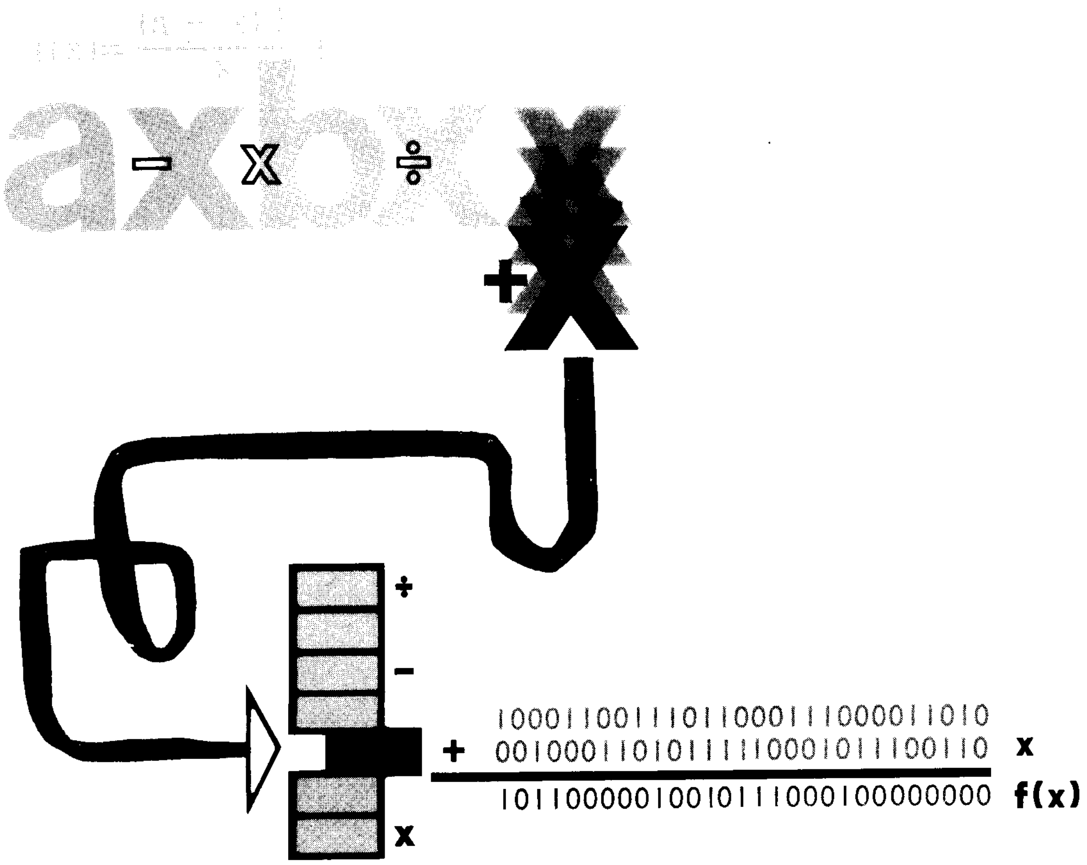
THE Bendix CORPORATION · COMPUTER DIVISION

5630 Arbor Vitae Street, Los Angeles 45, California

PROGRAMMING FOR THE G-15


ax bx
+
x
−
x
100011001110110001110000011010
+ 001000110101111100010111001 10 x
1011000001001011100010000 0000 f(x)

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.

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

TABLE OF CONTENTS (Cont'd.)

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

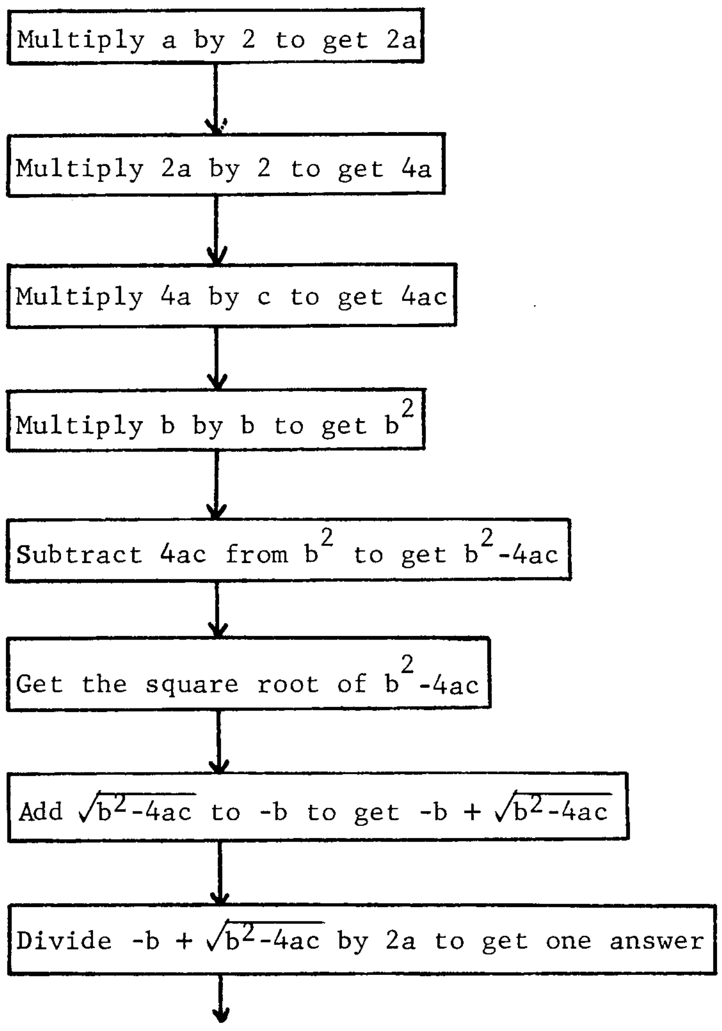
When the programmer has adequately defined the problem and the data available, he must choose a method of solution. Usually there will be as many of these as there are programmers. Whether or not one method is better than another depends on several factors. Is the time consumed in generating solutions 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:


Multiply a by 2 to get 2a
Multiply 2a by 2 to get 4a
Multiply 4a by c to get 4ac
Multiply b by b to get b^2
Subtract 4ac from b^2 to get b^2-4ac
Get the square root of b^2-4ac
Add $\sqrt{b^2-4ac}$ to -b to get -b + $\sqrt{b^2-4ac}$
Divide -b + $\sqrt{b^2-4ac}$ by 2a to get one answer

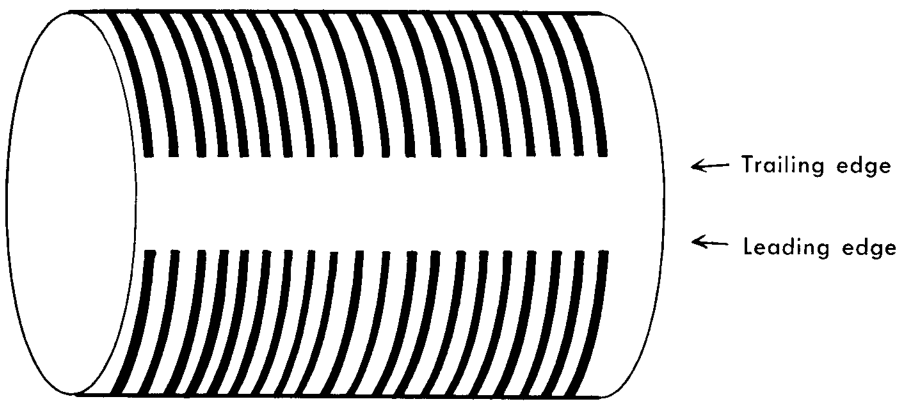
$$\begin{tikzcd} {} \arrow[d] \\ \boxed{\text{Subtract } \sqrt{b^2-4ac} \text{ from } -b \text{ to get } -b - \sqrt{b^2-4ac}} \arrow[d] \\ \boxed{\text{Divide } -b - \sqrt{b^2-4ac} \text{ by } 2a \text{ to get the other answer}} \arrow[d] \\ \boxed{\text{Stop}} \end{tikzcd}$$

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.

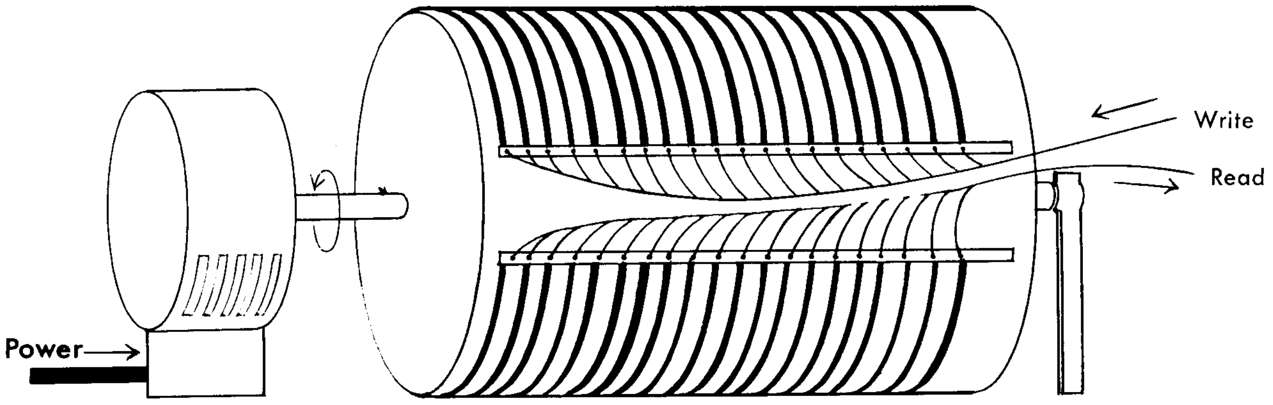
Before we go any further into the development of the program we must explore the operation of the computer, with an eye toward making up the proper commands to achieve a desired result.

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 revolve. 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

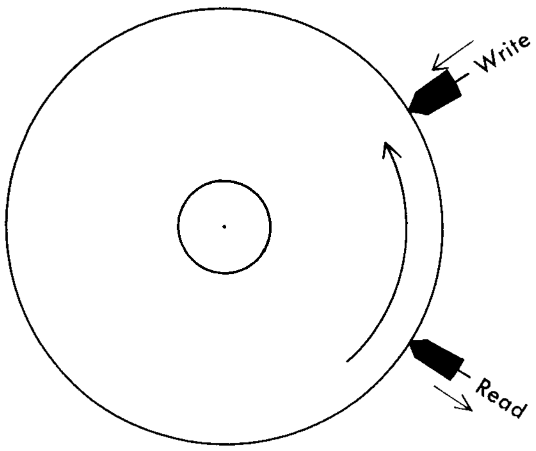
Part II: a designation of the location of the word within that line.

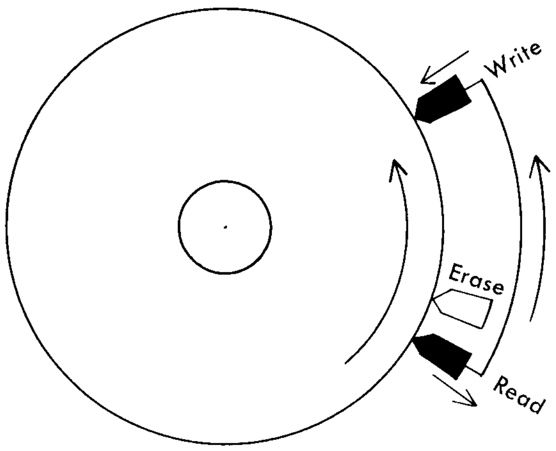
The 20 long lines are numbered 00 through 19, and the words in each one, starting at the leading edge, are numbered 00 through 99, u0 (100) through u7 (107). (The substitution of the hex digit u for the decimal digits 10 is for the purpose of holding the word-number to two digits.) The addresses of all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

$$\begin{tikzcd}[row sep=small]
\underline{\text{line : word}} \\
00 : 00 \arrow[d] \\
00 : \text{u}7 \\
01 : 00 \arrow[d] \\
01 : \text{u}7 \arrow[d] \\
19 : 00 \arrow[d] \\
19 : \text{u}7
\end{tikzcd}$$

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



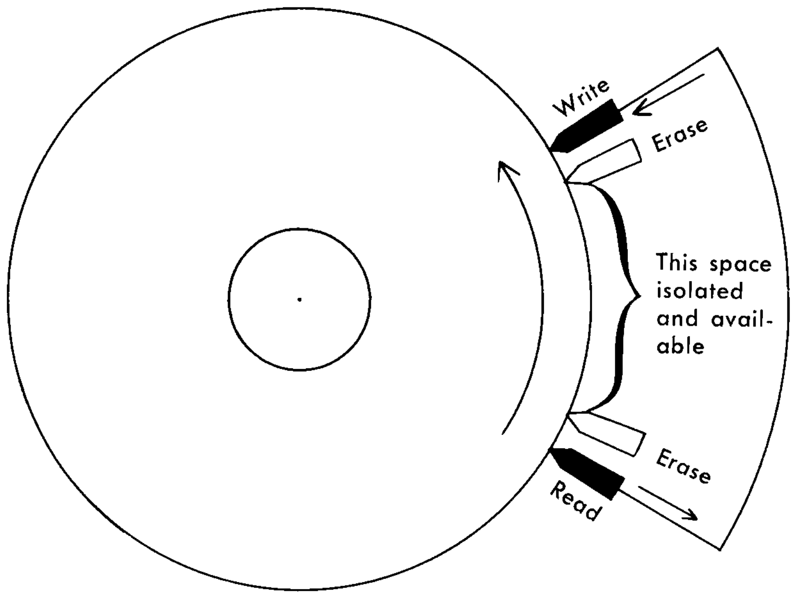


read-head; this is true for each long line in memory. The read-head feeds the write-head with electrical pulses mirroring the contents of the bit-positions as they pass under the read-head. Each bit in each line, as it is read, is moved ahead, along the circumference of the drum. Of course the trailing 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".

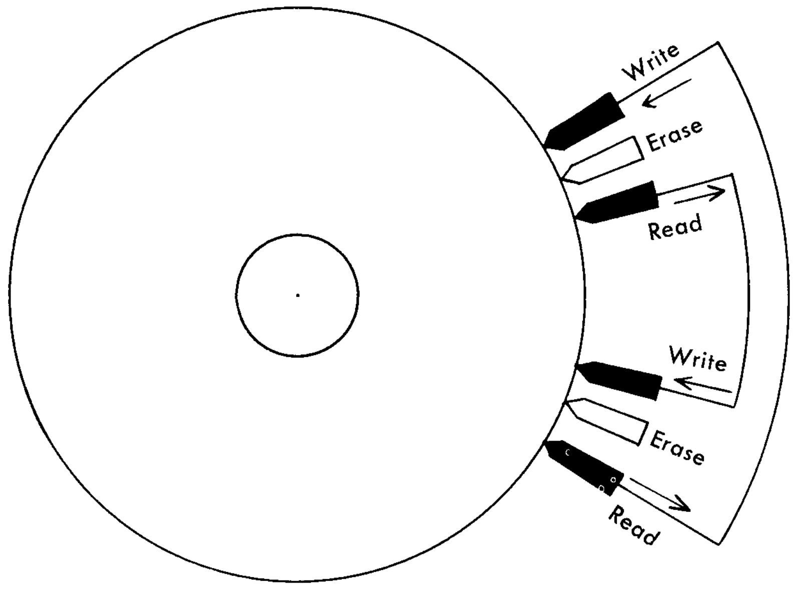
The spacing between the read- and the write-heads is such that there are approximately 2070 drum cycles per minute, or 34.5 per second. Each drum cycle requires .029 seconds. Each word is read or written in .00027 seconds. We refer to $\frac{1}{1000}$th's of a second as "milliseconds". Therefore,

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:

$$\begin{tikzcd}
20.00 \arrow[d] \\
20.03 \\
21.00 \arrow[d] \\
21.03 \\
22.00 \arrow[d] \\
22.03 \\
23.00 \arrow[d] \\
23.03
\end{tikzcd}$$

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.

L	P	T	N	C	S	D	BP	NOTES

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	NOTES
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	NOTES
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	NOTES
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.

```
   1000110110010001000011111100|0
   0111001001101110111100000100|1
   ----------------------------+-
 1 0000000000000000000000000000|1
 └────────────────────────────>|1
   ----------------------------+-
   0000000000000000000000000000|0
```

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.

```
   1000110110010001000011111100|1
   1000110110010001000011111100|0
   ----------------------------+-
 1 0001101100100010000111111000|1
 └────────────────────────────>|1
   ----------------------------+-
   0001101100100010000111111000|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		$10.02\text{-}03 \xrightarrow{+} PN_{0,1}$
04		06	08	6	10	30		$\lvert 10.06\text{-}07 \rvert \longrightarrow PN_{0,1+}$
08		10	N	5	26	10		$PN_{0,1} \xrightarrow{+} 10.10\text{-}11$

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		$\lvert 10.04\text{-}05 \rvert \longrightarrow PN_{0,1+}$
06		08	10	6	10	30		$\lvert 10.08\text{-}09 \rvert \longrightarrow PN_{0,1+}$
10		12	N	5	26	10		$PN_{0,1} \xrightarrow{+} 10.12\text{-}13$

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		$10.01\text{-}08 \longrightarrow 11.01\text{-}08$

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		$10.02\text{-}00 \xrightarrow{+} AR_+$
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		$AR \longrightarrow 18.05\text{-}04$
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) $\longrightarrow$ ID_1
06		07	08	0	00	28		00.07 $\longrightarrow$ 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) $\longrightarrow$ AR
53		59	N	0	28	00		x = (AR) $\longrightarrow$ 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.

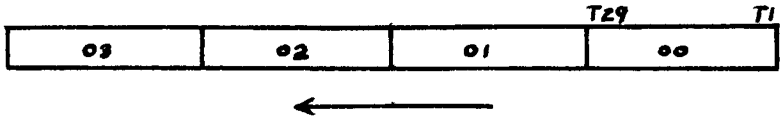
INPUTS

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:

$$\begin{array}{l r}
.0001654(=0001.654\cdot10^{-4})\cdot.0100000(=01.00000\cdot10^{-2}) & =.0000016(=000001.6\cdot10^{-6}) \\
.0003987(=000398.7\cdot10^{-6}) & \underline{=.0003987(=000398.7\cdot10^{-6})} \\
 & .0004003(=000400.3\cdot10^{-6})
\end{array}$$

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 $1\cdot10^2$ is $1\cdot10^{-2}$. Therefore, instead of multiplying .0003987 by 10.0000000($=1\cdot10^2$), we can divide .0003987 by .0100000($=01.00000\cdot10^{-2}$):

$$\begin{array}{l r}
.0001654(=0001.654\cdot10^{-4}) & =.0001654(=0001.654\cdot10^{-4}) \\
.0003987(=000398.7\cdot10^{-6})\div.0100000(=01.00000\cdot10^{-2}) & \underline{=.0398700(=0398.700\cdot10^{-4})} \\
 & .0400354(=0400.354\cdot10^{-4})
\end{array}$$

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

$$a\cdot10^{n}\cdot b\cdot10^{m}=a\cdot b\cdot10^{n+m}$$

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

$$\frac{a\cdot10^{n}}{b\cdot10^{m}}=\frac{a}{b}\cdot10^{n-m}$$

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		$21.00\text{-}01 \xrightarrow{+} PN_{0,1}$ 10^{-8}
03		04	07	5	21	30		$21.00\text{-}01 \xrightarrow{+} PN_{0,1}$ 10^{-8}
07		10	12	7	21	30		$21.02\text{-}03 \xrightarrow{-} PN_+$ 10^{-8}
12		14	14	0	29	31		Overflow?
14		16	N	5	26	22		No $PN \xrightarrow{+} 22.00\text{-}01$ 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.

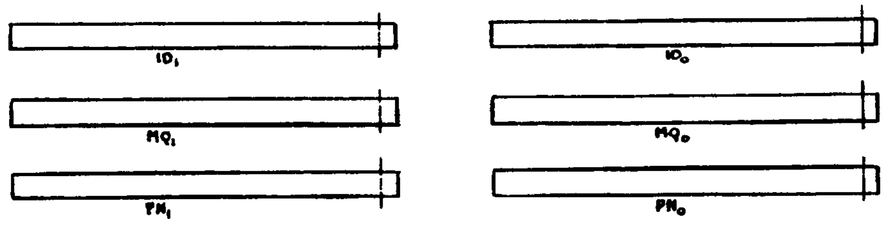
MULTIPLY

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		$10.05 \longrightarrow ID_1$	10^{-5}
06		07	09	0	20	24		$20.03 \longrightarrow MQ_1$	10^{-5}
09		56	66	0	24	31		Multiply	10^{-10}
66		68	N	4	26	21		$PN_{0,1} \longrightarrow 21.00\text{-}01$	10^{-10}

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

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		06	09	6	20	24		$20.02 \longrightarrow MQ_1$	10^{-5}
09		56	66	0	24	31		Multiply	10^{-10}
66		67	N	0	26	11		$PN_1 \longrightarrow 11.67$	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		$10.02\text{-}03 \longrightarrow ID_{0,1}$ 10^{-6}
04		06	09	4	21	26		$21.02\text{-}03 \longrightarrow PN_{0,1}$ 10^{-8}
09		v6	18	5	25	31		Divide 10^{-2}
18		22	N	4	24	20		$MQ_{0,1} \longrightarrow 20.02\text{-}03$ 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..............................	$N \xrightarrow{+} PN_{0,1}$
	Return command $\longrightarrow$ 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		$00.57 \longrightarrow AR_c$
57	[	u0	99	0	20	31	]	Return command
58		60	62	5	21	26		$21.00\text{-}01 \xrightarrow{+} PN_{0,1}$
62		64	94	1	21	31		Go to square root subroutine
63		u6	N	5	26	19		$PN_{0,1} \xrightarrow{+} 19.u6\text{-}u7$

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 $\neq 0$
06		08	00	0	16	31		= 0 Halt
07		09	N	1	28	21		$\neq 0$ 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 $\neq 0$
05	u	10	06	0	21	20		= 0 Line 21 $\longrightarrow$ Line 20
06		07	11	1	20	28		$\neq 0$ $20.03 \xrightarrow{+} AR_c$
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

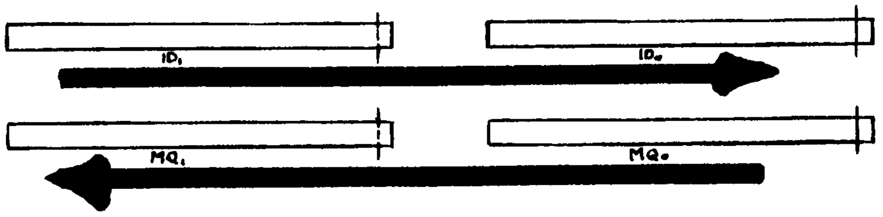
So far the only operations upon data which we have discussed are arithmetic, and we have assumed that the data numbers represent quantities. But we spoke earlier of another meaning for numbers: code. Programs can be written to process data which is in code form, where each bit, or group of bits, in a 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

```
ID1  1010110101100001011111100110|0   ID0  1101111010101001001111010111|0
MQ1  0110001111000111111100011011|10  MQ0  1101001111000001111111010100|1
AR   0011110111100000000000000110|0
```

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

After Execution

```
ID1  0000000000000000000001010110|0   ID0  1100001011111100110011011110|0
MQ1  0011011101101001111000001111|1   MQ0  1101010010000000000000000000|0
AR   0011110111100000000000000110|0
```

Before Execution

```
ID1  1010110101100001011111100110|0   ID0  1101111010101001001111010111|0
MQ1  0110001111000111111100011011|10  MQ0  1101001111000001111111010100|1
AR   0000000000000000000000000000|0
```

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

After Execution

```
ID1   00000000000000000000010101101|0   ID0   11000010111111001100110111110|0
MQ1   00110111011010011110000011111|1   MQ0   11010100100000000000000000000|0
AR    00000000000000000000000010100|0
```

Before Execution

```
ID1   10101101011000010111111001100|0   ID0   11011110101010010011110101111|0
MQ1   01100011110001111111000110111|0   MQ0   11010011110000011111110101000|1
AR    11111111111111111111111111000|1
```

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

```
ID1   00000000101011010110000101111|1   ID0   11001100110111101010100100111|0
MQ1   11000111111000110111011010011|1   MQ0   11000001111111010100100000000|0
AR    00000000000000000000000000000|0
```

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:

```
11111111000000000000000000000|1
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   0011100000111000011001110100|0

10.07   0110111000110101111100001010|1

10.08   0001110000111001110111100001|0

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 $\longrightarrow$ 20.01-00
01								zzz0000+
02								z000000+
03								0000zzz-
04								z000zzz+
05	u	10	10	0	10	21		10.06-09 $\longrightarrow$ 21.02-01
10	u	15	N	0	31	22		20·21 $\longrightarrow$ 22.03-02

After Execution

```
22.02   0011000000000000000000000000|0

22.03   0000000000000000111100001010|1

22.00   0001000000000000110111100001|0

22.01   1001110000010000000000000000|0
```

Before Execution

10.05   10110111111010001001101111001|1

L	P	T or L_k	N	C	S	D	BP	N O T E S
00		01	02	0	00	20		00.01 $\longrightarrow$ 20.01
01								zz00000-
02		05	06	0	10	21		10.05 $\longrightarrow$ 21.01
06		09	11	0	31	28		20·21 $\longrightarrow$ AR_c
11		05	N	0	28	10		AR $\longrightarrow$ 10.05

After Execution

10.05   10110111000000000000000000000|1

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   10110111111010001001101111001|1

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

After Execution

```
10.05   00000000110100010011011110011|0
```

Before Execution

```
10.06   00111000011100001100111010000|0
10.07   01101110001101011111100001010|1
10.08   00011100001110011101111100001|0
10.09   10011100000100111100111110000|1
```

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

After Execution

```
22.02   00001000011100001100111010000|0
22.03   01101110001101010000000000000|0
22.00   00001100001110010000000000000|0
22.01   00000000000000111100111110000|1
```

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: 20·21. 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 $\longrightarrow$ 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

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 $\xrightarrow{\pm}$ 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:

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

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

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:

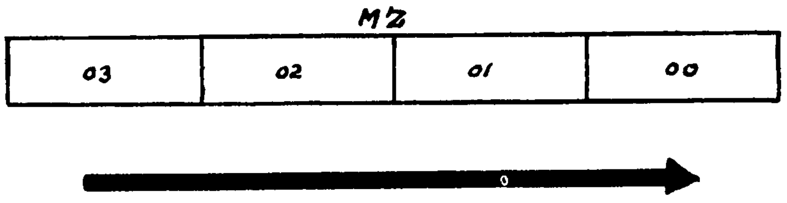
1. copy the format from its location in memory into a special buffer, called the MZ buffer, where it can control the output, and

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

```
                                                s
100 0,00 00,0 011, 000 0,00 00,0 111, 111 1,1|0 001,
```

```
02.03  803007z-
02.02  1000000+
```

Example 2:

SDDDDDDDTSDDDDDDDTSDDDDDDDTSDDDDDDDCE

```
                                              s
100 0,00 00,0 000, 000 0,00 00,0 000, 110 1,0|0 000, 000 0,00 00,0 000, 000 0,00
          s                                        s
11,0 100, 0|00 00,0 000, 000 0,00 00,0 000, 110 1,0|0 00,0 000 0,00 00,0 000, 000
0,00 01,0 001,
```

```
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		57	N	1	25	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.
or							
L		57	N	5	25	31	

L or L	v6 v6	N N	1 5	25 25	31 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.
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 $20\cdot21$. 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 $\overline{20}\cdot21$. 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		T	N	C	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.
or							
L	w	T	N	C	21	31	
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,

```
  1110101010111100110011010110|0,
```

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

```
  1110101010111100110011010110|0
  0001010101000011001100101001|1
  ----------------------------+-
1 0000000000000000000000000000|1
└---------------------------->|1
  ----------------------------+-
  0000000000000000000000000000|0,
```

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),

```
  1110101010111100110011010110|1
  1110101010111100110011010110|0
  ----------------------------+-
1 1101010101111001100110101100|1
└---------------------------->|1
  ----------------------------+-
  1101010101111001100110101100|0
```

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-