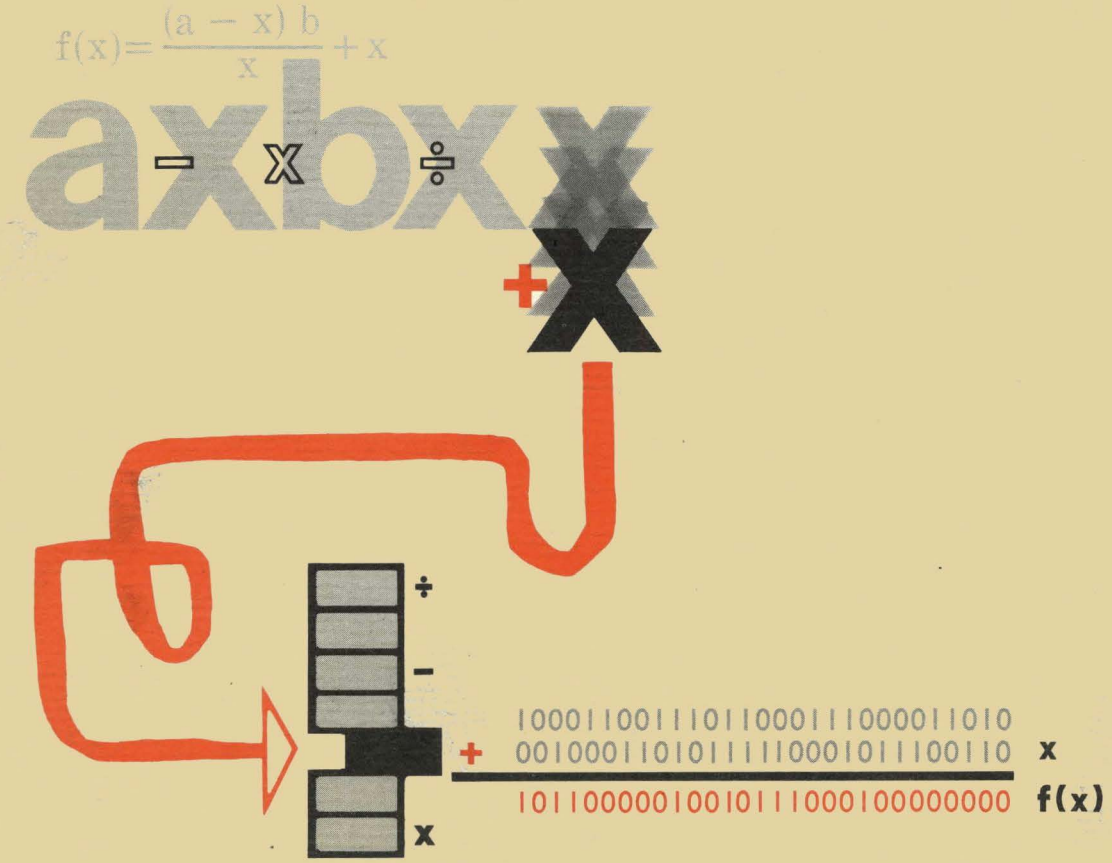
DIVISION OF BENDIX AVIATION CORPORATION
5630 Arbor Vitae Street, Los Angeles 45, California

BITS OF MEANING



INTRODUCTION TO THE BENDIX G-15

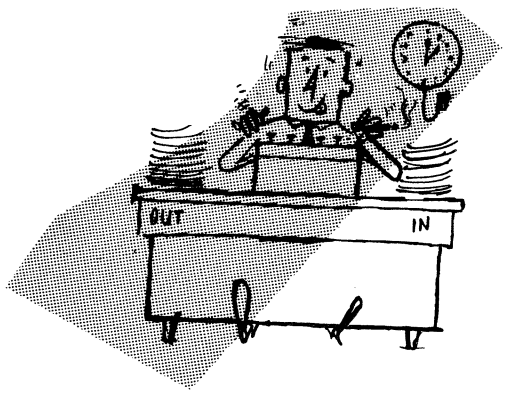
One of man's first problems was to find shelter; he solved it by walking over the land until he found a cave. This was a simple, direct solution to a simple problem. As centuries merged into Ages, he faced more problems and learned and remembered more facts; concurrently his brain became more active. His thinking-power increased, both in rate and complexity so that he faced more problems and learned more facts, at an ever-increasing rate. Thus caught in a maelstrom of mental activity, man swirled dizzily into the present. Because of the increasing complexity of the problems, men were forced to become specialists, dividing the problems into subproblems, and then dividing the subproblems, and so on. One of the categories in which some men specialized was mathematics.

Other men came to the mathematicians for solutions to those problems whose factors could be expressed numerically. But the mathematicians soon found that two types of problems were arising more and more frequently:

1. problems of such complexity that their solutions were impossible by the pencil-and-paper method;

2. problems whose solutions for any given case were possible, but which had to be repeated for case after case, until they became so time-consuming that the mathematicians had no time left for other work.

So the mathematicians turned to the engineers, and asked for development of machines to help men in the generation of solutions, by increasing the speed at which each mathematician could work. The engineers responded with various assorted combinations of keys, wheels, gears, cams, etc., which were referred to as "calculating machines". A calculating machine was placed on a mathematician's desk and called a "desk calculator". This man-machine system generated solutions at a highly increased rate, making possible more solutions per mathematician.



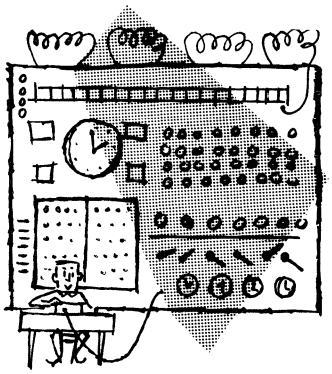
But the increased demands upon mathematicians soon surpassed even the most efficient man-machine system so far devised. Engineers were called on again, and this time they switched to

electronic equipment. Use of electronic circuits greatly reduced the time necessary to perform arithmetic operations, to the point where a single operation, such as an addition, could be performed in a few microseconds ($\frac{1}{1000000}$ ths of a second). This new type of machine was called a "digital computer".

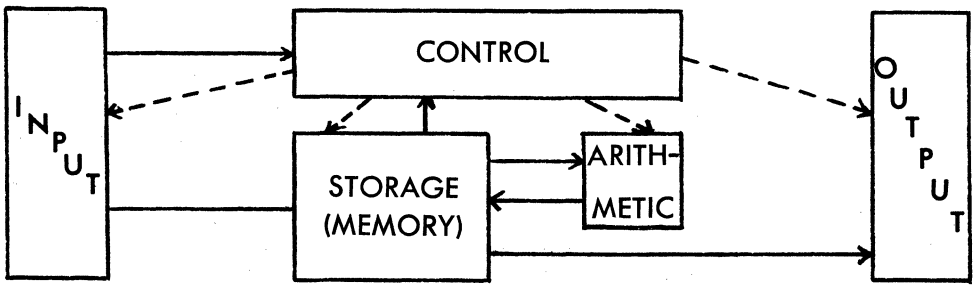
Some engineers did not switch entirely to electronic circuits: they combined them with the mechanical features of the old calculating machines, developing a computing machine which used mechanical operations to represent arithmetic and physical quantities, such as degrees of rotation of a shaft, to represent quantities, or numbers. This type of computer is called an "analog computer". The accuracy of an analog computer is limited by the accuracy with which a physical quantity can be measured.

The digital computers, using electronic circuits, represent numbers and quantities with digits, and any desired degree of accuracy can be obtained through adding more digits. Therefore, greater accuracy can be obtained in this type of computer. Most digital computers are general purpose, designed for a multitude of uses, scientific and commercial. Because the Bendix G-15 is a digital computer, we shall limit our discussion to digital computers.

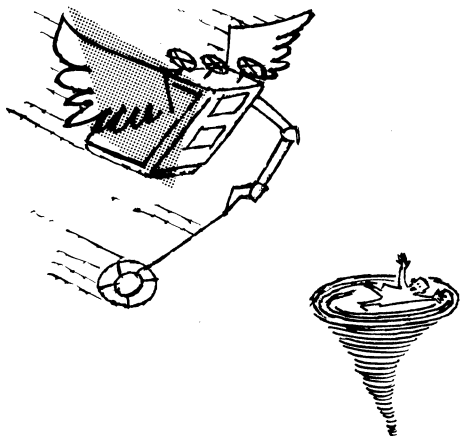
A digital computer consists essentially of five sections:



1. Input section, through which it assimilates data and control information;
2. Storage section, called "Memory", in which it stores assimilated data and control information;
3. Control section, in which it interprets the available control information;
4. Arithmetic section, in which it performs arithmetic operations on the available data, as indicated by the Control section;
5. Output section, through which it communicates solutions to the outside world.



The first digital computers were very large and very expensive, leaving many individuals and companies without the ability to purchase or lease one. As these people sank deeper and deeper into the vortex of problems, Bendix Aviation Corporation came to their aid by establishing the Bendix Computer Division, which developed a low-cost, medium-sized general-purpose computer, called the "G-15".



WHAT IS A PROGRAM?

A digital computer is capable of automatically receiving information from an external source, storing numbers representing data and control information in memory, transfering them from memory to the arithmetic element, performing specific operations on them, and transfering the results from the arithmetic element back to memory, and transmitting this information to the outside in intelligible form. In order for the computer to perform these operations, some control information (instructions) must be supplied to it. A sequence of individual instructions designed to accomplish a specific purpose is called a program. A program designed to work on specific data numbers in the computer is also stored in the memory of the computer, along with the data upon which it will operate. Because the computer can read only numbers, individual program instructions must be coded in number form, and this means the computer must be able to distinguish between data and instructions. Instructions coded in numerical form are referred to as "commands".

COMMAND vs. DATA

The distinction the computer makes between data and commands is based upon the time during which a number is inspected. There are two types of machine-time common to all digital computers: read-command-time (RC), during which commands are interpreted, and execution-time (EX), during which commands are executed. If a number is inspected during RC, the computer will regard it as a command; if it is inspected during EX, the computer will regard it as data. Following each RC there will be an EX, and following each EX, unless the computer is stopped by the execution of a command, there will be an RC.

ORDER OF COMMANDS

An order of commands has thus far been implied; for any given command, except the first, another can be found which must precede it, and its operation on data is dependent upon the execution of the previous command. There are basically two ways of determining such an order. The first is by building a predetermined sequential order into the machine, electronically. In this case, a program may be started at a given location in the memory, and, from there on, until the computer is commanded to stop, commands will be taken, one at a time, from each succeeding memory location, in order. Thr programmer can deliberately insert a com-

mand at any point, which will, upon execution, tell the computer to break this sequence and start a new one, beginning at a new memory location.

The second method of ordering commands is by program control, in which each command contains, among other information, the memory location of the next command to be obeyed. In this manner, the logical path followed by the computer is determined by the programmer. Such an ordering of commands is used in the Bendix G-15.

MEANINGS OF NUMBERS

Thus far two possible uses of a number have been discussed, and a third has been implied. The two which have been discussed are:

1. to represent quantity, and

2. to logically express, in the form of a numerical code, an instruction.

The third use, which has been implied, is to designate a location as an address. Every number in memory, whether it be a command or data, has a unique address. A command calling for an operation to be performed on a data number must contain, as well as a numeric code for the operation, the address of the data number (operand). The address of an operand in no way describes the quantity represented by the operand, any more than the address of a house describes the contents of the house.

In some computers, and the G-15 is one of them, each command must also contain the memory location (address) to which the operand is to be transferred. This is because not all operands are destined for the arithmetic element every time they are called for; it is possible, and sometimes desirable, to merely transfer a number from one location to another, within memory.

G-15 COMMAND

The basic parts, then, of a G-15 command, are:

1. Operation code;

2. Address of operand;

3. Address to which operand is to be transferred;

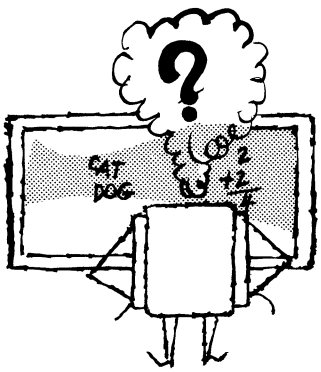
4. Address of the next command to be obeyed.

MACHINE FORM OF A NUMBER

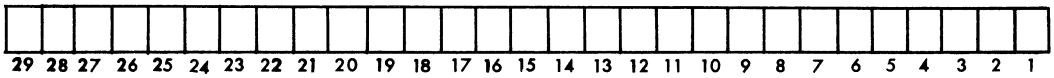
Numbers in the G-15 are in a form different from the customary decimal form.

Since we have ten fingers, we count in groups of ten; the G-15 has no fingers, being of an advanced design, but it has two electronic states between which its components will fluctuate: "ON" and "OFF". The G-15 therefore desires to count in groups of two. Counting in groups of ten requires ten digits, 0 through 9; counting in groups of two requires only two digits, 0 and 1. When we say we count in groups of certain magnitude, we are really stating by implication that we are using a certain number system, in which any quantity can be expressed by a unique series of available digits within that system. An infinite number of such systems can be defined, each having a different number of unique digits.

Similarly, we grew up among people who speak the English language, so we speak English; the G-15 is the child of electricity and magnetism, who do not speak English, but who speak only in terms of existance or nonexistance, i.e., "ON" and "OFF". Instructions, in order to be intelligible to the G-15, must be exressed in these "ON-OFF" terms, or as a series of 0's and 1's. As has been mentioned previously, instructions for the computer are coded in numerical form, and these numbers are called "commands".



All numbers in memory, whether they be data or commands, are referred to as "words", and each word has 29 digit-positions. The number system in which numbers are expressed as a series of digits, each having either of two possible values, is called the binary number system; of course the number system which utilizes ten unique digits is the decimal number system.



NUMBER SYSTEMS

Any number system has a base; the number of unique symbols in the system equals the base.

Binary	Decimal	Sexadecimal	
0 } Base	0	0	$10_{(10)}$ = u
1 } = 2	1	1	$11_{(10)}$ = v
	2	2	$12_{(10)}$ = w
	3	3	$13_{(10)}$ = x
	4 } Base	4	$14_{(10)}$ = y } Base
	5 } = 10	5	$15_{(10)}$ = z } = 16
	6	6	
	7	7	
	8	8	
	9	9	

Quantities are expressed in any number system by a series of digits ordered about a base point. Integral values are represented to the left of this point; fractional values, to the right of it. In a given series of digits, as you move to the left, away from the base point, each digit-position represents a positive power of the base one greater than that represented by the preceding digit-position. The first digit-position to the left of the base point represents units, or the base raised to a power of zero. As you move to the right, away from the base-point, the represented powers of the base increase by 1 in a negative direction, starting at the minus one power of the base immediately to the right of the point.

$$
\begin{aligned}
& abcd.efgh_{(10)} \\
= \; & a \times 10^{3} \\
& +b \times 10^{2} \\
& +c \times 10^{1} \\
& +d \times 10^{0} \\
& +e \times 10^{-1} \\
& +f \times 10^{-2} \\
& +g \times 10^{-3} \\
& +h \times 10^{-4}
\end{aligned}
$$

With a knowledge of the number system employed, you can interpret a number written in that system:

Binary	Decimal	Sexadecimal
$11001.101_{(2)}$	$1765.49_{(10)}$	$x23y.z6_{(16)}$
$= 1 \times 2^{4}$	$= 1 \times 10^{3}$	$= x \times 16^{3}$ (13×16^{3})
$+1 \times 2^{3}$	$+7 \times 10^{2}$	$+2 \times 16^{2}$
$+0 \times 2^{2}$	$+6 \times 10^{1}$	$+3 \times 16^{1}$
$+0 \times 2^{1}$	$+5 \times 10^{0}$	$+y \times 16^{0}$ (14×16^{0})
$+1 \times 2^{0}$	$+4 \times 10^{-1}$	$+z \times 16^{-1}$ (15×16^{-1})
$+1 \times 2^{-1}$	$+9 \times 10^{-2}$	$+6 \times 16^{-2}$
$+0 \times 2^{-2}$		
$+1 \times 2^{-3}$		

NUMBER CONVERSIONS

Note that, in the above example, when we "interpreted" the Binary and Sexadecimal numbers, what we really did was to convert them to unique corresponding decimal values. For most of us this is the only number system in which we feel "at home", and, when we are faced with picturing the quantity represented by a number in any other system, we must convert that number to a decimal number. Since the computer works in binary, and you and I work in decimal, it is apparent that we must be able to handle each number system and convert numbers from each system to the other with ease. It is also apparent that, if these conversions are possible between binary and decimal, similar conversions are possible among all number systems. This, of course, is true, because:

1. a given quantity can be expressed in any number system, to any desired accuracy, by a unique number, and

2. all expressions of the same quantity are equivalent.

One method of number conversion is "by inspection". For example, given the binary number,

$$1100111.101_{(2)},$$

to be converted to its decimal equivalent, we can inspect the number digit-by-digit and generate an equivalent decimal number to represent the same quantity:

$$1100111.101_{(2)} = 1\cdot2^6+1\cdot2^5+0\cdot2^4+0\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0+1\cdot2^{-1}+0\cdot2^{-2}+1\cdot2^{-3}$$

$$= 64 + 32 + 0 + 0 + 4 + 2 + 1 + 1/2 + 0 + 1/8$$

$$= 103\ 5/8 = 103.625_{(10)}.$$

Or, given the decimal number,

$$103.625_{(10)},$$

we can inspect it, and, through the reverse of the preceding process, we can extract from this decimal number each power of 2 which is contained in it. When we have done this, we will have all the information necessary to generate a unique binary number as a series of binary digits around a binary point. Since $128(=2^7)>103.625>64(=2^6)$, we know that the coefficient for each power of 2 greater than or equal to 2^7 must be 0; the binary number we generate can have any number of leading 0's, but usually we don't write them. The first non-0 coefficient of a power of 2 is in the 2^6 digit-position; this coefficient is 1. After 2^6 has been extracted from 103.625, 39.625 remains. This, in turn, is inspected for the highest power of 2 it contains. This process is repeated until the entire number has been converted; for each power of 2 found during the inspection, a 1 is placed in the binary number, and for each power of 2 not found, a 0 is placed in the binary number, as shown in the above example.

Decimal		Binary
103.625		
- 64.000	$= 2^6$, $\therefore$	$1\cdot2^6$
39.625		
- 32.000	$= 2^5$, $\therefore$	$1\cdot2^5$
7.625	$< 2^4$, $\therefore$	$0\cdot2^4$
7.625	$< 2^3$, $\therefore$	$0\cdot2^3$
- 4.000	$= 2^2$, $\therefore$	$1\cdot2^2$
3.625		
- 2.000	$= 2^1$, $\therefore$	$1\cdot2^1$
1.625		
- 1.000	$= 2^0$, $\therefore$	$1\cdot2^0$
0.625		
- 0.500	$= 2^{-1}$, $\therefore$	$1\cdot2^{-1}$
0.125	$< 2^{-2}$, $\therefore$	$0\cdot2^{-2}$
- 0.125	$= 2^{-3}$, $\therefore$	$1\cdot2^{-3}$
0.000		

$$103.625_{(10)} = 1\cdot2^6+1\cdot2^5+0\cdot2^4+0\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0+1\cdot2^{-1}+0\cdot2^{-2}+1\cdot2^{-3}$$

$$= 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1\ .\ 1\ \ 0\ \ 1_{(2)}$$

Note that an even fractional value in one number system may, upon conversion to another number system, yield an unended fraction:

$$1/10 = 1/16 + 1/32 + 1/256 + 1/512 + \ldots$$

In such a case, of course, the conversion by inspection is carried to the desired degree of accuracy in the new number system.

You can see that conversion of numbers from one system to another by inspection becomes progressively more difficult as the magnitude of the number increases or as the length of the fractional part of the number increases. Fortunately, a general rule for number conversions can be derived from the inspection method. First, note that an integral value in a given number system must be equivalent to a unique integral value in any other system. Similarly, a fractional value in a given number system must be equivalent (or nearly equivalent) to a fractional value in any other system. With these two points in mind, reconsider the conversion of $103.625_{(10)}$ to its binary equivalent, treating the integral and fractional parts separately.

If we divide 103 by 2, we will determine whether or not 103 is an exact multiple of 2. We will also know how many times greater than 2 it is. Conversely, we will also know if 103 is not an exact multiple of 2. In such a case, the remainder can be no greater than 1, and this remainder will indicate that 1 (or $1 \cdot 2^0$) must be added to some multiple of 2 in order to yield 103.

$$2\overline{)103}\quad 51 \text{ , remainder} = 1,$$

or,

$$1. \quad (51 \cdot 2^1) + (1 \cdot 2^0) = 103_{(10)}.$$

If we now proceed to divide 51 by 2, we will determine what multiple of 2 it contains, and whether or not it also contains a remainder = 1.

$$2\overline{)51}\quad 25 \text{ , remainder} = 1,$$

or,

$$2. \quad (25 \cdot 2^1) + (1 \cdot 2^0) = 51_{(10)}.$$

By substitution in (1), we get

$$\left[(25 \cdot 2^1) + (1 \cdot 2^0)\right] \cdot 2^1 + (1 \cdot 2^0) = 103_{(10)},$$

$$(25 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 103_{(10)}.$$

We can continue this process until no more divisions are possible, and the remainder in each case will equal the coefficient of some positive power of 2 in the ascending order, 2^0, 2^1, 2^2, etc., as shown in the following series of divisions, where the remainder in each case is shown to the right of the dotted line. Associated with each remainder is a corresponding term in the binary expansion of $103_{(10)}$. We can then write these remainders as a sum equal to $103_{(10)}$:

0	$1 \sim 1 \cdot 2^6$
$2\overline{)1}$	$1 \sim 1 \cdot 2^5$
$2\overline{)3}$	$1 \sim 1 \cdot 2^4$
$2\overline{)6}$	$0 \sim 0 \cdot 2^3$
$2\overline{)12}$	$0 \sim 0 \cdot 2^2$
$2\overline{)25}$	$1 \sim 1 \cdot 2^1$
$2\overline{)51}$	$1 \sim 1 \cdot 2^0$
$2\overline{)103}$	

$$1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 103_{(10)}.$$

And thus we have determined the unique binary number equivalent to $103_{(10)}$:

$$1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1_{(2)} \cdot$$

Similarly, for the fractional part of the number, if we multiply $.625_{(10)}$ by 2 and achieve a result equal to $1.000_{(10)}$, we know that this fractional number is exactly equal to $1/2$ (2^{-1}). If the result is greater than 1.000, the original fractional number is greater than, and thus contains, $1/2$. If the result is less than 1.000, the original fractional number is less than, and thus does not contain, $1/2$.

$$\begin{array}{r} .625 \\ \underline{\times \quad 2} \\ 1.250, \end{array}$$

or,

$$3. \quad (1 \cdot 2^{-1}) + (.250 \cdot 2^{-1}) = .625_{(10)} \cdot$$

If we now proceed to multiply .250 by 2, we will determine whether or not it contains $1/2$.

$$\begin{array}{r} .250 \\ \underline{\times \quad 2} \\ 0.500, \end{array}$$

or,

$$4. \quad (0 \cdot 2^{-1}) + (.500 \cdot 2^{-1}) = .250_{(10)} \cdot$$

By substitution in (3), we get

$$(1 \cdot 2^{-1}) + \left[(0 \cdot 2^{-1}) + (.500 \cdot 2^{-1})\right] \cdot 2^{-1} = .625_{(10)},$$

$$(1 \cdot 2^{-1}) + (0 \cdot 2^{-2}) + (.500 \cdot 2^{-2}) \qquad = .625_{(10)} \cdot$$

We can continue this process until no more multiplications are possible or until the desired degree of accuracy in the binary result is achieved, and the integral value in each case (0 or 1) will equal the coefficient of some negative power of 2 in the descending order, 2^{-1}, 2^{-2}, 2^{-3}, etc., as shown in the following series of multiplications, where the integral value in each case is shown to the left of the dotted line. Associated with each integer is a corresponding term in the binary expansion of $.625_{(10)}$. We can then write these as a sum equal to $.625_{(10)}$:

$$\begin{array}{rcl} & & .625 \\ & & \times \quad 2 \\ 1 \cdot 2^{-1} & \sim & 1.250 \\ & & \times \quad 2 \\ 0 \cdot 2^{-2} & \sim & 0.500 \\ & & \times \quad 2 \\ 1 \cdot 2^{-3} & \sim & 1.000 \end{array}$$

$$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = .625_{(10)} \cdot$$

And thus we have determined the binary equivalent of $.625_{(10)}$:

$$.1 \quad 0 \quad 1_{(2)} \cdot$$

By combining this with the previously derived integral binary value for $103_{(10)}$, we get

$$103.625_{(10)} = 1100111.101_{(2)}.$$

If the above methods work for the conversion of decimal numbers to binary, it stands to reason that they will also work for conversions between any number systems. This is true, although a proof of it will not be included here.

The general rule we can derive has two parts:

Part I: The integral part (x) of a number in the system with base = N can be converted to a unique corresponding integral part (x') of a number in the system with base = M by successively dividing x and each succeeding quotient by $M_{(N)}$ until a quotient of 0 is reached. The first remainder (may equal 0) is the coefficient of M^0, and each succeeding remainder is the coefficient of the next succeeding positive power of M, in ascending order.

Part II: The fractional part (y) of a number in the system with base = N can be converted to a corresponding fractional part (y') of a number in the system with base = M by successively multiplying y and the fractional part of each succeeding product by $M_{(N)}$ until a product of 0 or the desired accuracy is reached. The integral part (may equal 0) of the first product is the coefficient of M^{-1}, and the integral part of each succeeding product is the coefficient of the next succeeding negative power of M, in descending order.

BINARY ARITHMETIC

Before doing a binary-to-decimal conversion as a further illustration of the general rule, we must briefly note that the rules of arithmetic apply as well to binary as to decimal numbers.

Binary Addition Table

	0	1
0	0	1
1	1	0 carry 1

Binary Multiplication Table

	0	1
0	0	0
1	0	1

By inspection, you can see that $010_{(2)} = 2_{(10)}$ and $110_{(2)} = 6_{(10)}$. If we add these two binary numbers, we should get the binary equivalent of $8_{(10)} = 1000_{(2)}$.

$$\begin{array}{r} 010 \\ 110 \\ \hline 1000 \end{array}$$

If we multiply these two binary numbers (010 and 110), we should get the binary equivalent of $12_{(10)}$.

$$\begin{array}{r} 010 \\ \underline{110} \\ 000 \\ 010 \\ \underline{010} \\ 01100 \end{array} = 12_{(10)}$$

We know, from our previous conversion by inspection, that $1100111.101_{(2)} = 103.625_{(10)}$; now we'll convert this binary number, using the derived general rule. Note that $10_{(10)} = 1010_{(2)}$.

$$\begin{array}{r} 1010 \\ 1010\overline{)1100111} \\ \underline{1010} \\ 1011 \\ \underline{1010} \\ 11 \end{array} \text{ = remainder: } 11_{(2)} = 3_{(10)}$$

$$\begin{array}{r} 1 \\ 1010\overline{)1010} \\ \underline{1010} \\ 0 \end{array} \text{ = remainder: } 0_{(2)} = 0_{(10)}$$

$$\begin{array}{r} 0 \\ 1010\overline{)1} \\ \underline{0} \\ 1 \end{array} \text{ = remainder: } 1_{(2)} = 1_{(10)}$$

$$\begin{array}{r} .101 \\ \underline{1010} \\ 1010 \\ \underline{1010} \\ 110010 \end{array} = 110.010 : 110_{(2)} = 6_{(10)}$$

$$\begin{array}{r} .010 \\ \underline{1010} \\ 0100 \\ \underline{0100} \\ 010100 \end{array} = 010.100 : 010_{(2)} = 2_{(10)}$$

$$\begin{array}{r} .100 \\ \underline{1010} \\ 1000 \\ \underline{1000} \\ 101000 \end{array} = 101.000 : 101_{(2)} = 5_{(10)}$$

Combining, we get

$$1100111.101_{(2)} = 1\cdot10^{2}+0\cdot10^{1}+3\cdot10^{0}+6\cdot10^{-1}+2\cdot10^{-2}+5\cdot10^{-3},$$

$$1100111.101_{(2)} = 1 \quad 0 \quad 3 \quad . \quad 6 \quad 2 \quad 5_{(10)}.$$

COMPUTER DATA

With this preparation, we are now ready to discuss data numbers as they appear in the G-15 memory. Two features of the machine already mentioned are:

1. the G-15 works in binary only; and

2. every word (number) in memory has 29 digit-positions (a binary digit is referred to among computer personnel as a "bit", and, from now on, we will speak of bits, rather than digits).

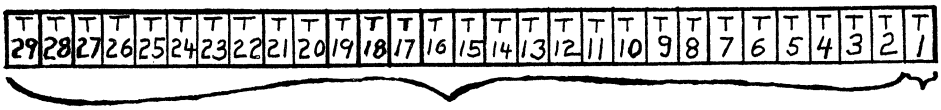
Notice that an addition of two words could very well result in a sum which would require more than 29 bits for expression. In such a case,

the result generated in the computer will be erroneous. An attempt to generate a number too large for the computer to hold causes a condition called "overflow", which the G-15 is equipped to detect.

```
a  :   100000000000000000000000000000
b  :   100000000000000000000000000000
a+b:  1000000000000000000000000000000
```

If the computer attempts to add the two numbers shown, the sum will exceed 29 bits.

It is also important to notice here that the machine has no facility for storing or recognizing the symbols, + and -. These, as well as all other information, must be represented in the computer in binary notation. One of the bits in each word has been set aside for this purpose; it is



the right-most (least significant) bit, T1, as shown above. A zero here will indicate plus, and a one will indicate minus, to the machine. The computer interrogates this "sign" bit before operating on the associated 28 bits of magnitude. During an addition of two numbers, the signs are added, but there is no "carry" from the sign-position into T2, the least significant bit of magnitude. Thus, the addition of two positive numbers, in the computer, would yield the correct positive result, provided no overflow occurred:

```
0000000000000000000000000010  :  +1
0000000000000000000000000100  :  +2
0000000000000000000000000110  :  +3 ,
```

which is correct. But notice that the addition of a positive and a negative number would result in an erroneous answer:

```
0000000000000000000000000011  :  -1
0000000000000000000000000100  :  +2
0000000000000000000000000111  :  -3 ,
```

which is incorrect. Similarly, the addition of two negative numbers would yield an erroneous result:

```
0000000000000000000000000011  :  -1
0000000000000000000000000101  :  -2
0000000000000000000000000110  :  +3 ,
```

which is incorrect. In order to enable itself to add numbers correctly, regardless of sign, the computer "complements" all negative numbers prior to addition. The word, "complement" can have any of various meanings when used to describe an arithmetic operation; in the case of the G-15,

1. only negative numbers are complemented, and

2. for each negative number possible there is a unique complement, formed by retaining the negative sign and the least significant bits of magnitude up to, and including the first, least significant, 1 encountered, and from there on, changing all 1's to 0's and all 0's to 1's.

For example, the complement of

000000000000000000000000001|1 (-1) is 111111111111111111111111111|1.

The complement of

111010110000000000010101100|1 is 000101001111111111101010100|1.

In an addition in the computer, if there is a "carry" out of T29, an "end-around-carry" is performed, and the carried 1 is added into the sign position (T1). Following these two rules, the addition of -1 and +2, shown to yield an erroneous result on page 12, will yield the correct result:

```
  000000000000000000000000001|1   :   -1
  000000000000000000000000010|0   :   +2 ,
```

complement the negative number, and then add:

```
  111111111111111111111111111|1   :   -1 (complemented)
  000000000000000000000000010|0   :   +2
1 000000000000000000000000001|1
  └──────────────────────────>|1   :   (end-around-carry)
  ----------------------------|-
  000000000000000000000000001|0   :   +1 ,
```

which is correct. Similarly, the addition of -1 and -2, shown to yield an erroneous result on page 12, will yield the correct result:

```
  000000000000000000000000001|1   :   -1
  000000000000000000000000010|1   :   -2,
```

complement the negative numbers, and then add:

```
  111111111111111111111111111|1   :   -1 (complemented)
  111111111111111111111111110|1   :   -2 (complemented)
1 111111111111111111111111101|0
  └──────────────────────────>|1   :   (end-around-carry)
  ----------------------------|-
  111111111111111111111111101|1   :
```

in order to interpret this result, it must be recomplemented. Just as all negative numbers are complemented prior to addition, so, any negative results are in complement form. The rule for recomplementation is, of course, the same as the rule for complementation. The complement of a complement is the original form of a number (sign and magnitude).

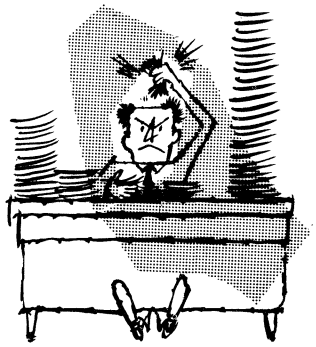
The complement of

111111111111111111111111110|1 is 000000000000000000000000001|1,

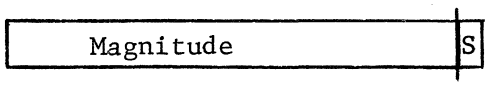
or -3, which is the correct result.

If two numbers of like sign are added, there should be no change of sign. If both are positive, and, during the addition there is a carry out of T29, overflow is present, and, in this case, the end-around-carry causes an erroneous sign (-), and the magnitude of the result will be erroneous. If both numbers are negative and complemented (when the computer adds them it will assume that each has been complemented), there should be an end-around-carry, and the absence of it will indicate overflow. In this case also, both the sign and the magnitude of the result will be erroneous. Prove this to yourself by writing down two uncomplemented negative numbers (28 bits of magnitude and a 1 in the sign-bit) of such magnitudes that, when combined, more than 28 bits will be needed to express the resultant magnitude. Complement them and add the complements. Recomplement the result, if necessary (if the result is negative), and compare it with the correct solution.

It is an invaluable aid to a programmer to be able to manipulate numbers with pencil and paper in the form in which they are in the machine (binary). With 29 bits per word, this is the type of chore that drives people out of a profession, if nothing else. It is affectionately called "bit-chasing" by some programmers. In order to alleviate this situation and to simplify the inputs and outputs for the computer, a "short-cut" number system is needed. The selected number system must be easily converted to binary, and vice-versa, by inspection, in order to avoid tedious paper-work. With 28 bits of magnitude (the sign will be the same for all number systems), use of the decimal system for this purpose is prohibitive.



Remember that a word of data in the G-15 has 28 bits of magnitude and a sign,



and choose a random example:

1101000011110101001100011001|1

Now expand the 28 bits of magnitude into a series of terms, as would be done in the inspection method of conversion (after all, we want to retain the inspection method and just save ourselves from the necessity of adding up the terms):

$$1\cdot2^{27}+1\cdot2^{26}+0\cdot2^{25}+1\cdot2^{24}+0\cdot2^{23}+0\cdot2^{22}+0\cdot2^{21}+0\cdot2^{20}+1\cdot2^{19}+1\cdot2^{18}+1\cdot2^{17}+$$
$$1\cdot2^{16}+0\cdot2^{15}+1\cdot2^{14}+0\cdot2^{13}+1\cdot2^{12}+0\cdot2^{11}+0\cdot2^{10}+1\cdot2^{9}+1\cdot2^{8}+0\cdot2^{7}+0\cdot2^{6}+0\cdot2^{5}+$$
$$1\cdot2^{4}+1\cdot2^{3}+0\cdot2^{2}+0\cdot2^{1}+1\cdot2^{0}.$$

Notice that, if 2^4 is factored out of each term as many times as possible, we will wind up with a series of terms of the form:

$$(2^4)^n(a2^3+b2^2+c2^1+d2^0).$$

Such terms would satisfy the conditions of a new number system, with base 2^4, and coefficients ranging from 0 through 2^4-1 (which is equal to $2^3+2^2+2^1+2^0$). This is called the sexadecimal number system (abbreviated hex).

The expansion above would look like this:

$$(2^4)^6(1\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0)+(2^4)^5(0\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0)+(2^4)^4(1\cdot2^3+1\cdot2^2+$$
$$1\cdot2^1+1\cdot2^0)+(2^4)^3(0\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0)+(2^4)^2(0\cdot2^3+0\cdot2^2+1\cdot2^1+1\cdot2^0)+(2^4)^1$$
$$(0\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0)+(2^4)^0(1\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0).$$

This series of terms can now be written in the following manner:

$$13\cdot16^6+0\cdot16^5+15\cdot16^4+5\cdot16^3+3\cdot16^2+1\cdot16^1+9\cdot16^0.$$

Following the list of unique symbols to be used as digits in the hex system, as shown on page 5,

$$x\cdot16^6+0\cdot16^5+z\cdot16^4+5\cdot16^3+3\cdot16^2+1\cdot16^1+9\cdot16^0.$$

This, in turn, can be written as a hex number: $x0z5319._{(16)}$.

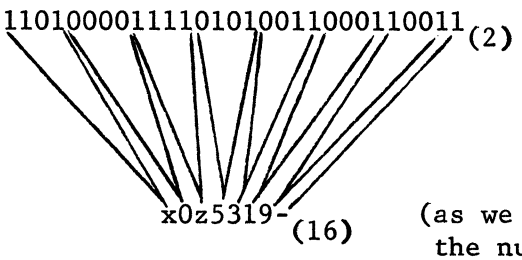
To summarize the decimal and binary equivalents for each hex digit:

Hex	Decimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
u	10	1010
v	11	1011
w	12	1100
x	13	1101
y	14	1110
z	15	1111

By assuming leading 0's in the above binary equivalents for the various hex digits, it can be seen that each hex digit exactly corresponds to a unique set of four bits. This is also true for decimal digits, but note

that we run out of single decimal digits before all possible combinations of four bits have been exhausted.

Conversion from hex to binary and from binary to hex is a very simple process by inspection, and, once you have mastered the above table, you will be able to make these conversions almost as rapidly as you can read the numbers. Seven hex digits exactly cover 28 bits, each hex digit corresponding to a group of four bits in a corresponding position in the binary number:



(as the number appears in the computer)

(as we represent the machine form of the number in a "short-hand" system)

PROGRAMMING THE G-15

It is the function of the programmer to analyze a problem in order to determine what inputs are to be called for, how they are to be processed, and what outputs are to be derived. It is then his function to use his knowledge of the operations and capabilities of the computer to prepare a sequence of commands which will be able to automatically control the computer and cause it to perform the necessary operations in the proper order.

To understand the actual form of commands for the G-15, you must be presented with more information than we can present in this short space. A machine-language programming manual is available. Of course all arithmetic operations, addition, subtraction, multiplication, and division are available, as are many other mathematical operations, such as the extraction of square roots and the generation of trigonometric functions. There are also available many "logical" operations, dealing with the manipulation of any specific bits within data words.

In addition to machine-language programming, there is an even simpler coding system available with the G-15, called INTERCOM. This is an interpretive system in which operations called for by the programmer are performed by a program already in the memory of the computer. Each "command" written in the INTERCOM system by the programmer is interpreted by the program in the machine, and, on the basis of what is called for, that program will operate a sequence of machine-language commands. There is also a manual available for programming and coding in the INTERCOM system.