added to PN. ID will then look like this:

	Word 1	Word 0
ID:	00000000000000000000000000000	00000000000000000000000010000

Since this is the only 1 that will be found in MQ, this is the only addition to PN that will take place. Therefore, upon completion of the multiplication, PN will also look like the above. Notice that, in the full double-precision magnitude of the two-word register (we previously stated that, during both multiplication and division, the operation of the two-word registers is essentially double-precision in nature, even if single-precision numbers are actually involved in the operation), the answer is $4 \cdot 2^{-56}$. This is to be expected;

the computer was given two fractions to multiply together, each of which was very small. Naturally, the resultant fraction will be even smaller, and, in fact, it is so small that, if you demand a 28-bit expression of its value (take the single-precision answer from PN_1), the nearest value to it that can be expressed in 28 bits is 0.

Notice, then, that multiplication can result in an answer whose scale factor will require more than the 28 bits of PN_1 for expression. As long as you are aware of this, and can devise methods for using the answer, fine. But, if you want the answer expressed in 28 bits, rescale the two numbers entering into the multiplication before you multiply, in such a way that the scale factor of the product (equal to the product of the scale factors of the two numbers) will lie in the range $2^0 - 2^{-28}$.

In division, the scaling rule is: the quotient, in MQ, will be scaled by a factor equal to the quotient of the scale factors of the numbers being divided. For example, if a/b is desired, a is scaled 2^{-15}, and b is scaled 2^{-13},

$$\frac{a \cdot 2^{-15}}{b \cdot 2^{-13}} = \frac{a}{b} \cdot 2^{-2}$$

Another example:

$$\frac{a \cdot 2^{-28}}{b \cdot 2^{0}} = \frac{a}{b} \cdot 2^{-28}$$

And finally, one more example:

$$\frac{a \cdot 2^{-28}}{b \cdot 2^{-28}} = \frac{a}{b} \cdot 2^{0}$$

In each of the above examples, there is a basic assumption that a appears in the machine to be smaller than b, in accordance with the rule for division. Notice that a can appear smaller than b, in the machine and yet, as in the second example above, we interpret it as being of greater magnitude than b. The scaling we associate with a number in the machine is unknown to the computer; it merely aids us in interpreting the numbers the computer works with. In the second example, we know that a, as it appears in the machine, represents a 28-bit binary integer, counting any leading 0's (because the true binary point is 28 places to the right of the machine binary point), while b, in the machine, represents a 28-bit binary fraction, counting any trailing 0's, (because the true binary point coincides with the machine binary point). In reality, then, as we interpret these numbers in the machine, we are dividing a relatively large magnitude by a relatively small magnitude. The computer, not realizing this, will perform the division correctly, without generating overflow, as long as the 28-bit value in the machine representing a is less than the 28-bit value in the machine representing b.

We have discussed scaling in the light of single-precision numbers in order to minimize the number of bits you have to keep track of. All of the principles and rules of scaling that have been mentioned apply equally well to either single- or double-precision numbers.

Now all four basic arithmetic operations (+, -, x, ÷) are available, and you know how to arrange numbers in the computer to suit your purposes. You also know how to interpret the results. In the solution of the quadratic equation, there is one operation that has not, as yet, been described: it is taking the square root of a number ($\sqrt{b^2-4ac}$). There is no one command that will cause the computer to do this, because the computer is not wired to do it directly. We can generate the square root of any number through a combination of the four basic arithmetic operations, repeated over and over again, but we will, for the present, postpone a discussion of this.

We are ready to expand the original flow diagram of our solution of the quadratic equation, as it appears on page 4, but first, we must decide on what ranges of values we will allow for a, b, and c. Remember the formula is:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

We previously decided to use single-precision, so the scale factor for each value must lie in the range 2^0 to 2^{-28}. Let's arbitrarily allow 7-bit fractional accuracy in the binary numbers, so that a, b, and c will be scaled 2^{-21}. In terms of decimal equivalents, this means that our program will be able to process values accurate to the nearest 1/100th, since $1/2^7 = 1/128$, and this is even a smaller value than 1/100. Notice that 6 fractional bits would not give accuracy to the nearest 1/100, since $1/2^6 = 1/64$. Now let's also assume that we want x to the same accuracy, scaled 2^{-21}. We could do whatever shifting is necessary to insure that the numerator, prior to the division, is scaled 2^{-21}. The question, therefore, is, how do we determine what scaling the denominator needs? We will find the answer from the following equation:

$$x \cdot 2^{-21} = \frac{N \cdot 2^{-21}}{D \cdot 2^{n}}$$

The solution of the above equation is: $n = 0$. This means that the true binary point of D would have to coincide with the machine binary point, meaning that $-1 < D < 1$. Since $D = 2a$, $-1 < 2a < 1$, or $-1/2 < a < 1/2$. This is, of course, too great a restriction on a; our program could, in no sense, be called a general program.

Let's go in the other direction:

$$x \cdot 2^{-21} = \frac{N \cdot 2^{n}}{D \cdot 2^{-21}}$$

The solution of the above equation is: $n = -42$. We know that we can position N in such a way as to meet this requirement, through shifting. This would seem to work out quite well, so let's do it.

This means that the integral portion of 2a is going to be expressed in 21 bits. 2a is scaled 2^{-21} in the machine. Since this is so, we better make sure that the integral portion of a does not, in any case, exceed 20 bits, even though a will originally be scaled 2^{-21} (in other words, a, as originally stored in the machine, will always have at least one leading 0). In 20 bits we can express all integral values up to, and including, $2^{20}-1$; this, then, becomes the limit for a. $2^{20}=1048576_{(10)}$. Therefore, $-1048575 \leq a \leq 1048575$. Now that a set of limits has been found for a, let's find a similar set for b and c. Notice that we are going to generate 4a. If 2a requires 21 bits, $2^{1} \cdot 2a$ will require 22 bits. We would like to shift the product 4ac in such a way as to scale it 2^{-42}. The reason for this is, if b is scaled 2^{-21}, as it was agreed it would be, $b \cdot b$ will be scaled 2^{-42}. If 4ac is scaled the same way, we can subtract immediately, without having to rescale b^2. If 4ac is to be shifted to be scaled 2^{-42}, its integral value must not, under any conditions, require more than 42 bits for expression. We have already seen that 22 bits will be necessary for 4a. If $c = 2^{20}$, $4a \cdot 2^{20}$ will require 42 bits for expression. Therefore c cannot exceed $2^{20} = 1048576_{(10)}$.

It is possible that what looks like a subtraction in the formula, b^2-4ac, might very well become an addition, if either a or c, but not both, is negative. Therefore, it might be possible, if we allow 42 bits for the integral portion of both b^2 and 4ac, that the combination of b^2 and 4ac will cause overflow. Since this is undesirable, we must prevent it. We can do this by limiting the integral value possible, in the generation of 4ac, to 41 bits, thus being sure that in all cases, as it is expressed in 42 bits, it will have at least one leading 0. If we similarly limit b^2, no overflow will be possible when we add b^2 and -4ac. Therefore, we will revise our limit for c. Whereas we originally suggested that c not exceed 2^{20}, we will now say that c may not exceed $2^{19} = 524288_{(10)}$.

If we limit both b^2 and -4ac to 41 integral bits, the result of b^2-4ac will be limited to 42 bits. When we take the square root of that number, we will get a number whose integral value is limited to 21 bits, and this number will be scaled 2^{-21}, since the square root of 2^{-42} is 2^{-21}. When this is combined with b, however, to form the final numerator, overflow might result. The square root will have to be limited, in its integral portion, to 20 bits, scaled 2^{-21}, meaning that it will have a leading 0. If this is so, the radicand, b^2-4ac, will have to be limited, in its integral portion, to 40 bits. This means that b^2 and -4ac will have to be limited to 39 integral bits, to assure no possibility of overflow when they are combined. The limits on a and c,

up to this point, will limit the integral value of 4ac to 41 bits. To reduce this to 39 bits, we could further cut down on c, but it would be preferable to cut the limit on a, assuming that, in the equation, ax^2+bx+c, greater values will be desired for c than for a. We have previously seen that presently 4a will require 22 integral bits. If we cut this down to 20, and c retains its limit of 19, the limit of the integral bits in 4ac will be the desired 39. Since $4a = 2 \cdot 2 \cdot a$, to get a result limited to 20 integral bits, we must limit a to 18 integral bits, meaning that the maximum a expressable will be $2^{18}-1$. $2^{18} = 262144_{(10)}$; therefore $-262143 \leqslant a \leqslant 262143$.

Similarly, the integral portion of b^2 is limited to 39 bits. If b contains 20 integral bits, b^2 may contain 40. If b contains 19 integral bits, b^2 may contain 38, which meets our requirement. So we will limit b to 19 integral bits, the maximum b then being $2^{19}-1$. $-524287 \leqslant b \leqslant 524287$.

Now no overflow will be possible in either the generation of b^2-4ac or the generation of $-b \pm \sqrt{b^2-4ac}$.

We have thus set up the following limits and scale factors, for this program:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

where:

$-262143 \leqslant a \leqslant 262143$, or

$-(2^{18}-1) \leqslant a \leqslant (2^{18}-1)$, where a is scaled 2^{-21};

$-524287 \leqslant b \leqslant 524287$, or

$-(2^{19}-1) \leqslant b \leqslant (2^{19}-1)$, where b is scaled 2^{-21};

$-524288 \leqslant c \leqslant 524288$, or

$-(2^{19}) \leqslant c \leqslant (2^{19})$, where c is scaled 2^{-21};

x will be scaled 2^{-21}.

With these ranges of values for a, b, and c, we can truly say that this program can be used in almost any application, in order to solve for the roots of a quadratic equation. There is one further restriction:

if $$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$, then

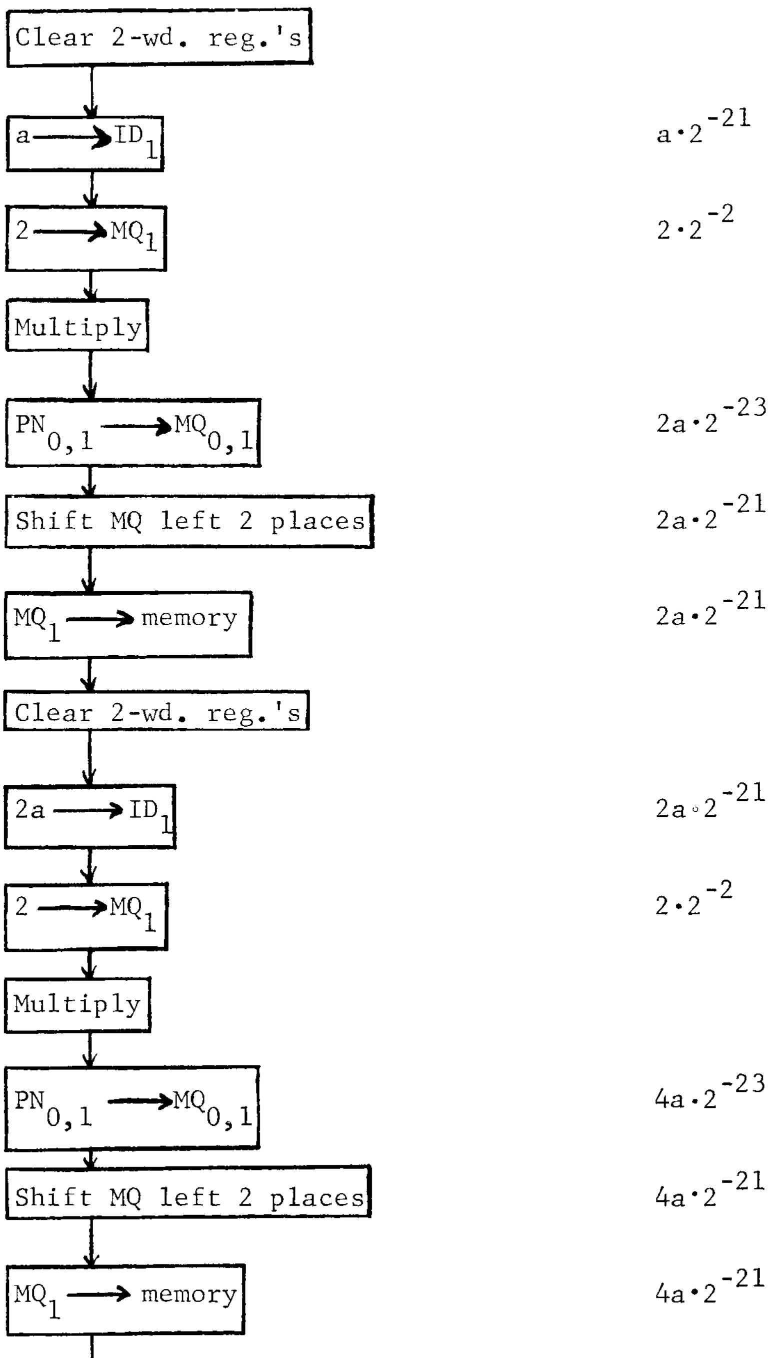
as a approaches 0,

$$x \text{ approaches } \frac{-b+b}{0} = \frac{0}{0} \text{ or } \frac{-2b}{0},$$

and the division will yield an erroneous result. In any case, a must be unequal to 0. If $|a| < 1/2$, the limit for $|b|$ will decrease in proportion.

FLOW DIAGRAM

We can now go on to expand the original flow diagram, as it was developed on page 4.



$a \cdot 2^{-21}$

$2 \cdot 2^{-2}$

$2a \cdot 2^{-23}$

$2a \cdot 2^{-21}$

$2a \cdot 2^{-21}$

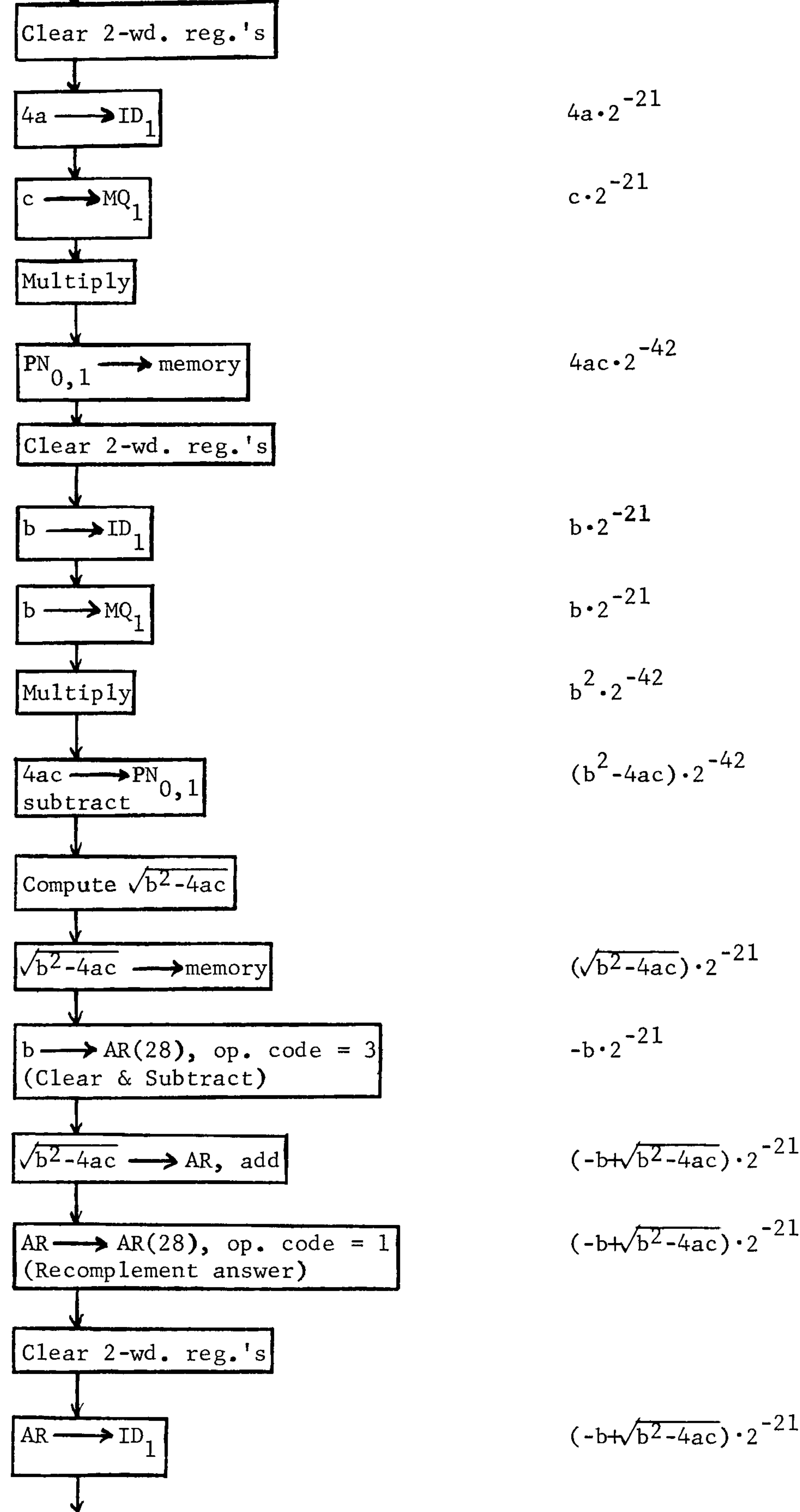
$2a \cdot 2^{-21}$

$2 \cdot 2^{-2}$

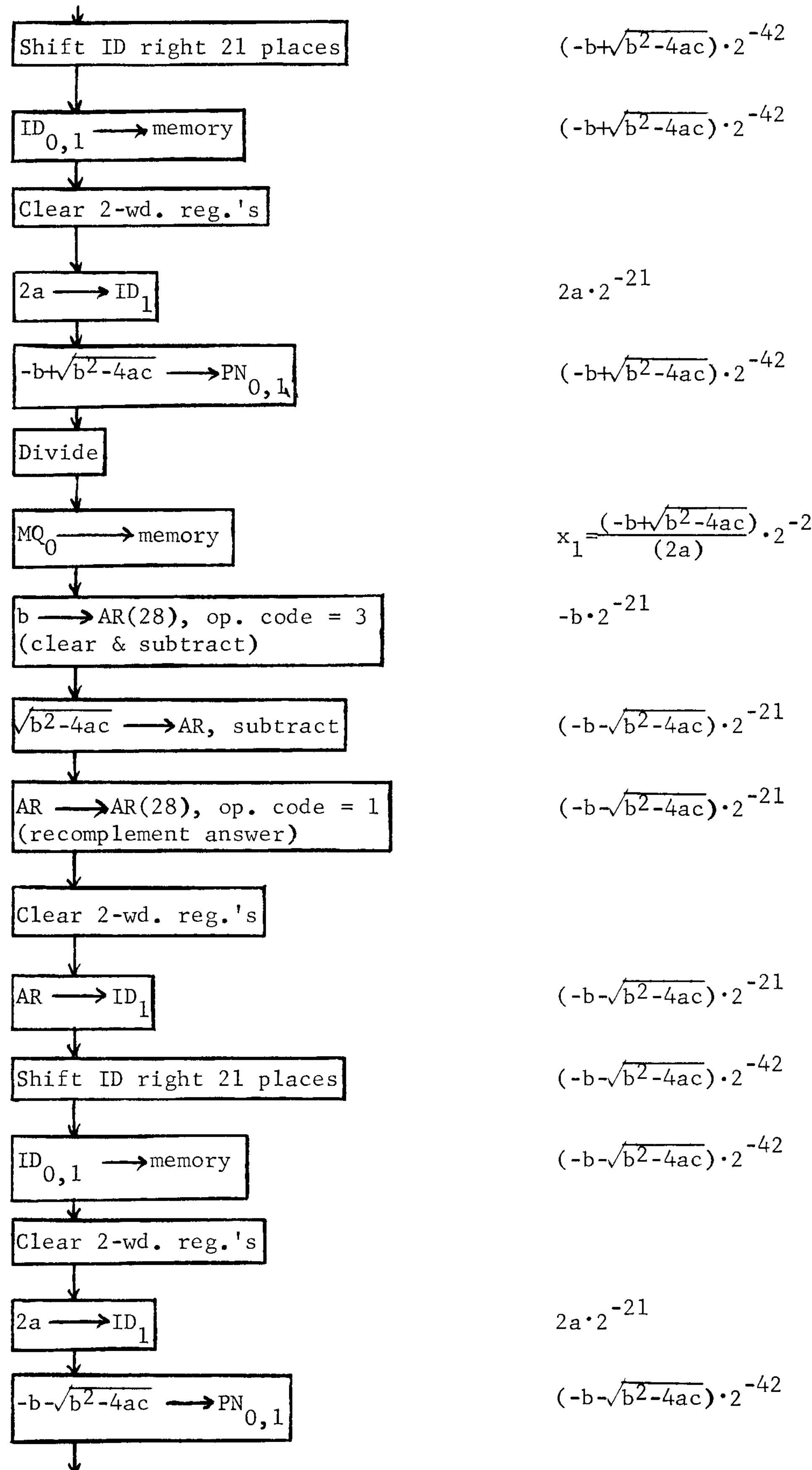
$4a \cdot 2^{-23}$

$4a \cdot 2^{-21}$

$4a \cdot 2^{-21}$



Clear 2-wd. reg.'s
4a → ID1
4a·2^-21
c → MQ1
c·2^-21
Multiply
PN0,1 → memory
4ac·2^-42
Clear 2-wd. reg.'s
b → ID1
b·2^-21
b → MQ1
b·2^-21
Multiply
b^2·2^-42
4ac → PN0,1 subtract
(b^2-4ac)·2^-42
Compute √b2-4ac
√b2-4ac → memory
(√b2-4ac)·2^-21
b → AR(28), op. code = 3 (Clear & Subtract)
-b·2^-21
√b2-4ac → AR, add
(-b+√b2-4ac)·2^-21
AR → AR(28), op. code = 1 (Recomplement answer)
(-b+√b2-4ac)·2^-21
Clear 2-wd. reg.'s
AR → ID1
(-b+√b2-4ac)·2^-21


Shift ID right 21 places
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
$ID_{0,1}$ → memory
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
Clear 2-wd. reg.'s
2a → ID_1
$2a\cdot 2^{-21}$
$-b+\sqrt{b^2-4ac}$ → $PN_{0,1}$
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
Divide
MQ_0 → memory
$x_1=\frac{(-b+\sqrt{b^2-4ac})}{(2a)}\cdot 2^{-21}$
b → AR(28), op. code = 3
(clear & subtract)
$-b\cdot 2^{-21}$
$\sqrt{b^2-4ac}$ → AR, subtract
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
AR → AR(28), op. code = 1
(recomplement answer)
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
Clear 2-wd. reg.'s
AR → ID_1
$(-b-\sqrt{b^2-4ac})\cdot 2^{-21}$
Shift ID right 21 places
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$
$ID_{0,1}$ → memory
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$
Clear 2-wd. reg.'s
2a → ID_1
$2a\cdot 2^{-21}$
$-b-\sqrt{b^2-4ac}$ → $PN_{0,1}$
$(-b-\sqrt{b^2-4ac})\cdot 2^{-42}$

In this expanded flow diagram you can see the program begin to take shape. The arrows connecting the boxes indicate the logical path of the program, from step to step. Each box in this flow diagram represents one command, with perhaps one or two exceptions, one of which is the box containing "Compute $\sqrt{b^2-4ac}$". A former statesman once said, "My job is to reduce problems to manageable proportions." That is exactly the function and purpose of a flow diagram. Initially, a problem may seem quite complicated and unmanageable, but, when it is disected into individual little parts, each part becomes easily understandable and manageable. Each programmer develops his own method of flow-diagramming. The diagram above is perhaps a little more detailed than need be, but limiting the logical size of the boxes in a flow diagram to approximately one command per box is a fine idea when starting out as a programmer.

Notice that arrows have been used inside boxes to indicate the direction of a transfer.

Certain transfers are usually named in order to simplify references to them. The transfer of a number into AR or PN, replacing what was originally there (D = 28 for AR, 26 for PN), prior to an addition (the characteristic will usually equal 0 or 1), is called "clear and add". The transfer of a number into either of the same two registers, to be combined with their present contents (D = 29 for AR, 30 for PN), thus performing an addition, is called "add". The transfer of the magnitude of a number into AR (C = 2), replacing what was originally there (D = 28), prior to an addition, is called "clear and add magnitude". The transfer of a number into AR or PN with the same characteristic and D = 29 or 30 is called "add magnitude". The transfer of a number into AR with a C = 3, and D = 28, is called "clear and subtract". The transfer of a number into AR or PN with the same characteristic and D = 29 or 30, is called "subtract". The transfer of the result of any of these operations from either AR or PN to some storage location in memory is referred to as "storing" the result.

Remember that if any two-word register, with the exception of PN = 30, (which is greater than or equal to 28), is the destination of a transfer whose C code is 2, 3, 6, or 7, the operation called for will be a transfer via AR, and the even half of the two-word register will be cleared. Therefore, "clear and add magnitude" into PN, which would require a destination of 26, is out. But a transfer of a number into PN with C = 4 will divorce the sign from the magnitude and load it into IP. This leaves the magnitude of the number in PN, with a positive sign (0), in T1 of PN_0, where we want the sign of a double-precision number which is to be involved in an addition.

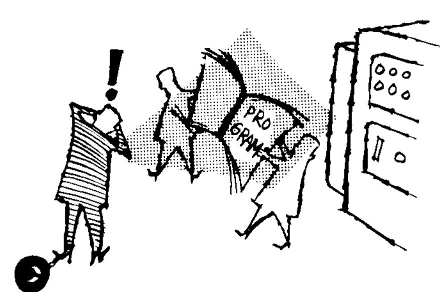
Similarly, "clear and subtract" into PN is out, since its C code is 7, and PN must be referred to as 26. But the same thing can be accomplished by first clearing PN, and then subtracting a number from 0.

THE NEED TO AUTOMATICALLY CHECK COMPUTATIONS

There are certain conditions which might arise in the operation of this program, as it is written, which would result in erroneous answers. Despite the scaling and the limitations on a, b, and c that we have chosen, in either of the two divisions we have incorporated, the numerator could exceed the denominator in apparent value in the machine. As has been pointed out, this would cause an overflow and an erroneous quotient. But how will the person using the program know when this has occurred? How will he know which answers can be trusted, and which cannot? We must include something in our program which will prevent the output of an erroneous answer.

We have been very careful, in our choice of scaling and limitations, to prevent the possibility of overflow in any of the necessary additions or subtractions. Does this mean that no overflow can occur as a result of any of these? Ideally, yes; practically, no. If the limitations, as we set them, on a, b, and c, are obeyed, no overflow will occur. But, never trust anyone else to follow your limitations when using your program. Anyone who knows *how* to operate the computer, without knowing *why* it does what it does, might try to use this program for his purposes. To him, because he might not understand *why* the limitations have been imposed, they may be meaningless. If he attempts to use values outside the prescribed ranges, overflow might result. He will, of course, be unaware of this, and treat the answers he receives as accurate, unless they are obviously wrong. To prevent this sort of thing from happening, even though we have taken steps to prevent it, we must include in the program something which will prevent an output in the case of overflow resulting from addition or subtraction.

Our program, as diagramed, includes a computation of a square root. It is possible that the radicand might be negative. We will assume that we do not want imaginary numbers as answers. We must, therefore, make sure the radicand is not negative before proceeding to compute and put out an answer.



In short, there are two types of deleterious conditions that might arise during the operation of almost any program. One is that type of situation that cannot adequately be prevented through an "ounce" of caution, because it might arise from given data which, on the face of it, seems to be perfectly acceptable. The other is that type of situation that arises when someone other

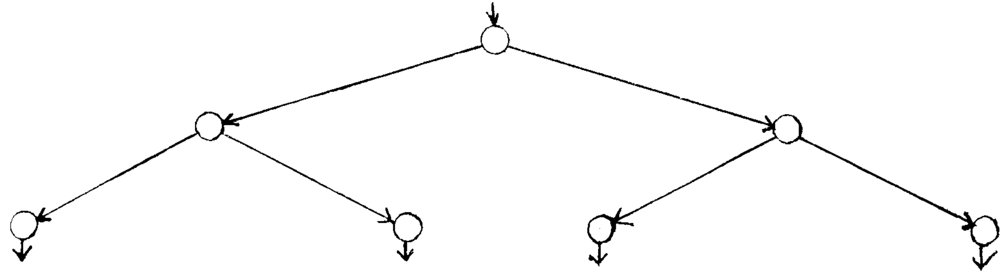
than the programmer, himself, attempts to twist the program to suit his own needs, heedless of warning. In this case, the programmer might very well like the output to consist of a few well-chosen four-letter words, but, for our purposes, we will be content with merely frustrating the offender by refusing to give him an answer.

TEST COMMANDS

The G-15, like many other digital computers, has the ability to determine the presence or absence of any one of several conditions, and a program can be written with two alternate logical paths, either of which will be followed, during operation of the program, depending on the decision made by the computer. The program itself will tell the computer when to "test" a particular condition, and the commands which do this are called "test" commands. Depending on which state the tested condition is in (off or on), the computer will take its next command from N (as it usually would) or N + 1, respectively. Always remember that only one of two answers to the test is possible: there is no "maybe" in the computer.

This simple "decision-making" power of computers is what has led laymen to use the term "electronic brain", and other equally erroneous terms, when referring to computers. You can see that, actually, the G-15 does not "think"; it merely tests, upon command, the condition of a circuit or component as to "on" or "off", and, in this respect only, it can answer "Yes" or "No" to a particular, properly chosen, question or "test".

A limitless number of tests can be included in any program, each with two alternate paths, so a program's flow diagram, unlike the straight, unswerving one we generated, can take on the shape of a "tree".



The following tests are available in the G-15:

1. test for overflow, D = 31, S = 29, C = 0;

2. test for sign of AR (neg.), D = 31, S = 22, C = 0;

3. test for "ready", D = 31, S = 28, C = 0;

4. test for punch switch on, D = 31, S = 17, C = 1;

5. test for non-zero, D = 27, S = any memory line.

Test for overflow:

You can see that this is a special command (D = 31). It commands the computer to test the condition of the overflow flip-flop. If it is off (no overflow), the next command will be taken from N (as usual). If it is on (overflow), the next command will be taken from N + 1 (thus changing the path of the program). In our program, this path will not contain further computation, but will halt the program (the command to halt computer operation has not yet been discussed).

We will use this test immediately following additions and subtractions.

The manner in which the computer determines the existence of an overflow condition is somewhat indirect, and should be understood. We will discuss it in relation to a single-precision addition in AR.

Three questions are automatically asked by the computer when the addition is performed:

1. Is the intermediate sign of the result 0 (= +)?

2. Did the inverting gates complement the last number to enter AR?

3. Was there an end-around-carry out of bit T29 of AR?

The computer uses the answers to these questions in order to determine whether or not an overflow was generated in the following way:

1. If the answer to question (1) is "no", the intermediate sign of the result is 1 (= -), the two numbers added were of unlike sign, and overflow could not result. If the answer is "yes", the intermediate sign of the result is 0 (= +), the two numbers added were of like sign (both + or both -), and overflow could result. In this case only, proceed to (2).

2. If the inverting gates did not complement the last number to enter AR, both numbers were positive; if the inverting gates did complement the last number to enter AR, both numbers were negative. In either case, proceed to (3).

3. If both numbers were positive, and an end-around-carry did occur, overflow is present; if both numbers were positive, and an end-around-carry did not occur, overflow is not present.

   Examples:

   if a + b = c,

If both numbers were negative, and an end-around-carry did occur, overflow is not present; if both numbers were negative, and an end-around-carry did not occur, overflow is present.

Examples:

if a + b = c,

The importance of understanding the method in which the computer determines the presence or absence of overflow is pointed up by the following example:

Assume we wish to double a negative number by adding it to itself:

$$2\cdot(-a) = -a + (-a) = -2a.$$

We could transfer the number (-a) into AR, replacing the original contents of AR, with a properly coded command containing C = 1 and D = 28. The number will be complemented on its way to AR, and it will be ready for addition. Now we could transfer the contents of AR to AR, calling for an addition (S = 28, D = 29). In this latter command, however, C must equal 0, so that the complement form of the negative number will be retained. We would therefore write a command with C = 0, S = 28, D = 29.

But look at what happens to us when we attempt to check overflow.

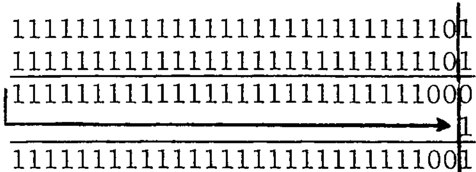
Assume the number was $-2\circ2^{-28}$:

0000000000000000000000000010|1.

After execution of the first command mentioned above, AR will contain:

1111111111111111111111111110|1,

which is the complement of the original negative number, ready for addition. Because of the C = 0 in the second command discussed above, the following addition will be performed in AR:



and this is a valid answer, being the complement of $-4 \cdot 2^{-28}$. We can see that no overflow occurred. But the computer believes that two positive numbers were added, because (1) the intermediate sign of the result is 0 (= +), and (2) the inverting gates did not complement the last number to enter AR (indeed they could not, because C = 0, and the number did not pass through them at all). The computer is aware that an end-around-carry has occurred in AR, through the addition of two positive numbers, and overflow is indicated, the overflow flip-flop being automatically turned on. Therefore, if we test for a possible overflow after this addition, the test will be answered "yes", even though, in reality, no overflow occurred.

Therefore, if we have a number, a, whose sign could be either + or - at the time the program is operated, and if we wish to generate 2a by adding a to itself, and if this could result in a true overflow, necessitating an overflow test following the addition, the best method would be to transfer a from its storage location in memory to AR twice, each time with C = 1. In the first transfer D will be 28, and in the second transfer D will be 29. Now the sum can be checked reliably for overflow.

Test for sign of AR (neg.):

This, too is a special command. It commands the computer to test the sign-bit of the number in AR.

If it is off (0), the next command will be taken from N (as usual). If it is on (1), the next command will be taken from N + 1 (thus changing the path of the program). In our program, this path will not contain further computation, but will halt the program.

We will use this test to determine the sign of the redicand prior to taking the square root; we want to halt rather than take the square root of a negative number.

Test for "ready":

This is a special command. It commands the computer to test for the presence of the "ready" state of the input/output system which we have not, as yet, discussed. We will explain the effect and use of this test later, when we discuss inputs and outputs.

Test for punch switch on:

This is a special command. It commands the computer to test the setting of an external switch. Again, since this test is associated primarily with outputs, we will discuss it later.

Test for non-zero:

Notice that D = 27; this is the only possible line number that has not yet been discussed. This is the only case in which this number is permissible as a destination.

If C = 0, 29 bits of S.T will be tested for non-zero.

If C = 4, 58 bits from the double-precision number contained in S.T and T + 1 will be tested for non-zero.

If C = 1, 29 bits of S.T, after passing through the inverting gates, will be tested for non-zero.

If C = 5, 58 bits of the double-precision number contained in S.T and T + 1, after passing through the inverting gates, will be tested for non-zero.

If S = 28, and C = 2, the magnitude of the number will be tested for non-zero.

If S < 28, and C = 2, all 29 bits of the original contents of AR will be tested for non-zero, and the contents of S.T will be placed in AR.

If C = 6 (S < 28), during the first word-time of execution (even), all 29 bits of the original contents of AR will be tested for non-zero, and S.T will be placed in AR. During the next word-time of execution (odd), AR's contents (S.T) will be tested for non-zero and S.T + 1 will be placed in AR.

If S = 28, and C = 3, the sign of AR will be changed and all 29 bits, after passing through the inverting gates, will be tested for non-zero.

If C = 7 (S < 28), the operation will be the same as for C = 6, except that numbers entering AR will enter via the inverting gates.

In the case of two-word registers, IP will never be tested for non-zero.

. . . . .

Now that we know the test commands that are available, we can incorporate them into our program at strategic places, in order to prevent the output of erroneous answers.

One situation we want to prevent is the division of a number by another which appears to be smaller in value in the computer. In the case of each of the two divisions we have called for, we generate the numerator by shifting $(-b \pm \sqrt{b^2-4ac})2^{-21}$ right 21 places, arriving at a double-

precision value in ID equal to $(-b \pm \sqrt{b^2-4ac})2^{-42}$. We want to assure ourselves that this value is less than the double-precision value, $2a \cdot 2^{-21}$. To do this we must subtract the absolute value (positive magnitude) of $2a \cdot 2^{-21}$ from the absolute value of $(-b \pm \sqrt{b^2-4ac})2^{-42}$, and inspect the sign of the result.

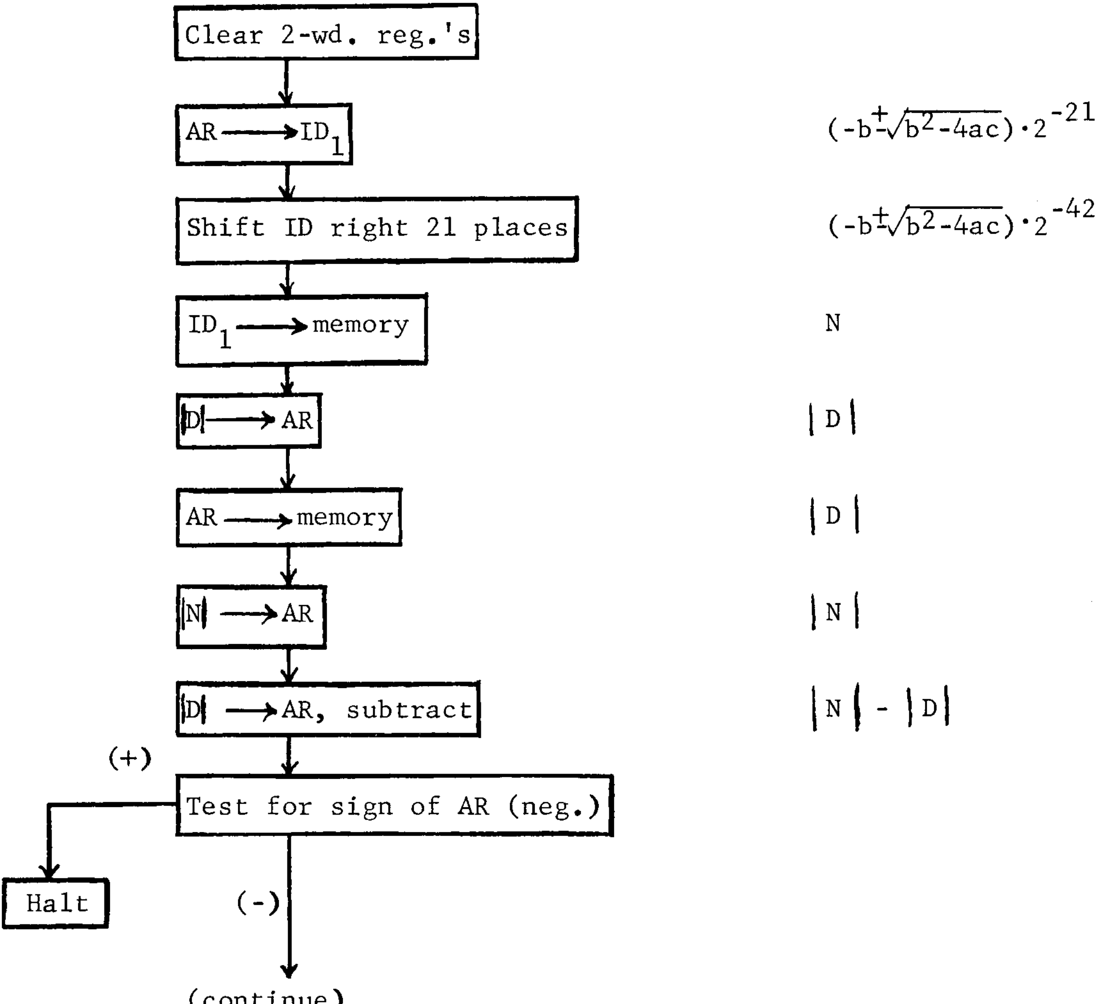
TO SUBTRACT A MAGNITUDE

You probably noticed, in the discussion of the various transfers possible in the G-15, there was no mention of "subtract magnitude" or of "clear and subtract magnitude", although mention was made of "add magnitude" and "clear and add magnitude". The reason for this is that no single transfer available will cause the subtraction of a magnitude. This is most easily accomplished through a series of transfers. We can clear and add the magnitude of D into AR, then store AR's contents in memory. In memory, then, we will have $|D|$. Now we can clear and add the magnitude of N into AR. AR will now contain $|N|$. Now we subtract from AR the number in memory which equals $|D|$. The result in AR is $|N| - |D|$. This number may be either positive or negative, depending on whether $|D|$ exceeds $|N|$ or not. Notice that the result in AR will be positive if the two magnitudes are equal (+0). If we subtract the absolute value of the denominator from the absolute value of the numerator, and get a negative result, we know that the magnitude of the denominator exceeds that of the numerator, and division is permissible. If the result is positive, we know that the magnitude of the numerator either exceeds, or is equal to, that of the denominator, and in either case division is not permissible.

We therefore want to clear and add the absolute value of the numerator into AR. But, the numerator is a double-precision value, scaled 2^{-42}. The denominator, likewise, will be treated as a double-precision value, scaled 2^{-21}, but we know that, when D is in ID, all the least significant magnitude bits, from T1 of ID_1 through T2 of ID_0, will be 0's, because all we do to generate it is to transfer the single-precision number, $2a \cdot 2^{-21}$ into ID_1, being careful to clear the rest of ID. If the most significant 28 bits of magnitude of the numerator equal or exceed the most significant 28 bits of the denominator, we can be sure that the numerator at least equals the denominator, and we cannot divide. We can pick up the first 28 magnitude bits of the numerator from ID_1, following the shift, and leave T1 of that word behind, by transferring out of ID_1 with a C = 0. Carried with the 28 magnitude bits will be the original sign of the numerator, from IP. We'll store this number in memory. Then we'll clear and add the magnitude of D ($= 2a \cdot 2^{-21}$) into AR, and store it back in memory, calling it $|D|$. Now we can clear and add $|N|$ (C = 2) into AR, and subtract $|D|$. The result, in AR, will be $|N| - |D|$. If this result is positive, we cannot divide; if it is negative, we can. Therefore, we will use the "test for sign of AR (neg.)" command. Our computation will proceed with the command located at an address one greater than the N of the test command. The command at the location with an address equal to N of the test command will call for a halt.

The "halt" command is another special command; D = 31, S = 16, C = 0. This command is very easy to explain: its execution causes the computer to stop.

Now we can rewrite those two portions of the flow diagram preceding the divide operations.



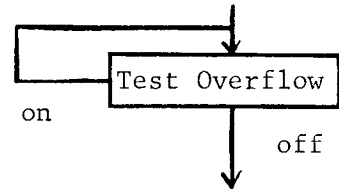
As for overflow errors, unfortunately we cannot test for overflow after shifting MQ left in order to rescale 2a and 4a, since the overflow flip-flop is not connected to MQ during a shift operation, although it is during a divide. If the limit for a is exceeded, an overflow may occur at either or both of these times, and we will be unable to detect it. But we can prevent computation from proceeding if any of the additions or subtractions causes an overflow.

The first point in the program where we can detect an overflow is the generation of b^2-4ac in PN. If we follow this subtraction with an overflow test, the computer will test for the overflow flip-flop being set. Unfortunately, it could have been set earlier, by another program. Once the overflow flip-flop has been set, it can only be turned off in either

of two ways. One is to turn off the computer (not with a halt command, but through an actual switch action which turns off the power). The other way it can be turned off is by the overflow test itself. In addition to testing the flip-flop, this command resets it to the "off" position if it was on. A previous program run in the computer may have turned on the overflow flip-flop and never tested it. In such a case it will remain on.

In order to be sure the overflow flip-flop is off prior to the execution of those steps in our program which could turn it on, we will precede them with an overflow test whose only function is to turn off the flip-flop. The fact that this test command will start either of two alternate paths through our program now becomes a hindrance rather than a help, because we want to continue with the same sequence, regardless of the condition of the overflow flip-flop. We can solve this problem by placing the same command at both N and N + 1, so that, no matter which will be taken as the next command, the same operation will be performed. These two commands will have the same N, so that, following either of them, the same path will be followed through the logic of our program.

Another method is commonly used, however, to achieve the same net effect. We will use it in this program, in order to familiarize you with it. We know that, if the overflow flip-flop is on, the next command will be taken from N + 1. Suppose we choose an N for the test command such that N + 1 = the location of the overflow test command itself. If the overflow flip-flop is off, the program will continue with the command located at N, which is one word-time earlier than the test command. This is fine; the test command will not be read and interpreted again. If the overflow flip-flop is on, the next command will be taken from N + 1, which is the location of the overflow test command itself. This means the test will be repeated. But, the first time the test was made, the flip-flop was reset to the "off" condition. Therefore, this time, when it is tested, it will be found to be off, and the program will continue at N. No matter which condition the flip-flop is in when the test is initially given, the program will eventually continue at N.



If we precede the generation of b^2-4ac with this operation, we can follow the subtraction with another overflow test. This time, the program will halt if overflow is found (N + 1), and it will continue if overflow is not found (N). Notice that, from now on, if the program continues, we need not reset the overflow flip-flop in the manner described above, prior to testing it after a series of arithmetic operations.

Other points at which we want to test for overflow will be following the generation of:

1. $-b + \sqrt{b^2-4ac}$;

2. $\frac{-b + \sqrt{b^2-4ac}}{2a}$;

3. $-b - \sqrt{b^2-4ac}$;

4. $\frac{-b - \sqrt{b^2-4ac}}{2a}$.

We want to include one other test in the program; a test of the sign of (b^2-4ac) prior to attempting to compute the square root of it. If this difference is negative, we want to halt. We will, of course, use the command which tests the sign of AR (neg.). If the answer is yes, the next command will be taken from N + 1, where we will place a "halt". If the answer is no, the program will continue at N.

This completes the use of test commands in the computation.

SUBROUTINES

Up to this point we have very neatly evaded the issue of computing a square root in a computer not wired to do it directly. It must be done through a series of basic arithmetic operations. We can no longer evade it, however; it's the only portion of the computation remaining unplanned. How are we going to do it? A mathematician-turned-song-writer-and-performer has answered our question in one of his songs:

"Plagiarize, plagiarize, plagiarize.
Let no one else's work evade your eyes."

Bendix Computer Division, of course, does not recommend or condone plagiarism, but it does supply a standard "package" of programs designed to make life easier for its customers. This package is standard equipment with every G-15 computer. Each of these programs is designed to perform a commonly needed function among computer users. Typical examples are calculation of square roots and trigonometric functions. These programs are called "subroutines". A subroutine operates out of a prescribed command line in memory, with certain inputs, which are also prescribed, and it is usually designed to generate a solution, or set of solutions, which will appear at a prescribed location in memory.

There is a square root subroutine available. The command line prescribed for its execution is line 01. The input, the number whose square root is desired, must be placed in $PN_{0,1}$, prior to execution of the subroutine. The first command is at word-time 94. The answer will appear in $PN_{0,1}$. All of these facts, and others, can be found on a specifications sheet which accompanies a write-up of the subroutine. All subroutines are written-up, and specifications similar to those above are supplied.

If this subroutine is to occupy line 01, certainly our program should not. Assume that our program will occupy line 00. The question, then, is, at the right point in our program, after we have the number whose square root we desire in $PN_{0,1}$, how do we cause the computer to change command lines from 00 to 01, and take its next command from word 94 in the new command line?

There is a special command (D = 31), whose function is to cause the computer to change command lines. This is called the "mark and transfer control" command. In it, S = 21, and C = the line number to which control is to be transferred. The line number specified could be 0, 1, 2, 3, 4, 5, 6 (referring to line 19), or 7 (referring to line 23). The word in the new line at which the new sequence is to start will be located, as usual, in the N portion of the mark and transfer control command.

The mark and transfer control command that we want to incorporate in our program, then, will contain D = 31, S = 21, C = 1, and N = 94.

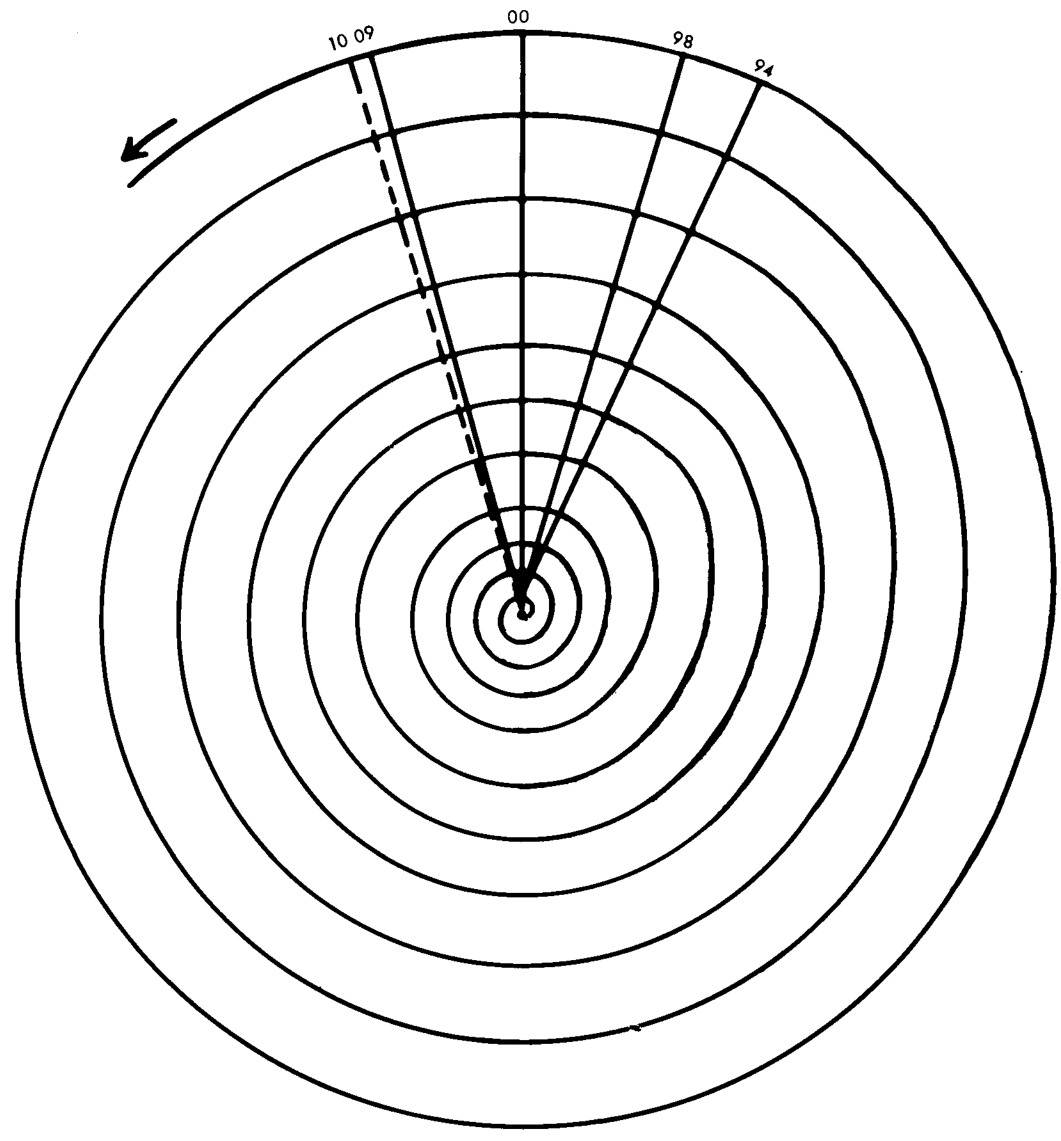
The word "mark" in the name of this command has special significance, other than just making the command sound complicated. In the case of a subroutine, be it one from the "standard package", or one that you write for yourself, there will come a time, after the subroutine has done its work, when you would like it to return to your own main program, in whatever line that might be (in this case, line 00), at a given word-time. Each of the subroutines in the standard package is equipped with a "return" command, which is similar in nature to a mark and transfer control command. In the return command, which is also a special command, D = 31, S = 20, and C = the line number to which control is to be returned. We will call this the "return line". The determination of the word-time at which the sequence in the return line will begin, however, is a bit different than in the preceding case.

When a mark and transfer control command is executed, a "mark" is generated electronically in the computer, at the word-time immediately preceding the word-time of execution. In other words, if the command is executed at word-time 10, the mark will be at word-time 09. If the return command is properly made up, it will cause the computer to sense this mark, after the return line has been selected, and the computer will take its next command, in the return line, from the next location. If the mark is at word-time 09, the next command will be taken in the return line from word-time 10. Notice that this was the time of execution of the original mark and transfer control command in that same line: it was not the word-time in which that command was located.

Thus, if the mark and transfer control command and an accompanying return command are properly made up, a transfer of control to a new line will be effected at the proper word-time in the new line, and, while still taking commands from the new line, the computer gets a command directing it to return to a return line (which usually will be the line from which control was originally transferred, in our case, line 00, although this does not have to be the case); it will return control to the command line specified, and take its next command at

a word-time corresponding to the word-time of execution of the original mark and transfer control command.

In the following drawing the spiral indicates the passage of time, in the direction shown by the arrow, which is outward. As the spiral passes the heavy vertical line, which represents word-time 00, a complete drum cycle has elapsed. Ten complete drum cycles are shown, in the center of the spiral a fraction of another cycle is shown, and, on the outside of the spiral, a fraction of still another drum cycle is shown.



Let us refer to that drum cycle of which only a fraction is shown in the center of the spiral as drum cycle 1. Then, counting outward, drum cycles 2 through 11 are completely shown, followed by a fraction of drum cycle 12.

During drum cycle 1, assume control of the computer is in command line 00. It remains there up through word-time 10 of cycle 2. Sometime before word-time 10 of cycle 2, a mark and transfer control command is read in line 00, calling for a transfer of control to line 01 at word-time 94. This command is executed at word-time 10 of drum cycle 2, shown in the drawing. Beginning with word-time 11 in drum cycle 2, control is in command line 01, and the computer is waiting for the next command, which it will read at word-time 94 of drum cycle 2. This command will come from word-time 94 in command line 01. An "electronic mark" was generated by the mark and transfer control command, at the word-time whose number is one less than that of the word-time of execution of the command itself.

If the command is executed, as we say, at word-time 10, this mark will be at word-time 09, as shown in the drawing. Such a mark will last indefinitely, being turned off, or erased, only by either the creation of another "mark" by a similar command, or turning off the computer. The square root subroutine, in line 01, is now operating, starting at word-time 94 of drum cycle 2. It continues for approximately 9 drum cycles (stated in the specifications), through word-time 99 of drum cycle 11, in the drawing. At word-time 98 in command line 01, which is finally reached in the last drum cycle of execution during the square root subroutine, a return command is located. This is specified in the write-up of the subroutine, and will be, for all subroutines. This command specifies the command line to which control is to be transferred (returned), in our case, line 00, and it enables the computer to sense the mark (currently coming up at the next word-time 09). Beginning at word-time u0 (100) in drum cycle 11, control has been returned to line 00, and the computer is looking for the next command. It will find the next command at either of two locations: the location specified by the N of the return command, or the marked location. Which of these will contain the next command the computer will read is determined by which arrives earlier. The early word gets control.

It therefore becomes the programmer's responsibility to see to it that the return command in the subroutine is timed in such a way that the marked location cannot be missed. He cannot place this return command in any location in the subroutine other than the one specified, which, in this case, is word-time 98. But he can set up the command so that the marked word-time will have to come up before the word-time specified by N in the return command.

We have set word-time 10 as the next command in our program, upon the return from the square root subroutine, for purposes of example. We want to be sure that word-time 09, which bears the mark, will come up before N of the return command. We could set N of the return command in the subroutine equal to 10. In that way, the location picked by the mark and the location picked by N of the return command would coincide, and there would be no doubt as to which word in line 00 would be the first to be interpreted as the next command.

But notice, if we could make up a general return command in such a way that we would always return to the marked word-time in the same return line, no matter what that marked word-time might be, we could then use

the same return command in the subroutine, no matter how many times in the course of the program we wanted to enter the subroutine. In our particular example, we only use the square root subroutine once, but it is not inconceivable that some programs would use it literally dozens of times. It is possible to make up a return command for any subroutine in such a way that it can be used over and over again, each time returning control to the same line, but at a different word-time, depending on where the mark is currently located. Remember it was said that a mark is erased by the setting of a new one. Only one mark may exist in the computer. This general return command is ideal, because now the place at which the main program picks up, after receiving control back from the subroutine can, in each case, be picked through the setting of the mark and transfer control command which transfers to the subroutine. One time we could set the T of the mark and transfer control command equal to 10, the next time, to 90, and so on. The main program will pick up at word-time 10 after the first use of the subroutine, at word-time 90 after the second use of the subroutine, and so on.

Such a general return command is made up in the following manner, for the reasons indicated. Make the command immediate, and let it be executed for one word-time. This can be done by setting T equal to the location plus two. In this case, the immediate command will be read at word-time L (location), and the immediate execution will begin in the next word-time, L + 1. The T number will act as a flag, as mentioned previously in the discussion of immediate commands. Since T = L + 2, the operation will be stopped after word-time L + 1, and therefore will last only one word-time. During this word-time, the computer will begin to search for the existing mark. The mark, when found, will be rejected unless it is in the last word-time of execution of the return command or later. Since there is only one word-time of execution of this command when coded in this form, it is also the last. Therefore, the search for a mark will begin at L + 1 and continue until the mark is found, and the next command in the return command line will be specified by the location of the mark. Let N = L + 1 in the return command; it cannot specify the next command in the return command line, because N = L + 1 and cannot be effective for one whole drum cycle. The mark must be found and become effective at some time during the drum cycle; the worst case would be the one in which the mark is at L, determining L + 1 (= N) as the location of the next command. (A detailed description of the occurrence of machine signals in this regard follows: do not attempt to master it on the first reading of this text.)

Drawing 1 shows pictorially what will be the effect of a mark when the return command is coded properly: T = L + 2, N = L + 1. The return command is located at word-time 12 and executes during word-time 13, as shown on the time-spiral. Because word-time 13 is also the last word-time of execution a mark sensed at that time will be effective.

The first possible location of the next command in the return command line, therefore, is 14, as indicated by the X in the drawing. If the mark is not found at 13, the search will continue until it is found, and in the drawing this is seen to be word-time 37. The next command therefore, in the return command line, will be taken at 38. The worst case

would be the one in which the search continues for a whole drum cycle. The mark would be found during word-time 12. During word-time 12 there will also be a signal telling the computer to read a command from the next word-time, this signal having been generated by the N specified in the return command itself. Thus, in the worst possible case both the mark signal and the N signal will be present during the same word-time, 12. The N signal is interrogated at T21, whereas the mark signal is interrogated at T13. Therefore, even in the worst case, it will be the mark signal that picks the next command, and not the N signal.

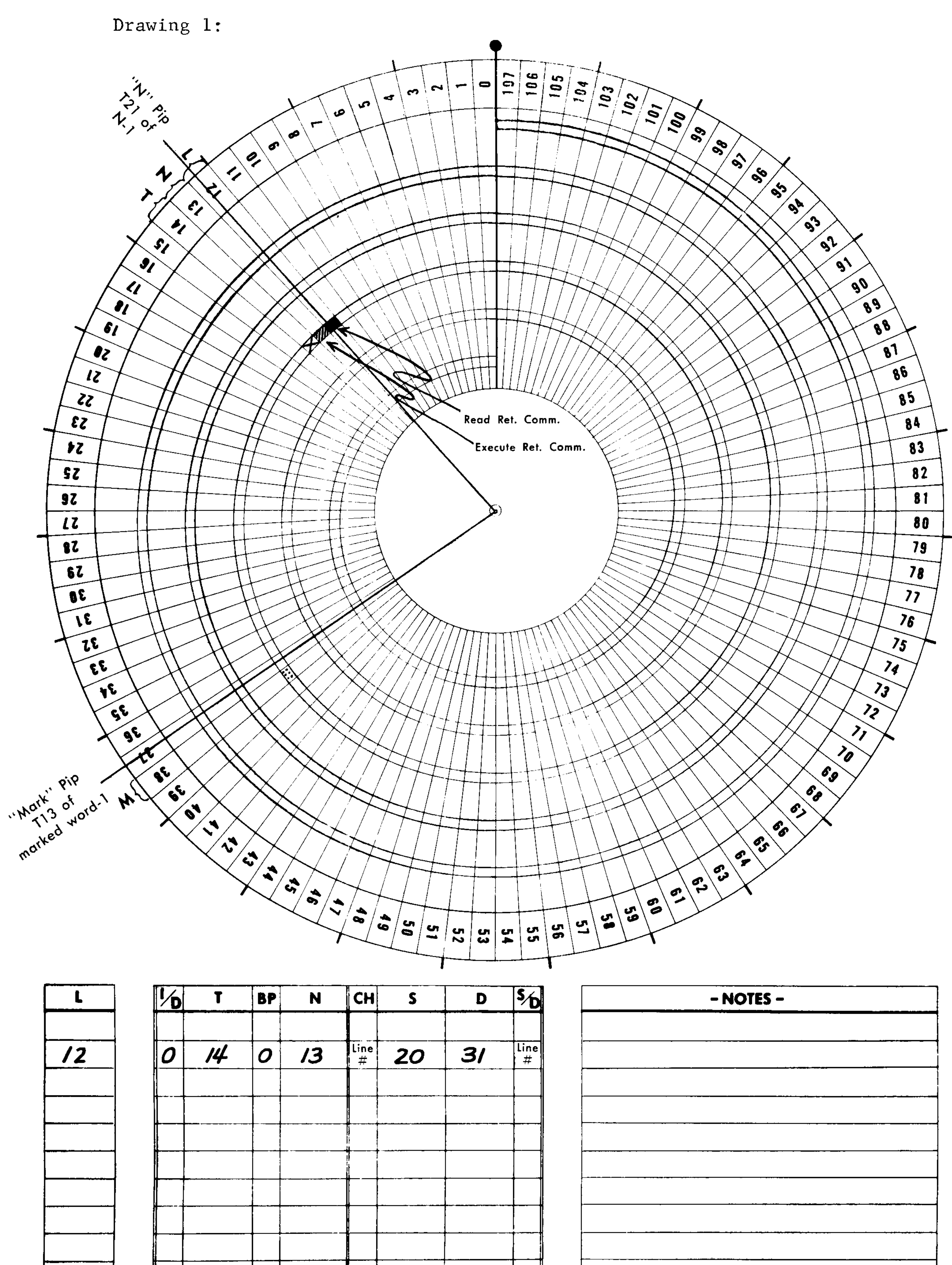
Drawing 2 shows pictorially the operation of the return command when coded in a different manner. In this case the command is executed dur- word-times L + 1 through N - 1. Shown in the drawing, L = 12, N = 27, and the word-times of execution are 13 through 26. The location of the next command to be taken from the return command line cannot be determined by any signal occuring during execution time, unless that signal occurs during the last word-time of execution. The N signal does occur during word-time 26, because N = 27; therefore, the location = N will arrive earlier than any existing marked location, unless the mark is also present during word-time 26. In any event, the next command will be taken from the return command line at word-time N. This is a convenient way to program a transfer of control to line C, word N, without setting a new mark, and therefore allowing an already existing mark to remain.

In drawing 3, where no care has been used in formulating the return command, L = 12, T = 44, and N = 20. The command will execute from 13 through 43, so the first mark or N signal occuring at 43 or later will determine the word-time at which the next command will be taken in the return command line. The drawing shows an existing mark at word-time 37, and we know that the N signal will be, as shown in the drawing, at 19. It is evident, then, that the next such signal will not occur until the following drum cycle, at word time 19. This will be the N signal, and so the next command in the return command line will be at 20 (N). This third case makes it apparent that care should be exercised in making up return commands. The first method, T = L + 2, N = L + 1, is the best.

Up to this point we have been speaking rather blithely about setting the return command in a subroutine; now we will see how. Every subroutine requires, as an input, a return command. Most subroutines require it to be placed in AR. Regardless of the location specified, the return command must be placed there prior to transferring control to the subroutine. One of the first steps in any subroutine, then, is to pick up the word containing this return command and transfer it to the proper location in the subroutine, so that, when that location is finally reached, in the course of the subroutine, the proper return command will be there to be read and executed. In the case of the square root subroutine, the return command must be placed in AR prior to entering the subroutine.

Notice that a command, in this case the return command, can be treated as data. If it is read during execution time, rather than during read command time, the computer will be unable to tell the difference between

Drawing 1:

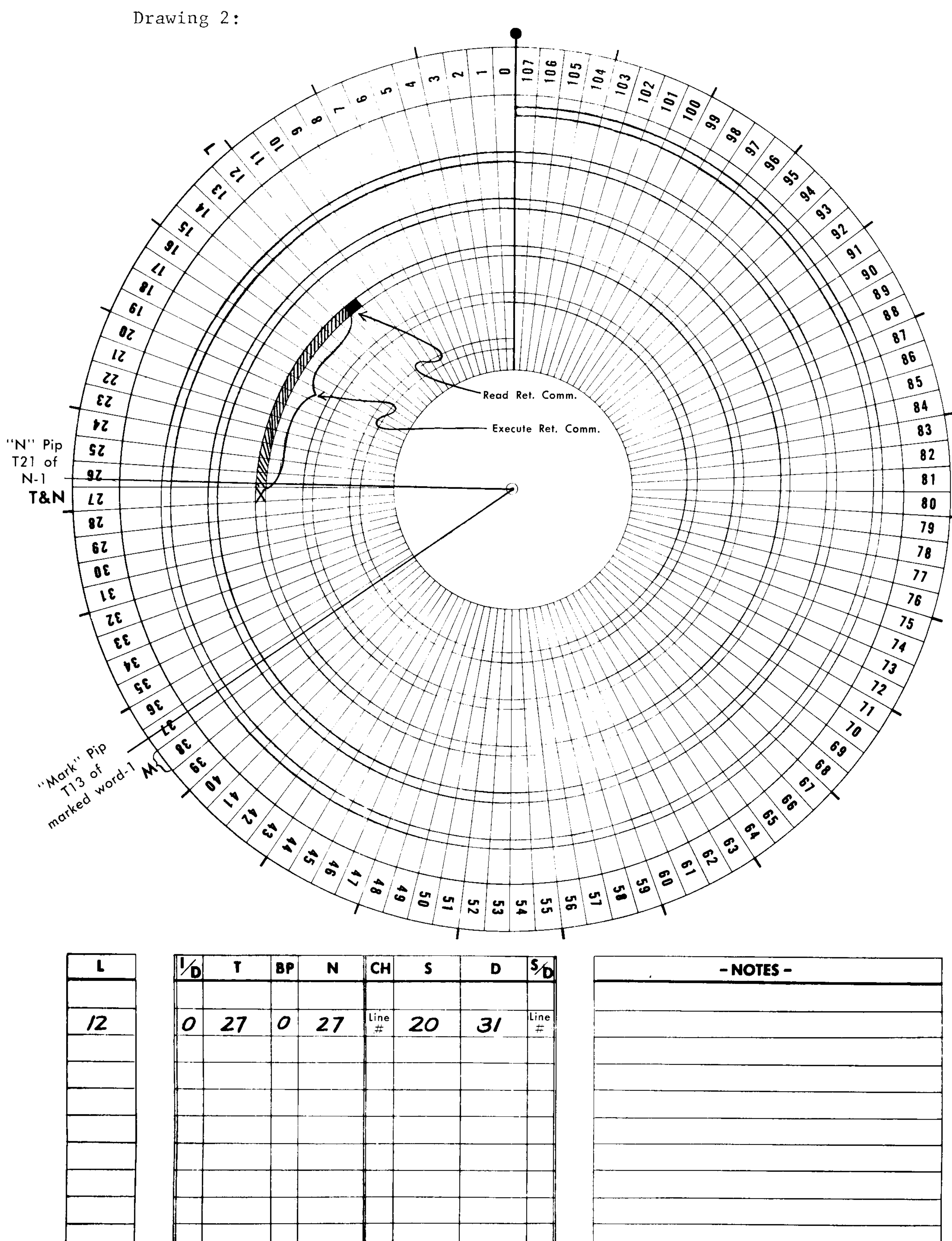


L
12

I/D	T	BP	N	CH	S	D	S/D
0	14	0	13	Line #	20	31	Line #

- NOTES -

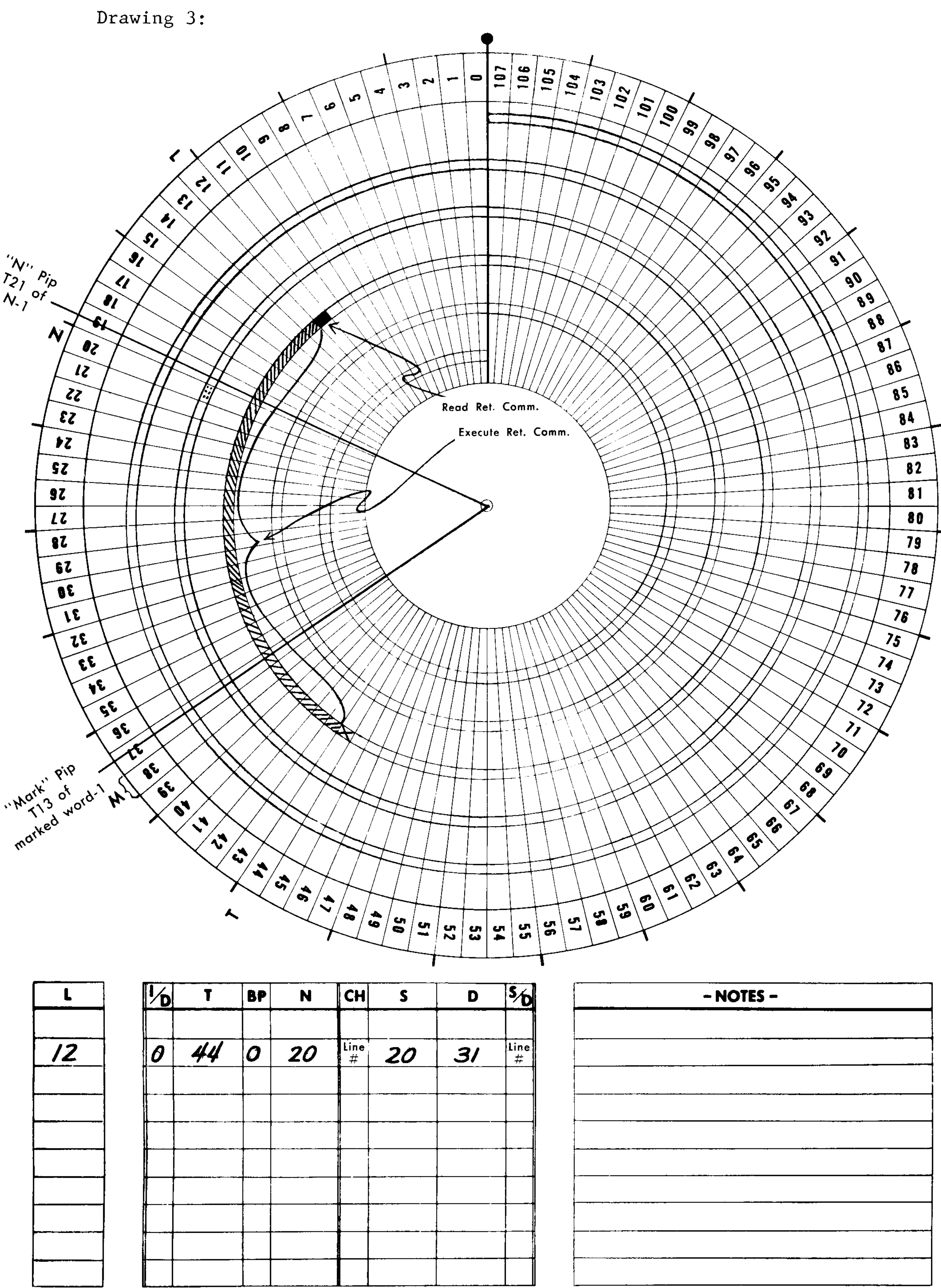
Drawing 2:



L
12

I/D	T	BP	N	CH	S	D	S/D
0	27	0	27	Line #	20	31	Line #

Drawing 3:



L
12

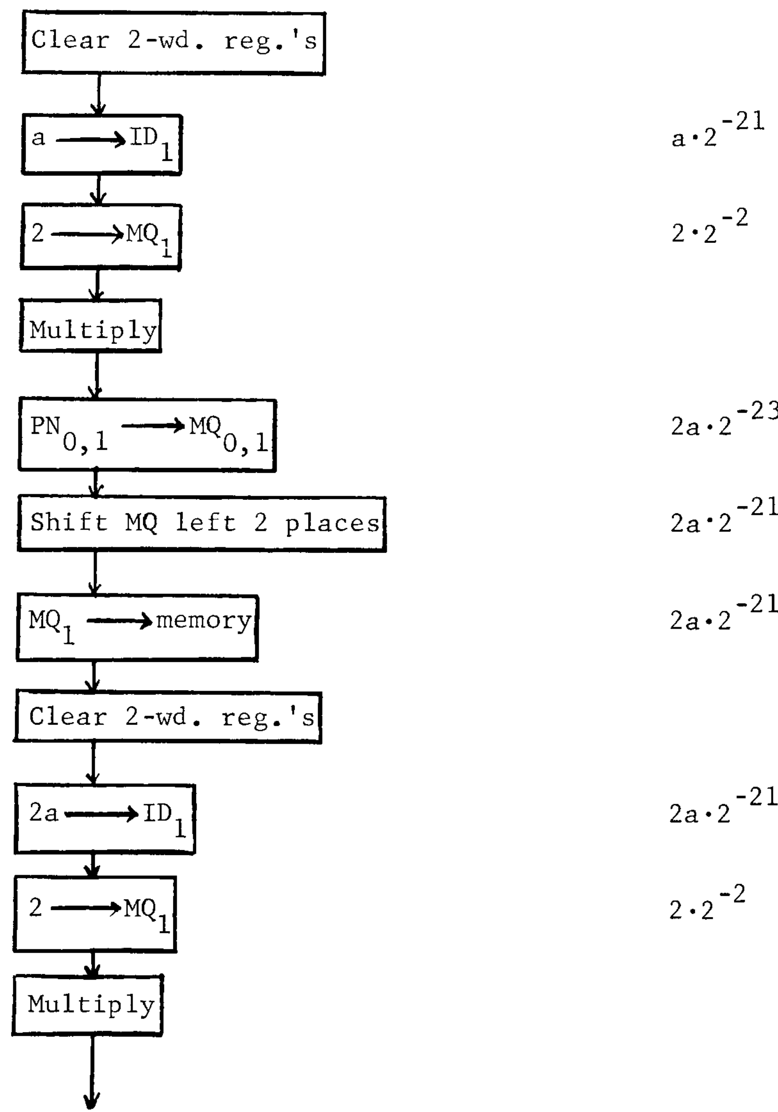
I/D	T	BP	N	CH	S	D	S/D
0	44	0	20	Line #	20	31	Line #

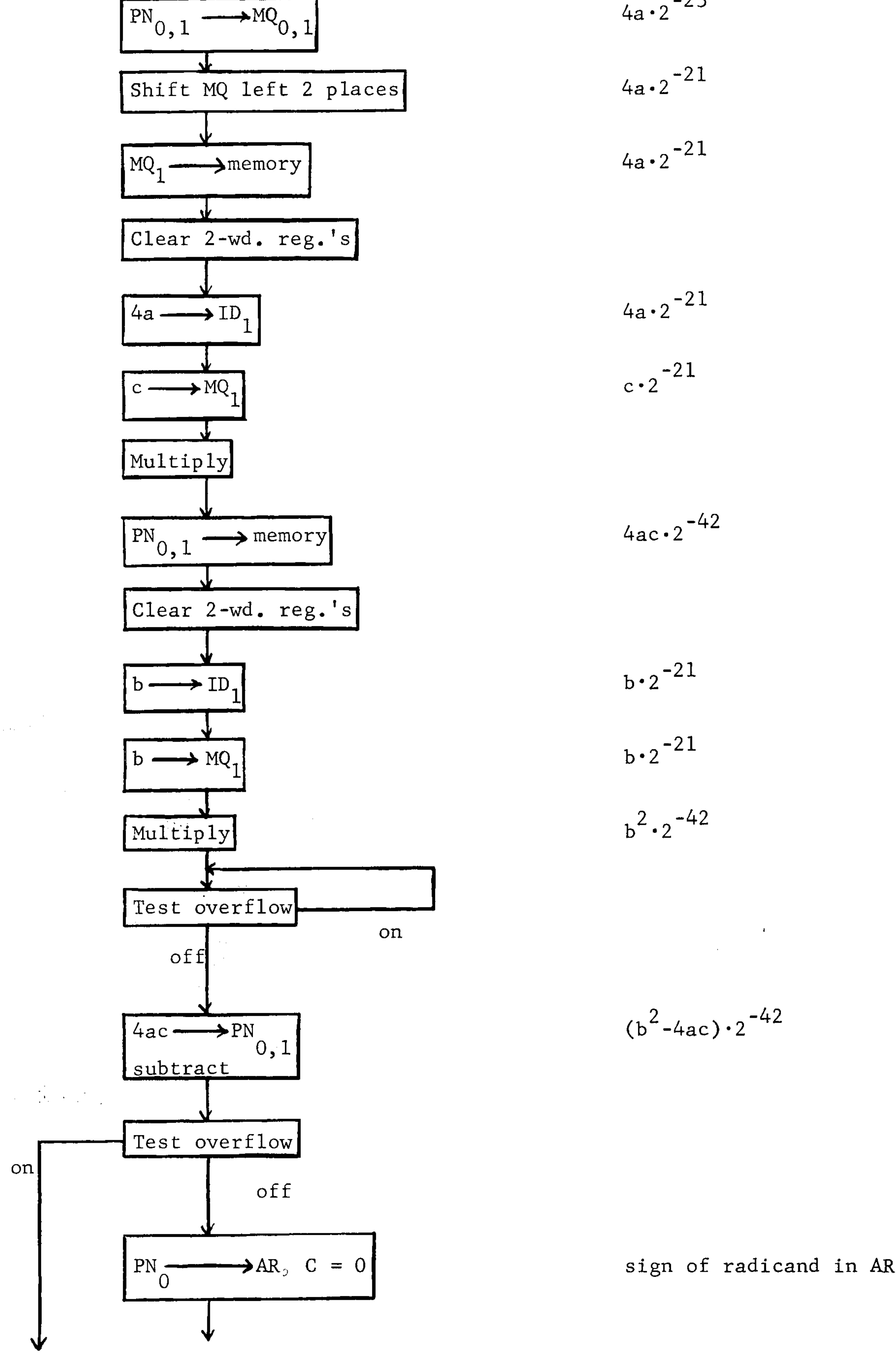
it and legitimate data. Thus, a command could not only be placed in AR, but it could, while there, be modified through the addition of a constant.

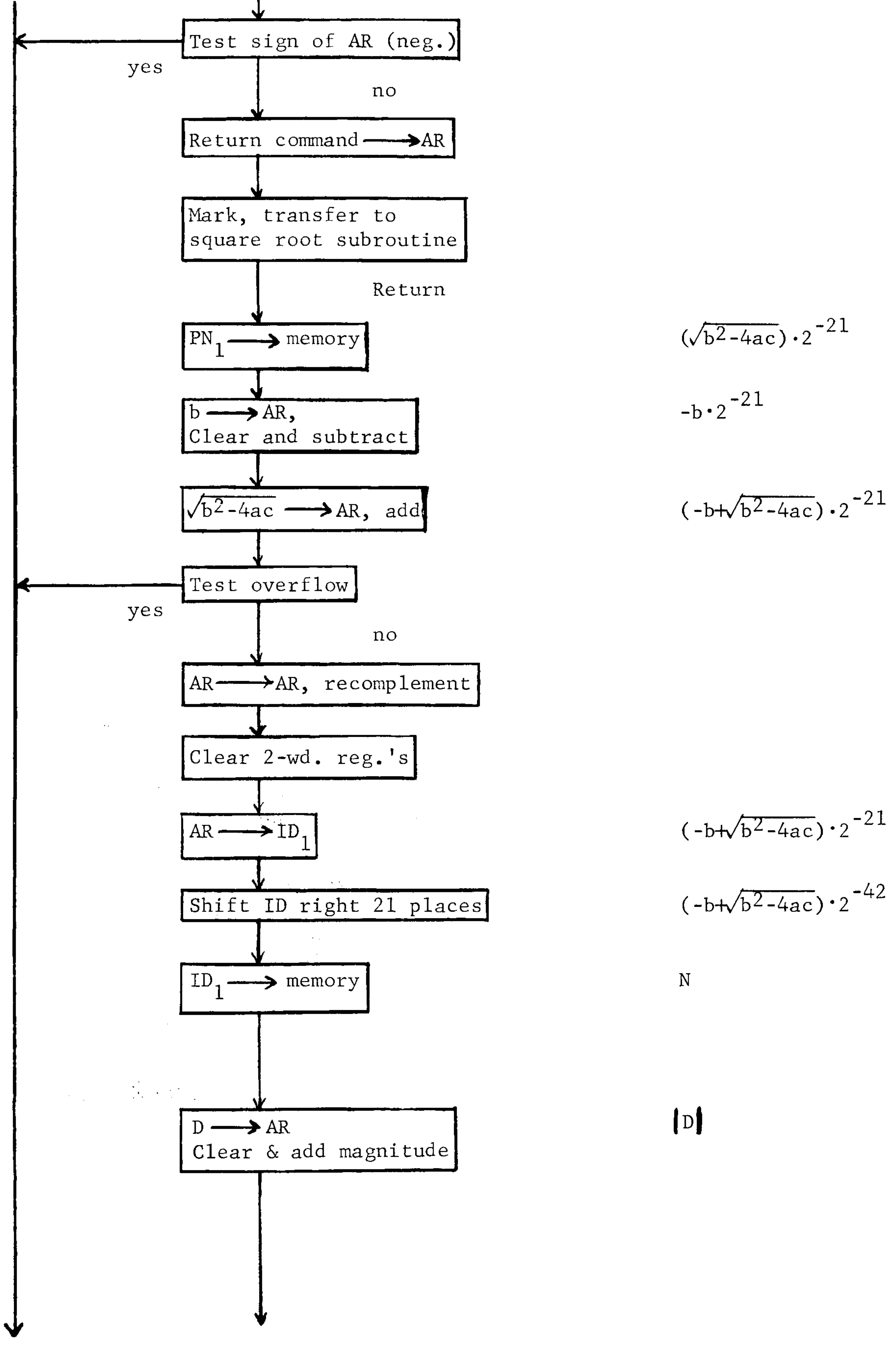
This leads us to a brief discussion of another special command, which tells the computer to "take the next command from AR". This, too, is a special command, containing D = 31, S = 31, and C = 0. Thus, a command could be transferred to AR, modified there, and executed out of AR, in its modified form. The N of the "take next command from AR" command will be the word-time at which the command in AR will be read.

REVISED FLOW DIAGRAM

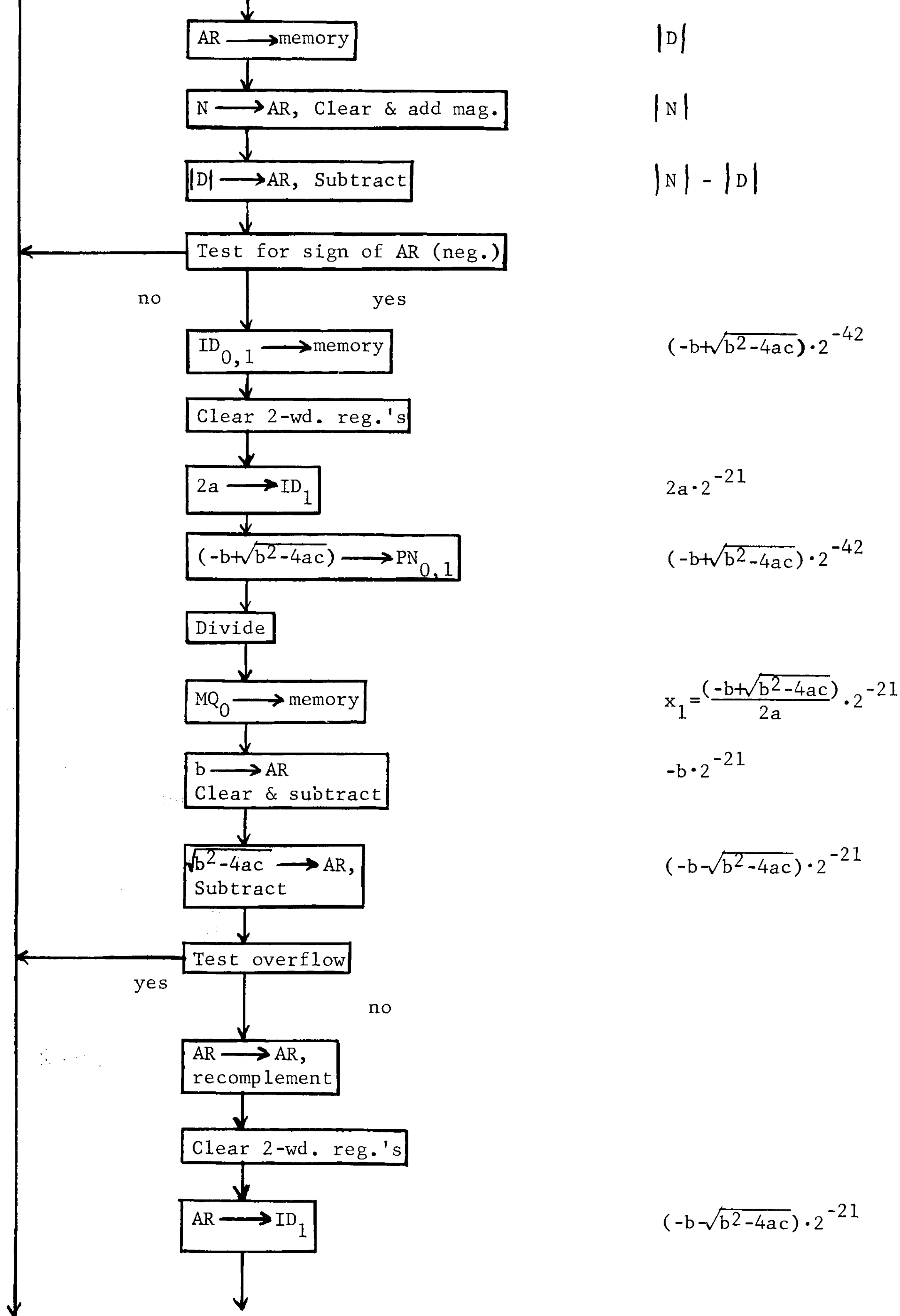
Having determined which test commands we must incorporate, and how to set up for, enter, and exit from, the square root subroutine, we can revise the expanded flow diagram for the solution of the quadratic equation.

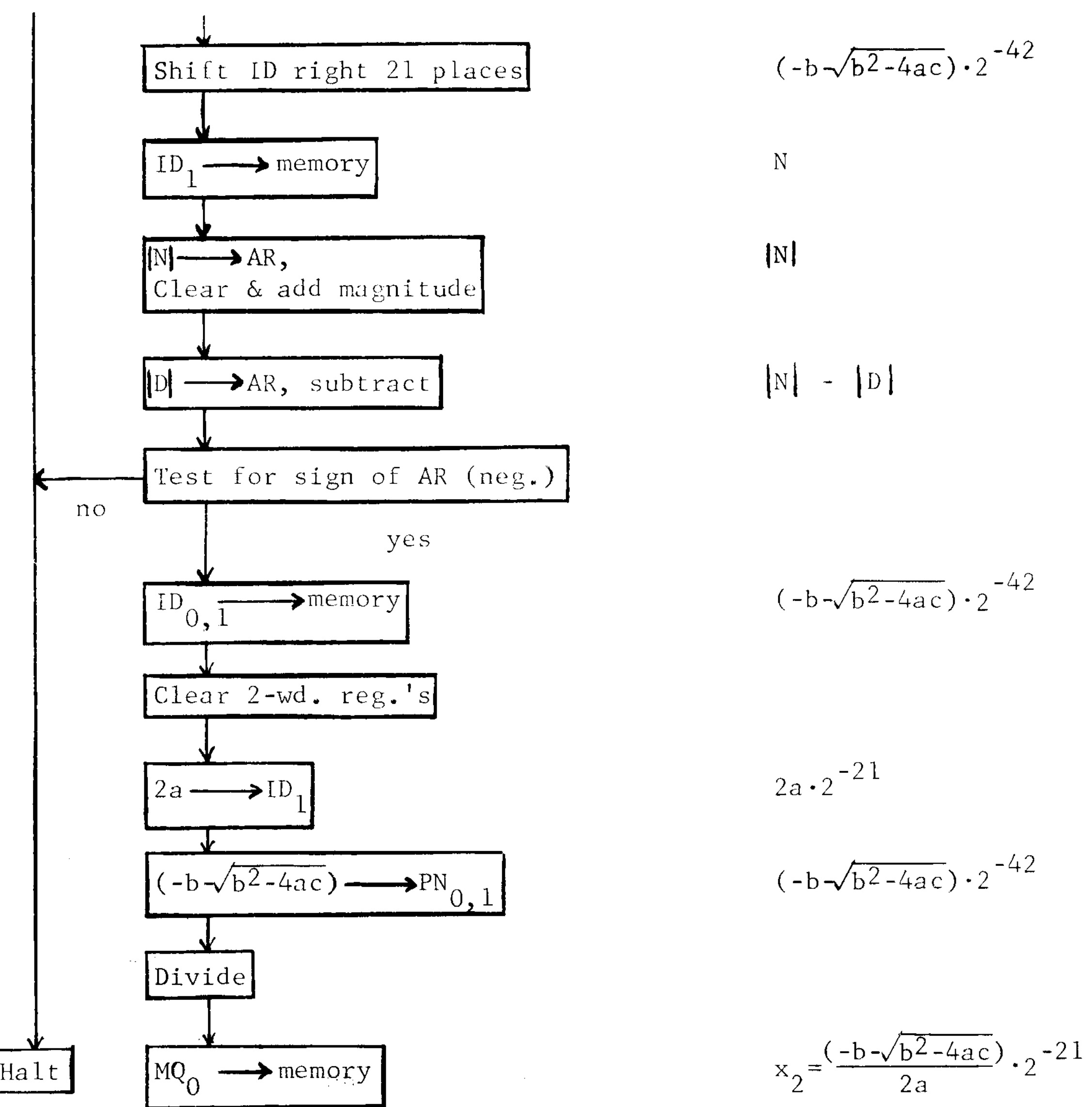



PN0,1 → MQ0,1
4a·2-23
Shift MQ left 2 places
4a·2-21
MQ1 → memory
4a·2-21
Clear 2-wd. reg.'s
4a → ID1
4a·2-21
c → MQ1
c·2-21
Multiply
PN0,1 → memory
4ac·2-42
Clear 2-wd. reg.'s
b → ID1
b·2-21
b → MQ1
b·2-21
Multiply
b2·2-42
Test overflow
on
off
4ac → PN0,1
subtract
(b2-4ac)·2-42
Test overflow
on
off
PN0 → AR, C = 0
sign of radicand in AR



Test sign of AR (neg.)
yes
no
Return command → AR
Mark, transfer to square root subroutine
Return
PN_1 → memory
$(\sqrt{b^2-4ac})\cdot 2^{-21}$
b → AR, Clear and subtract
$-b\cdot 2^{-21}$
$\sqrt{b^2-4ac}$ → AR, add
$(-b+\sqrt{b^2-4ac})\cdot 2^{-21}$
Test overflow
yes
no
AR → AR, recomplement
Clear 2-wd. reg.'s
AR → ID_1
$(-b+\sqrt{b^2-4ac})\cdot 2^{-21}$
Shift ID right 21 places
$(-b+\sqrt{b^2-4ac})\cdot 2^{-42}$
ID_1 → memory
N
D → AR
Clear & add magnitude
$|D|$


AR → memory
|D|
N → AR, Clear & add mag.
|N|
|D| → AR, Subtract
|N| - |D|
Test for sign of AR (neg.)
no
yes
ID0,1 → memory
(-b+√b2-4ac)·2-42
Clear 2-wd. reg.'s
2a → ID1
2a·2-21
(-b+√b2-4ac) → PN0,1
(-b+√b2-4ac)·2-42
Divide
MQ0 → memory
x1 = (-b+√b2-4ac)/2a ·2-21
b → AR
Clear & subtract
-b·2-21
√b2-4ac → AR,
Subtract
(-b-√b2-4ac)·2-21
Test overflow
yes
no
AR → AR,
recomplement
Clear 2-wd. reg.'s
AR → ID1
(-b-√b2-4ac)·2-21



This is the complete flow diagram for the main computation part of the program: it will occupy line 00. The square root subroutine will occupy line 01. However, the program still lacks a method or "scheme" of input, and any provision for output. We call for a, b, and c from memory, but as yet have made no provision for initially storing them there. Similarly, we generate two answers, x_1 and x_2, the two roots of the quadratic equation, but we have no provision as yet for communicating these carefully derived answers to the outside world. They're still stored away inside the computer. We have also made no provision for stopping the computer, or in any way terminating the main body of the program, although we do halt the computer in the case of error.

The next step in development of the program, now that we know the exact form in which we want the inputs, is to devise an input scheme; it, too, will be flow diagramed, and we will treat it almost as a separate program, although the input scheme is really an integral part of any program.

Because of their similarities, we will discuss inputs and outputs together, and then flow-diagram each method chosen for this particular program. They will, of course, be much shorter than the main body of computation.

INPUTS/OUTPUTS

A general-purpose computer is worthless unless it can receive inputs and yield outputs. The requirements of any input or output system are:

1. compatibility with the central portion of the computer,

2. ability to handle any type of information that may be needed or yielded by the computer,

3. accuracy,

4. speed.

These four requirements are listed in their relative order of importance.

Certainly the input system must be compatible with the central portion of the computer. It must be able to convert, if necessary, incoming information into a form recognizable by the computer. In the case of the G-15, the information must be in the form of electrical pulses which can generate magnetized spots on the surface of the drum, called "bits". In special cases, certain inputs to the G-15 may be electronic "signals", capable of activating a specific circuit or component directly. Such signals might, for example, cause an operation within the computer similar to that which could be caused by the execution of a command in a program. Most signals of this sort, you will see later, will call for an input or output, in the same manner it might be called for by a command in a program. This will not always be the case, however. In any event, whether an input of pulses or signals is necessary, a human operator is not anatomically equipped to supply them directly. He is therefore supplied with a set of buttons and switches, which he can manipulate, and which close and open circuits, supplying the computer with the pulses and signals it needs.

In some cases, the computer can be linked directly to some other system. In such a case, the input system receives electrical inputs, rather than manual inputs, and it must convert these to other pulses and signals in the form needed by the computer. This type of operation is sometimes referred to as "on-line" operation because the computer is on a line with the external system. A typical example of this type of operation might be one in which radar receivers are linked to the computer through some type of device which converts the range pulses and angle of deviation from North of the set itself to binary form. Cases of this type, where the computer is used "on-line" require special peripheral equipment for the computer, which is not supplied as standard equipment. In some cases, actual modification of the computer itself might be necessary.

The inputs with which we will concern ourselves at present are those which can be handled by the standard input equipment, supplied with the computer. These will fall into the "operator" category. We will hereafter refer to them as "normal" inputs.

NORMAL INPUTS

Normal inputs to the G-15 can come from either of two sources:

1. an electric typewriter, which supplies the operator with the buttons and switches he needs, and

2. a photo-electric tape reader, which reads punched tape somewhat similar in appearance to teletypewriter tape.

Since the second source is merely a speedier substitute for the first (a reel of tape is punched in codes which simulate the activation of a typewriter key), and since the typewriter contains a button or switch for every possible pulse and signal, we will discuss the typewriter first.

TYPEWRITER INPUTS

The electric typewriter is connected to the G-15 by a cable, which contains many individual lines. Over these lines pass the various inputs from the keyboard and switches on the typewriter. As has already been mentioned, there are essentially two types of inputs that can be supplied to the computer. One type is electrical pulses, which set up information in the memory of the computer; the other type is "signals", which cause the computer to act, these signals having the same general effect as a programmed command.

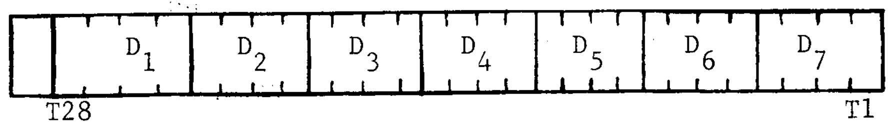
We will consider first the inputs from the typewriter which enter the memory of the computer. They are supplied by certain of the typewriter keys shown in the illustration on page 130. Notice they include all of the sexadecimal digits, 0 through z, the minus sign, the tab, the carriage return, and the slash (/) key. Inputs which enter the memory of the computer, enter at word-times, of course, and they become parts of words. Therefore, they must be in binary notation. The hex number system, as was pointed out earlier, is merely a short cut for binary representation. These inputs could just as well be entered from only two keys on a differently wired typewriter and with some modification of the input-system, one key for "0" and another for "1". It would take 29 punches of these keys to enter one complete word into memory. With the use of hex digits, a complete word can be entered through striking only 7 or 8 keys: seven hex digits and a sign. If the sign is positive, no key is struck; if it is negative, the minus sign key is struck.

Mounted within the typewriter, beneath the key-board, are a set of switches, one for each of these keys. As a key is struck, the associated switch is activated. The corresponding 4-bit code is generated and transmitted to the computer.

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
u	1010
v	1011
w	1100
x	1101
y	1110
z	1111

The minus sign, tab, slash (reload), and carriage return key do not enter 4-bit codes, although they do affect what is stored in memory, as will be explained shortly.

The 4-bit code, when it is generated, is transmitted into word 00 of short-line 23, at the least significant end, so that it occupies bits T1 through T4 of that word. When the next key is struck, assuming for the moment that it, too, will be a hex digit, a new 4-bit code is entered into these same bits, and the preceding one is shifted to the left, into the next four bits, so that the first "character" of input will occupy bits T5 through T8, and the second character will occupy bits T1 through T4, of word 00 in line 23. This process will continue as long as you keep on striking hex digit keys. Finally, after seven of these keys have been struck, word 00 of line 23 will look like this:



All of short line 23 will have been shifted to the left 28 bits. T1 of 00 will be in T29 of 00, the rest of the bits from 00 will be in 01, and so on, and 28 bits from word 03 will have been lost.

If you continue to enter hex digits, line 23 will continue to shift left, four bits at a time, until eventually, if you enter enough hex digits, the first characters of input will begin to "fall out" of 23.03, and will be lost. There must be a stopping-point. But, before we discuss that, let's continue with the input for a moment, where we have seven hex digits in 23.00.

Notice that, in this case, T1 of 23.00 (the sign-bit) contains the least significant bit of the seventh 4-bit code entered. A shift of one bit is necessary if we want to merely complete word 23.00, without losing any bits from the first code entered.

PERMIT ALPHANUMERIC TYPE-IN

EXECUTE ONE COMMAND

RETURN TO MARKED PLACE

READ TAPE & SET COMMAND LINE TO 23, N= 00

STOP INPUT OUTPUT

PERMIT NUMERIC TYPE-IN

N ⟶ AR

TYPE AR

SET N = 00

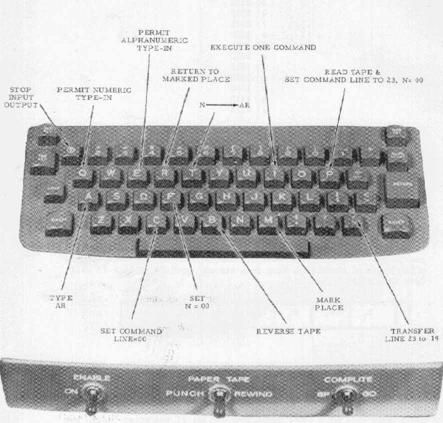
MARK PLACE

SET COMMAND LINE=00

REVERSE TAPE

TRANSFER LINE 23 to 19

TYPEWRITER CONTROL KEYS AND SWITCHES



TYPEWRITER CONTROL KEYS AND SWITCHES

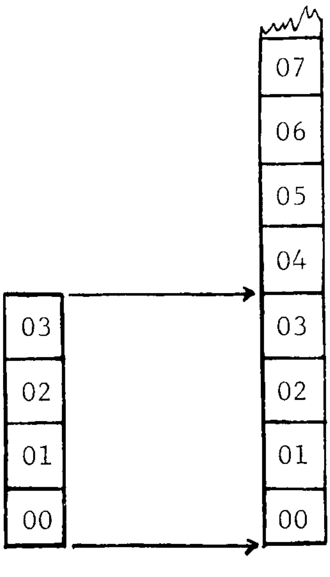
Associated with the input system is a flip-flop, which, like all flip-flops, can be in either of two states. It can contain a 0 or a 1. This particular flip-flop is referred to as the "sign flip-flop". It can be set to 0 (cleared) by striking either the tab key or the carriage return key on the typewriter. It will also be cleared after termination of any input. Once cleared, it can only be set to 1 by the striking of the minus sign key on the typewriter. When either "tab" or "carriage return" is struck, line 23 is shifted by one bit, rather than four, and the content of the sign flip-flop is "dumped" into the sign-bit (T1) of 23.00.

Therefore, if we desire to shift the seven hex digits we have just set up in 23.00 to the left one bit-position, causing them to occupy bits T2 through T29 of 23.00, and place a sign in T1, we strike either the tab or the carriage return key. The present content of the sign flip-flop will become the sign of the binary (hex) number in 23.00.

If the tab key was struck twice in succession, the first input would cause all of line 23 to shift left one bit, and the content of the sign flip-flop (call it x, since it could be either 0 or 1, depending on whether or not the minus key was struck previously) would be entered into T1 of 23.00. The second input would again shift line 23 to the left one bit, placing x in T2 of 23.00, and the content of the sign flip-flop (now known to be 0, since the sign flip-flop was cleared) would be entered into T1 of 23.00. Any succession of inputs is permissible, and the result in line 23 will be predictable. Notice that a minus sign can be struck at any time, but will not enter the computer until the tab or the carriage return key is struck.

Finally, when line 23 is filled as desired, it seems we have run out of space for the storage of inputs. But this is not correct. There remains one typewriter key in the illustration, which has not yet been discussed: the slash (reload) key. Striking this key will cause all of long line 19 to be shifted towards its upper end by four full words, and cause the transfer of all four words in line 23 into the now vacated low-order four words of line 19 (23.00 - 03 $\longrightarrow$ 19.00 - 03).

These four words will also remain in line 23. Input may continue, and eventually line 23 will be refilled with new inputs (the old words will be pushed out of the high-order end of the line; where they go, nobody knows. It has been said they go where a light goes when it goes "out".), and the slash key may be struck again. Line 19 will shift by four words again, and the four words in line 23 will be transferred into the four low-order words of line 19. Thus eight words have entered the computer. If we number them in reverse of the order in which they were entered (call the first word entered 07; the last, 00), then the situation in the computer will look like this:



Eventually, with 27 reloads, we could fill line 19 with input. If, at that time, another reload is struck, the first four words of input will be lost. Provided this is not allowed to happen, and line 19 is filled with input, the first word of input will be in 19.u7. Four more words of input can be accommodated in line 23, but no reload should be given after they have been entered. This is the limit of one input. No more information can be absorbed by the computer during one input.

Now we come to the method for stopping any normal input. Any normal input is stopped by a "stop" code. The "s" key on the typewriter will supply this code. This brings us to the second type of input from the typewriter: signals which control the computer directly.

The "s" key supplies the computer with a "stop" code, which is a signal capable of controlling the computer directly. The computer is capable of handling only one normal input or output operation at one time. When one is in progress, the input/output system will be in a "not ready" status, which can be determined by inspecting the neon lights on the front panel of the computer. (See page 208. The bottom row of neons contains a group of five lights pertaining to inputs and outputs. One of these is marked "R"; it is the "ready" light.) If the "ready" light is not on, the input/output system is not ready, because an input or an output is in progress. If the computer's input/output system is not ready, the process currently being carried out must be stopped before another is begun. The stop code will do this.

The only way the stop code can be provided to end a typewriter input is via the striking of this key. It may be done at any point during the input, even before any information has been placed in line 23. In that case, of course, line 23 will retain its original contents.

In order for an input or output to be processed by the input/output system, it must be called for. This can be done by command in a program, but this will be discussed later. It can also be done from the keyboard of the typewriter.

We now seem to be on the horns of a dilemma. We have just said that a typewriter input, or, for that matter, any normal input, cannot be processed until it is called for. We then proceeded to say that we would originally call for it through an input from the typewriter.

"ENABLE" ACTIONS

Notice again the difference between the two types of input from the typewriter: one, already discussed, places information from the keyboard into the memory of the computer; the other, which we are now discussing, supplies control signals directly from the keyboard to the computer. The latter type is called for in only one way: through a switch action. It cannot be called for by a program. In the preceding drawing of the typewriter, you will notice a switch mounted on the front of the base of the typewriter, called the "enable" switch. This switch has only two positions: to the left, it is on; in the center position, it is off. When the enable switch is on, the control keys on the keyboard are enabled to send control signals to the computer; when it is off, these keys are not connected to the computer.

There is only one exception to this rule, and we have discussed it; when a type-in of information for the memory of the computer is called for, the stop code to end it can be, and must be supplied by striking the s key. In this case, the enable switch need not be on. In all other cases, including use of the s key to stop any other input or output, the enable switch must be on in order to activate the control keys on the keyboard. Because this is the only way these inputs can be called for, and because they are not really inputs, in the sense that they don't place information in the memory of the computer, they are not usually referred to as inputs. Rather, they are referred to as "enable actions" or "control actions". The custom adopted as a short-hand for specifying one of these actions is to underline the appropriate letter (e.g., $\underline{s}$). We will drop the reference to these as inputs, and adopt the name, "enable actions".

As seen in the diagram on page 130, $\underline{q}$ will call for a typewriter input. We have already mentioned that $\underline{s}$ will stop any normal input or output, and set the input/output system ready.

Two other enable actions should be mentioned here: they are $\underline{c}$ and $\underline{f}$. Much earlier in this book, a question was deliberately left unanswered, with the excuse that it would be covered later. The question was: How do we initially select a command line, and how do we change control from one command line to another during the operation of a program, if that is necessary? The latter part of the question has been answered: we change command lines under program control through use of either the mark and transfer control command or the return command. The former

part of the question will be answered now. We can initially select a command line through the use of the two enable actions, c and f. You will notice in the drawing that c will signal the computer to set the command line equal to the following number, or, if no number follows, to command line 00. This implies that c would be followed by the typing of a number from 0 through 7, corresponding to the desired command line. This is correct. A 0 may follow c or not; there will be no difference in the effect of the signal.

Selecting a command line, however, does not fully establish the address of the first command to be obeyed. There still remains the word-time portion of the address. In the two commands that transfer control to a specified command line, this is accounted for, either in the command itself, or in the timing of the command, combined with the existence of a "mark" in the computer. In the case of enable actions, a word-time must be supplied by another enable action, f. This signals the computer to take the next command from word 00 of the selected command line. Notice, there is no provision to specify any word-time other than 00. If however, this is not specified, and the computer is allowed to start operating, it will take the first command in the selected command line at whatever word-time it received as the N of the last command read, which, in most cases, will lead to an erroneous result, since it is a good bet that, when you choose to change command lines, the desired word-time for the start of the new sequence will be different.

Because of this feature of the f action, it has been the experience of many programmers of the G-15 that it is best, wherever possible, to start a program at word-time 00.

As you can see, in the drawing on page 130, there are many other enable actions which, as yet, remain uncovered. They will be discussed, one by one, as they arise during the further discussion of inputs and outputs.

Discussion of punched tape input requires some knowledge of what punched tape contains. Therefore, we will next bring up punched tape output, followed by punched tape input.

PUNCHED TAPE OUTPUT AND OUTPUT FORMAT

The use of punched tape for output serves two purposes. It preserves information in a form in which it can be retained for later use as an input for the computer; in other words, it acts as an interim storage device. The second purpose is to speed up the output operation of the computer (the other normal output is via the typewriter, and is a good deal slower), and yields an output in a form which can be processed off-line (not involving the computer), on any suitable tape-reading device which can read this type of punched tape and type out the contents, much on the order of some teletypewriters.

When a punched tape output is called for, the output information will be taken from line 19 in a manner prescribed by a format stored in the memory of the computer.

A format is a series of binary codes, each of which calls for a type of output character. The types of characters, their abbreviations, and the related format codes are shown below:

Type of Output Character	Abbreviation	Format Character
Digit	D	000
End (stop)	E	001
Carriage Return	C	010
Period (point)	P	011
Sign	S	100
Reload	R	101
Tab	T	110
Wait (skip one digit)	W	111

The complete format for the punching of tape is contained in four words in memory, 02.00 - 02.03. The desired format characters are placed end-to-end, beginning with T29 of word 03 in line 02, and working backwards, ignoring word-boundaries, towards T1 of word 00. The format may be any desired length within the limit of four words. Since all of line 19 may contain information to be transferred to the tape punch, it is readily apparent that not enough 3-bit format characters can be placed in the available bits in four words to call for every digit and every sign of the output. The reload code in the format will cause all of the preceding format characters to be reinspected, as the processing of line 19 continues. At this point we must investigate the processing of line 19.

Each D in the format will call for the output of bits T29 down through T26 of word 19.u7 (the most significant four bits of the word, and therefore, the most significant hex digit of the word) as a hex digit. Line 19 will then be shifted up four bits, losing the four which have just been inspected, and vacating the four least significant bits (T4 down through T1) of word 19.00. Thus, successive D's in the format will cause a succession of hex digits in the output, and they will also cause a succession of shifts in line 19.

Each S in the format will cause an inspection of bit T1 of word 19.u7, and an output of either a plus, or a minus sign. No shift will occur in line 19. Notice that the sign of a number will have to be called for at the time it is in T1 of 19.u7.

Each W in the format will cause a 4-bit shift of line 19, but there will be no output of the corresponding hex digit; instead, an output which will be treated as a blank character is substituted.

Each T in the format will cause a special tab code to be punched on tape. Line 19 will be shifted one bit.

Each C in the format will cause a special carriage return code to be punched on tape. Line 19 will be shifted one bit.

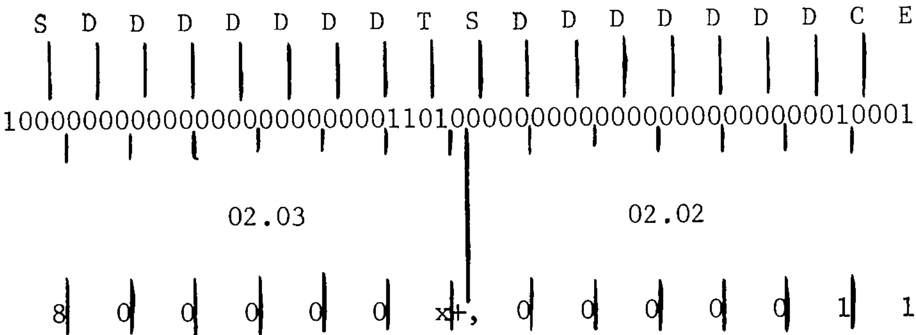
Each P in the format will cause a period to be punched on tape. Line 19 will not be shifted.

The only two remaining format characters are R and E. Each R will cause a special reload code to be punched on tape. There will be no shift of line 19. The inspection of the entire format, beginning at T29 of 02.03 will be repeated. Thus, once an R has been placed in a format, the format is essentially closed in a loop. Any remaining bits in the allotted four words in line 02 will never be inspected, and the output will never end. Use of R in a format requires caution. A way to stop an output under control of such a format by program command is available, and will be discussed later. Striking s on the typewriter keyboard will also stop it.

Use of an E character in a format is the normal method of stopping an output. In addition to punching a stop code on the tape, it supplies a stop code for the output. But the operation of this character of format is very special, and requires closer scrutiny. You have noticed that, as characters (hex numbers and signs) in line 19 are used up, during an output, they are shifted out of the line (processing of a sign does not accomplish this), and bits are vacated at the low end of the line, in word 00. In any shift, the vacated bit-positions are filled with 0's. When line 19 contains nothing but 0's, we want output to cease. Thus, by clearing line 19 and then properly positioning the output data in line 19, prior to the output, we can control the duration of the output. This is made possible through the computer's interpretation of an E character in a format. When the E character is encountered, as the format is inspected, character-by-character, line 19 is searched for at least one non-0 bit. If a 1 is found anywhere in the line, the E character is automatically interpreted as an R, and causes the same sequence of events as is caused by an R character. Eventually line 19 will contain all 0's. When the E character is encountered, the search of line 19 is performed, it is found that the entire line is clear, and the E character is interpreted as calling for a stop code to be generated on the tape, and for the output to be stopped. At this point, the input/output system will be "ready".

Consider, as an example, the case of the program we have already developed, in which two answers will be generated, each a signed single-precision number. We could first clear line 19. The best method for this is an immediate command, allowed to work for one complete drum cycle, with S containing 0's in each of its words, and D = 19. We could, for example, clear the two-word registers (and IP) with the clear command, then use any one of them as S, with C = 0, and D = 19. We would make this an immediate command (I/D = 0), and set T (flag) equal to L_1. During each word-time either the even or the odd half of the specified two-word register will be copied into the specified

word-time of line 19. After execution of this command, every word in line 19 will contain 29 0's. After line 19 has been cleared, we could transfer x_1, generated by our program, into 19.u7, and x_2 into 19.u6. Then we set up the following format in line 02.



After both the data and the format are properly loaded, we initiate the output to the tape punch. Let's examine the results.

The first character of the format is inspected, and found to call for a sign. Therefore, bit T1 of 19.u7 is inspected, and a code for plus or a code for minus is punched on tape.

The second character of the format is inspected, and found to call for a digit. Therefore, the hex digit representing the four bits T29 - T26 in 19.u7 is punched on tape. Call this the first hex digit. Line 19 is shifted up four bits, so that T29 - T26 of 19.u7 receive the bits which previously occupied bits T25 - T22 of the same word; call this the second hex digit. Bits T4 - T1 of 19.u6 receive 0's from the previous word (all of line 19 has been cleared to 0, except words u6 and u7).

The third character of the format is inspected, and found to call for a digit. The same sequence of events occurs again, the third hex digit of the number x_1 arriving in T29 - T26 of 19.u7, while the second hex digit is punched on tape. The shift of line 19 leaves bits T8 - T1 of 19.u6 cleared.

The same thing happens for the fourth, fifth, sixth, seventh, and eighth characters of the format, which all call for digits. After the eighth format character has been processed, and the character it calls for has been punched, line 19 is shifted again. The result, up to and including this shift will be: the sign and seven hex digits representing x_1 have all been punched on tape, sign first, followed by the most significant digit down through the least significant digit. Line 19 has been shifted so that 28 bits from 19.u7 have been lost, T29 of 19.u7 contains what was originally the sign-bit of that same word, bits T28 down through T1 of 19.u7 contain the magnitude bits of x_2 (originally in 19.u6), and T29 of 19.u6 contains the bit that was originally the sign-bit of the same word. Bits T28 through T1 of 19.u6, like the rest of the line, contain 0's.

The ninth character of the format is inspected, and found to call for a tab. A tab code is punched on tape, and line 19 is shifted by one bit. 19.u7 now contains all of x_2, while the rest of the line is cleared.

The tenth character of the format is inspected, and found to call for a sign. Bit T1 of 19.u7 is inspected, and the proper sign code is punched on tape.

The eleventh through the seventeenth characters of the format are inspected, and, since they also call for a series of digits, are processed in the same fashion as were the second through the eighth format characters. After the seventeenth character of the format has been processed, its corresponding character of output has been punched, and the corresponding shift of line 19 has been carried out, the result will be: the sign and seven hex digits representing x_2 have all been punched on tape, sign first, followed by the most significant digit down through the least significant digit. Line 19 contains only one of the original data bits, in T29 of 19.u7, which was originally the sign-bit of x2 in 19.u6. The rest of line 19 is clear.

The eighteenth character of the format is inspected, and found to call for a carriage return. A carriage return code is punched on tape, and line 19 is shifted one more bit. Now the entire line is cleared to 0.

The nineteenth character of the format is inspected, and found to call for a stop code. Line 19 is searched, and, since it is found to contain nothing but 0's, this character is treated as an E character. The stop code called for is punched on tape. A stop code is also generated which stops the output and sets the input/output system "ready" for another input or output.

At this point there is a tape hanging out of the computer (top, front). There will be no doubt in your mind where the punch is after you have once activated it. A toggle switch on the face of the computer allows you to feed blank tape through the punch until you can tear off the piece of tape containing the entire contents of the output you called for. How will you know when you reach the end of valuable information? You know which character was punched last; it was the stop code. Therefore, if you can recognize a stop code, you can tell the end of the information on the tape. The following table shows the codes punched on tape corresponding to each character which can be punched. In the punched codes, as shown, a 1 represents punch; a 0, no punch. It will be seen that there are five "channels" on the tape.

The length of tape so generated is referred to as a "block" of tape. Every block is ended by a stop code. Its length will be determined by the lowest-ordered word in line 19 containing non-0 data when the output is called for which generates the block of tape.

Output Character	Code Punched on Tape
0	10000
1	10001
2	10010
3	10011
4	10100
5	10101
6	10110
7	10111
8	11000
9	11001
u	11010
v	11011
w	11100
x	11101
y	11110
z	11111
Space	00000
Minus	00001
CR	00010
Tab	00011
Reload	00101
Period	00110
Stop	00100
Wait	00111

The particular block of tape generated in our example will contain the equivalent of two words, since, by the end of the first inspection of the format, all of line 19 will contain 0's. The order of punched codes on the tape would be:

1st	sign
2nd	digit
.	.
.	.
.	.
8th	digit
9th	tab
10th	sign
11th	digit
.	.
.	.
.	.
17th	digit
18th	CR
19th	stop

Notice that the tape now contains the same characters that you might choose to supply, were you to "gate" type-in, and enter the two numbers x_1 and x_2 as typewriter inputs.

A block of punched tape can be read (by a photo-reader mounted on the front of the computer), upon command to the computer; this is the other normal input. Were this block of tape to be mounted on the drive mechanism of the photo-reader, and a p action taken (see drawing on page 130); it would be read as a computer input. The rules governing the entry of its information are the same as those governing a type-in. The first character entered, being a sign, would not immediately enter memory, but would enter the sign flip-flop. Then seven digits would be entered into word 00 of line 23. The next character of input, the tab, would shift line 23 one bit, placing the seven previously entered digits in bits T29 - T2 of 23.00. The sign of the number would be dumped into T1 of the same word from the sign flip-flop, and that flip-flop would be cleared. The next eight characters would be entered in the same way, the first complete 29-bit number (x_1), being shifted into bits T28 - T1 of 23.01 and T29 of 23.00, while bits T28 - T1 of 23.00 receive seven new digits. The following character, a carriage return, will have the same effect as the tab, and the result will be x_1 in 23.01 and x_2 in 23.00. The next character, the stop code, automatically reloads, shifts line 19 by four complete words, and places words 23.00 - 03 in 19.00 - 03 *. These words also remain in line 23. Then it terminates the input, and sets the input/output system "ready" for another operation. This, then, is the pattern for the entry of a block of tape into the memory of the G-15. All of line 19 could be loaded in this manner, and the last four words to be entered into line 19 would remain in line 23. This fact will be important to us a little later. Punched tape input is preferable to typewriter input in one respect, at least: speed.

TYPEWRITER OUTPUT

The format which controls the output from line 19 to punched tape also controls the output from line 19 to the typewriter, in exactly the same way, except in the case of typing, keys, tabs, and the carriage return on the typewriter are affected, rather than punch-heads. You can actually see the keys move as the contents of line 19 are typed out.

The contents of AR may also be typed out, under control of a format made up in exactly the same way as the format for line 19. The AR output format must be placed in line 03, words 02 - 03, prior to calling for the type-out of AR. Again, the inspection of the format will begin with T29 of word 03 and move toward the low-order end of the line. During a type-out of its contents, AR will be shifted in the same manner as line 19.**

---

* Note: This automatic reload feature of the stop code is not true when the s key is used to stop a type-in.

** Note: Because four words' worth of format should, in all cases, be sufficient to "cover" one word of output, it should be unnecessary for an "end" code in the format to be automatically changed to a "reload" code. For this reason, the "end" code in an AR format will never be changed to cause a reload.

The type-out of the contents of AR brings us to another topic. You will notice, from inspection of the drawing on page 130, a will cause the contents of AR to be typed out. You will also notice that this is the only output of the three mentioned (and these are all of the normal outputs) which can be called for through an enable action.

Consider for a moment the function of the whole class of enable actions. It is to enable the operator to give the computer commands directly, not in the normal binary command form. When would this be useful? Primarily, when the computer does not have loaded in its memory the desired program. These actions enable the operator to get a program into the memory of the computer, either one command at a time, through type-in, or a block of tape at a time, through the reading of tape. Once the program has been loaded, control can be given to it, within the computer, and it will operate the machine. For instance, when the computer is first turned on, perhaps in the morning, there will be no information in its memory. Turning it off the night before cleared memory. The enable actions enable an operator to start the computer. At such a time it is hard to conceive of the need for an output. Quite the contrary, when outputs are required, a program will have generated them, and that same program can call for them with commands, none of which we have yet defined.

DEBUGGING

The reason for providing for this one output, the type-out of the contents of AR, through enable action, is to assist the programmer in "debugging" his program. It has been painfully established by almost all the programmers who have preceded you, no matter what computer or programming system has been employed, that very few programs work successfully in all respects as originally written. There are usually a few flaws, perhaps stemming from carelessness, or from lack of knowledge, or from a change in requirements. Finding these flaws by inspection of the program is sometimes almost impossible, especially in very long and complicated programs. In such cases, the programmer will usually resort to making up a "test case", for which he will calculate the correct answer(s). He will then enter his program into the computer, and allow it to operate with the inputs of the test case. He will cause the program to halt temporarily at various strategic points, and inspect the partial results he has achieved. In this way he can eventually isolate the steps in the program which are causing the trouble. How does he inspect these results? He stops the program at points where AR contains vital information, and inspects AR. Thus, the provision for type-out from AR.

BREAK-POINT

Now we come to the only remaining question which was intentionally left open earlier, and answer it. In the machine form of a command, bit T21 was left undefined. This bit in a command is called the BP bit. BP stands for Break-Point. If a command contains a 1 in this bit, the

computer will halt upon execution of the command, provided a switch action has been taken previously. (Do not break-point a return command.)

On the front of the typewriter base is a switch called the "compute" switch. Other than performing an enable action, the computer will not operate until this switch is on. The center position for this switch is the off position. The switch is on when thrown either to the left or to the right. If thrown to GO, it will cause the computer to continue operating until either a halt command is reached or the compute switch is moved back to the off position. If thrown to BP, it will cause the computer to operate until a halt command is reached, the switch is thrown back to the off position, or a command with BP = 1 (called a "break-pointed" command) is reached. If you want the computer to be sensitive to these inserted break-points, then, you must move the compute switch to BP rather than GO to operate your program. A rule is that the enable switch and the compute switch should never be on simultaneously. If you have stopped at a break-point, turn compute off before turning enable on.

The entire process of debugging encompasses far too many techniques and far too much effort to be thoroughly discussed here, but one of the important facets of it is this periodic inspection of AR. Combined with the ability to type out the contents of AR, are certain other enable actions. For instance, t, will place in AR the address of the next command the computer will obey if the compute switch is turned back on. This enable action, followed by a, should help you determine whether or not your program is following the predicted path.

Notice that the contents of AR will shift as it is typed out. This means that, following the type-out, AR will no longer contain what it did. If, after inspection of AR, you wish to return to your program, it is quite conceivable that this destruction of AR's contents will cause errors in the rest of the program. Prior to a, you should take the m action, which will save the contents of AR and mark the location of the next command. Following a, r will restore this information.

"SINGLE CYCLE"

Another enable action which is of help in the debugging process is i, which causes only one step to be executed. You could operate a whole program through a long enough series of i's. If you will look at the drawing on page 208, you will see that among the neons on the front of the computer, there is a set for S and another for D. These lights will contain the S and D number, respectively, of the command being executed. By following your program, as it is written on paper, and these lights through a series of i's, you can often spot errors in the path of your program. Do not single cycle return commands.

INPUT/OUTPUT COMMANDS

It is necessary for most programs to call for their own inputs and outputs, by command. The commands which will do this are:

"Gate" type-in: D = 31, S = 12, C = 0.

Read punched tape: D = 31, S = 15, C = 0.

Type AR: D = 31, S = 08, C = 0.

Type line 19: D = 31, S = 09, C = 0.

Punch line 19 on tape: D = 31, S = 10, C = 0.

These inputs and outputs will behave in the manner described.

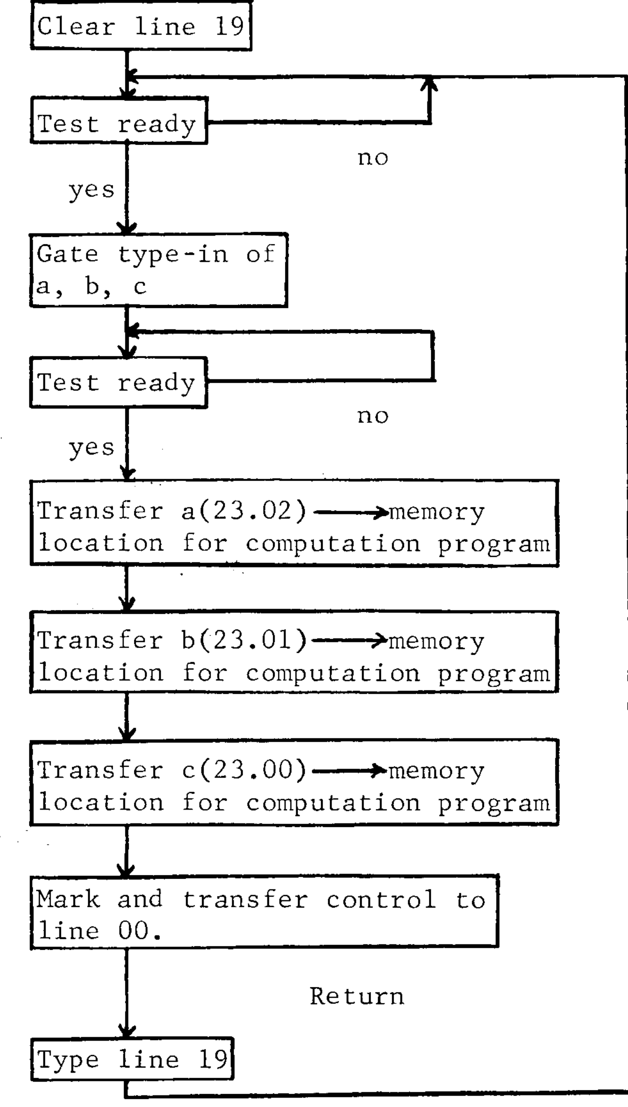
The G-15, although it can handle only one normal input at a time, has no interlock to prevent the initiation of another before the input/output system is "ready". In such a case, the input or output called for will be a logical sum of the two special (S) codes of the conflicting commands. In short, the results of allowing your program to make this mistake are disastrous. For example, if, during a type line 19 operation (S = 09), you executed a "gate" type-in command (S = 12), you would suddenly find you were, as far as the computer is concerned, requesting an input/output operation with S = 13, their logical sum. It just so happens that this special code for an input/output operation calls for "read magnetic tape". You might not even have a magnetic tape drive at your installation, but nevertheless, the computer would attempt to read from one. Of course it would never receive a stop code, and thus the attempted input would never terminate, to say nothing of the fact that neither your desired input nor your desired output will be accomplished.

$$\begin{aligned} 09_{(10)} &= 1001 \\ 12_{(10)} &= \underline{1100} \\ & 1101 = 13_{(10)} \end{aligned}$$

In order to prevent such distressing occurrences, you are equipped with a test command which can be incorporated in your program. It was mentioned earlier, but not defined: "test for 'ready'". If the input/output system _is_ "ready", the next command will be taken from N + 1; if _not_, the next command will be taken from N. The most common use of this test is to set N = the location of the test itself, so that the test will be repeated until the input/output system is ready for a new operation, at which time the test will be met, and the program will proceed at N + 1. At this point another input or output might be called for. Another use of the ready test so programmed is to prevent the program from trying to use a set of inputs until they have been completely received. In order to achieve the most benefit from the ready test you will usually want to make it immediate, and let it be executed as often as possible. If N is set equal to L, you want the last word-time of execution to be L - 1, so that no delay will be involved waiting to take the next command from N. If the last word-time of execution is to be L - 1, the flag (T) in the test command must also equal L. The ready command then, programmed in this recommended way, will contain D = 31, S = 28, C = 0, T = L, N = L, and the command will be immediate. This command should precede all normal input or output commands.

There is a very important implication in what has just been said. The G-15 continues to operate your program during any normal input or output. In many computers, input and output time is "dead" time as far as computation goes, but this is not so in the G-15. While you are typing out one answer, for example, you can be computing the next. As a matter-of-fact, experienced programmers write programs in which almost none of the input/output time required is left unused as far as computation goes.

At this point we can develop another program, not very long, which will handle the inputs and outputs for our main computation program. Let's assume we want to type the answers we derive, x_1 and x_2, out of line 19. Since we will have to use a line 19 format (we already developed it; see page 137), we might just as well use command line 02 for this program. Thus, the program and its output format will occupy the same line.



In addition, we must include a command at the end of the program in line 00, a mark and transfer control command, which will transfer control back to this program at the correct point, for the type-out of the answers.

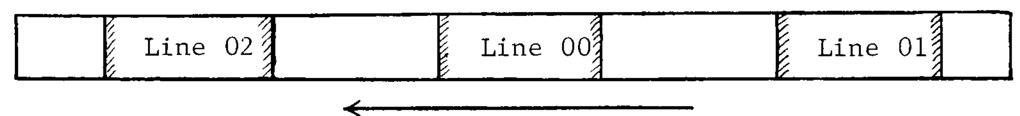
In the computation program, when we generate x_1, we will store it in 19.u7. Similarly, we will store x_2 in 19.u6, so that these numbers will be properly positioned, ready for output, prior to returning to the output portion of the program above.

Notice that the first thing this input/output program does is to clear line 19, so that it will always be clear when we attempt to type out our answers. If this were not done, and line 19 contained some garbage, we would type out this garbage as well as our valid answers during the first output.

The input/output program continues in a "loop". After completion of one run of the whole program, it immediately returns to an earlier command and eventually calls for type-in of a new set of a, b, and c. (N of last command = L of test.)

In this way, our program will continue forever, always calling for a new set if inputs after typing out the last set of answers. Of course we can stop this at any time we want by turning off the compute switch on the typewriter and walking away from the computer. More will be said later concerning loops and their uses.

Now we know that we can enter our program a line at a time, into the computer and punch out that line on tape. Suppose we punch line 01 (the square root subroutine) on tape. Then, on the same tape, we follow it with line 00, the main computation program. Then, still on the same tape, we follow line 00 with line 02. Finally when we run the tape out of the punch, we will have a long piece of punched tape containing three blocks, as shown in the drawing below, where the arrow indicates the direction in which the tape would be read.



BLANK "LEADER"

Notice that a blank space is located before the first block, between blocks, and after the last block. When a tape is mounted on the photo-tape-reader, it must have some leader which can be fed through the mechanism and onto the winding-spool, similar to the loading of a movie film. After an input, the drive mechanism coasts to a stop, and we don't want valuable information from the next block to slip past the photo-reader during this coast-time. The blank tape at the end will result simply from manually feeding the tape out of the machine, prior to tearing off the desired length, after it has been punched. Approximately 9" to 1' of blank tape should be left before the first block

as "leader", for initial winding purposes. Approximately 6" to 9" of blank tape should be left between blocks, to allow the tape drive mechanism to coast to a stop without allowing any information in the next block to slip past the photo-reader. This means that, when an input from punched tape is called for, an indeterminate length of blank tape will be "read" prior to the reading of any valuable information. The "reading" of blank tape will cause no input to the computer.

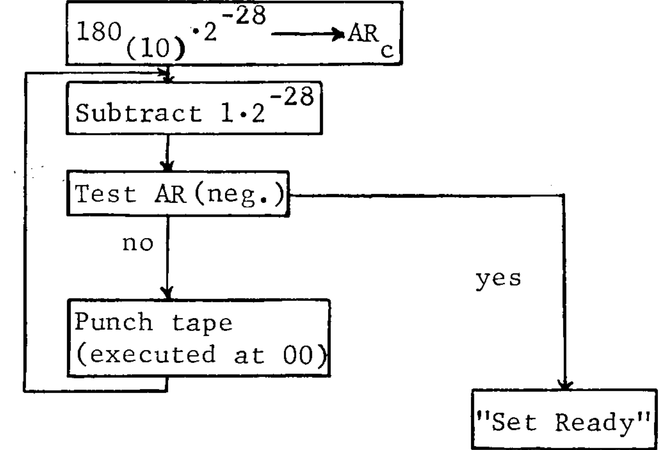
There is an acceptable method for generating blank leader automatically, under program control. This method is based on the fact that the "punch line 19" command not only initiates the punch operation, but also reloads the line 19 format (in line 02) into a four-word inspection buffer.

If the punch command is given before the end of the line 19 format has been reached during its inspection, the format will be automatically reloaded, and its inspection will begin anew, from the first format character. In this way, the end code in the format might never be reached, and the output would continue indefinately. As a matter of fact, if the punch command is repeated often enough, only the first character of the output format will ever be inspected.

Therefore, as tape is punched, a series of characters will be transmitted to tape; it will be a series of whatever is called for by the first format character. Of all data transmitted to the tape punch, the only one which causes no punch is a + sign. We therefore will cause a series of + signs to be transmitted, thus causing blank tape to be fed out of the punch. The first format character in the line 19 format (contained in line 02) must be a sign character, and the sign-bit of 19.u7 must be 0 (= +). In this manner, for as long as punching continues we will get only blank tape.

Ten strokes of the punch will yield one inch of tape. Two drum cycles are necessary for each punch stroke. Therefore, 20 drum cycles are necessary to generate one inch of blank tape. The generation of nine inches of blank tape would require 180 drum cycles.

In order to achieve this, the punch tape command should be executed at word-time 00 of every drum cycle for 180 drum cycles.

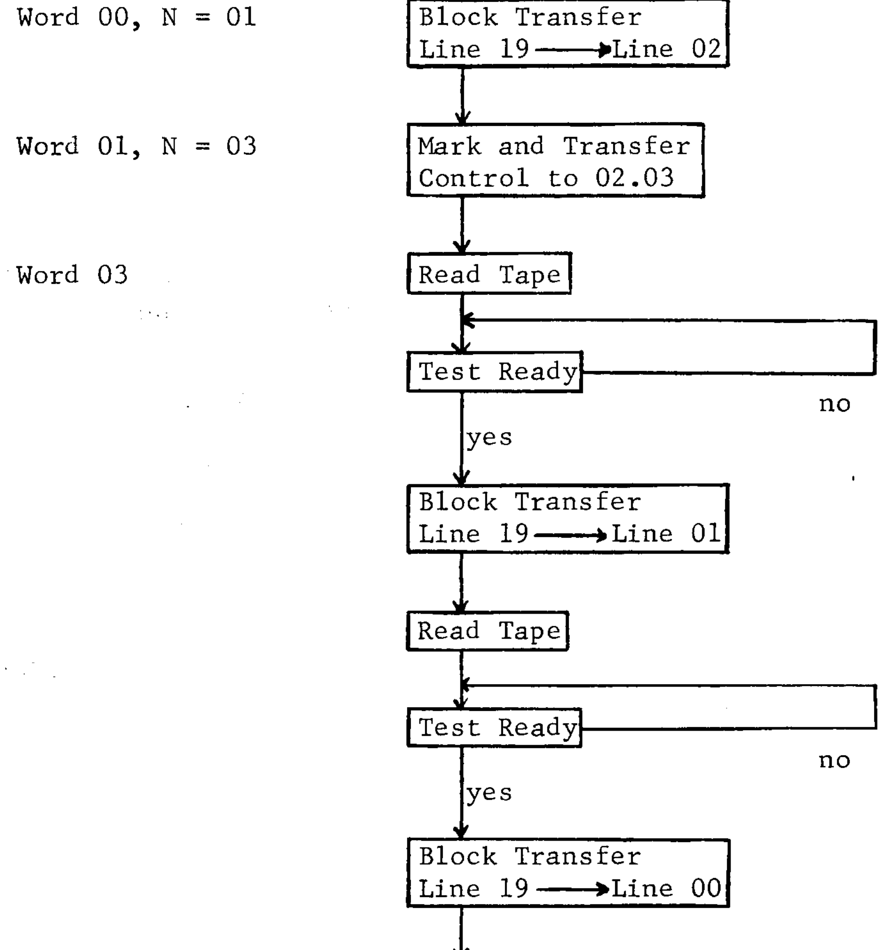


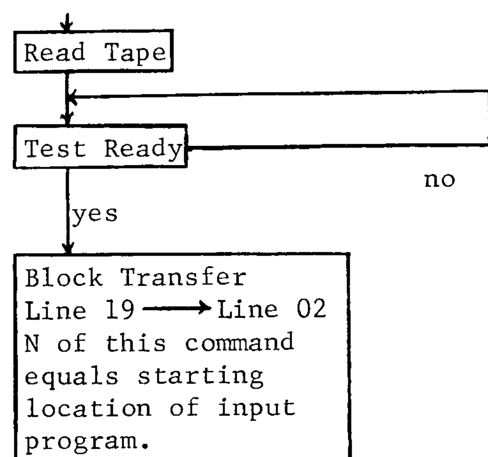
In the program flow-diagrammed on the preceding page, N in the punch tape command will equal the word-time in which the subtract command is located. The program will continue looping and counting the elapsed drum cycles by subtracting 1 from 180 for each drum cycle. Eventually AR will contain +0, and when 1 is subtracted from it, it will contain -1, the answer to the test will be "yes", and the program will exit from the loop. At this point the proper length of blank tape will have been punched.

You have noticed the use of a "set ready" command in the flow diagram. This is a special command with D = 31, S = 00, and C = 0. When this command is executed, whatever input or output is in progress will be automatically and arbitrarily stopped. No "stop" code will be punched on the tape. This command must be used with caution; it may shift the contents of line 19. Do not place the valid outgoing information in line 19 until after the set ready command has been executed.

LOADER PROGRAM

Suppose now, we entered the following program, and punched it on tape, and then we spliced this block of punched tape onto the other, preceding line 01.





This is called a "loader" program. Its value is derived from the fact that, if p is used to cause the reading of a block of tape, and if, after the input has ceased, the enable switch is turned off and the compute switch is turned to GO, the computer will take its next command from 23.00, which, as you know, will be the same as 19.00. Now we can trace the operation of this program.

The loader will be in line 19, and its first four words will also be in line 23, following the input caused by p.

When the compute switch is thrown to GO, the next command is taken from 23.00.

This command, word 00 of the loader program, causes the transfer of all of line 19 into line 02. The next command to be executed, still in the same command line, (command line 07, which is line 23) is at word 01.

This command is a mark and transfer control command, and control is transferred to line 02. The N number of this command selects the word-time of the first command to be read from line 02: it is word-time 03.

The program in line 02, beginning with word 03, will now be executed. But this is the loader program, itself. Word 03 calls for an input from punched tape. A preceding "ready" test is unnecessary, because it can be firmly predicted that no input or output is already in progress. The block of tape read into line 19 will be the next block on the program tape, following the loader. This is the block containing the square root subroutine, destined for line 01. Therefore, after this input is completed, line 19 is transferred, word-for-word, into line 01. Then the next block of tape is read, still under control of the loader program in line 02. This block of tape contains the main computation program, and therefore, upon completion of the input, line 19 is transferred into line 00. The remaining block of tape is then read into line 19. This block contains the input/output program designed to accompany the computation program, and is destined for

line 02. But line 02 is already in use; it contains the loader program. After the input is finished, the loader program will execute one more command, which will be its last. It calls for all of line 19 to be transferred into line 02. The loader program thus destroys itself, and line 02 contains the input/output program we desire. The command line is still line 02; nothing has been done to change that. The next command, as always, will be taken from the same command line, at a word-time specified as N of the previous command. Therefore, if, as indicated in the flow diagram, the N of the last command of the loader program equals the location of the initial command of the input/output program, during the next read-command time the input/output program will start its normal execution, just as we originally planned it.

A loader of this type is called, rightly enough, a "self-destroying" loader. Its purpose is to set up the memory of the computer for the operation of a given program completely, and yet occupy no part of memory after the set-up has been completed and it is no longer needed.

This type of procedure, involving the use of a loader program, is sometimes given the picturesque name of "bootstrap", for obvious reasons. Once such a tape has been mounted on the drive mechanism of the photoreader, the only actions necessary at the typewriter are:

1. with the compute switch off, put the enable switch on;

2. strike p;

3. after one block of tape has been read into the computer, make sure the enable switch is off, and move the compute switch to GO.

From that point, in our example, the rest of the program will pick itself up by its own bootstraps, enter the computer's memory at the proper locations, and proceed to operate until it reaches the point where it gates the type-in of the first set of a, b, and c. At this point the S and D neons on the face of the computer will not be flickering rapidly as step after step is executed, because always the same step is being executed. It is the "ready" test. The neons will remain steady, indicating D = 31, S = 28, and an input/output code of 12 (the code for a "gate type-in").

You will enter the numbers in the following order, as determined by the way we originally formulated the input program: a (tab) b (tab) c (tab) s. Each number will consist of seven hex digits and a sign. The s will set the input/output system "ready", the test on which the program was "hung up" will be met, and the program will proceed. Provided the numbers entered don't generate erroneous results, the computation program will place the two answers in 19.u7 and 19.u6 and transfer control back to the input/output program, which will type them out in the following order: x_1 (tab) x_2 (carriage return). It will then hang up again on the "ready" test, awaiting a new set of inputs. It will keep on performing this cycle until we simply don't supply any more inputs.

Notice that we will have to convert decimal numbers for a, b, and c, to hex numbers, prior to the input, and that, when these hex numbers are typed in, they will be scaled, to our knowledge, 2^{-21}. The decimal numbers should therefore be converted to binary, rather than hex, 21 bits being allowed for the expression of the integral value, and 7 bits for the fractional value. From the resultant series of bits, a corresponding hex number can very easily be made up, and it will be this number that will be typed in.

The output will also be in hex, representing a binary value scaled 2^{-21}. This binary value must be converted to its decimal equivalent, which can be done quite easily, by inspection. A table of corresponding powers of 10 and 2, as well as corresponding powers of 10 and 16, is located in the back of this book (page 207).

Only two tasks remain to be performed before we can punch a complete program tape of the type described above.

One is to choose word-locations in the appropriate command lines for all of the necessary commands. Because of the nature of the memory in the G-15, as has been pointed out previously, timing becomes a consideration in the writing of a program. We wish to minimize the amount of wait-time preceding both the reading of commands and their execution, in order to enable the program, as a whole, to operate in the shortest amount of time possible. We therefore will have to choose wisely the locations into which we place the commands, the times at which we will execute them, the words in which we store constants used by our program, and its inputs.

The other task is to code, in binary, each command and constant that our program needs. We know the binary make-up of a command, so this will not be difficult. From the binary number, we will have to get a hex number which can be typed into the computer. When this has been done for all the words in a line, that line's contents will be in line 19. We can then punch a block of tape. We must repeat this for each block of tape necessary. Although this is not a difficult task, it is time-consuming.

PROGRAM PREPARATION ROUTINE (PPR)

Fortunately, Bendix Computer Division has developed a program, called the Program Preparation Routine (PPR), which will do this. As inputs, it needs commands composed of decimal numbers for T, N, C, S, and D, in the form shown below.

T N C S D

The decimal number for T will contain two digits, ranging from 00 through u7. Similarly, that for N will also contain two digits, within the same range. The decimal number for C will contain one digit, ranging from 0 through 7. The decimal number for S will contain two digits, ranging from 00 through 31. Similarly, for D, the decimal number will contain two digits, within the same range.

PPR needs to be told the location for the command, as well as being given the "decimal form" of the command. From these two facts, it will cause the proper binary command to be entered into the specified word-time of the line it uses for storage. This is line 18. Finally, line 18 will contain all of the commands we want to appear in, say, line 00, at the appropriate word-times. We can then give PPR a command to punch a block of tape with this line. It will transfer line 18 to line 19, and then punch line 19 on tape. Now we can give PPR another command to clear line 18, and then proceed to set up a new line of our program in the same manner. Eventually, we will generate the entire program tape we desire.

When reference is made to giving PPR a command to do something, the question of how this is done arises. PPR gates type-in after completion of every operation. At that time, the operator types in what we call a "pseudo-command" which can be recognized by PPR, telling it what is desired next. Some of these are:

"accept a decimal command input, and store it in word __";

"punch out the program now stored in line 18";

"clear line 18".

There are many more such pseudo-commands for PPR, and they, along with PPR's functions and capabilities, are discussed elsewhere in this book.

At the present time, assume the availability of such pseudo-commands, and we will proceed to see how the commands of a program, in particular, the program we have developed, are coded, in decimal form, prior to being supplied to PPR.

The decimal form has already been shown, but some additions must be made to it:

P T N C S D BP

P is a "Prefix". PPR does not know whether a command, as coded in decimal form, is intended to be an immediate or a deferred command. It will assume that all commands with $D \neq 31$ will be deferred, and all commands with $D = 31$ will be immediate. If this assumption is correct as pertaining to a command, the P is left blank. If it is incorrect, in that you wish to make a command with $D \neq 31$ immediate, you must supply a P of "u". If it is incorrect in that you wish to make a command with $D = 31$ deferred, you must supply a P of "w".

BP stands for breakpoint. If you wish a command to be breakpointed, you must supply a minus sign at this point. If you do not wish the command to be breakpointed, you supply no sign, which is tantamount, as we have seen, to supplying a plus sign.

Usually the commands of a program are written on standard, printed forms, called "coding sheets". The following pages contain our program, its loader, and line 02, coded on such sheets. Notice that the location of each command and constant needed by our program is also specified, along with the command or constant, itself. This location will be supplied to PPR, but not as an integral part of the coded command. You will see that most of the commands are explained somewhat in a "NOTES" column on the sheet, just as a matter of convenience.

Pay close attention to the timing numbers, T and N in each command, and how they have been chosen to reduce wait time during the operation of the program.

Following the coding sheets, there will be some discussion of the timing numbers chosen for individual commands.

You will notice that no coding for line 01 is included. This is the square root subroutine. It has already been written, and we will use it in its present form. We will simply reproduce a block of tape containing that subroutine, and include this reproduction in our own program tape.

Bendix Computer

Los Angeles 45, California

Page 1 of 6

G-15 D

Prepared by ____________

Date: ____________

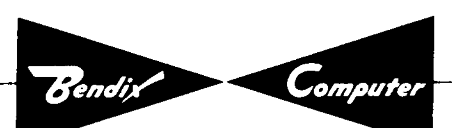
PROGRAM PROBLEM: Loader

Line 02

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
32 33 34 35
36 37 38 39
40 41 42 43
44 45 46 47
48 49 50 51
52 53 54 55
56 57 58 59
60 61 62 63
64 65 66 67
68 69 70 71
72 73 74 75
76 77 78 79
80 81 82 83
84 85 86 87
88 89 90 91
92 93 94 95
96 97 98 99
u0 u1 u2 u3
u4 u5 u6

L	P	T or L_k	N	C	S	D	BP	NOTES
00	u	01	01	0	19	02		line 19 ⟶ line 02
01		03	03	2	21	31		Mark, Transfer ⟶ 02.03
03		05	05	0	15	31		Read Tape
05		05	05	0	28	31		Test Ready
06	u	07	07	0	19	01		line 19 ⟶ line 01
07		09	09	0	15	31		Read Tape
09		09	09	0	28	31		Test Ready
10	u	11	11	0	19	00		line 19 ⟶ line 00
11		13	13	0	15	31		Read Tape
13		13	13	0	28	31		Test Ready
14	u	15	00	0	19	02		line 19 ⟶ line 02
								(next command 02.00)

FORM BCD 108-5.0



Los Angeles 45, California

Page 2 of 6

Date: ________

Line 00

G-15 D

Prepared by ________

PROGRAM PROBLEM: Computation

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
00		03	03	0	23	31		Clear 2-wd. Registers
03		05	06	0	21	25		a = (21.01) $\longrightarrow$ ID_1
06		07	09	0	00	24		2 = (00.07) $\longrightarrow$ MQ_1
09		56	66	0	24	31		Multiply
66		68	71	4	26	24		$PN_{0,1} \longrightarrow MQ_{0,1}$
71		04	76	0	26	31		Shift MQ left 2 bits
76		77	78	0	24	20		2a = (MQ_1) $\longrightarrow$ 21.01
78		81	81	0	23	31		Clear 2-wd. Registers
81		85	86	0	20	25		2a = (20.01) $\longrightarrow$ ID_1
86		07	11	0	00	24		2 = (00.07) $\longrightarrow$ MQ_1
11		56	68	0	24	31		Multiply
68		70	73	4	26	24		$PN_{0,1} \longrightarrow MQ_{0,1}$
73		04	79	0	26	31		Shift MQ left 2 bits
79		81	82	0	24	28		$MQ_1 \longrightarrow$ AR (4a)
82		85	85	0	23	31		Clear 2-wd. Registers
85		87	88	0	28	25		4a = (AR) $\longrightarrow$ ID_1
88		92	91	0	23	24		c = (23.00) $\longrightarrow$ MQ_1
91		56	40	0	24	31		Multiply
40		42	44	4	26	20		4ac = ($PN_{0,1}$) $\longrightarrow$ 20.02,03
44		47	47	0	23	31		Clear 2-wd. Registers
47		49	50	0	22	25		b = (22.01) $\longrightarrow$ ID_1
50		53	55	0	22	24		b = (22.01) $\longrightarrow$ MQ_1
55		56	05	0	24	31		Multiply
05		07	04	0	29	31		Test Overflow
04		06	08	7	20	30		4ac = (20.02,03) $\xrightarrow{-}$ PN+
08		10	12	0	29	31		Test Overflow

FORM BCD 108-5.0

Los Angeles 45, California

Page 3 of 6

G-15D

Prepared by ______

Date: ______

PROGRAM PROBLEM: Computation

Line 00

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
12		14	14	1	26	28		$PN_0 \xrightarrow{+} ARc$
13		15	00	0	16	31		Halt
14		16	16	0	22	31		Test for sign of AR (neg.)
16		18	19	0	00	28		Return Command=(00.18) ⟶ AR
17		19	00	0	16	31		Halt
18	[	u0	99	0	20	31	]	Return Command
19		21	22	0	20	03		2a = (20.01) ⟶ 03.21
22		25	26	0	22	03		b = (22.01) ⟶ 03.25
26	w	20	94	1	21	31		Mark, Transfer ⟶ 01.94
20		21	23	0	03	20		2a = (03.21) ⟶ 20.01
23		25	61	0	03	22		b = (03.25) ⟶ 22.01
61		63	60	0	29	31		Test Overflow
60		00	27	0	00	00		Go to 27
27		29	30	3	22	28		b = (22.01) $\xrightarrow{-}$ ARc
30		31	32	0	20	29		$\sqrt{\ }$ = (20.03) ⟶ AR+
32		34	34	0	29	31		Test Overflow
34		35	36	1	28	28		AR $\xrightarrow{+}$ ARc
35		37	00	0	16	31		Halt
36		39	39	0	23	31		Clear 2-wd. Registers
39		41	43	0	28	25		AR ⟶ ID_1
43		42	87	0	26	31		Shift ID right 21 bits
87		89	90	0	25	23		N = (ID_1) ⟶ 23.01
90		93	94	2	23	28		$\lvert N \rvert = \lvert (23.01) \rvert$ ⟶ ARc
94		95	96	0	28	23		$\lvert N \rvert$ = (AR) ⟶ 23.03
96		97	98	2	20	28		$\lvert D \rvert = \lvert (20.01) \rvert$ ⟶ ARc
98		u1	u2	0	28	23		$\lvert D \rvert$ = (AR) ⟶ 23.01

FORM BCD 108-5.0

Bendix Computer

Los Angeles 45, California

Page 4 of 6

G-15D

Prepared by ____

Date: ____

PROGRAM PROBLEM: Computation

Line 00

0	1	2	3	L	P	T or L_k	N	C	S	D	BP	NOTES
4	5	6	7	u2		u3	u4	0	23	28		N = (23.03) ⟶ ARc
8	9	10	11	u4		u5	u6	3	23	29		N = (23.01) $\xrightarrow{-}$ AR+
12	13	14	15	u6		00	01	0	22	31		Test for sign of AR (neg.)
16	17	18	19	01		03	00	0	16	31		Halt
20	21	22	23	02		04	10	4	25	21		N = ($ID_{0,1}$) ⟶ 21.00,01
24	25	26	27	10		13	15	0	23	31		Clear 2-wd. Registers
28	29	30	31	15		17	21	0	20	25		2a = (20.01) ⟶ ID_1
32	33	34	35	21		24	29	4	21	26		N = (21.00,01) ⟶ $PN_{0,1}$
36	37	38	39	29		57	89	1	25	31		Divide
40	41	42	43	89		30	u5	0	24	28		MQ_0 ⟶ ARc
44	45	46	47	u5		u7	24	0	28	19		AR ⟶ 19.u7
48	49	50	51	24		25	28	3	22	28		0 = (22.01) $\xrightarrow{-}$ ARc
52	53	54	55	28		31	33	3	20	29		$\sqrt{\ }$ = (20.03) $\xrightarrow{-}$ AR+
56	57	58	59	33		35	37	0	29	31		Test Overflow
60	61	62	63	37		38	41	1	28	28		AR $\xrightarrow{+}$ AR
64	65	66	67	38		40	00	0	16	31		Halt
68	69	70	71	41		44	45	0	23	31		Clear 2-wd. Registers
72	73	74	75	45		47	49	0	28	25		AR ⟶ ID_1
76	77	78	79	49		42	92	0	26	31		Shift ID right 21 bits
80	81	82	83	92		93	95	0	25	28		ID_1 ⟶ AR
84	85	86	87	95		96	97	2	28	28		|N| = |(AR)| ⟶ ARc
88	89	90	91	97		u1	u3	3	23	29		|D| = (23.01) $\xrightarrow{-}$ AR+
92	93	94	95	u3		u5	51	0	22	31		Test for sign of AR (neg.)
96	97	98	99	51		53	00	0	16	31		Halt
u0	u1	u2	u3	52		54	56	4	25	21		N = ($ID_{0,1}$) ⟶ 21.02,03
u4	u5	u6		56		59	59	0	23	31		Clear 2-wd. Registers

FORM BCD 108-5.0

Bendix Computer Los Angeles 45, California

Page 5 of 6

G-15D Prepared by ______ Date: ______

PROGRAM PROBLEM: Computation Line 00

L	P	T or L_k	N	C	S	D	BP	NOTES
59		61	62	0	20	25		2a = (20.01) ⟶ ID_1
62		66	69	4	21	26		N = (21.02,03) ⟶ $PN_{0,1}$
69		57	25	1	25	31		Divide
25		26	31	0	24	28		MQ_0 ⟶ ARc
31		u6	48	0	28	19		AR ⟶ 19.u6
48		50	50	2	21	31		Mark, Transfer ⟶ 02.50

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 u0 u1 u2 u3 u4 u5 u6

FORM BCD 108-5.0

Bendix Computer Los Angeles 45, California

Page 6 of 6

G-15D Prepared by ______ Date: ______

PROGRAM PROBLEM: Input/Output Line 02

L	P	T or L_k	N	C	S	D	BP	NOTES
02								0000011
03								800000x
00		01	04	2	28	28		Clear AR
04		05	06	3	28	29		
06	u	07	07	0	28	19		Clear line 19
07		07	07	0	28	31		Test Ready
08		10	10	0	12	31		Gate Type-in
10		10	10	0	28	31		Test Ready
11		14	15	0	23	28		a = (23.02) → ARc
15		17	18	0	28	21		a = (AR) → 21.01
18		21	22	0	23	22		b = (23.01) → 22.01
22		28	00	0	21	31		Mark, Transfer → 00.00
50		52	07	0	09	31		Type line 19

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 u0 u1 u2 u3 u4 u5 u6

FORM BCD 108-5.0

At this point let's discuss the program as it is coded. The first part of the program to be operated will be in the loader. With the entire program tape, generated in a manner already described, loaded on the photo-reader mechanism, you will strike the p key, and one block of tape (the loader) will be read into line 19. Striking the p key also sets the computer to take its next command from word 00 of command line 7 (line 23). This word will, of course, be the same as word 00 in line 19, and, as a matter of fact, so will words 01, 02, and 03. The command located at 23.00 is a "block copy" of all of line 19 into all of line 02. Remember the function of the T number in an immediate command: it is to serve as a flag, stopping the execution of the command after word-time T - 1, and before word-time T. Word-time T, in this case, 01, will be the first word-time following the end of the execution of the command at 23.00, and it therefore is the first word-time available for the reading of another command. No time will be lost while the computer waits to read the next command if it is located in word 01. This, then, explains the choice of word 01 as N in the command at 23.00. Of course, the next command in the program must be located at word 01 in the same command line. It is, and it is a "mark and transfer control" command, causing the computer to take its next command from word 03 in line 02. This is also an immediate command, and must be executed for one word-time, 02. Thus 03 is the best possible location for the next command, and that's why it was chosen as N in the command located at 23.01.

Now, notice that line 19 contains, in words 00, 01, 02, and 03, the same four words that are contained in line 23, and that line 02 contains the same words as line 19. Therefore, words 00, 01, 02, and 03, in line 02 are the same as the four words in line 23. So command 03 in line 02 is the command as shown on the coding sheet for the loader. At 03 a command from line 02 will be read which will call for the reading of a block of tape, and the program will continue to 05, the next command. Up to this point, no time has been lost in either waiting to read the next command or in waiting to execute any command.

The command located at 02.05 will cause the computer to test for the normal input/output system "ready". If it is not ready (there is an input or an output in progress), the answer to the question asked of the computer will be "no", and the program will continue at N. Remember that the G-15 will continue to compute while an input or output is in progress. Since N of the test command is 05, the test will be repeated, over and over again, until the input/output system is ready, which will be the case only when the block of tape has been read. Notice that the computer has not stopped operating; it is merely repeating the same command until a given condition exists, at which point it will go on to a new command, and the only reason it is doing this is that this is the way the program was written. Notice that the ready test is also an immediate command, executing during word-times 06 through 04, when it is stopped by the flag (05) in T. 05 is therefore the next available word-time for reading a command, and N equals 05, so there will be no lost time in waiting to read the next command.

Further study on your own should indicate that, because of the T's and N's chosen in the commands of the loader, there will be no wait-time of either variety involved in the operation of this part of the program. Eventually, at word-time 14, another immediate command will be read, and this one, during word-times 15 through 14, will "block copy" all of line 19 (at this point containing the input/output portion of the program) into all of line 02, destroying the loader, which has, at this point, outlived its usefulness. Here the first wait-time is encountered. The last word-time of execution of this command is 14, so the next available word-time at which the next command could be read will be 15. But the next command is called for from word 00 of line 02 (the program continues in the same command line, although the contents of that line have changed). Word-times 15 through u7 will be lost time during the current drum cycle, while the computer waits to read the next command (at 00).

The next command in the program (02.00) calls for placing the magnitude of AR's present contents in AR. Its execution time will be 01, and no time will be lost in waiting to execute. You might ask yourself, looking at the coding for 00 in the input portion of the program, in line 02, why a drum cycle would not be lost. It has been stated previously (page 151) that PPR will make all commands with $D \neq 31$ deferred. And a long time ago (page 69) it was stated that, when the computer is directed to defer execution of a command until the word-time indicated in T, it must wait one word-time at least before it can execute the command. If, then, a deferred command is given at word-time 00, to be executed in 01, why is it that the computer will not have to wait until 01 in the next drum cycle to execute the command? The reason is that, the authors of PPR, thinking all the time, were one jump ahead of you. When PPR is directed to make a command deferred ($D \neq 31$), it first tests to determine the relationship between T and L. If T is one greater than L, PPR increases T by one (T = L2) and makes the command immediate. In other words, the command at 00 of the input part of the program, could have been coded in the following form, and the same effect in the resultant machine-language program would be achieved:

L	P	T	N	C	S	D
00	u	02	04	2	28	28

Although no time will be lost waiting to execute this command, notice that two word-times will be lost waiting to read the next command, since it is called for at 04, even though the next available word-time for its location will be 02. The reason for this is that line 02 must contain, as well as this part of the program, the output format, which has already been discussed, and which in this case, occupies words 02 and 03 of the line. Obviously no command can be stored in either of these locations without disturbing part of this format.

You should be able to carry out the rest of this analysis of the program, as coded, for yourself, spotting any wait-to-execute or wait-to-read command time. Notice that an attempt has been made to cut down on this wasted time as much as possible, although this program still wastes quite a bit of time. You might devise for yourself ways in which to cut down

on this waste, by arranging the incoming data (a, b, and c) differently, performing some of the operations in a different order, using different temporary storage locations, etc.

Notice that, towards the end of the computation portion of the program, in line 00, the amount of wait-time increases, due to the increasing unavailability of storage locations for the necessary commands.

This program is not the most efficient method of arriving at the solution for the roots of a quadratic equation; it is straight-forward, however, and coincides with the original flow-diagram for the solution of the problem, as developed on pages 122 - 126.

It was operated with the inputs:

$a = 1$ (0000080)

$b = 1$ (0000080)

$c = -6$ (0000300-)

and the results were:

x_1 = 0000101 = $2(2^{-21})$ + Princeton round-off (from division)

x_2 =-0000181 =$-3(2^{-21})$ + Princeton round-off (from division).

The Program Preparation Routine, as it appears on punched tape, is very long, consisting of many blocks of information. Of all these blocks, four are basic, in that they form the heart of the routine. With them you can prepare all of the commands in machine language for a long line; you can also enter hex constants into the long line at any desired word-time; you can punch a tape of the line's contents; and you can set up almost all of memory in any way you wish (not all of memory, however, because PPR has been written to protect itself, and, if you command it to destroy part of itself, it will reject your command).

The line in which PPR stores the commands and constants you give it is line 18. Word-times will be fixed in this line. The whole line itself may, however, be placed in any other long line in the memory of the G-15, with the exceptions of lines 05, 15, 16 and 17, since these are the lines occupied by the basic package of PPR. Thus, we could use PPR to enter the proper machine-language commands at proper word-times in line 18, along with the properly located hex constants, and then cause PPR to copy line 18 into line 00, thus setting up the main computation part of our program. We could also cause PPR to punch a block of tape containing this line of program, so that it would be preserved in its machine-language form, relieving us of the necessity of having to re-type all of the individual commands and constants, in case it is ever desired to re-enter the program into the computer.

When PPR has been read into the memory of the computer (p, then GO), its four basic lines will be occupying lines 05, 15, 16, and 17, and the computer will be "hung up" on a ready test, gating type-in. PPR is waiting for a command from you to do something.

The first thing you would probably want PPR to do is to clear any already existing garbage out of line 18; you would probably want to do this just prior to making up any line of program or constants. The command which will tell PPR to do this is x00 (tab) s. The neons on the front panel of the computer will flicker momentarily, as line 18 is cleared.

PPR, after performing an indicated task, will always return to that point where it gates type-in of a new command, eager to do more work. The next thing you might want to have it do is start a series of commands, accepting your commands in their "decimal command" form, converting them to their true binary machine language form, and storing them in line 18 at the proper word-times. PPR can be told to start such a series of commands at word-time ab by the command, yab (tab) s. It will immediately type a carriage return and L = ab, and then gate type-in of the command, itself. When the command has been typed in, followed by (tab) s, PPR will convert the various portions of it, make up the proper machine language command, and store it in word ab of line 18. On the next line, on the typewriter, PPR will then type a new L, equal to the N of the command just entered. PPR will be unable to notice the end of such a sequence of commands, and, at some point, after a new L has been typed out, you will not enter another decimal command to be converted and stored, but, instead, you will enter another command to be interpreted by PPR as telling it what to do next. PPR will be able to recognize this new command; there is no danger of it trying to incorporate it as another machine language command in the program you are making up.

This next command you give PPR might be to reproduce, on punched tape, the present contents of line 18; in this case it would be x06 (tab) s. All of the words in line 18, 108 of them in all, will be added, regardless of overflow, and the sum, called a "check sum", because it can be used to check for accurate read-in of the tape later, will be typed out, after which the contents of the line will be punched on tape. In our previous discussion of typing out or punching the contents of line 19, it was pointed out that the output will end when the end code of the output format is sensed, and a resulting check of all of line 19 indicates that the whole line is clear. PPR block copies all of line 18 to line 19, and then executes a command calling for the contents of line 19 to be punched on tape, under control of an "abbreviated" format (DDDDDDDDDDDDDDDDDDDDDDDDDDDDDE), which calls for four words of output at a time. If each D calls for an output character representing four bits, 29 D's will "cover" 116 bits = four full words. This tape will be relatively unintelligible upon inspection but will serve as interim storage of all bits. When it is read into line 19, line 19's original setting, bit-by-bit, will be reproduced. If words 00, 01, 02, and 03 in line 18 are all clear, but there is some non-zero information in the next higher group of four words, the last word of output would be word 04.

When such a tape is read into the computer, the last word from tape will remain in 00 of line 19, leaving the first word of input in word u3 of line 19. This would not be too good, since all the words in the line would be removed from their correct location by four word-times. In order to avoid such an embarrassing state of affairs, PPR will, when called upon to punch line 18's contents on tape, check words 00 through 03 for non-zero. If they contain only zeroes, PPR will insert two bits in word 03, so that its hex form will be 4400000. Thus, as the output continues, every time the end code of the format is reached, a check of line 19 will yield non-zero, causing output to continue for four more word-times, until, finally, words 03 down through 00 will be punched on tape, at which point line 19 will be entirely cleared to zero, and the output will cease. This will assure us of a tape, which, when read into the computer, will load word 00, and therefore all other words, correctly. Remember that when such a tape is read, word 03, although you originally set it with nothing, will now contain the hex number 4400000.

"PRECESSION", AS USED BY PPR

When you call for the output of line 18's contents, there may be many full words of zeroes at the high end of the line. Output of these words is unnecessary since they need not be filled, during input, to guarantee that the non-zero words will be set correctly. If only the last four words contain non-zero information, only four words will be punched on tape. When this tape is read into the computer, only the last four words of line 19 will be set, but they will be properly set. PPR gets rid of words containing all zeroes prior to punching out the contents of line 18, by four-word groups. As soon as the first non-zero four-word group, counting from u7 down, is encountered, output is initiated. To understand how PPR eliminates words from a line four-at-a-time, you must understand the effect of the following command, where both S and D are less than 28, and C = 2:

L	P	T	N	C	S	D
L	u	L_k	N	2	S	D

Consider first a short line. S = D = 20, T = L_6. A C code of 2 calls for an exchange of AR with memory. During each word-time of execution, (AR) ⟶ D.T and (S.T) ⟶ AR. Let the command be:

L	P	T	N	C	S	D
00	u	06	N	2	20	20

word-time of execution:	AR holds contents of:	this word in line 20 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
03	(02)	(02)	(03)
04∼00	(03)	(03)	(00)
05∼01	(00)	(00)	(AR) = (01)

The contents of line 20 have moved up one word-time: what was in 00 is now in 01, what was in 01 is now in 02, what was in 02 is now in 03, and what was in 03 is now in 00. AR's original contents have been restored to AR.

Notice that a fifth word-time of execution is necessary, even though we are moving four words. It is necessary to complete the cycle and restore AR.

In the case of a long line, where 108 words are to be moved up, it is impossible to get a 109th word-time of execution in one immediate command. Yet this word-time of execution is necessary, to complete the cycle and restore AR. Consider the command,

L	P	T	N	C	S	D
00	u	01	N	2	19	19

word-time of execution:	AR holds contents of:	this word in line 19 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
.	.	.	.
.	.	.	.
.	.	.	.
u7	(u6)	(u6)	(u7)
00	(u7)	(u7)	(00)

All 108 words in long line 19 have been moved up one word-time with the exception of word 00, which never reached 01, although it would have, had the execution been continued for one more word-time. AR currently holds the contents of word 00, and its original contents are in word 01, where we desire to place the contents of word 00. One more word-time of execution, during 01, would cause the contents of AR and 19.01 to be exchanged, restoring AR with its original contents, and placing the contents of 19.00 in 19.01. Therefore, the command above, in order to achieve the desired effect on the entire long line, must be followed by one more command, which exchanges the contents of AR and 19.01. Why can't one immediate command be executed for 109 word-times? This question will be left for you to answer.

Let the above command be followed by another:

L	P	T	N	C	S	D
01		01	N	2	19	19

These two commands, taken together, will cause all of line 19 to be moved up by one word-time, and AR to be restored. Notice that a whole drum cycle will, of necessity, be lost in wait-time, either waiting to read the next command or waiting to execute a command.

It is essential that no commands affecting either AR or 19.01 intervene between these two. For that reason you should try to keep them together as a pair.

It is interesting to note that a long line can be moved up two word-times, with the cycle completed and AR restored, in exactly the same number of commands and time:

L	P	T	N	C	S	D
00	u	01	01	2	19	19
01	u	02	N	2	19	19

word-time of execution:	AR holds contents of:	this word in line 19 receives:	AR receives:
01	(AR)	(AR)	(01)
02	(01)	(01)	(02)
.	.	.	.
.	.	.	.
.	.	.	.
u7	(u6)	(u6)	(u7)
00	(u7)	(u7)	(00)
----------	----------	----------	----------
02	(00)	(00)	(02)=(01)
03	(02)=(01)	(01)	(03)=(02)
.	.	.	.
.	.	.	.
.	.	.	.
00	(u7)=(u6)	(u6)	(00)=(u7)
01	(00)=(u7)	(u7)	(01)=(AR)

All words in the long line have been moved up two word-times, and AR has been restored.

Therefore, since PPR desires to eliminate words containing all zeroes by groups of fours, from the high-order end of line 19, it is done through pairs of commands of the type shown above. Two such pairs move all words in any long line up by four word-times. By assumption, if PPR tests and finds such may be done, four words containing zeroes will move "up" from 19.u4 - u7 to 19.00 - 03.

The movement of words through a line in this fashion is called "precession".

OTHER PPR OPERATIONS AVAILABLE

Before commanding PPR to punch a tape containing the contents of line 18, you may want to place some hex constants in the line at strategic locations. The command which tells PPR to accept a seven-digit hex number and store it at location ab in line 18 is: zab (tab) $\pm$ $d_1d_2d_3d_4d_5d_6d_7$ (tab) s. Of course, this must be repeated for each constant to be so entered.

If, in accepting a sequence of decimal commands, PPR finds that a location which is about to receive a new command is already filled, PPR will type out the contents of that location before accepting the new command. You may disregard this typeout and proceed to enter the new command into the location, if you desire to destroy the word currently contained there. If you desire to keep that word, you may, of course, choose a new location for the command you desire to enter, merely by giving PPR a new yab (tab) s command, specifying a new ab. If the last yab, which started the current sequence of commands, was followed by a minus (-) prior to the (tab) s, the contents of any location called for which is found to be non-zero, will be converted by PPR to decimal command form prior to the typeout. If, in the last yab (tab) s command, there was no minus inserted, the contents of any such location will be typed out as a hex number.

If you cause PPR to enter a hex number in any location, it will enter the number, and then it will type out the location into which the number was entered, followed by a typeout of the original contents of that location, if non-zero.

After a line of program and/or constants is made up in line 18, PPR may be commanded to block copy all of line 18 into any other line of memory, excepting lines 05, 15, 16, and 17, all of which are occupied by PPR itself (PPR will refuse to destroy itself). The command which will cause PPR to do this is klx03, where kl stands for the desired line number.

If you find a mistake in a line of your program and wish to correct it through the use of PPR, you can cause PPR to block copy any line of memory into line 18, and, as a bonus, to type out any given word in that line. The command to do this is: klijx02 (tab) s, where line kl will be copied into line 18, and word ij will be typed out. If a minus precedes the (tab) s, the word typed out will first be converted to decimal command form; otherwise, it will be typed out as a hex number.

PPR may also be commanded to type out the entire contents of line 18: x05 (tab) s. All words will be typed out as hex numbers.

Line 18 will be typed out, and, at the same time, punched on tape, if PPR is given the command, x07 (tab) s.

PPR will punch a block of tape containing the number track if it is given the command x01 (tab) s.

PPR will read a block of tape which you have mounted on the photo-reader mechanism if given any one of the following commands:

w00 (tab) s .... read tape, type check sum

w01 (tab) s .... read tape, type check sum, type out

w02 (tab) s .... read tape, type check sum, punch tape

w03 (tab) s .... read tape, type check sum, type and punch

In any of these cases, the contents of the block of tape read will be block copied into line 18. In all of these cases, if the compute switch is on GO, after the operation called for has been performed on one block of tape, a new block will be read and the same operation will be performed on it. This will continue until all the tape has been read, and, eventually, the photo-reader will be activated, even though no tape is passing through it, in which case, the input will never cease, since no stop code will be read. If the compute switch is thrown to BP, PPR will stop at a breakpointed command in the read tape sequence of commands which follows the reading of the current block.

If PPR is stopped at any time in this manner, remember that, if the compute switch is thrown to GO, it will immediately continue in the same sequence of steps. To return to that portion of PPR which gates the type-in of PPR commands, strike sc5f. As a matter of fact, if, for any reason, you are not currently in PPR, but in some other program, and you know that PPR remains in memory intact, you can always return to that portion of PPR which gates type-in of PPR commands through this keyboard action.

Finally, PPR can be commanded to transfer control to any line from 0 through 4, at any word-time, by the command, cijx04 (tab) s, where c = 0 through 4, and ij is the location of the desired command in that line.

Other operations are possible with PPR, including certain auxiliary operations, of primary interest in three main areas:

1. automatic compilation of loader programs;

2. automatic compilation of output formats;

3. check-out and correction of programs, including listing of commands and constants.

PPR and all its auxiliary routines are thoroughly discussed in the Bendix G-15 operating manual.

DECIMAL NUMBER INPUTS AND SCALING

If PPR is capable of taking decimal numbers (for the various parts of a command), and generating the proper binary equivalents in the machine, it stands to reason that, we, too, could write our program to do this, so that we would not have to convert each decimal input to binary and then to hex prior to typing it in. Let's figure out a method for accepting decimal inputs, rather than requiring hex inputs.

Obviously, the keys on the typewriter, since there is one for each hex digit, will be sufficient for the decimal digits 0 - 9. If a digit is struck, we know that four bits will enter line 23, word 00. If a

complete seven-digit decimal number and its sign are entered, the signed number will appear in 23.00. But, unlike a hex entry, it will not be in the form of a binary magnitude and sign; it will be in a code, each four bits corresponding to a digit of the decimal number. The four most significant bits in 23.00 will contain a four-bit code for the most significant decimal digit; the next four bits will contain a similar code for the next decimal digit entered, and so on. This four-bit code for decimal digits is called "Binary-Coded-Decimal". Remember that a complete BCD (Binary-Coded-Decimal) number, signed, as it occupies 29 bits of a word in the computer, is not the binary equivalent of the represented decimal magnitude, signed. As an example, suppose we entered the decimal number, (7196323-) into the computer, digit-by-digit. In 23.00 we would have:

0111000110010110001100100011|1

representing

7 1 9 6 3 2 3 -

The problem now is to convert this code into a true, corresponding, binary magnitude. Let's treat the seven-digit decimal number entered as an integral number, so that

$$7196323 = 7 \cdot 10^6 + 1 \cdot 10^5 + 9 \cdot 10^4 + 6 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Now consider the first four bits of the BCD number in 23.00: they represent a multiple of 10^6. If we had, stored in the computer's memory, as a constant, a binary value for 10^6, we could multiply that value by the four-bit multiplier (in this case, $7_{(10)}=0111$). Similarly, we could multiply each digit times the corresponding binary equivalent of a power of 10, and, when we added all these products together, the sum would be the binary equivalent of the original decimal number. Because the decimal number was integral, so would the binary equivalent be integral. In order to prevent overflow in the additions, we would have to be sure to scale each individual product in such a way that their sum could not possibly overflow out of the most significant bit-position in the accumulator. The maximum decimal number which could be specified is 9999999. Since this is less than 2^{24}, twenty-four bits would be sufficient to express it. Therefore, if the individual products are scaled 2^{-24} in the computer, no overflow can result from their sum; of course they will all have to be scaled similarly. It is convenient, however, to scale a converted decimal integer 2^{-28}, for other reasons, which will be discussed later. This will be alright; it merely means that the binary equivalent will have at least four leading 0's.

The first four bits of the BCD number to be converted to binary are, of course, scaled 2^{-4}. They represent, in our example, 7. If we scale the binary equivalent of 10^6 by 2^{-24}, the resultant product, the binary equivalent of $7 \cdot 10^6$, will be scaled 2^{-28}, according to the rules discussed previously.

The binary (hex form will be used) equivalents of the powers of 10 involved in this case are:

10^6 = 00z4240

10^5 = 00186u0

10^4 = 0002710

10^3 = 00003y8

10^2 = 0000064

10^1 = 000000u

10^0 = 0000001

Scaled 2^{-24}, rather than 2^{-28}, these become:

10^6 = 0z42400

10^5 = 0186u00

10^4 = 0027100

10^3 = 0003y80

10^2 = 0000640

10^1 = 00000u0

10^0 = 0000010

} As these magnitudes would appear in the machine, for our purposes, scaled 2^{-24}.

Now, suppose we clear the two-word registers, and then we store the BCD number (7196323-), which we wish to convert, in MQ_1. Its sign will go to IP. Since we are loading MQ, it will be added to what is already there, but we know that is 0, since we previously cleared the two-word registers. The 28 bits of BCD code will be in the 28 most significant bits of MQ_1. Now we load $10^6 \cdot 2^{-24}$ into ID1, but with a C of 1, rather than 0. Since this C is not even, the sign of the number will accompany the magnitude, and will not be sent off to IP; of course the sign will be 0. If ID is loaded with an odd C, the setting of it will not cause PN to be cleared. We know that PN will contain 0's, though, because we previously cleared the two-word registers. Now suppose we multiply these two numbers, but allow the multiplication to last only eight word-times, rather than the usual 56. We said previously, we could cut a multiplication short at any time, provided an even number of word-times has been allowed, and be able to predict the result in PN. Eight word-times would be just sufficient for four bits from the most significant end of MQ_1 to be inspected. The effect, in PN, will be to generate the product of these four bits times the contents of ID. In other words, in PN we will have $7 \cdot 10^6$ scaled 2^{-28}, which is what we want. MQ will have been shifted left by four bits. Thus the four-bit binary code for the first decimal digit is gone, and the four most significant bits in MQ_1 contain the next decimal digit, scaled 2^{-4}.

Suppose now, we reload ID_1, again with a C equal to 1, with $10^5 \cdot 2^{-24}$. The sign in IP and the product in PN will remain undisturbed. Now we can again multiply for eight word-times. The effect will be to generate a sum in PN equal to the old product plus a new one which is, in our example, $1 \cdot 10^5 \cdot 2^{-28}$. The sum will equal $(7 \cdot 10^6 + 1 \cdot 10^5) \cdot 2^{-28}$.

We can do this seven times in all, arriving at a sum in PN equal to

$$(7 \cdot 10^6 + 1 \cdot 10^5 + 9 \cdot 10^4 + 6 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0) \cdot 2^{-28}$$

which is the desired answer. It is a binary integer, scaled 2^{-28} in the machine, representing the decimal integer we started with.

This number could be manipulated satisfactorily by a program; as far as the computer is concerned it is a binary fraction, with the machine binary point preceding the most significant bit. But, since the decimal number may have been scaled, itself, in the decimal system, containing fractional digits, this procedure would eventually lead to misery, since both a decimal and a binary scale factor would have to be remembered by the programmer. We would like to treat it as a decimal fraction in the machine, represented in binary, and, just as we previously spoke of the machine treating every number as a fraction, scaled 2^{-28}, we would now like to speak of the machine treating every number as a fraction, scaled 10^{-7}. If only seven decimal digits are allowed per number during input, then the machine value will always be a fraction, if the machine treats this number as scaled 10^{-7}.

But there is something further that must be done to the converted number, as we left it, before we can say it is scaled 10^{-7} in the machine. We have a decimal integer scaled 2^{-28}; if we divide it by $10^7 \cdot 2^{-28}$ (which, in hex, would be represented in the machine as 0989680), we will truly have in the machine the binary equivalent of a decimal fraction, and from here on out, we can drop all references to binary scaling, and speak only of decimal scaling. Therefore, in our conversion process, one more major step is required: we have an integer scaled 2^{-28} in PN_0; it is already properly located as the numerator for a division. If we load the denominator (0989680_{16}) into ID_1 with a C of 1, leaving PN undisturbed, we can perform a single-precision divide, and MQ_0 will contain the quotient. IP will still contain the sign of the number, which is what we want. The quotient will equal the original decimal integer, call it "D", divided by 10^7, or $D \cdot 10^7 \cdot 2^0$. We have thus completely eliminated binary scaling, and have substituted decimal scaling, which jibes more closely with the way in which we are used to handling numbers.

Applying the same rules here to decimal scaling that we applied earlier to binary scaling, if we say that

$$A^* = A \cdot 10^{-7},$$

where A* is the machine representation of A, we are saying that there are no fractional digits in A; its true decimal point is seven decimal

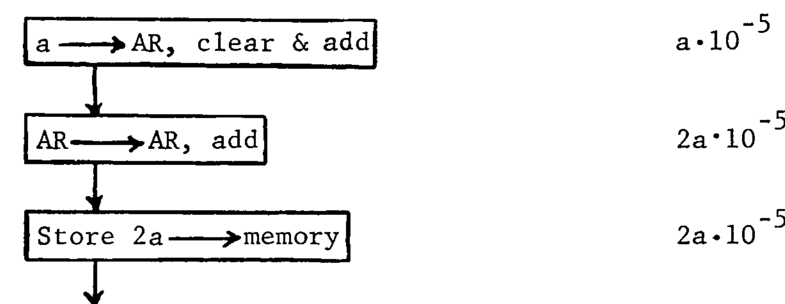
places to the right of the machine decimal point, or, following the least significant decimal digit.

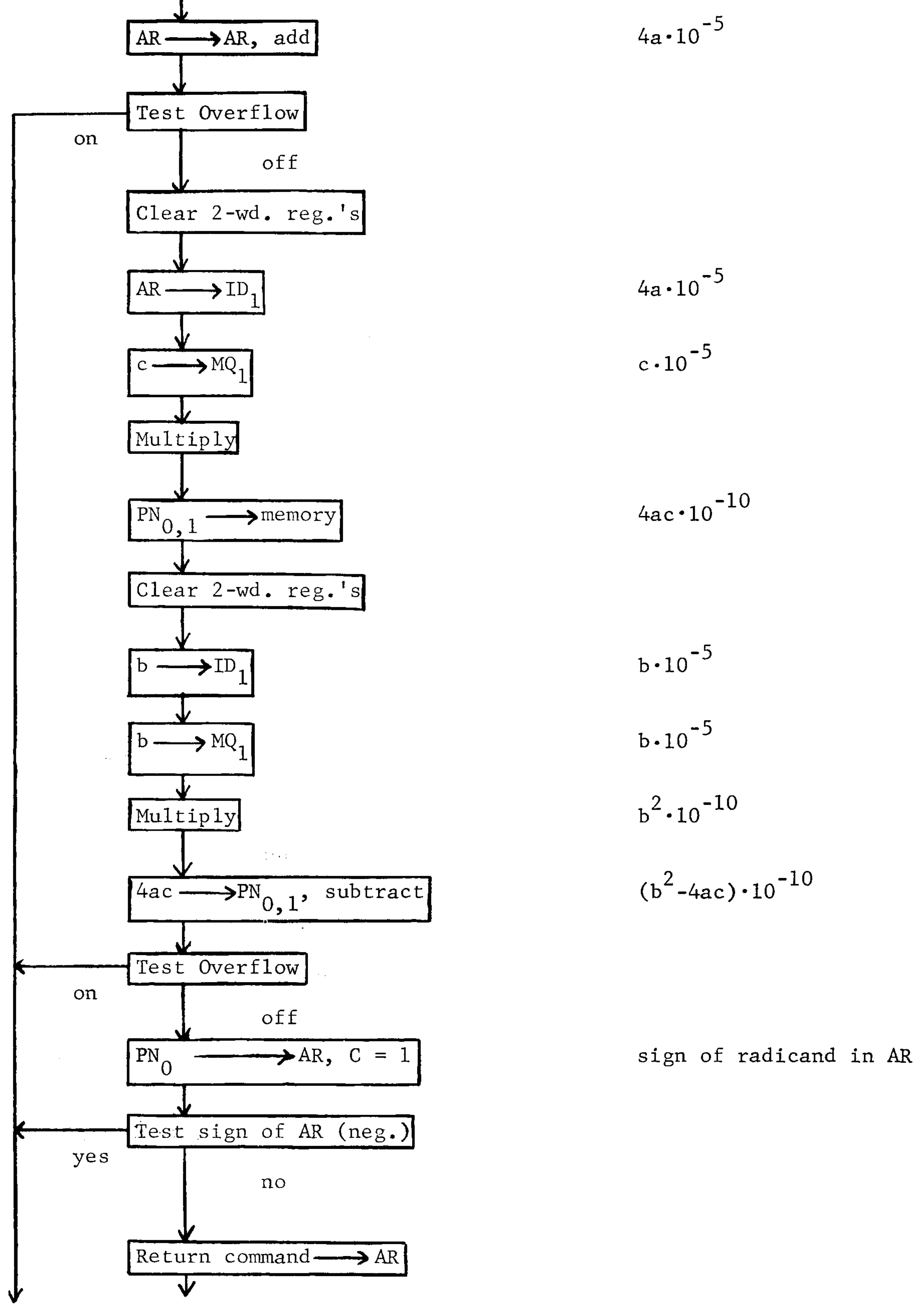
In our problem, we originally chose to allow seven binary digits for fractional accuracy, in order to carry accuracy at least to the nearest 1/100th. Now we can carry accuracy exactly to the nearest 1/100th, merely by rescaling a, b, and c. Rather than enter them as seven-digit decimal numbers with the true decimal point following the least significant digit, we can understand the true decimal point to be to the left two places allowing fractional accuracy to the nearest 1/100th. We are saying, then, that the machine will contain a, b, and c, all scaled $10^2 \cdot 10^{-7}$, or 10^{-5}. In other words, the true decimal point is five decimal places to the right of the machine decimal point, and the following two decimal digits are fractional, in the true sense of the word.

This will make the inputs easier (we haven't yet discussed the outputs, but they will be easier, too), and it will reduce the scaling problems to a form with which we are familiar. But notice what else it will do; it will greatly restrict the range of values we can have for these three numbers, and therefore, for the answers. If seven decimal digits are allowed for a number, and two of them must be interpreted as fractional, the range of values for any number is $-99999.99 \leq N \leq 99999.99$. Compare this with the ranges we were able to accommodate when we used binary scaling and did our own conversions of inputs and outputs.

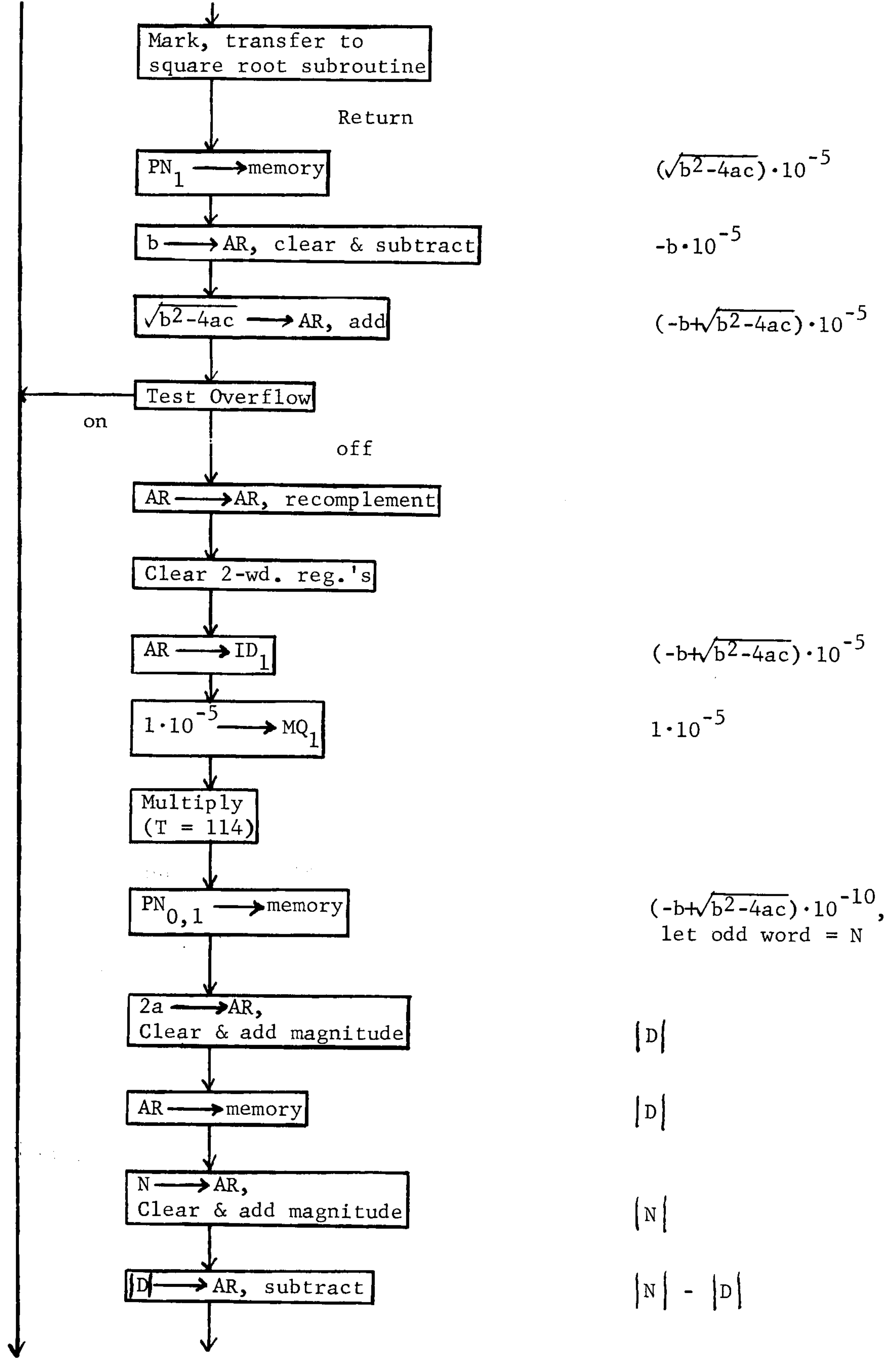
Whether you decide to use decimal or binary scaling will depend to a great deal on the requirements of your particular problem, but you should always be aware that operation of the computer is essentially designed for work with binary numbers representing binary quantities. Once you become familiar with the new number system, you will find it no harder to use than the decimal number system; as a matter of fact, you will find it easier.

The flow diagram of our computation will be simplified, if we treat the numbers in the machine as decimal numbers. Notice that we will again resort to double-precision numbers in order to retain both wide range and the necessary accuracy. Notice also the mental gymnastics we play in order to avoid repositioning an answer by shifting. Shifting rescales a number in a binary manner. It will not usually be adaptable to dealing with decimally scaled numbers.

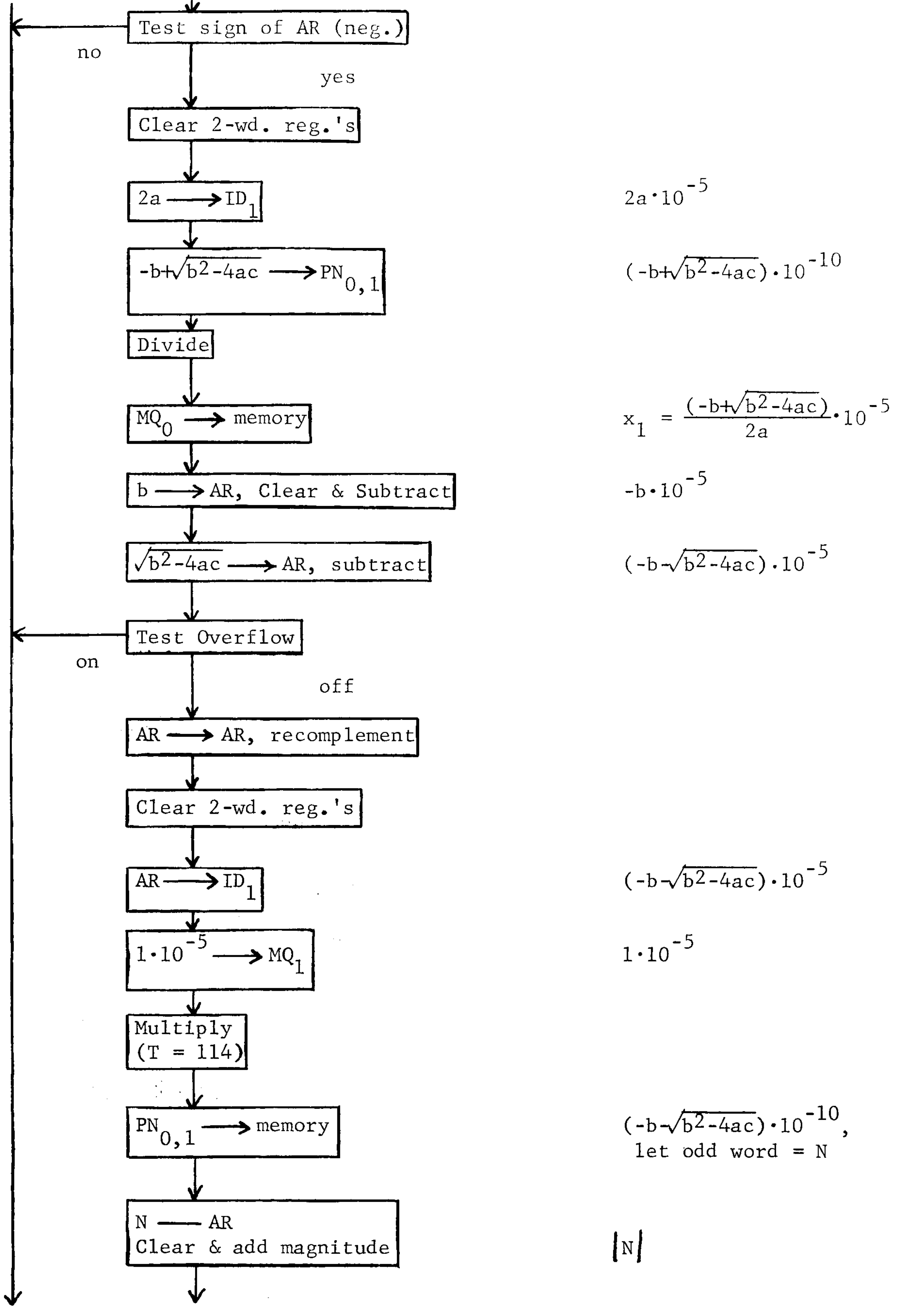


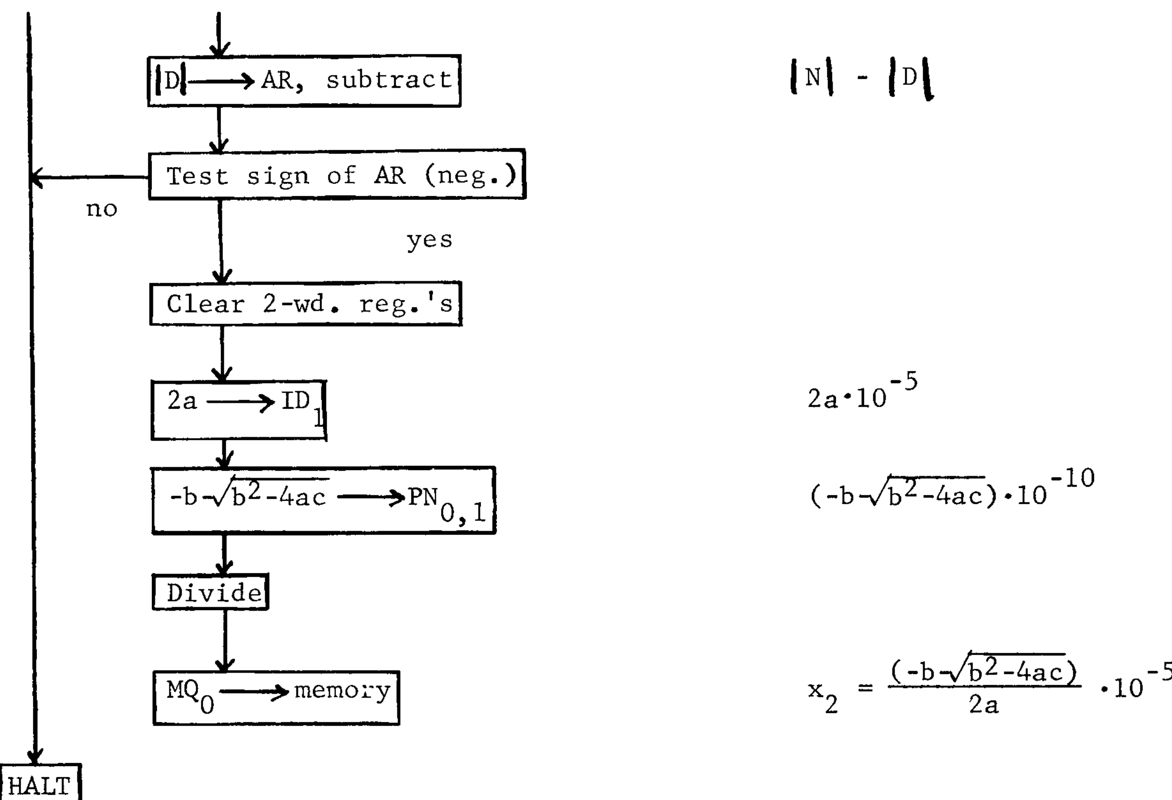


AR ⟶ AR, add
Test Overflow
on
off
Clear 2-wd. reg.'s
AR ⟶ ID_1
c ⟶ MQ_1
Multiply
$PN_{0,1}$ ⟶ memory
Clear 2-wd. reg.'s
b ⟶ ID_1
b ⟶ MQ_1
Multiply
4ac ⟶ $PN_{0,1}$, subtract
Test Overflow
on
off
PN_0 ⟶ AR, C = 1
Test sign of AR (neg.)
yes
no
Return command ⟶ AR
$4a \cdot 10^{-5}$
$4a \cdot 10^{-5}$
$c \cdot 10^{-5}$
$4ac \cdot 10^{-10}$
$b \cdot 10^{-5}$
$b \cdot 10^{-5}$
$b^2 \cdot 10^{-10}$
$(b^2-4ac) \cdot 10^{-10}$
sign of radicand in AR



Mark, transfer to square root subroutine
Return
PN$_1$ → memory
$(\sqrt{b^2-4ac})\cdot 10^{-5}$
b → AR, clear & subtract
$-b\cdot 10^{-5}$
$\sqrt{b^2-4ac}$ → AR, add
$(-b+\sqrt{b^2-4ac})\cdot 10^{-5}$
Test Overflow
on
off
AR → AR, recomplement
Clear 2-wd. reg.'s
AR → ID$_1$
$(-b+\sqrt{b^2-4ac})\cdot 10^{-5}$
$1\cdot 10^{-5}$ → MQ$_1$
$1\cdot 10^{-5}$
Multiply (T = 114)
PN$_{0,1}$ → memory
$(-b+\sqrt{b^2-4ac})\cdot 10^{-10}$, let odd word = N
2a → AR, Clear & add magnitude
$|D|$
AR → memory
$|D|$
N → AR, Clear & add magnitude
$|N|$
$|D|$ → AR, subtract
$|N| - |D|$


Test sign of AR (neg.)
no
yes
Clear 2-wd. reg.'s
2a → ID_1
$2a \cdot 10^{-5}$
$-b+\sqrt{b^2-4ac}$ → $PN_{0,1}$
$(-b+\sqrt{b^2-4ac}) \cdot 10^{-10}$
Divide
MQ_0 → memory
$x_1 = \frac{(-b+\sqrt{b^2-4ac})}{2a} \cdot 10^{-5}$
b → AR, Clear & Subtract
$-b \cdot 10^{-5}$
$\sqrt{b^2-4ac}$ → AR, subtract
$(-b-\sqrt{b^2-4ac}) \cdot 10^{-5}$
Test Overflow
on
off
AR → AR, recomplement
Clear 2-wd. reg.'s
AR → ID_1
$(-b-\sqrt{b^2-4ac}) \cdot 10^{-5}$
$1 \cdot 10^{-5}$ → MQ_1
$1 \cdot 10^{-5}$
Multiply
(T = 114)
$PN_{0,1}$ → memory
$(-b-\sqrt{b^2-4ac}) \cdot 10^{-10}$,
let odd word = N
N — AR
Clear & add magnitude
$|N|$



We have already discussed the manner in which we treat the input data, in order to supply this computation program with the numbers it needs, but, before we flow diagram a new input scheme, we should consider output. We will write an input/output program for this case, just as we did before.

We will, upon completion of the computation, have two answers, each scaled 10^{-5}, in 19.u7 and 19.u6. These will be binary numbers, but they will be meaningless to us in their binary form. We must convert them back to decimal and type out decimal answers. We can indicate the proper scaling of the answers by the positioning of the decimal point in the type-out. We know, if these answers are scaled 10^{-5} in the computer, each of them should be typed out in the form: SDDDDDPDD.

The method we developed for the conversion of a BCD number to binary grew directly out of the inspection process: we developed the binary equivalent for each multiple of a power of ten, and then added these binary equivalents. It stands to reason that, from the inspection process, as it was defined in the Introduction, we can develop a reverse conversion process for our program, to convert a binary number to its decimal equivalent.

The binary scaling of our numbers was eliminated by generating a binary scale factor of 2^0, and leaving remaining only a decimal scale factor. If each number, as it appears in the machine actually has a binary scale factor of 2^0, it is truly a fraction. The general rule for converting a binary fraction to its decimal equivalent is: multiply the binary

fraction by the binary equivalent of 10, which is 1010. Retain the integral portion of the product as the coefficient of 10^{-1}. Perform the same operation on the fractional portion of the product. Retain the integral portion of the new product as the coefficient of the next power of 10 in the decreasing series, which would be 10^{-2}. Repeat the process as often as necessary (the fractional portion of some product equals 0), or until the desired accuracy in the converted form of the original fraction is achieved.

As an example, consider the conversion to decimal of the binary fraction,

```
                .1011100000111001110101000011
                                         1010
              -------------------------------
                1 0111000001110011101010000110
              101 1100000111001110101000011 0
              -------------------------------
7(10)        0111.0011001001000010010010011110
                                         1010
              -------------------------------
                0 0110010010000100100100111100
              001 1001001000010010010011110 0
              -------------------------------
1(10)        0001.1111011010010110111000101100
                                         1010
              -------------------------------
                1 1110110100101101110001011000
              111 1011010010110111000101100 0
              -------------------------------
9(10)        1001.1010000111100100110110111000
                                         1010
              -------------------------------
                1 0100001111001001101101110000
              101 0000111100100110110111000 0
              -------------------------------
6(10)        0110.0101001011110000100100110000
                                         1010
              -------------------------------
                0 1010010111100001001001100000
              010 1001011110000100100110000 0
              -------------------------------
3(10)        0011.0011110101100101101111100000
                                         1010
              -------------------------------
                0 0111101011001011011111000000
              001 1110101100101101111100000 0
              -------------------------------
2(10)        0010.0110010111111001011011000000
                                         1010
              -------------------------------
                0 1100101111110010110110000000
              011 0010111111001011011000000 0
              -------------------------------
3(10)        0011.1111101110111111000111000000 0
```

The decimal equivalent of the above binary fraction is .7196323. Compare the binary equivalent of this fraction with the BCD number corresponding to this fraction.

We can use the above method to convert each answer generated by our program to its BCD equivalent. Then we can type out this BCD number as the answer, so that, on the typewriter, a decimal number will appear which can be read directly. Of course, the BCD equivalent will be in the form of a fraction: it will be a decimal number scaled 10^{-7}.

If we choose to interpret it, as we do, scaled 10^{-5}, we can move the decimal point to the right five decimal places, arriving at a number, as we have already seen, of the form SDDDDDPDD. Thus, we can use the output format itself to properly scale the answer, as it is typed out, so that it can be read directly.

We will place the answer in ID_1 and the multiplier, $1010_{(2)}$, in MQ_1, in the four most significant bits. When we multiply a number scaled 2^0 (the answer) by a number scaled 2^{-4} (the multiplier), we get an answer scaled 2^{-4} (the integral portion of the product will be in the first four bits of PN_1). If we had a way of "extracting" these four bits and saving them, we could then reload ID with the remaining bits from PN, which constitute the fractional portion of the product, reload MQ_1 with the same multiplier (the first one was shifted out as it was inspected), and perform the process all over again. Eventually, after seven such operations, each time saving the integral portion of the product, we would have the seven BCD digits corresponding to our answer. If we had a way of recombining them, end-to-end in one word, we would have exactly the number we wish to type out as an answer.

EXTRACT, AND ITS USE IN NUMBER CONVERSION

There is an extract operation available in the G-15. It is called for by a special command of slightly different form from the other special commands we have thus far considered. S in this command equals 31; again the computer will know, since there is no line numbered 31, that a special operation is being called for. C equals 0. Any destination may be specified. The command will operate during whatever word-time(s) is (are) specified; it may be either immediate or deferred. The number out of which certain bits are to be extracted must be in line 21, in whatever word you desire (of course, you must be sure that it will be in the word available during the word-time of execution of this command). In the corresponding word of line 20, there will be a "mask", which will specify to the computer which bits you wish extracted. The mask in line 20 will contain a 1 in each bit-position to be extracted, a 0 in each bit-position you wish left behind. For instance, the mask in our case above, would be: 1111000000000000000000000000, causing only the first four bits of the product (the integral portion, a BCD digit) to be extracted. The result of the extraction, containing 0's in all those bit-positions not extracted, will be transferred to the destination during the same word-time. The number from which the bits were extracted will remain intact in line 21, while the mask will also remain intact in line 20.

There is a second extract operation available in the G-15, which causes exactly the same sequence of operations to occur, with the exception that the bits extracted from the number in line 21 will be those bits corresponding to 0's in the mask, while the others, corresponding to 1's in the mask, will be left behind. In short, the extract operation is the same, but interpretation of the mask is exactly reversed. The command for this is: S = 30, C = 0, D may be any line.

Consider now, the first product we arrived at, on page 176.

01110011001001000010010010011110

This requires more than 29 bits, and therefore, we would have to use two words in lines 20 and 21. This means we would have to make the extract operations immediate for two word-times of execution ($T = L_3$). We could load this number in line 21, and a mask in line 20, as shown below (word-boundaries have been ignored):

01110011001001000010010010011110	Line 21 (number)
11110000000000000000000000000000	Line 20 (mask)
01110000000000000000000000000000	Result 1
00000011001001000010010010011110	Result 2

The first extraction, S = 31, yields result 1, in which we have only the first binary-coded-decimal digit, in its proper position. We could now store this in memory.

The second extraction, S = 30, yields result 2, in which we have only the remaining fractional portion of the product, similar to that shown in the first product on page 176, except that here, there are four leading 0's. This is fine; if we transfer this to ID (double-precision) and multiply it by $1010 \cdot 2^{-4}$, we will get the second product shown on page 176, except that it will be removed four places to the right. In short, this second product will be: 000000011111011010010110111000101100. If we load this into line 21, and a mask into line 20, as shown below, we can perform the extractions all over again:

000000011111011010010110111000101100	Line 21 (number)
000011110000000000000000000000000000	Line 20 (mask)
000000010000000000000000000000000000	Result 1
000000001111011010010110111000101100	Result 2

The first extraction, S = 31, yields result 1, in which we have only the first binary-coded-decimal digit, in its proper position (the second group of four bits). We could now store this in memory.

The second extraction, S = 30, yields result 2, in which we have only the remaining fractional portion of the product, similar to that shown in the second product on page 176, except that here, there are eight leading 0's. This, again, is fine; if we transfer this to ID and multiply it by $1010 \cdot 2^{-4}$, we will get the third product shown on page 176, except that it will be removed eight places to the right.

We could continue this process until seven multiplications have been performed, with all of the accompanying extractions. At that point, we would have saved seven results from the first extractions, and they would be,

eliminating trailing 0's, and expressing them as single-precision 28-bit magnitudes:

```
0111000000000000000000000000
0000000100000000000000000000
0000000010010000000000000000
0000000000000110000000000000
0000000000000000001100000000
0000000000000000000000100000
0000000000000000000000000011
```

If we now add these together, we will get:

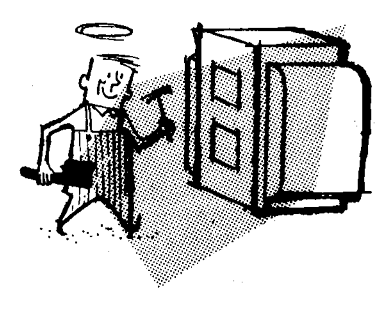
```
0111,0001,1001,0110,0011,0010,0011
```

If this were typed out, it would yield:

```
7   1   9   6   3   2   3
```

which is the number we expected.

This method will work, and we could program it, but it involves many, many commands. Prior to each set of extractions, the product must be transferred from PN to line 21, and a new mask must be transferred into line 20. Prior to each multiplication, the two-word registers must be re-set-up. This method will also require quite a few storage locations in memory.



The Bendix Computer Division engineers, always aiming to make life easier for the programmer, built into the G-15 a special extract command, designed especially for the purpose of saving bits in PN and resetting ID for further multiplication. It is a special command: D = 31, S = 23, C = 3. It may be either deferred or immediate. In our case, we will make it immediate, operating for two word-times, thus covering both halves of PN and ID.

During each word-time of execution of this command, the word in that word-time of line 02 will serve as a mask for an extraction. The bits in those bit-positions in PN which correspond to the bit-positions in the mask containing 0's, will remain in PN. The bits in PN for which there are corresponding 1's in the mask in line 02 will be transferred to ID. PN will retain only the results of the extraction which treated the mask in reverse (extracting bits corresponding to 0's in the mask); ID will receive only the results of the extraction which treated the mask in the normal manner (extracting bits corresponding to 1's in the mask). Thus, if we set up masks in line 02 which will have 0's corresponding to the integral portions of PN, and 1's corresponding to the remaining bits (the fractional portions

in PN), we will, through execution of this one command, generate the sum of the integral portions of the products in PN_1, while we directly reload ID for the next multiplication. Remember that the resultant product, in each case, is moved to the right by four more places, so that the integral portion of PN, as it grows longer, will never be disturbed by the succeeding multiplication.

The extractor, or mask, used after the first multiplication will be:

odd word: 0zzzzzz-
even word: zzzzzzz

After the second multiplication:

odd word: 00zzzzz-
even word: zzzzzzz

After the third multiplication:

odd word: 000zzzz-
even word: zzzzzzz

After the fourth multiplication:

odd word: 0000zzz-
even word: zzzzzzz

After the fifth multiplication:

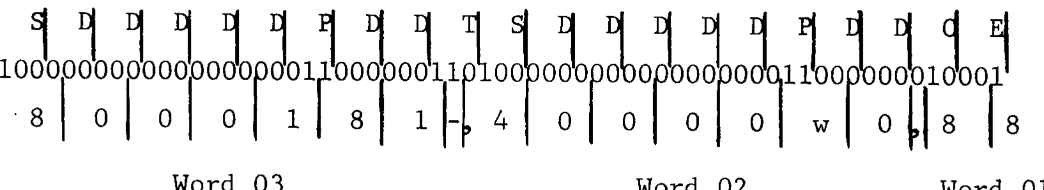
odd word: 00000zz-
even word: zzzzzzz

After the sixth multiplication:

odd word: 000000z-
even word: zzzzzzz

After the seventh multiplication, no extraction will be necessary: the seventh BCD digit will occupy the desired four bits, and no further multiplications are necessary.

The BCD number which results will be the desired answer in decimal. It can be placed properly in line 19 for type-out under control of the output format. Several times we have mentioned that we want the form of the decimal answers, as they are typed out, to be SDDDDDPDD. Our output format, therefore, for two such answers, will be:

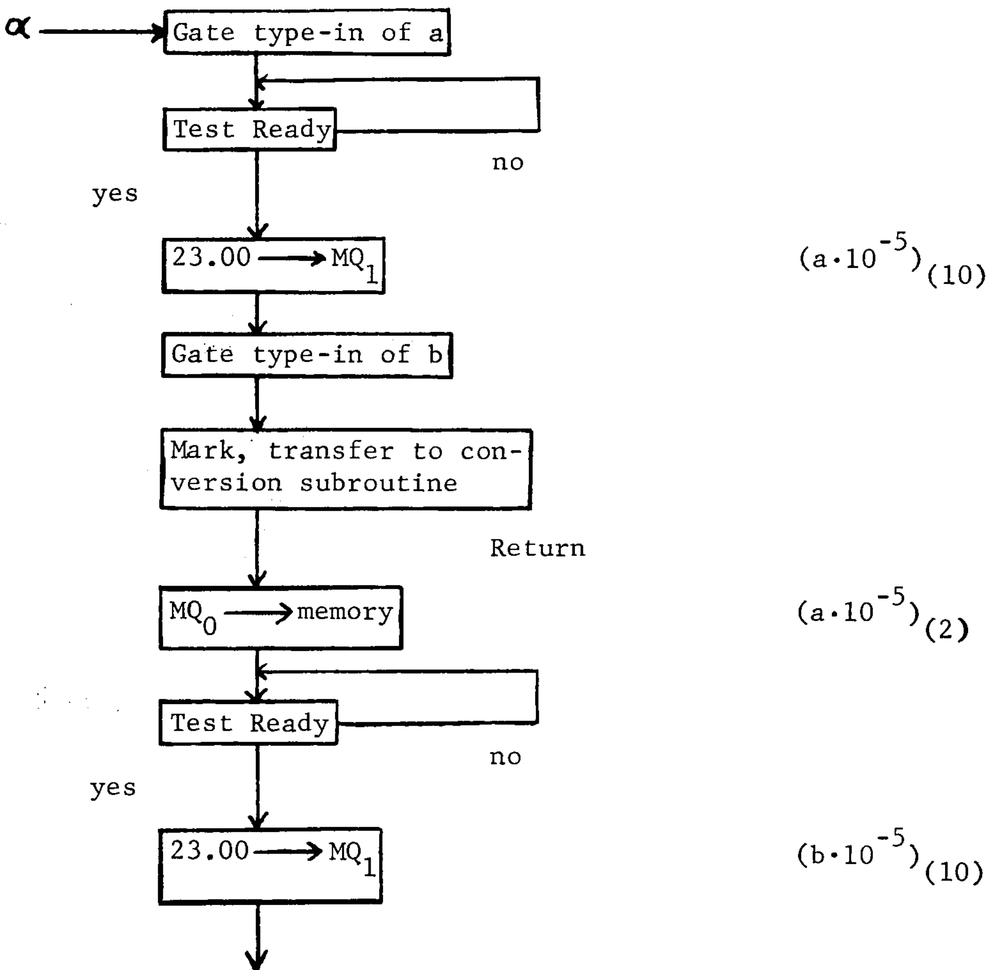


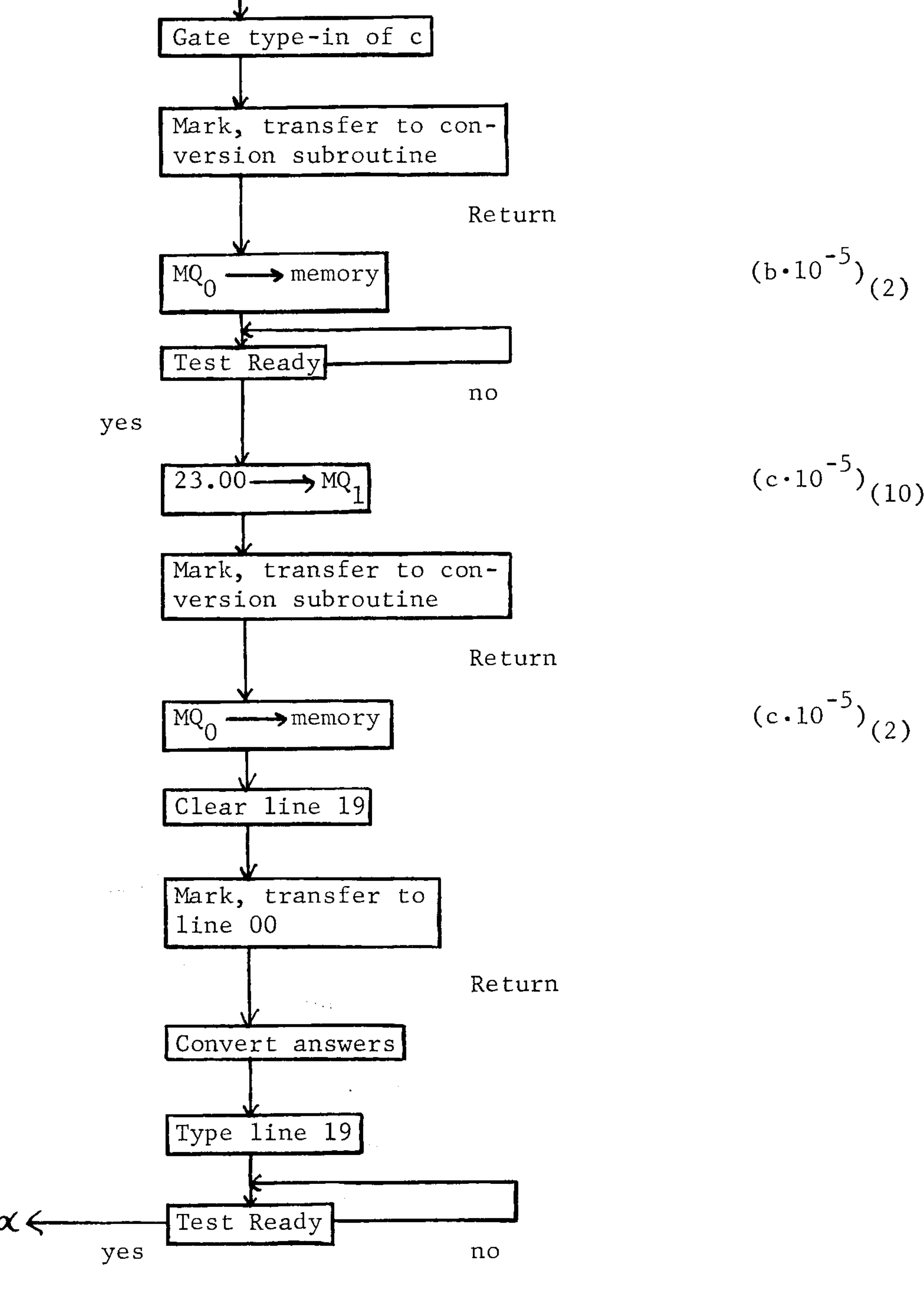
One further point remains, before we flow-diagram this new input/output program. We have mentioned, up to this point, that we will reload the multiplier into MQ_1 before each multiplication during the conversion for output. The multiplier is 1010. Why not place seven multipliers in MQ_1 all at once, and multiply for only eight word-times (therefore four bits) each time? In that case, our multiplier will look like this: 0101010101010101010101010101. Cutting short a multiplication is fine, and this will work. But, if we are going to do this, notice that each multiplication will end with a multiplication by 0. This merely means to the computer that it is not to add the shifted contents of ID to what it already has in PN. Why must we tell the computer that? There is no good reason for it, so we can eliminate the last 0 in each group of four bits, making our multiplier look like this:

1011011011011011011010000000.

We will limit the word-times of execution of the multiplication, in each case, to six, meaning that only three bits from MQ_1 will be inspected. The hex equivalent of this binary number is v6xv680.

Now let's flow-diagram the new input/output program, for line 02.

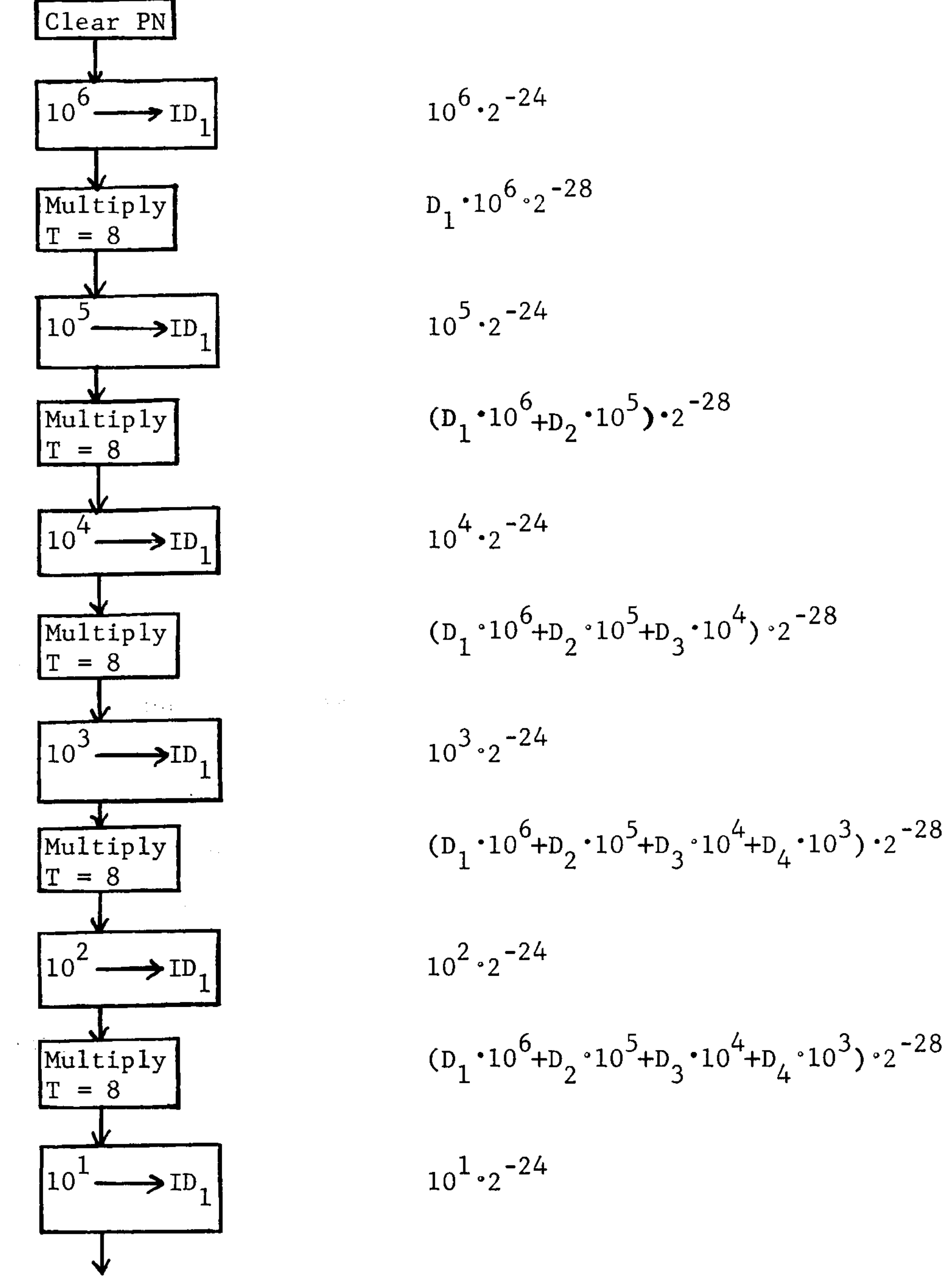


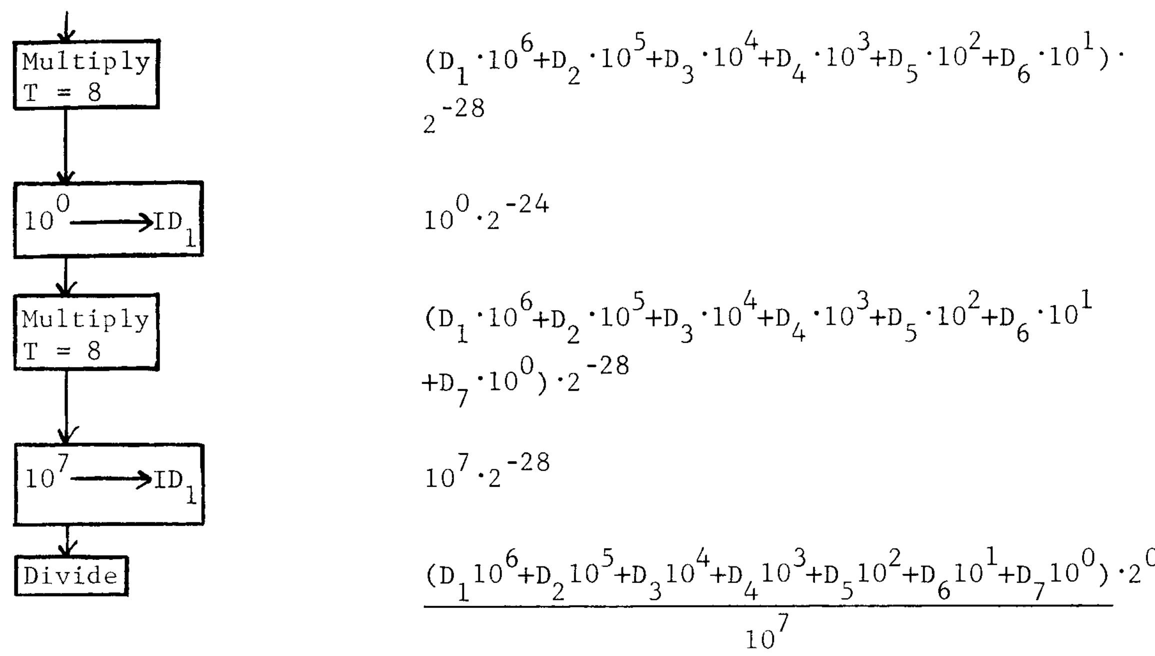


Notice that, in the preceding flow diagram, we made reference to a conversion subroutine. This we are also going to write, since we now know how. It will be in line 02, along with the input/output program. The mark, transfer command need not cause control to be transferred from one line to another; it may simply transfer control within the same line to another word-time, as it will do in this case. We need not supply this subroutine with a return command each time we enter it, because we will code the subroutine with a return command already in it.

The reason for that provision, as we mentioned earlier, is to allow very general use of a subroutine by any number of users, all of whom may not wish to return to the same line upon completion of the subroutine.

The following is a flow diagram of the conversion subroutine for BCD to binary, written assuming that the BCD number to be converted is already in MQ_1.





Another conversion subroutine is also necessary in the program, and it, too, will be in line 02. This is the subroutine for converting the binary answers to their BCD equivalents. When we discussed this conversion process earlier, we omitted one point at that time, which now must be mentioned, since you will see provision for it in the flow diagram which follows.

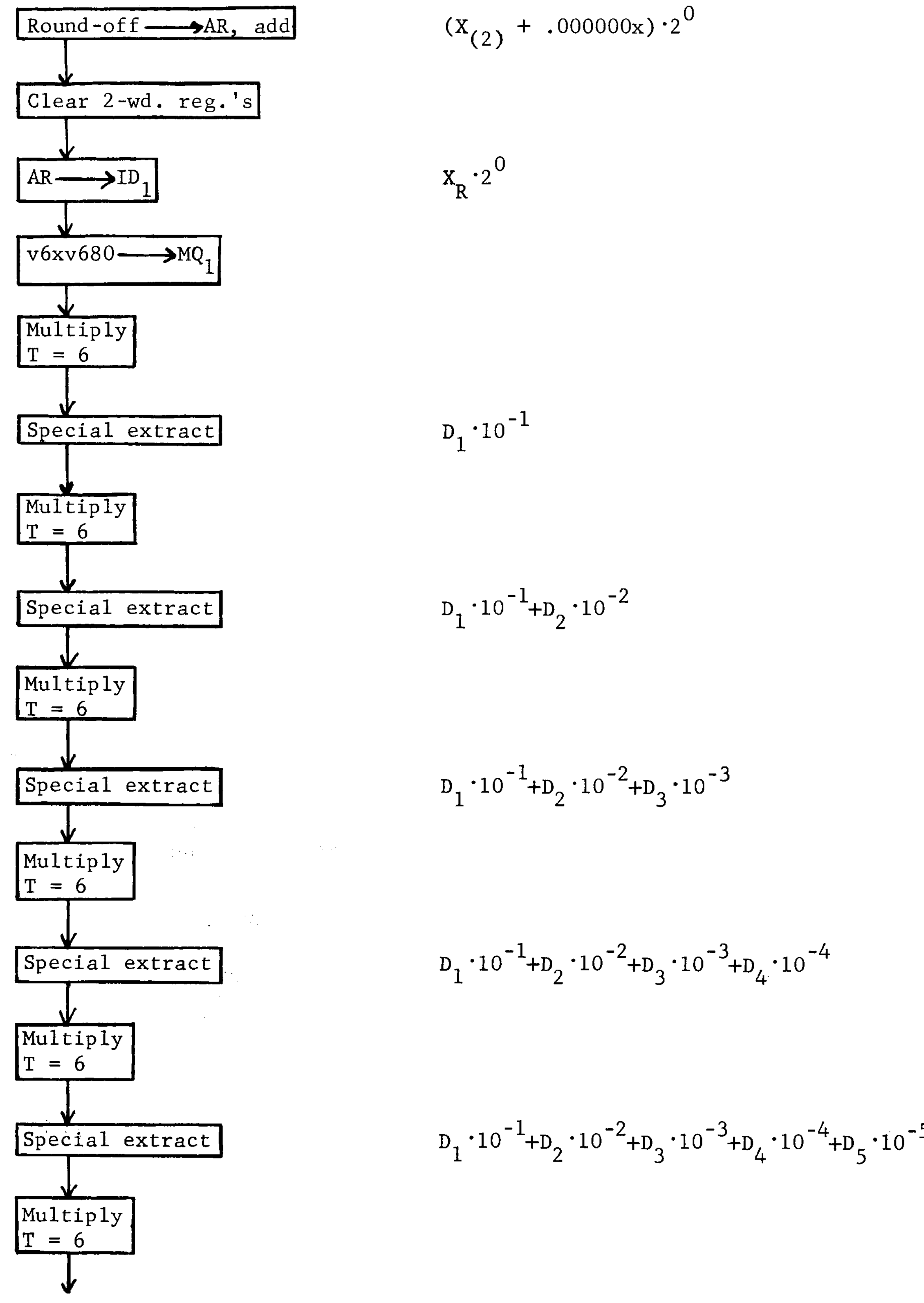
The least significant BCD digit generated will be 10^{-7}, or $1/10^7$. This quantity is a good deal greater than 2^{-28}, or $1/2^{28}$. For this reason, the last bit in the binary number prior to conversion can have no effect on the BCD result. As a matter of fact, $2^{-23} > 10^{-7} > 2^{-24}$. In order to arrive at the seven-digit decimal number closest to the value represented in 28 bits, we must round off the binary number prior to conversion.

$$2^{-28} = .000000003725$$
$$13 \cdot 2^{-28} = .000000048425$$
$$14 \cdot 2^{-28} = .000000052150$$

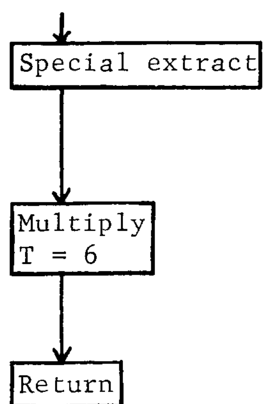
$13 \cdot 2^{-28}$ is very close to $.5 \cdot 10^{-7}$; so is $14 \cdot 2^{-28}$. But the latter exceeds $.5 \cdot 10^{-7}$, and, faced with a choice, we choose to round up to the next higher decimal digit only if we are at least within 1/2 of it. We will therefore choose the smaller of the two round-off numbers, since it will require a value in the original binary number of more than $.5 \cdot 10^{-7}$. Our round-off, expressed in hex, will be:

.000000x

Now we can flow-diagram the binary - BCD conversion subroutine, assuming the number to be converted has already been placed in AR, with a C of 1.



Round-off → AR, add
$(X_{(2)} + .000000x) \cdot 2^0$
Clear 2-wd. reg.'s
AR → ID_1
$X_R \cdot 2^0$
v6xv680 → MQ_1
Multiply
T = 6
Special extract
$D_1 \cdot 10^{-1}$
Multiply
T = 6
Special extract
$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2}$
Multiply
T = 6
Special extract
$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3}$
Multiply
T = 6
Special extract
$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4}$
Multiply
T = 6
Special extract
$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5}$
Multiply
T = 6



$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5} + D_6 \cdot 10^{-6}$

$D_1 \cdot 10^{-1} + D_2 \cdot 10^{-2} + D_3 \cdot 10^{-3} + D_4 \cdot 10^{-4} + D_5 \cdot 10^{-5} + D_6 \cdot 10^{-6} + D_7 \cdot 10^{-7}$

The loader program and the square root subroutine will remain the same. The square root subroutine just "cranks out" square roots of binary numbers. What those numbers represent makes no difference at all to the subroutine.

The program tape thus prepared can be read into the computer with an p switch action, followed by moving the compute switch to GO. The program will gate type-in, at which time a must be typed in. Type-in will be gated again, and this time b must be entered. Type-in will be gated a third time, and c must be entered. Notice that the program has been written to make as much use as possible of the time required to type in a number. You will be unable to type in the number so fast that the computer will not have to wait for you. After the third type-in, computation will take place, and two decimal answers will be typed out. These can be read directly, since their decimal points will be properly positioned. Type-in will again be gated, this time for a new set of values.

OTHER PROGRAMMING TECHNIQUES

A commonly needed programming technique remains undiscussed because the program we have been considering up to this point does not require it. However, because it is such a common technique, it cannot remain unmentioned any longer. It is called looping. Consider the case in which you are given 50 random positive numbers in 19.00 - 49, and you must sort them, placing the least number in 00, and the greatest number in 49.

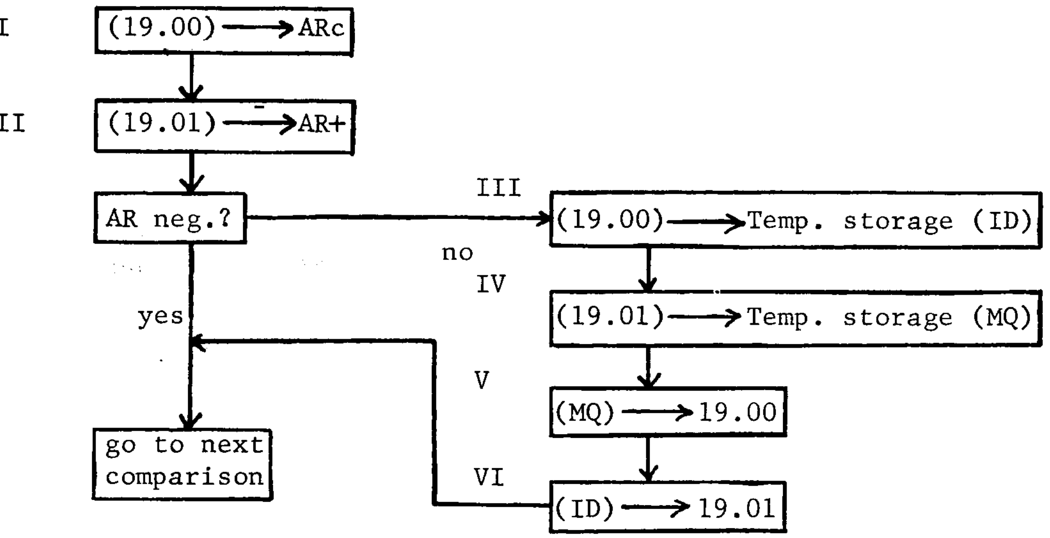
We know that commands are in the same binary form as constants and data in the memory of the computer. In fact, we know that the only distinction the computer can make between commands and constants or data is based on when it reads the words: if a word is read during RC time, it will be treated and interpreted as a command; if it is read during EX time, it will be treated and interpreted as data. Thus, any command could be treated as data by the computer, if it were read during EX time. A command could be called into AR, and have something added to it, or subtracted from it. We also know that a command can be executed out of AR, the computer being told to do this by a special command, D = 31, S = 31, C = 0. When this special command is executed, the computer is set up to take the next command from AR at time N of the special command. After

the command in AR is executed, the computer will go to N (of the command executed from AR) in the same command line from which it was previously taking commands.

The requirements of the proposed problem are to pick a number, say 19.00, and compare it with all 49 others, each time exchanging if necessary, to assure that, at the end of the series of comparisons, the least number of the 50 is in 19.00, and then repeat the process in order to get the smallest number in 19.01, etc.. The total number of comparisons necessary will be:

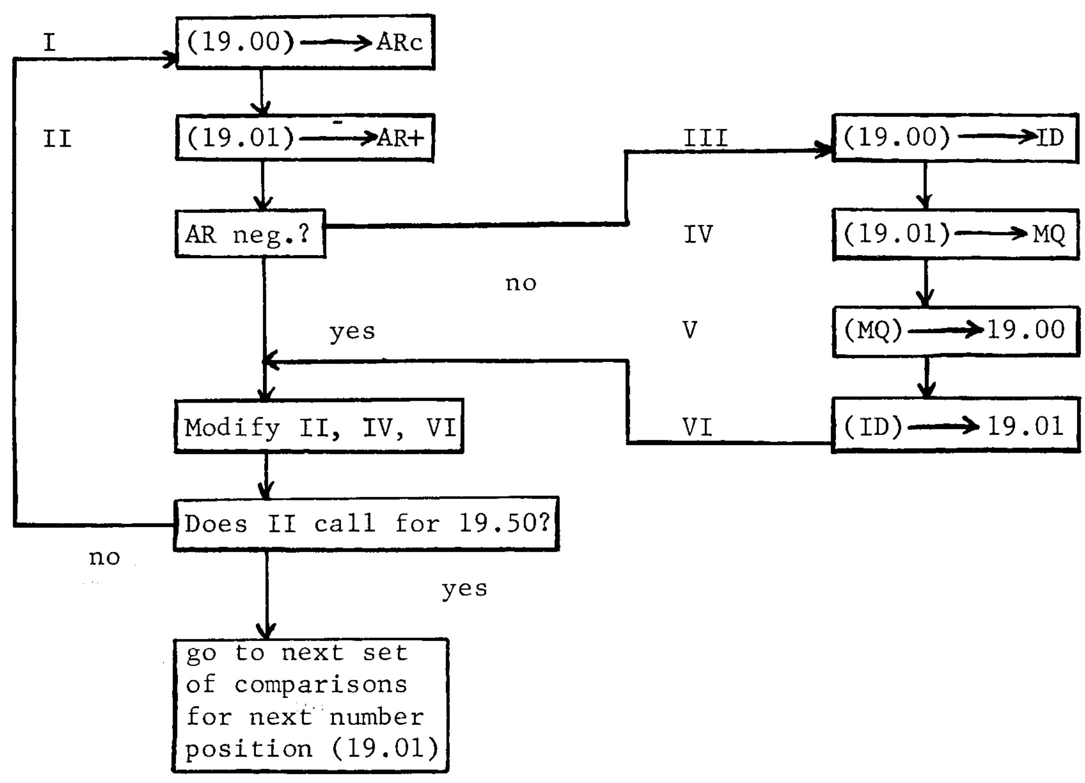
for 00:	49
for 01:	48
for 02:	47
.	.
.	.
.	.
for 48:	1
for 49:	0
	1225 = total number of comparisons necessary.

It's an easy job to flow-diagram the comparison and exchange, if an exchange is necessary:

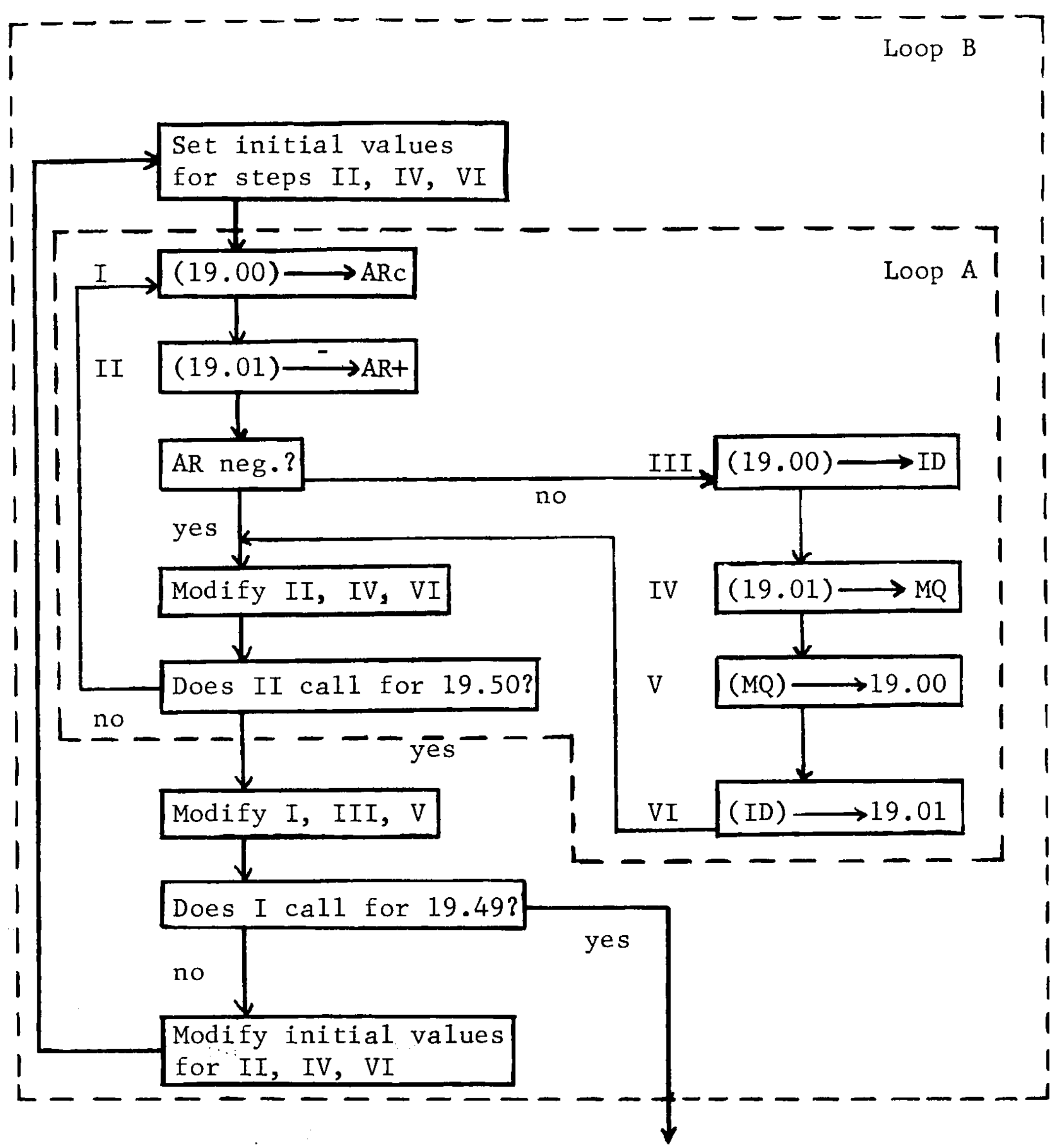


It would be an easy job to write this same sequence 1225 times, each time for the comparison of a new pair of numbers, but the resultant program would be too long to store in the memory of the computer. In the flow diagram above, the next comparison would be of 19.00 against 19.02. The same diagram would be sufficient for our purposes if boxes II, IV, and VI were modified to affect 19.02, rather than 19.01. We could then modify them again, to affect 19.03, etc.. Eventually they would affect 19.49, and after that, we could be sure that the least

number of the 50 would be in 19.00. If, instead of an arrow in the preceding flow diagram pointing to the next comparison, we inserted steps to modify steps II, IV, and VI, and then drew an arrow going back to step I, we would be indicating a loop which would be repeated over and over again, each time comparing 19.00 with a new word from line 19. This loop would be unending, and this, of course, would be disastrous. We will therefore establish a limit beyond which the loop will not continue; that limit will be after 19.00 has been compared with 19.49. At that point we will continue with the program, and the flow diagram will be as follows:



Now 19.01 must be compared with each succeeding location, resulting in the placement of the next least value in 19.01, according to the terms of the problem. This could also be done with a loop involving the above flow diagram. After the above flow diagram is followed up through the limit for step II, we can insert steps to modify I, III, and V to affect the next higher location in line 19. But, notice that steps II, IV, and VI will all be set to affect 19.50. They must all be reset, but not to their original values. Every time the above flow diagram is entered for a new set of comparisons, steps II, IV, and VI must be set to initially affect a word in line 19 whose number is one greater than the word affected by steps I, III, and IV. New initial values for II, IV, and VI will have to be set in the program every time I, III, and V are modified.



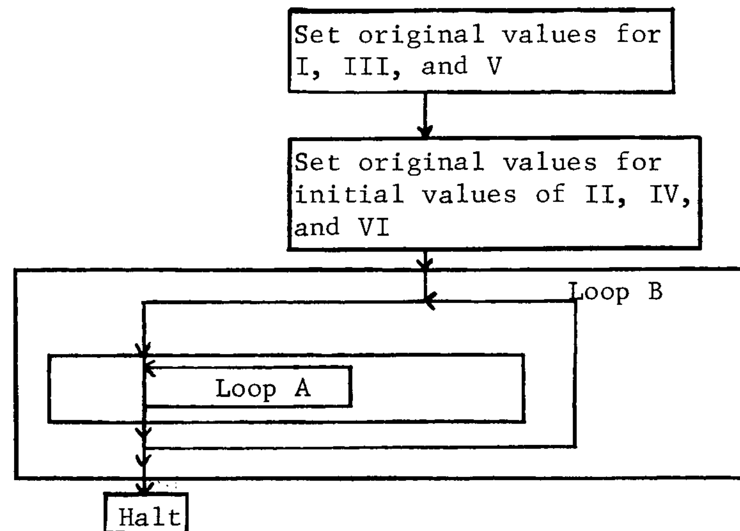
Notice that we have established a limit for the looping back to step I after modifying it, in order to prevent undesired continued operation of the program after a certain point. Only 50 numbers are to be ordered; the fiftieth is in 19.49. When all the preceding ones have been ordered, the one remaining in 19.49 must be in its correct position and need not be compared at all. So, when step I would call for 19.49, we wish to leave the loop, having accomplished all that was originally asked for.

We have, then, in the above flow diagram, two loops, one within the other. We will call the smaller one (which is operated a varying number of times per each operation of the larger one) loop A. The larger loop we will call loop B; it will be operated 49 times. A pass through a loop we will call an "iteration". The two loops are shown in the above flow diagram.

After we leave loop B, we are done, and the 50 numbers are ordered as desired. We could halt at this point, giving the HALT command an N equal to the starting location of the entire program, so that it could be repeated again, if desired.

There is one remaining difficulty with this program, as flow-diagrammed on the preceding page. After it has operated once, steps I through VI will be modified, and the program would not operate successfully in another complete pass, from the beginning, without some restoring. Every program which modifies itself should also restore itself to its initial condition, so that it can be operated as many times as desired without having to be reloaded, in its entirety, into the memory of computer. Such restoration by the program itself is called "housekeeping".

Housekeeping should be done initially in a program. We will therefore, further alter the preceding flow diagram, as follows:



The only remaining question is, "how do you modify a command in a program?"

Take step II in the above flow diagram, for instance. It calls for the transfer of 19.01 to AR+, with the sign changed and via the inverting gates. A command corresponding to this might be:

L	P	T	N	C	S	D
u6		01	50	3	19	29

To modify this command, transfer it into AR: (19.u6)⟶ARc. Use a C of 0 to do this. If the command itself is negative, that is an

indication of the fact that the command calls for a double-precision operation; this does not mean that we want to complement the binary number, however. All we want to do is to modify it in its present form, by adding a 1 to T. After (19.u6) is in AR, add the following constant ("dummy command") to it:

L	P	T	N	C	S	D
	u	01	00	0	00	00

This is called a "dummy command" because it will never be read and interpreted as a command; it is merely a constant, entered in decimal command form when making up the program using PPR, for the sake of convenience. Transfer this constant into AR+. The result in AR will be:

L	P	T	N	C	S	D
		02	50	3	19	29

Now store the present contents of AR into word u6 of the command line containing step II, and the next time this word is interpreted as a command it will be executed at word-time 02.

Notice that the dummy command has a prefix of u. Why? (Hint: will this make the command immediate or deferred? What will be the effect of this on the binary number generated by PPR?)

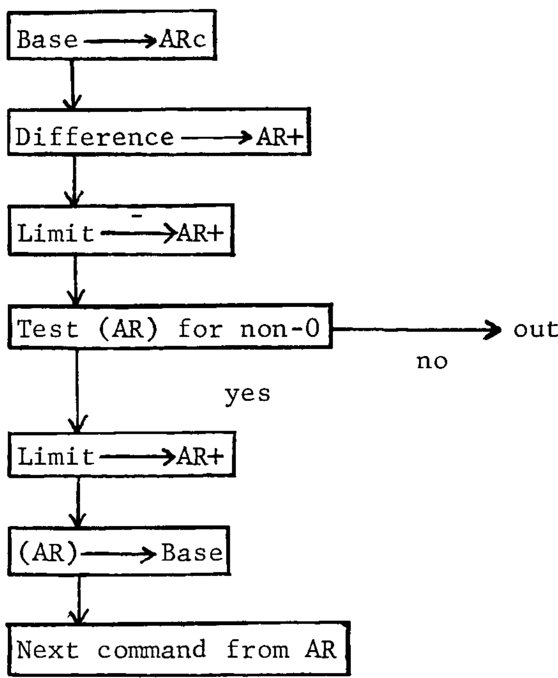
Rather than actually modify commands as they appear in a long line (which would necessarily entail great time delays, since the command would have to be picked up from a word in a long line, and then, in its modified form, be restored into the same word-position in the same long line), we would like to operate the modified command, in each case, out of AR, which is always available. Now the problem arises, if step I places a number in AR, how can we then modify a command there and execute it from there without destroying the number which was originally placed there by step I? The answer is that we will have to store the number called for by step I in a short line, then operate step II, with a destination of ARc, rather than AR+. Then we will place a command in our program which calls for the original number from its short-line temporary storage location. The destination of this command will be AR+.

Always be careful, when executing commands from AR, that you don't destroy valuable data already residing in AR.

INDEXING

"Indexing" is probably the easiest and most convenient way of modifying and operating a command out of AR. It involves a "Base" command, which is modified by a "Difference" (dummy command), and the result is restored into the Base, so that, on the next pass, the new base will again be modified, and so on. Usually there will be a "Limit" associated with such

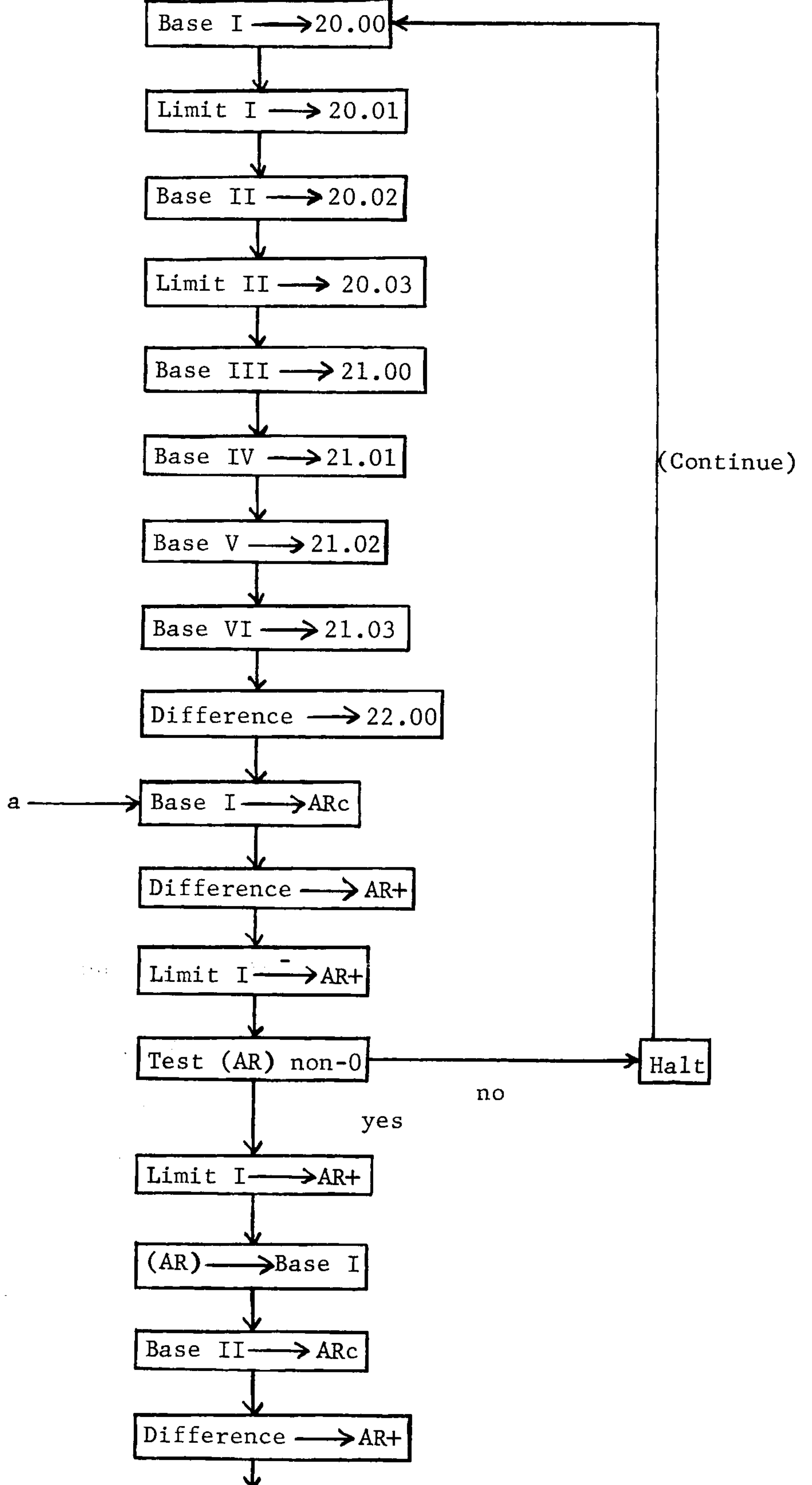
an index. If there is a limit, the modified version of the base will be checked against it each time, in order to determine whether or not the limit has been reached. All of this modification and checking will be done in AR, and the final contents of AR will then be operated as a command (unless the limit has been reached, of course). The sequence of steps might be:

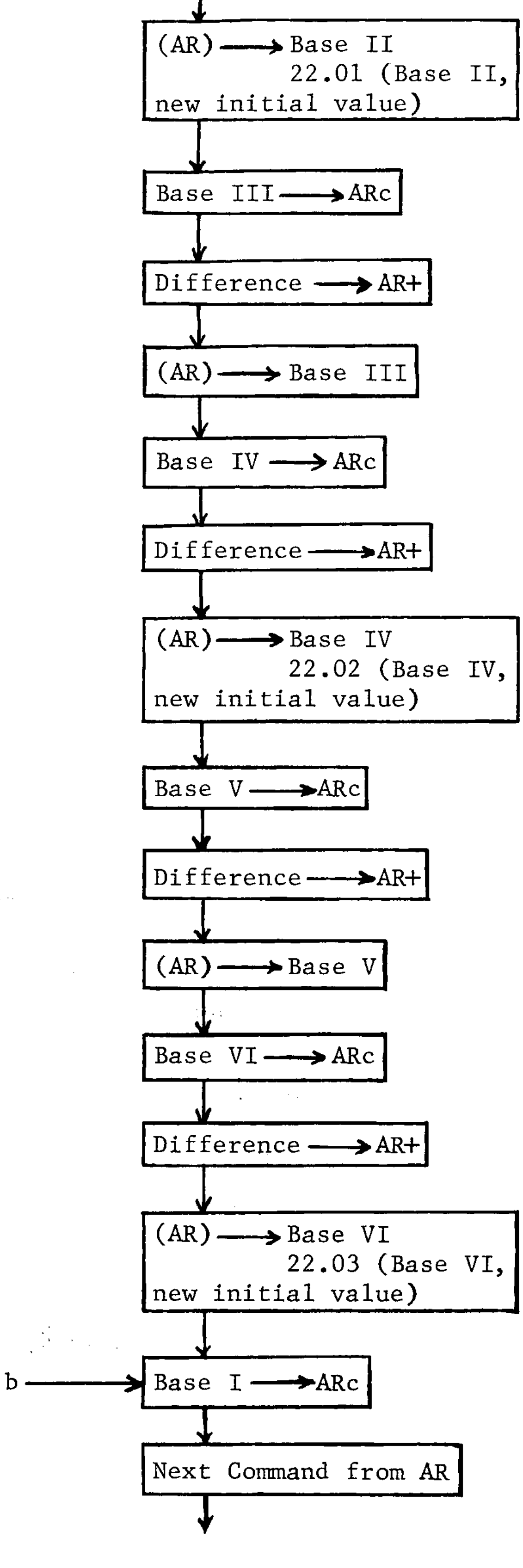


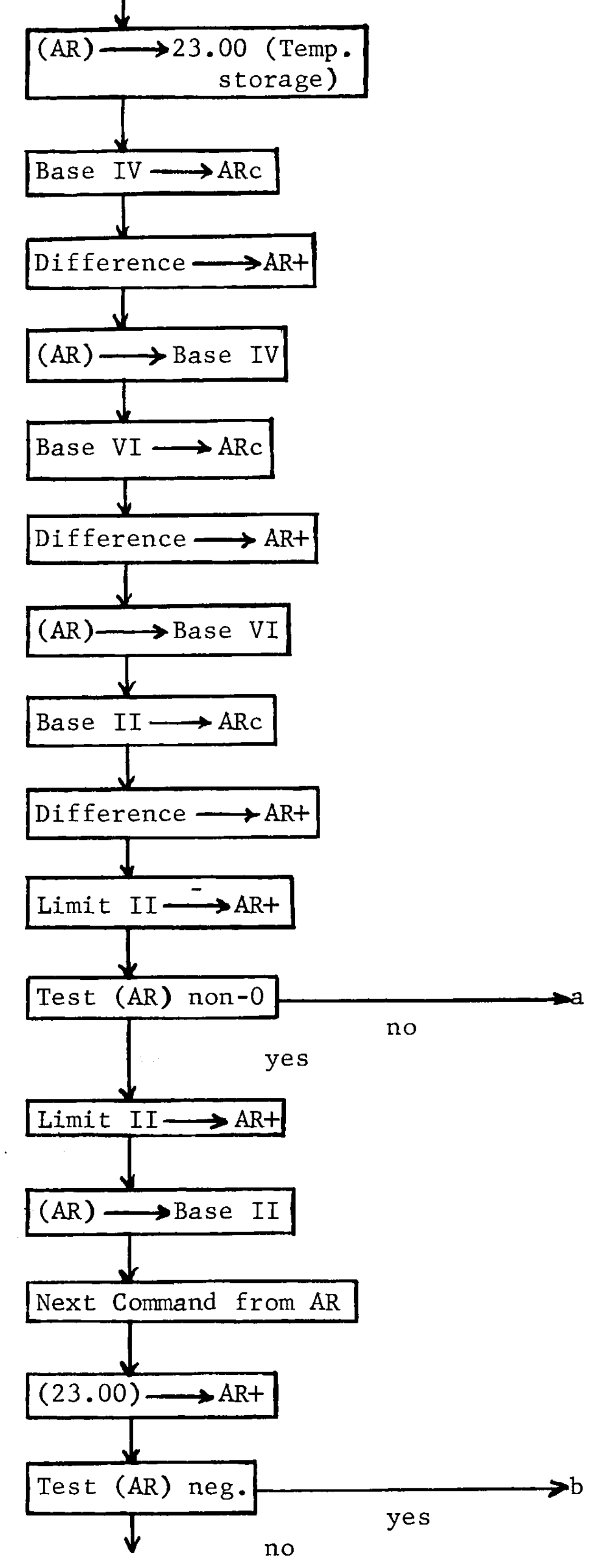
Such an index for step II might be:

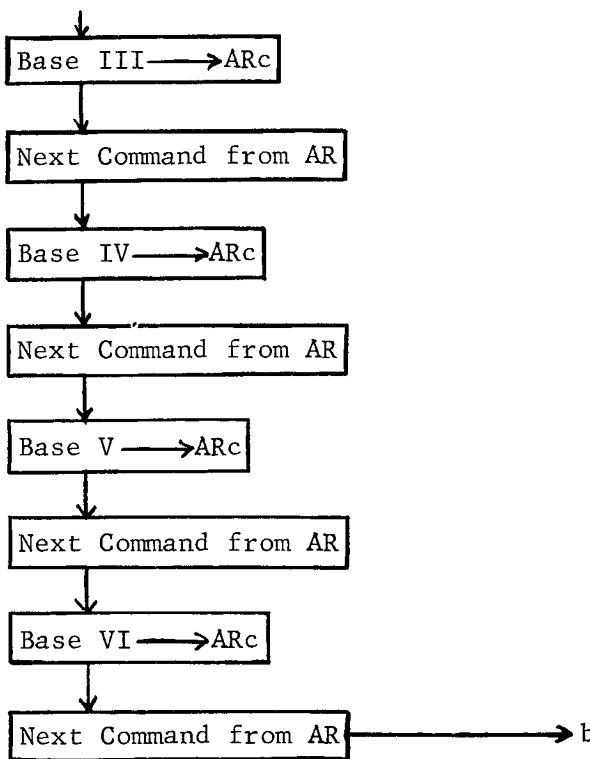
Base....................		00	50	3	19	28
Difference.............	u	01	00	0	00	00
Limit...................		50	50	3	19	28.

These could all be stored in a short line, and thus be available without undue delay during the modification process. Notice, in the preceding flow diagram, modification takes place prior to execution of the command, not after. If this is the case, and if we want to use the same steps for all passes through the loop, including the first, the Base must start out with a T number one lower than the first word-time at which we want to execute the command. Notice above that the Base has a T = 00, even though step II should initially contain T = 01. The previous flow diagrams for this problem must now be revised slightly. Notice in the revision that not all indices have been assigned a limit, because we know that some bases can never be increased too far, due to the fact that their modification is dependant upon modification of other bases, where limits have been imposed.


Base I → 20.00
Limit I → 20.01
Base II → 20.02
Limit II → 20.03
Base III → 21.00
Base IV → 21.01
Base V → 21.02
Base VI → 21.03
Difference → 22.00
a
Base I → ARc
Difference → AR+
Limit I –→ AR+
Test (AR) non-0
no
yes
Halt
(Continue)
Limit I → AR+
(AR) → Base I
Base II → ARc
Difference → AR+


(AR) ⟶ Base II
22.01 (Base II,
new initial value)
Base III ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base III
Base IV ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base IV
22.02 (Base IV,
new initial value)
Base V ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base V
Base VI ⟶ ARc
Difference ⟶ AR+
(AR) ⟶ Base VI
22.03 (Base VI,
new initial value)
b
Base I ⟶ ARc
Next Command from AR


(AR) → 23.00 (Temp. storage)
Base IV → ARc
Difference → AR+
(AR) → Base IV
Base VI → ARc
Difference → AR+
(AR) → Base VI
Base II → ARc
Difference → AR+
Limit II → AR+
Test (AR) non-0
a
no
yes
Limit II → AR+
(AR) → Base II
Next Command from AR
(23.00) → AR+
Test (AR) neg.
b
yes
no



The following coding sheets contain the individual commands in this program.

Notice that, in the Bases, T = w7 (127), and the command, so written, has deliberately been made immediate. The eight most significant bits in such a command will be:

(1) 01111111...

All of the Bases are initially modified, prior to being executed, through the addition of a Difference, whose eight most significant bits are:

(2) 00000001...

When (2) is added to (1), the result will be:

(3) 10000000...,

which will be a deferred command with T = 00. This is what is initially desired. Bases II, IV, and VI are further modified by the Difference, so that T = 01 prior to their execution for the first time.

This program, as written, requires slightly less than three minutes to sort numbers which are already in their proper order, and it requires eight minutes to sort numbers in the worst possible arrangement

initially. It is not by any means, a "good" program, because it is inefficient. Notice that it will exchange numbers which are equal, and it will "sort" fifty numbers which are already in proper order.

It was written in this manner to demonstrate the use of indexing in as straight-forward, and uncomplicated, a way as possible.

The "Notes" column on each coding sheet has been left blank, for you to fill-in, as a review.

Bendix Computer

Los Angeles 45, California

Page 1 of 3

G-15D

Prepared by ____________

Date: ____________

PROGRAM PROBLEM: 50-Word Sort

Line 00

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
00		01	04	0	00	00		
04	u	09	09	0	00	20		
05	[	49	49	0	19	28	]	Limit I
06	[u	w7	51	3	19	28	]	Base II
07	[	50	51	3	19	28	]	Limit II
08	[u	w7	49	0	19	28	]	Base I
09	u	14	14	0	00	21		
10	[u	w7	60	0	24	19	]	Base V
11	[u	w7	u4	0	25	19	]	Base VI
12	[u	w7	52	0	19	28	]	Base III
13	[u	w7	58	0	19	28	]	Base IV
14	u	19	19	0	00	22		
15								Base VI
16	[u	01	00	0	00	00	]	Difference
17								Base II
18								Base IV
19		20	21	0	20	28		a
21		24	25	0	22	29		
25		29	30	3	20	29		
30		31	32	0	28	27		
32		34	00	0	16	31		
33		37	38	0	20	29		
38		40	41	0	28	20		
41		42	43	0	20	28		
43		44	45	0	22	29		
45		46	47	0	28	20		

FORM BCD 108-5.0

Bendix Computer

Los Angeles 45, California

Page 2 of 3

G-15D

Prepared by __________

Date: __________

PROGRAM PROBLEM: 50-Word Sort

Line 00

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	

L	P	T or L_k	N	C	S	D	BP	NOTES
47		49	50	0	28	22		
50		52	53	0	21	28		
53		56	57	0	22	29		
57		60	61	0	28	21		
61		65	66	0	21	28		
66		68	70	0	22	29		
70		73	74	0	28	21		
74		78	79	0	28	22		
79		82	83	0	21	28		
83		84	85	0	22	29		
85		86	87	0	28	21		
87		91	92	0	21	28		
92		96	97	0	22	29		
97		99	u0	0	28	21		
u0		u3	u4	0	28	22		
u4		00	02	0	20	28		b
02		04	u6	0	31	31		
[u6]		[xx	49	0	19	28]		
49		52	54	0	28	23		
54		58	59	0	22	28		
59		60	62	0	22	29		
62		66	67	0	28	22		
67		71	72	0	22	28		
72		76	77	0	22	29		
77		79	80	0	28	22		
80		81	82	0	22	28		

FORM BCD 108-5.0

Bendix Computer Los Angeles 45, California

Page 3 of 3

G-15D

Prepared by ____________

Date: ____________

PROGRAM PROBLEM: 50-Word Sort

Line 00

0 1 2 3	L	P	T or L_k	N	C	S	D	BP	NOTES
4 5 6 7	82		84	86	0	22	29		
8 9 10 11	86		87	88	3	20	29		
12 13 14 15	88		89	19	0	28	27		
16 17 18 19	20		23	24	0	20	29		
20 21 22 23	24		25	02	0	28	22		
24 25 26 27	[u6]		[xx	51	3	19	28]		
28 29 30 31	51		52	55	0	23	29		
32 33 34 35	55		57	u3	0	22	31		
36 37 38 39	u3		u4	02	0	21	28		
40 41 42 43	[u6]		[xx	52	0	19	28]		
44 45 46 47	52		54	56	4	28	25		
48 49 50 51	56		58	02	0	22	28		
52 53 54 55	[u6]		[xx	58	0	19	28]		
56 57 58 59	58		60	63	4	28	24		
60 61 62 63	63		66	02	0	21	28		
64 65 66 67	[u6]		[xx	60	0	24	19]		
68 69 70 71	60		63	02	0	22	28		
72 73 74 75	u6		xx	u4	0	25	19		
76 77 78 79									
80 81 82 83									
84 85 86 87									
88 89 90 91									
92 93 94 95									
96 97 98 99									
u0 u1 u2 u3									
u4 u5 u6									

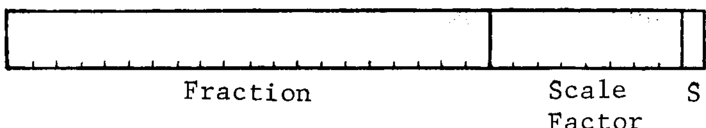
FORM BCD 108-5.0

FLOATING-POINT OPERATION

When the range of numbers which is anticipated during a calculation is either very large or unpredictable, it may become extremely difficult to work with fixed-point data. In such situations, floating-point arithmetic is usually used.

In floating-point operation a data word is divided into two major portions, similar in function to the characteristic and the mantissa of a logarithm. One portion contains a fraction; the other, an integral power of 2 by which the fraction is to be multiplied. The sign-bit of the data word is associated with the fraction, as its algebraic sign. Any arbitrary decision as to how many bits of a word will be used to express the fraction and how many will be reserved to express the accompanying exponent of the base of 2 is permissible.

The Bendix programmers have generally adopted the following rule. The fraction part of the data word is contained in bit positions T10 through T29. The sign of the fraction is contained in bit T1 of the data word and the scale factor is contained in bits T2 through T9.



As T1 of the data word indicates the algebraic sign of the fraction and, since signed exponents are desirable, the scale factor is formed by adding 128 to the power of 2. Thus, the range of the scale factor is $0 \leq$ scale factor ≤ 255, while the range of the power of 2 is $-128 \leq$ power of $2 \leq 127$. Therefore, the larger the scale factor, the smaller the number that is represented in the fraction or mantissa of the data word.

In floating-point operation it is often desirable to keep as many significant bits as possible in the mantissa of a data word and to use the scale factor to express all leading zeroes. A special command is available in the G-15 that will allow the programmer to keep all significant bits of a data word and will also keep a count of how many leading zeroes were dropped. This command is the "normalize" command. It's normal form is:

L 54 N 0 27 31.

The "normalize" command operates in the following manner:

1. $MQ_{0,1}$ and AR are the only registers involved in the execution of this instruction.

2. $MQ_{0,1}$ will shift left until a 1 bit appears in T29 of MQ_1. For every leading zero that is lost from MQ_1 a 1 is added to AR in

the 2^{-28} position. A zero will be inserted in T1 of MQ_0 after each shift.

3. As in a shift command, the location for the "normalize" command must be odd.

Unlike some other machines, floating-point arithmetic operations are not automatic in the G-15. It is up to the programmer to keep track of the scale factors and fractions that he is working with. Thus multiplication and division become rather simple programming problems in floating-point because all the programmer must do is separate the fractional portions of the numbers from the scale factors, do the multiplication or division with the fractions, either add or subtract the scale factors to get the new scale factor of the product or quotient and then combine the product or quotient with the new scale factor for storage.

Addition and subtraction are now more difficult because the programmer must compare the scale factors of the numbers to be added or subtracted, and if they are unequal, must shift one of the numbers until the scale factors do become equal. He must keep track of the number of shifts to get the new scale factor of the number. A shift command is available to help the programmer keep a count of the number of shifts that have been made. This is a special command and its normal form is:

L 54 N 0 26 31.

This command will operate on ID and MQ in the same manner as the shift command mentioned earlier with the following exception, for each bit-position shift in ID and MQ a one will be added to AR scaled 2^{-28}. If at any time during the shift the one added to AR causes an end-around-carry in AR, the shift will terminate and no more bits will be shifted in ID and MQ. Therefore, by using this command the programmer can either count the shifts in AR or use AR, loaded with the complement of the number of bit-positions that he desires shifted, to control the number of shifts. The location of this command must be odd so that execution will start at an even word-time.

The two extract commands previously discussed enable the programmer to separate the fraction and the scale factor so that he can operate on them separately. After the operations have been performed on both the scale factors and the fraction, the programmer will want to combine the new fractions and scale factors for storage. Another extract command is available for this purpose. Its normal form is:

L T N 0 27 D.

This extract command operates in the following manner:

1. Where there are one bits in the mask at word-time T in line 20, the corresponding bits in line 21 are extracted to word-time T of the destination.

2. Where there are zero bits in the mask at word-time T in line 20, the corresponding bits in AR are extracted to word-time T of the destination.

Thus, by use of this extract command the programmer can unite his new scale factor with the new fraction for storage.

From the preceding paragraphs, it can be seen that floating-point operation in the G-15, although it presents a somewhat more difficult programming effort, can operate with very large or very small numbers that fixed-point operation could not handle.

MISCELLANEOUS TOPICS TO BE COVERED BEFORE CLOSING

So far, in discussing outputs, we have mentioned the possibility of either punching or typing the contents of line 19. If it is desired to get both a tape and a typed copy of the line's contents, two separate outputs would have to be called for.

On the base of the typewriter, as shown on page 130, there is a punch switch. If this switch is on when a type-out of line 19's contents is called for, the characters of output, as well as activating the typewriter, will also activate the punch, and the two outputs will proceed simultaneously as the result of one command (Type line 19). Of course the speed of the punch will be slowed down to the speed of the typewriter, which is considerably slower than the normal speed of the punch, when used alone. This punch switch is merely a physical connection enabling the pulses which reach the typewriter to also reach the punch.

Of course, the punch switch must be on prior to execution of the "type line 19" command. And so the question arises, "How can you be sure the punch switch has been manually turned on?" A test command is available which tests for this condition. It is a special command, D = 31, S = 17, C = 1. If the switch is on, the next command will be taken from N + 1; if the switch is off, the next command will be taken from N.

Normally, you would use this test prior to calling for a type-out of line 19's contents, if you want to be sure that a tape will also be punched. The "type line 19" command would be available only after the test was answered affirmatively, the next command coming from N + 1.

If the answer is "no", you would normally want to repeat the test until the switch is turned on. In such a case, it would be desirable to call the operator's attention to the fact that he is to throw the punch switch on. It is reasonable to assume that the operator will not be aware of this desire of yours; he might not even be at the computer (coffee-break, of course). What then?

There is a special command available (D = 31, S = 17, C = 0) which rings a bell inside the computer once each time it is executed. At N you could give this command, and then go back to the test again. The bell would thus be rung once each time the test is executed and answered negatively. Presumably this continuous bell-ringing would cause somebody to come to

the computer. There would be the operating instructions for your program, containing one all-important sentence: "If the bell continuously rings, turn on the punch switch."

The ringing of the bell requires a physical action on the part of the computer: the movement of a solenoid, striking the rim of the bell. It is a safe bet that, whenever physical action is involved, timing problems occur. In this case, it is safe to allow one complete drum-cycle execution time for the command. This will be sufficient to cause the solenoid to ring the bell. Since D = 31 in the "ring bell" command, PPR will make the command immediate. Set T (the flag) equal to L + 1, allowing a complete drum cycle of execution.

Solenoids require a recovery time, and, if the solenoid which rings the bell is not allowed to recover after each ring, it will merely vibrate against the bell, causing a buzz, rather than a series of individual rings. Recovery time for this solenoid is three drum cycles. Therefore, three drum cycles must elapse between executions of the "ring bell" command. These can be achieved through purposeful "bad" coding of commands, requiring "maximum access-time". For instance, the following command will waste two drum cycles:

L	P	T	N	C	S	D
51		51	50	0	00	00
50		..	..	.	..	..

Notice that both the "ring bell" and "test punch switch" commands have a special S code of 17.

ring bell:	T	N	0	17	31
test punch switch on:	T	N	1	17	31.

The punch switch test will also ring the bell, if a full drum cycle of operation is allowed (T = L + 1). Of course, recovery time for the solenoid is still necessary.

Recovery time is also necessary in one other case already discussed in this text. When type-in is called for, the stop code of the input is supplied by striking the "s" key. There is a physical contact involved in this action, and that contact will remain closed for approximately 1/10 second (3 drum cycles). If another input or output is initiated prior to the opening of that contact, the stop code pulse will still be present, and that input or output will immediately stop.

Therefore, rule: after completion of typewriter input (ready test is successfully met), allow three drum cycles to elapse before initiating any other input or output.

Concerning punched tape output, one point should be made quite clear. The reloading of the format will cause a reload code to be punched on tape. If the tape being punched is later to be read into the computer

(interim storage) you must be sure that it is originally punched under control of an output format which calls for four full words prior to the reload.

The number track, mentioned previously, is a timing channel physically located on the surface of the drum. It occupies a long line similar to the long lines already discussed, and this long line recirculates once per drum cycle in the same manner as all other long lines. There is no way to program the loading of this channel: it is loaded automatically when the computer is turned on. Two blocks of punched tape will automatically be read: the computer will automatically load the number track with the information from the first; the second should be a loader program designed to read in a test routine, in the normal manner. The contents of this block of tape will occupy line 19. Turn-on procedure, including use of test routines, is fully discussed in the Operating Manual.

The function of the number track is to affix specific word-times to all words in memory. At each word-time, in the number track, the T and N portions of the word contain the number of the next word-time to come up under the read-heads. The computer compares T's and N's of commands being interpreted with the T and N available from the number track, and in this way is able to determine when to execute or read a command.

Page 206 contains a type-out of the number track. The words are typed in four-word groups, reading from left to right, one group per line. The first word typed out is u7, and the last is 00. Notice that in all words except u7, the I/D bit is set equal to 1.

Word u7 is unlike any of the others. You would expect its T and N to contain 00, but this is not the case. The counting of T and N is, of necessity, modulo 128 (there are seven bits for each). However, there are only 108 words per long line, and, therefore, only 108 word-times possible for either T or N. Word u7 in the number track contains 20 in each of these positions, so that, when this is added to the respective time counters, they will be cleared to 00 indicating that the next word-time will be 00. The meaning of other bits set in word u7 of the number track would require more engineering background than the reader is assumed to have at this point.

Notice, if you store a command in word u7 of a command line, and if you expect your program to read and interpret this command at word-time u7, it will be interpreted simultaneously with the jumping ahead of the counters by 20 word-times. Therefore, if you want to do this, the T and N portions of your command must equal the desired word-time plus 20 in each case. For example, if, at word-time u7, you desire to call for the transfer of word 10 from line 08 to line 09, and then you desire to take your next command at word-time 11, your command would be coded in the following manner:

L	P	T	N	C	S	D
u7		30	31	0	08	09
11		..	..	.	..	..

Number Track

-1414794	yv6v000	yu6u000	y969000
y868000	y767000	y666000	y565000
y464000	y363000	y262000	y161000
y060000	xz5z000	xy5y000	xx5x000
xw5w000	xv5v000	xu5u000	x959000
x858000	x757000	x656000	x555000
x454000	x353000	x252000	x151000
x050000	wz4z000	wy4y000	wx4x000
ww4w000	wv4v000	wu4u000	w949000
w848000	w747000	w646000	w545000
w444000	w343000	w242000	w141000
w040000	vz3z000	vy3y000	vx3x000
vw3w000	vv3v000	vu3u000	v939000
v838000	v737000	v636000	v535000
v434000	v333000	v232000	v131000
v030000	uz2z000	uy2y000	ux2x000
uw2w000	uv2v000	uu2u000	u929000
u828000	u727000	u626000	u525000
u424000	u323000	u222000	u121000
u020000	9z1z000	9y1y000	9x1x000
9w1w000	9v1v000	9u1u000	9919000
9818000	9717000	9616000	9515000
9414000	9313000	9212000	9111000
9010000	8z0z000	8y0y000	8x0x000
8w0w000	8v0v000	8u0u000	8909000
8808000	8707000	8606000	8505000
8404000	8303000	8202000	8101000

Powers of "2"

k = no. of pre-zeros

2^n	2^{-n}	k	n
1	1	0	0
2	.50000000	0	1
4	.25000000	0	2
8	.12500000	0	3
16	.06250000	0	4
32	.03125000	0	5
64	.01562500	0	6
128	.00781250	0	7
256	.k3906250	2	8
512	.k1953125	2	9
1024	.k9765625	3	10
2048	.k4882812	3	11
4096	.k2441406	3	12
8192	.k1220703	3	13
16384	.k6103516	4	14
32768	.k3051758	4	15
65536	.k1525879	4	16
131072	.k7629395	5	17
262144	.k3814697	5	18
524288	.k1907349	5	19
1048576	.k9536743	6	20
2097152	.k4768372	6	21
4194304	.k2384186	6	22
8388608	.k1192093	6	23
16777216	.k5960464	7	24
33554432	.k2980232	7	25
67108864	.k1490116	7	26
134217728	.k7450581	8	27
268435456	.k3725290	8	28
536870912	.k1862645	8	29

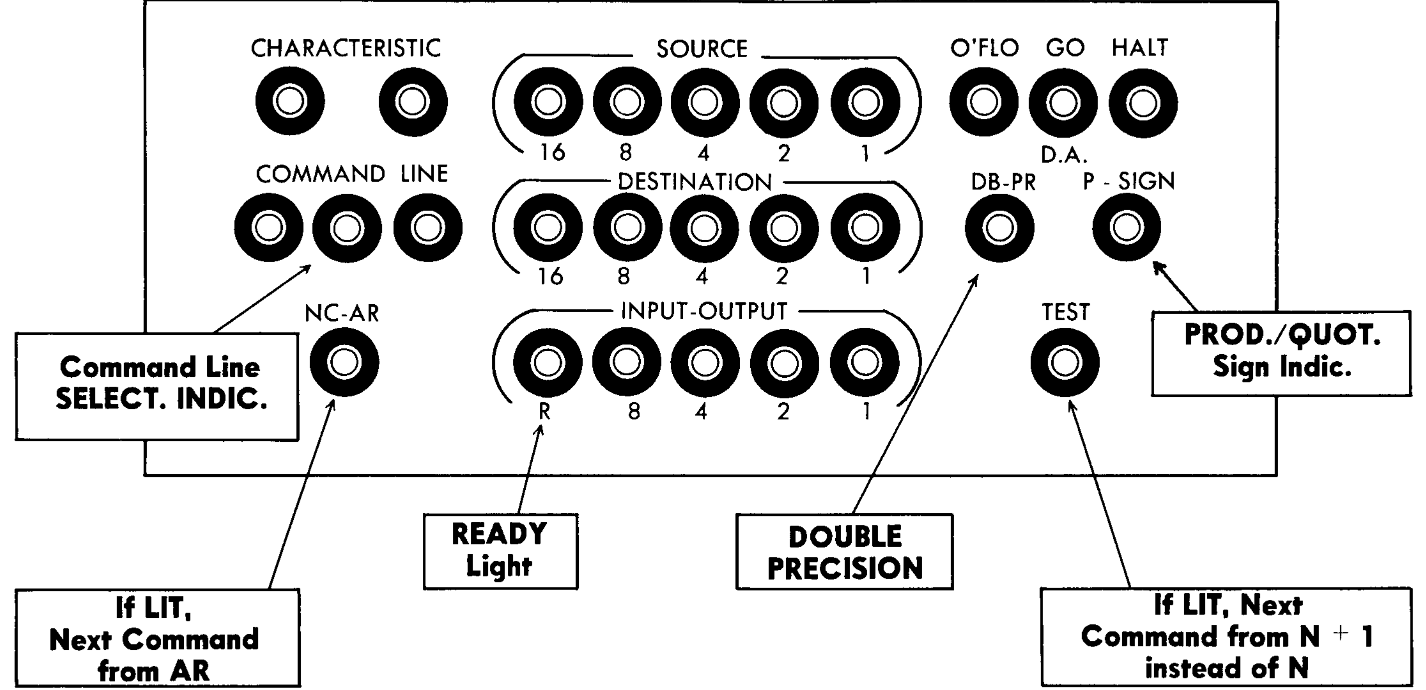
Hex Powers of "10"

k = no. of pre-zeros

10^n(Hex)	10^{-n}(Hex)	k	n
0000001	1	0	0
000000u	.199999u	0	1
0000064	.k28z5w29	1	2
00003y8	.k4189375	2	3
0002710	.k68xv8vv	3	4
00186u0	.ku7w5uw4	4	5
00z4240	.k10w6z7u	4	6
0989680	.klux7z2u	5	7
5z5y100	.k2uz31xw	6	8

Constants

π = 3.14159265
π^2 = 9.86960440
$\sqrt{\pi}$ = 1.77245385
e = 2.71828183
log e = 0.43429448
log 2 = 0.30103000
log = 0.49714987


CHARACTERISTIC
SOURCE
16 8 4 2 1
O'FLO GO HALT
D.A.
COMMAND LINE
DESTINATION
16 8 4 2 1
DB-PR P - SIGN
NC-AR
INPUT-OUTPUT
R 8 4 2 1
TEST
Command Line SELECT. INDIC.
PROD./QUOT. Sign Indic.
READY Light
DOUBLE PRECISION
If LIT, Next Command from AR
If LIT, Next Command from N + 1 instead of N

INDEX

Abbreviated Format (see Input/Output System)
Absolute Value (see Magnitude)
Access Time, 10, 150, 152, 159-161
Accumulator
    Double Precision (see PN Register)
    Single Precision (see AR Register)
"Add", 20, 103
"Add Magnitude", 22, 25, 103
Address, 7, 10-11, 15, 66
    of next command, 16, (see also Next Command)
Analysis (see Problem Analysis)
Arithmetic Operations, 20-27, 33-41, 202
AR Register, 11, 20

Bell (see Ring Bell)
Binary-Coded-Decimal (see Decimal Inputs)
Binary Point
    in machine (see Machine Point)
    true, 91
Binary Scaling (see Scaling)
Bits
    ordering of, 6
Blank Leader (see Leader)
Block of Punched Tape, 18, 138, 140
Block Operations, 25-26, 68-69, 163-165
"Bootstrap" (see Loader)
Break-Point Operation, 16, 141-142

"C" Codes, 13-15, 64-66
Characteristic (CH), 61-66
Check Sum, 162
Clear (see Erase)
"Clear & Add", 20, 103
"Clear & Add Magnitude", 22, 25, 103
"Clear & Subtract", 21, 62, 103
Coding Sheets, 152-158
Command
    binary form, 60-61
    decimal form, 13, 150-151, 162
    modification of, 122, 186-191, (see also Indexing)
    ordering of, 69
        next command from AR, 51, 122, 186-187, 191
    parts of with respect to computer operation, 13-17, 61-70, 141-142, 205
    restoration of (see "Housekeeping")
    special (see Special Command)

Command Line, 28, 69-70
    selection of, 134
Complement, 23-24, 63-64
Complementation (see Inverting Gates)
Compute Switch, 17, 130, 142, 167
Control Information (see Timing)
Copy, 13-14
Copy via AR, (see Transfer via AR)
Cycle, drum (see Drum Cycle)

Debugging, 141-142
Decimal Command (see Command)
Decimal Inputs, 31, 167-168
    conversion to binary, 28-30, 168-170
Decimal Scaling (see Scaling)
Decision-Making (see Test Commands)
Deferred Command, 15, 69, 160
Destination, 15, 66-68
Divide, 39-40, 76-84
    considered as a ratio, 81-82
    round-off (see Round-Off)
Double Precision, 11-12
Drum Cycle, 8
Drum Revolution, 6
Drum Memory, 5-13

Enable Actions, 17, 133-134, 142
Enable Switch, 17, 130, 133, 142
End-Around-Carry, 106-108
Erase, 8-9
Erase Head, 8-9
Exchange AR with Memory, 13-15, 61-62, 163-165
    D = two-word register (see Two-Word Registers)
Extract, 44-47, 177-180, 202-203

Fixed-Point Operation (see Scaling)
Flag (see Immediate Command)
Flip-Flop
    sign, 131
Floating-Point Operation, 201-203, (see also Scaling, Floating-Point)
Flow Diagrams
    description of, 4-5
Format (see Input/Output System)

Halt Command, 56, 111
"Housekeeping", 190

ID Register (see Two-Word Registers)
Immediate Command, 16, 18-19, 68-69, 160
Immediate-Deferred Bit, 68
Indexing, 50, 191-196
Input/Output System
    commands, normal, 18, 142-143
    enable actions (see Enable Actions)
    normal inputs, 17, 128-133
        punched tape, 18, 140
        typewriter, 17-19, 30-31, 128-134, 204
            drawing of, 130
    normal outputs, 51-52
        abbreviated format, 162
        format, 52-55, 135-136, 138, 204-205
        punched tape, 54, 134-139, 145-147, 162-163, 203-205
        typewriter, 140-141, 203-204
    ready (see Test Ready)
    requirements, 127
    simultaneous with computation, 19, 144
    stop code, 132, 136, 140
        recovery time for S key, 204
Introduction, 1-3
Inverting Gates, 13, 61, 67, 106
IP Flip-Flop (see Two-Word Registers, use of, in multiplication)

Leader, 145-147
Loader Program, 60, 147-149
Logical Addition, 27, 51, 143
Logical Operations, 42-47
Long Lines, 6-7
Loop
    simple, 47-49, 145
    through command modification and indexing, 50-51, 186-196

Machine Point, 89
Magnitude
    of a number (see following: "Clear & Add", "Add", and "Subtract")
Mark & Transfer, 28, 29, 114-121
Mask (see Extract)
Maximum Access (see Access Time)
Memory (see Drum Memory)
Method of Solution, 3-4
Millisecond, 8
Minimum Access (see Access Time)
MQ Register (see Two-Word Register)
Multiply, 35-38, 70-75
    Round-Off (see Round-Off)

Neons, front panel, 18, 132, 208
Next Command (see Commands, ordering of)
Next Command from AR, 51, 122, 186-187, 191
"Normalize", 201-202
Notes, 16
Number
    BCD (see Decimal Inputs)
    conversions, 27, 28-30, 55, 168-170, 175-180, 182-186
    machine form, 89
    table of powers, 207
Number Track, 12-13, 26-27, 205-206

Operand, 15, 66
Operation Code
    special (see Special Operations)
Output (see Input/Output System)
Overflow (see also Test)
    definition, 33
    indicator, to turn off, 34, 111-112
    resulting from divide, 80
    temporary, 79

Photo Reader, 18
PPR (see Program Preparation Routine)
PN Register (see Two-Word Register)
Precession, 163-165
Prefix, 16, 18, 151
"Princeton" Round-Off (see Round-Off)
Problem Analysis, 3
Problem Method (see Method of Solution)
Program Preparation Routine, 13, 18-20, 59-60, 150-151, 160-163, 165-167
Psuedo-Commands for PPR, 151, 162, 165-167
Punched Tape (see Input/Output System)
Punch Switch, 130, 203-204

Range of Values
    associated with scaling, 97-100, 171
Read Heads, 6, 8-9
Ready (see Test Ready)
Recirculating Memory, 8-9, 67-68
Recovery Time, 204
Relative Timing Number, 36, 39
Rescaling, 91-92
Return Command, 29, 114-121, 142
Return Line, 29, 114
Revolution, Drum (see Drum Revolution)
Ring Bell, 203-204

Round-Off
    after division, 82
    of binary number prior to conversion to BCD, 184

Scaling, Fixed Point
    binary, 89-97
    decimal, 32-33, 170-171
        in BCD output, 175, 180
Scaling, Floating Point, 201-202
Selector, Source & Destination, 67
Self-Destroying Loader (see Loader)
Set Ready, 146-147
Shift, 42-44, 93-94, 201-202
Short Lines, 9-12
Sign Flip-Flop (see Flip-Flop)
Sign Time (see Bits, ordering of)
"Single-Cycle", 142
Single-Double Precision Bit, 61, 64-66
Solenoid, 204
Special Commands, 16-17, 18, 70
    table of, 56-59
Sorting, 186-197
Source, 15, 66-68
Stop Code (see Input/Output System)
"Store", 103
"Store Magnitude", 22
"Subtract", 14-15, 20, 62, 103
"Subtract Magnitude", 110
Subroutines, 27, 113, 118
    square root, 40

"T" Numbers
    pertaining to bits, 6
Temporary Overflow (see Overflow)
Test Commands, 19, 105-109
    non-zero, 41, 109
    overflow, 33-34, 106-108
    punch switch on, 109, 203-204
        and ring bell, 204
    ready, 19-20, 108, 143
    sign of AR negative, 41, 108
Timing (see Machine Time)
    Problems where physical action is required, 204
Timing and Control Information, 12, (see also Number Track)
    in immediate commands (see Immediate Commands)
    relative timing number (see Relative Timing Number)
"TO" Pulse, 12
Transfer Control (see Mark & Transfer)

Transfer via AR
    divide, 82, 84
    multiply, 82-23
Two-Word Registers, 10-12
    double precision accumulator, PN register, 11, 21
    exchange of AR with memory, D = two-word register, 13-15
    summary of rules, 37-38
    use of, in division, 39-40, 76-84
    use of, in multiplication, 35-38, 70-75

Word-Time, 6-7, 8, (see also Number Track)
    as part of address, 15, 66
"Working Memory", 7
Write Heads, 6, 8-9

Offices:

BOSTON 16

114 Waltham Street
Lexington 73, Massachusetts
862-7976

CHICAGO 11

919 N. Michigan Avenue
MIchigan 2-6692

CLEVELAND 13

55 Public Square
CHerry 1-7789

DALLAS 1

2626 Mockingbird Lane
FLeetwood 1-9951

DAYTON 2

1900 Hulman Bldg.
BAldwin 6-2341, Area code 513

DETROIT 37

12950 West Eight Mile Road
JOrdan 6-8789

HUNTSVILLE

Holiday Office Center
Memorial Parkway
South 539-8471

LOS ANGELES

291 S. La Cienega Boulevard
Beverly Hills, California
OLeander 5-9610

NEW YORK 17

205 East 42nd Street
Room 1205
ORegon 9-6990

PHILADELPHIA

723 Street Road
Southampton, Pa.
ELmwod 5-0600, Area code 215

WASHINGTON 6, D. C.

1000 Connecticut Avenue, N.W.
STerling 3-0311

CANADA

Computing Devices of Canada

P.O. Box 508
Ottawa 4, Ontario, Canada
TAlbot 8-2711

OTHER COUNTRIES

Bendix International Division

205 E. 42nd Street
New York 17, New York
MUrray Hill 3-1100

Bendix Computer Division

LOS ANGELES 45, CALIFORNIA