An understanding of the binary number system is necessary before proceeding with a further examination of LGP-21 programming concepts. Each digit of a decimal number has a multiplier associated with it. Take, for example, the number 237.

Multipliers:	etc. ←	1000	loo	10	1
Digits:			2	3	7

Starting with the least significant digit (first digit to the left of the decimal point) the associated multiplier is 1 (or 10^0); moving one place to the left, the multiplier is 10 (or 10^1), then 100 (or 10^2), 1000 (or 10^3), etc. The multipliers, starting with the least significant digit and moving to the left, are consecutively higher powers of 10. The number 237, then means:

7 ones plus	7x 1 = 7
3 tens plus	3 x 10 = 3 0
2 one hundreds	2 x 100 = 200
Total	237

The binary number system is similar to the decimal system, with two important differences. First, the multipliers starting with the least significant digit and moving to the left are consecutively higher powers of 2: 1 (or 2^0), 2 (or 2^1), 4 (or 2^2), 8 (or 2^3), etc. The second difference is that any digit position may contain only a 0 or 1, whereas, in the decimal system, any digit position may contain 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. An example of a binary number, then, is

Multipliers:	etc. ←	128	64	32	16	8	4	2	1
Digits:		1	1	1	0	1	1	0	1

This binary number, 11101101, is constructed just like the decimal number 237, above.

By adding the respective multiplier values for each binary digit, starting with the least significant digit, we find that

1x1	= 1
0x2	= 0
1x4	= 4
1x8	= 8
0x16	= 0
lx 32	= 32
lx 64	= 64
lx 128	= 128
	237

Thus, the decimal number 237 is equivalent to 11101101 in binary. The decimal system is based on 10 digits, and the binary system on 2. The standard notation used to specify the base of a number is a subscript. Therefore, the equivalence could be written:

$$237_{10} = 11101101_2$$

To convert a binary number to its decimal equivalent, write the multipliersabove each of the binary digits, then total all the multipliers that have the digit "1" below them. For example, find the decimal equivalent of 110000110102:

1024	512	256	128	64	32	16	8	4	2	1
										0

1024
512
16
8
2

1562_{10} = 110000110102

One way to convert a decimal integer (whole number) to its binary equivalent is to divide the number by 2. The remainder becomes the least significant binary digit; the quotient (e.g. , 237 ÷ 2 gives a quotient of 118 and a remainder of 1) is again divided by 2 and the remainder becomes the next binary digit. This process continues until the quotient is zero. The remainders become the binary number, where the first remainder is the least significant binary digit and the last remainder is the most significant (far left) binary digit.

Example: Convert 23710 to its binary equivalent.

Quotient	Remainder
2| 237	
2| 118	1 least significant
2| 59	0
2| 29	1
2| 14	1
2| 7	0
2| 3	1
2| 1	1
0	1 most significant

Therefore, 237_{10} ▫ 11101101_2.

In the decimal system the digits to the right of the decimal point (fractions)also have multipliers. Take, for example, the number .6875:

Multipliers:	1/10	1/100	1/1000	1/10,000 ⟶ etc.
Digits:	6	8	7	5

The most significant fractional digit (first digit to the right of the decimal point) has a multiplier of 1/10 (or 10^{-1}); moving one place to the right, the multiplier is 1/100 (or 10^{-2}), then 1/1000 (or 10^{-3}), 1/10,000 (or 10^{-4}), etc. The multipliers, starting with the most significant digit and moving to the right, are consecutively lower powers of 10. The number .6875, therefore, constitutes a series of additions, as follows:

6 x 1/10 = .6
8 x 1/100 = .08
7 x 1/1000 = .007
5 x 1/10000 = .0005
.6875

Again, the binary system works similarly. The multipliers, starting with the most significant fractional digit and moving to the right, are consecutively lower powers of 2, namely 1/2 (or 2^{-1}), 1/4 (or 2^{-2}), 1/8 (or 2^{-3}), 1/16 (or 2^{-4}), etc. Again, a digit position can only contain a 0 or a 1. An example of a binary fraction is

Multipliers:	1/2	1/4	1/8	1/16 ⟶ etc.
Digits:	.1	0	1	1

To convert a binary fraction to its decimal equivalent, the multiplier values of the binary fraction are added again, just as in the decimal example:

1 x 1/2 = .50
0 x 1/4 = .00
1 x 1/8 = .125
1 x 1/16 = .0625
.6875

Therefore, $.6875_{10} = .1011_2$

One way to convert a decimal fraction to its binary equivalent is to multiply successively by 2, ignoring any digit to the left of the decimal point in the multiplicand, when performing the successive multiplications.

Example: Convert $.6875_{10}$ to its binary equivalent.

.6875
2
1x37 50
x 2 (ignoring the "1" to the left of the point in the multiplicand)
0.7500
2
1x5000
x 2 (ignoring the "1" to the left of the point in the multiplicand)
1.0000

Continue until there are all zeros to the right of the decimal point, as on the last multiplication above, or until the number of multiplicands equals the number of bits to the right of the binary point in the number. Going back to the first result, write down the digits to the left of the point in each product; place a point in front of these to get the binary equivalent of the decimal number.

Therefore, $.6875_{10} = {}_{\wedge}1011_2$

In the decimal system this is called a decimal point; in the binary system, a binary point. The binary point is usually represented as a caret (A). Also in binary terminology, the word "bit" is often used synonymously with "binary digit"-thus, "a 32 bit number" and "a 32 digit binary number" are the same thing.

ADDITION IN BINARY

Addition is the same as in the decimal system, except, 1 + 1 = 0 with a 1 carried.

Examples:

1	10	11	111
+1	+1	+1	+11
10	11	100	1010

SUBTRACTION IN BINARY

Subtraction is also the same as in decimal, except, 0 - 1 = 1 with a 1 borrowed: i.e. borrow 1 from the left and add 2 to the digit on the right, just as you would add 10 if working in decimal.

Examples

1	0	10	1010	100
0	-0	1	1.-111	-11

MULTIPLICATION AND DIVISION IN BINARY

. The rules are the same as in decimal.

Examples:

0 x 0 = 0	0 ÷ 0 = 0
1 x 0 = 0	0 ÷ 1 = 0
1 x 1 = 1	1 ÷ 1 = 1

Once the binary configuration of a number has been established, it is not difficult to imagine what it looks like in a memory location. For example,

$$.375_{10} = {}_{\wedge}011_2$$

appears in the LGP-21 as

0 01100000000000000000000000000000

position of binary point* t spacer is always 0

sign bit, always 0 for positive numbers.

NEGATIVE NUMBERS

Bit position zero of a computer word will indicate whether the number is positive or negative. However, the sign of a positive number is not changed by simply inserting a 1 in position zero. Instead, negative numbers are held in the computer as a 2's complement.

Consider, for example, the number -. 375 which is represented in.binary as:

0	1	2	3	4	5	6	7	8	9		29	30	31	bit positions of computer word
1	1	0	1	0	0	0	0	0	.	.	.	0	0	0

The quickest way to see why this is the computer's way of representing $-.375_{10}$ is to add it as a binary number to the representation of $+.375_{10}$:

	0	1	2	3	4	5	6	7	8	9		29	30	31	bit positions of computer word
$+.375_{10}$ =	0	0	1	1	0	0	0	0	0	0.	.	.	0	0	0
$-.375_{10}$ =	1	1	0	1	0	0	0	0	0	o.	.	.	0	0	0
	1	0	0	0	0	0	0	0	0	0	.	.	0	0	0

If the 1 to the left of bit position zero is dropped, the result is 0, just as $.375_{10} + (-.375_{10}) = 0$.

One way to obtain the representation of a negative number is the following:

1. Change its sign to + and write its binary representation.
2. Starting at the left, change all the l's to O's and all the O's to l's, until the last 1 is reached. This 1 and all the following zeros remain unchanged.

The largest positive number the LGP-21 Accumulator can holds is

01111111111111111111111111111110

---

* The binary point for a number is never actually stored in memory. The location of the imaginary binary point inside the computer is between bit positions 0 and 1. However, for convenience in expressing integer values, the binary point is often assumed to be moved to other positions. This relative position is referred to as "q" and is discussed later in this chapter.

If the negative value of this number is used, it appears as

10000000000000000000000000000010

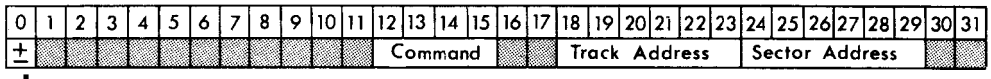
The number -1, which cannot be converted according to the above rule, appears as

10000000000000000000000000000000

Notice that the first bit position of all positive numbers contains a zero, and that of all negative numbers a 1. It is the sign bit of the Accumulator which is examined by the circuits associated with the Conditional Transfer instruction.

INSTRUCTION WORDS

The format of a word which is interpreted by the computer as an instruction is as follows (Figure 4.1):



L Interpreted only in conjunction with T, I, P, and Z instructions.

FIGURE 4.1 Instruction Word Format

The only bits the computer considers when interpreting an instruction word are 0, 12 through 15, and 18 through 29. Other positions in the word do not affect the meaning of the instruction.

Each of the command symbols or letters has a 4-bit code. This code is held in positions 12 through 15 of the instruction word. The 4 positions 12 through 15 allow for 16 different 4-bit patterns. The 16 eommand symbols and their 4-bit codes are listed in Figure 4.2 (Note: some of these have not yet been discussed.

Symbol	Command	Code
Z	Halt; Sense and Transfer	0000
B	Bring	0001
Y	Store Address	0010
R	Set Return Address	0011
I	Input or Shift	0100
D	Divide	0101
N	N Multiply (save right)	0110
M	M Multiply (save left)	0111
P	Print or Punch	1000
E	Extract	1001
U	Unconditional Transfer	1010
T	Conditional Transfer	1011
H	Hold	1100
C	Clear	1101
A	Add	1110
S	Subtract	1111

FIGURE 4.2 List of LGP-21 Commands

Examples of some instruction words:

DECIMAL	BINARY																															
												COMMAND						TRACK						SECTOR								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B0523	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	1	1	1	0	0
S 6317	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1	1	1	0	1	0	0	0	1	0	0

In the discussion of the Y instruction, it was explained that this instruction causes the address portion of the contents of the Accumulator to replace the address portion of the contents of location m. This means that the contents of bit positions 18 through 29 of the Accumulator replaces the contents of bit positions 18 through 29 of memory location m.

Also discussed earlier was half of the rule for track-and-sector arithmetic when adding two instruction words. The rule was that, when the sector comes to 64 or more, subtract 64 from it and add 1 to the track. Now, consider track modification. When a track address exceeds 64, a 1 is carried into bit position 17 (one of the bits which are ignored in an instruction). This allows "end-around" programming; i. e. , one could consider the tracks as being in sequence, numbered 00, 01, 02,. . .60, 61, 62, 63, 00, 01, etc. For example, if the address 1500 were to be added to the address 5329, the resulting address would be 0429 and a 1 bit would be carried into bit position 1'7. This carry is important if an address is used to terminate a loop which results in an "end-around" operation.

It was also noted earlier that adding Z is the same as adding zero. These rules are based on binary arithmetic. Some examples of arithmetic operations using two instruction words follow:

DECIMAL	BINARY																															
												COMMAND						TRACK						SECTOR								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B4218	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	1	0	0	0
+Z0056	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
B4310	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0
H3638	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0
+ 23300	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
HO538	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	1	1	0	0	0	1	0	0
S4215	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	1	1	0	0
+ 23551	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	1	1	0	0
S1402	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	1	0	0	0
HO301	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0
-H0500	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
-S6201	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	0	0

Notice that the bits in the Command portion of the instruction word can be manipulated, too. For example, if a Bring command is added to a Hold command, the result would be a Clear command.

DECIMAL	BINARY																														
												COMMAND						TRACK						SECTOR							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B1408	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0
+H1026	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0
C2434	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	0

Therefore, care must be exercised when adding or subtracting instructions (e.g., to test for the end of a loop) so that the desired result will be obtained.

DATA WORDS

The format of a word interpreted as data by the computer is shown in Figure 4.3. It consists of a sign (in bit position zero) and 30 bits of magnitude. The 31st, or spacer bit, is always zero in memory. A computer word can represent data in a number of different forms, including:

1. Binary
2. Binary-Coded Decimal (4-bit format)
3. Alphanumeric (6-bit format)

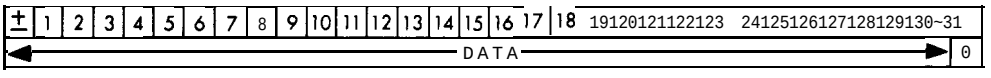


FIGURE 4.3 Data Word Format

Binary Data

When the number 125.25_{10} is handled in the computer as binary data, it appears in this form: $1111101_{\wedge}01_2$. Since there are 32 places in a computer word, the question arises: Where in the 32 places is the $1111101_{\wedge}01$ positioned. The answer is that it can be anywhere in the word. The convention for denoting the position of the number is to specify the value of q; q being the position of the least significant integer bit, and the caret symbol indicating the position of the binary point in the computer word. For example:

Decimal No.	Computer Word																														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
125.25@q = 12	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
125.25@q = 10	0	0	0	0	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The letter q is sometimes dropped and the decimal number written as 125.25 @ 12 or 125.25 @ 10. This convention also applies to instruction words. Therefore, for the example above, we could write that the command is @ 15, the track address @ 23, and the sector address @ 29.

Since the position of the binary point in a computer word is merely an assumption for the programmer's convenience, the computer does not know where it is, but assumes it to be between bits 0 and 1, or at a q of 0 for all numbers, including results of arithmetic operations. The programmer therefore considers a number (as interpreted by the computer) to be multiplied by 2^q, where q is the assumed binary point.

Example:

Computer Word	Number as Interpreted by Computer	Number as Interpreted by Programmer
0100 --------0	.5	$.5 \times 2^q$

If the programmer's q is 2, the number is $.5 \times 2^2$ or 2; if his q is 3, the number is $.5 \times 2^3$ or 4. This is analogous to multiplying by 10^x in the decimal system by moving the decimal point "x" places to the right.

When decimal data is to be entered into the computer, it can be read in and converted to binary by one of the data input subroutines available from General Precision. The q of the binarized data is specified by the programmer. Care must be taken to specify a q at which the data can actually be held. The q can be determined only when the largest value is known which the subroutine is being asked to read at a given time. This means that the programmer must specify a q at least large enough that the largest data value can be binarized to that q. Further, if the programmer wants to retain as much significance to the right of the binary point as possible, he should not make the q any larger than necessary.

By consulting the Powers of 2 Table, (Appendix C), it is easy to determine the largest number that can be held at any given q and the decimal places of accuracy possible to the right of the binary point. For example, $2^9 = 512$ means that at a q of 9, the computer can hold binary numbers ranging from -512 to almost $512, as shown below:

DECIMAL	BINARY (bits 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
-512 @ 9	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
511.9...9@9	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

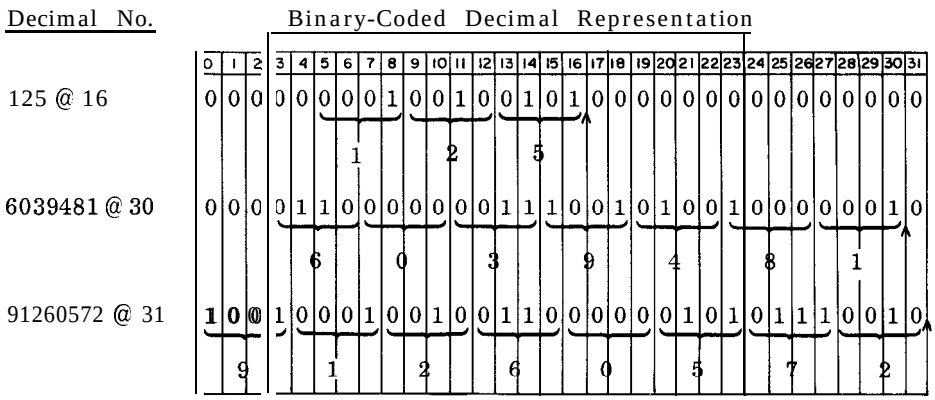
To determine the precision to the right of the binary point at a q of 9, one must consider how many binary places there are between positions 10 and 30 inclusive (position 31 is the spacer bit). This would allow 21 binary places. The Powers of 2 Table shows that $2^{-21} = .000000476...$ Therefore, the programmer can safely expect, at a q of 9, to hold in binary the equivalent of decimal numbers accurate to 6 decimal places to the right of the decimal point.

Binary-Coded Decimal Data

Each decimal digit has a 4-bit code as follows:

Decimal Digit	Code	Decimal Digit	Code
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Binary-coded decimal data is held in groups of 4 bits, each group representing a decimal digit. Up to 8 such digits can be held in a 32-bit computer word. Examples:



It should be observed that the same number in binary-coded decimal and in simple binary presents two entirely different bit patterns:

125 in BCD @ 30 o - - - - - 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 ~ 0
125 in Binary @ 30 o - - - - - 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 ~ 0

Decimal data enters the computer in binary-coded decimal; usually it is converted to binary and stored. However, there are some instances when this conversion is not necessary. If it is an identification number (such as a stock or employee number), has only numeric (no alphabetic) characters, and the problem requires merely that the program be able to determine its relationship to other identification numbers-equal, not equal, less than, or greater than-binarization may be unnecessary. For even though the computer performs pure binary, not binary-coded decimal arithmetic, it can subtract one binary-coded decimal number from another and use the sign of the difference to indicate relative magnitudes. This is possible because the two numbers, as interpreted by the computer, retain the same relative magnitudes as they have when they are interpreted by people as binary-coded decimal numbers. For example, assume X and Y are two binary-coded decimal numbers at the same q and that Y is greater than X. When they are interpreted by the computer as binary numbers at a q of 0, Y will still be larger than X. The result of subtracting Y from X will, of course, be meaningless except for the sign. Note however, that this type of arithmetic is not possible when either of the binary-coded decimal numbers has a binary 1 in bit position zero, as the computer would then consider it a negative number.

Data binarization is also unnecessary for a one digit number and for data which, after being entered, becomes part of the output but is used in no other way.

Alphanumeric Data

When data consists of a combination of alphabetic and numeric characters—such as names, identification numbers which also contain alphabetic characters, or typewriter control codes-it is called alphanumeric. This kind of data must be stored in 6-bit form. That is, 6 instead of 4 bits must be stored for each character, since four bits can only represent 16 different characters which is obviously insufficient for all the numeric and alphabetic characters available.

Appendix C contains a list of all available characters and their 6-bit codes. The first four bits are called the numeric bits and the last two, the zone bits. Notice that, in some cases, two characters have the same four numeric bits and can be distinguished only by their zone bits. The programmer must specify for every

character which enters the computer whether he wants it recorded in memory in 4-bit or 6-bit mode. When entering strictly numeric data, the 4-bit mode should be used, as no two digits have the same numeric bits. However, for alpha-numeric data input the 6-bit mode should be used, so that distinction between characters with identifical four numeric bits is possible; e.g., between F and U. A 32-bit word can hold five alphanumeric characters. Example: "LGP21" in 6-bit format at a q of 29 appears as follows:

```
00011010111010000100101000011000
L     G     P     2     1
```

There are other forms of internal data representation (such as floating-point), but their discussion is not necessary in this manual.

HEXADECIMAL NOTATION

Since it is awkward to write thirty-two O's and l's, a shorthand or hexadecimal notation for writing computer words has been devised. To find the hexadecimal representation of a computer word, divide its 32 bits into eight groups of four bits each. There are 16 possible combinations for any group of four bits. Therefore, each combination of four bits can be represented by one of a group of 16 characters, zero through W, used for this purpose, as well as the decimal equivalent of the 4-bit numbers. This is shown in Figure 4.4.

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	F	10
1011	G	11
1100	K	12
1101	Q	13
1110		14
1111	W	15

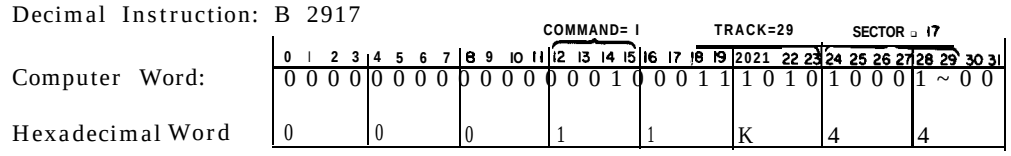
FIGURE 4.4 Hexadecimal Equivalences

Some examples follow:

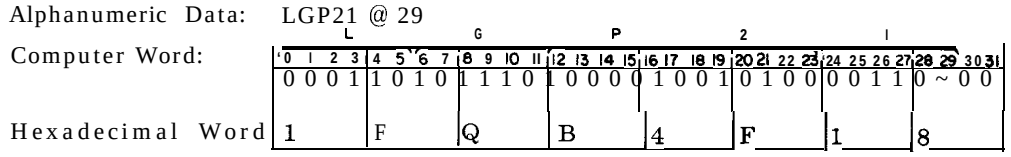
Decimal Number: 23.75 @ 14

	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31
Computer Word	0 0 0 0	0 0 0 0	0 0 1 0	1 1 1 ~	1 1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
Hexadecimal Word	0	0	2	W	8	0	0	0

Decimal Instruction: B 2917



Alphanumeric Data: LGP21 @ 29



Since the last character involves the spacer bit, it will normally be one of the eight even characters, which have zero as their fourth bit: 0, 2, 4, 6, 8, F, J, Q.

Decimal to Hexadecimal Conversion

A method for determining the appearance of numbers in the LGP-21 as sequences of thirty-two O's and l's was given earlier in this manual. Now a simpler method shall be explained which provides the eight hexadecimal characters which can be used to represent a given number at a given q.

Suppose 94.87654, at a q of 7, is to be expressed in hexadecimal. Two steps are required to find the first character:

1. Subtract the q of the given number from 3

   $3 - q = x$ therefore $3 - 7 = 4$

2. Evaluate 2^x and multiply this value by the given number:

   2^x(number) therefore $2^{-4}(94.87654) =$
   $(.0625)(94.87654) =$
   2.92978375

The first hexadecimal character is 5.

Each of the remaining characters requires a single process:

3. Multiply the fractional part of the previous product by 16 (always 16, regardless of q). The integer part of the new product is the next hexadecimal character.

Thus, in the example given:

```
.92978375          .30784
x       16         x   16
14.87654000        4.92544

.87654             .92544
x   16             x   16
14.02464           14.80704

.02464             .80704
x   16             x   16
0.39424            12.91264

.39424
x   16
6.30784
```

Since the hexadecimal characters equivalent to 14 and 12 are Q and J, respectively, the hexadecimal representation is

$$94.87654_{10} @ 7 = 5QQ064QJ$$

The fractional portion after the last multiplication, .91264, is greater than .5, so it may appear that the correct hexadecimal representation for 94.87654 at a q of 7 is closer to 5QQ064QK than to 5QQ064QJ. However, it may be recalled that the last character involves the spacer bit, and so must be even: 0, 2, 4, 6, 8, F, J, or Q. Therefore, 5QQ064QJ is the best possible approximation in the LGP-21.

This example illustrated a positive number. For negative numbers, one preliminary step is needed: subtract the negative number from the power of 2 which is 1 greater than the given q. For example, suppose the first two hexadecimal characters are to be found for the number -3.1415927 at a q of 3. First, the number must be subtracted from the power of 2 which is 1 greater than 3 (i.e., 2^4):

$$\begin{aligned} 2^4 &= 16.0000000 \\ \text{number} &= \underline{-3.1415927} \\ &\quad 12.8584073 \end{aligned}$$

Then, proceed as with positive numbers: multiply by $2^{3-3} = 2^0 = 1$. No multiplication is necessary for this step.

$$\underline{12}.8584073$$

Thus, the first character is J (decimal value 12).

$$\begin{array}{r} .8584073 \\ \times \quad 16 \\ \hline \underline{13}.7345168 \end{array}$$

The next character is K (decimal value 13), and so on.

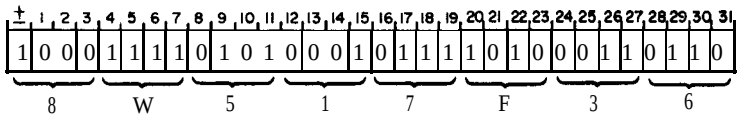
Hexadecimal Instruction Words

Instruction words as well as data words may be represented in hexadecimal. The Command portion of an instruction occupies bits 12 through 15 and can be represented by a hexadecimal character. A complete list of the LGP-21 commands and their hexadecimal designations is given in Figure 4.5.

COMMAND	BINARY	HEXADECIMAL	DECIMAL
Z	0000	0	0
B	0001	1	1
Y	0010	2	2
R	0011	3	3
I	0100	4	4
D	0101	5	5
N	0110	6	6
M	0111	7	7
P	1000	8	8
E	1001	9	9
U	1010	F	10
T	1011	G	11
H	1100	J	12
C	1101	K	13
A	1110	Q	14
S	1111	W	15

FIGURE 4.5 Hexadecimal Designation of Commands

Thus, the hexadecimal word 8W517F36 would appear in memory as



Since bits 12 through 15 are 0001 ($0001_2 = 1_{10}$), which is the binary equivalent of the Bring command, this word would be interpreted as a B instruction if it were to reach the Instruction Register.

The six bits, 18 through 23, contain the track portion of the operand address. In the above example, the bits are 111010. Their decimal equivalent is

etc. ← $1^{32}\ 1^{16}\ 1^{8}\ 0^{4}\ 1^{2}\ 0^{1}$ =

1 x 32 = 32
1 x 16 = 16
l x 8 = 8
o x 4 = 0
l x 2 = 2
o x 1 = 0
58

Therefore the track number is 58.

The next six bits, 24 through 29, contain the sector number. These bits are 001101, so the sector number is 13, according to the same conversion process:

etc. ← $0^{32}\ 0^{16}\ 1^{8}\ i\ 0^{2}\ i$ =

0 x 32 = 0
0 x 16 = 0
l x 8 = 8
l x 4 = 4
o x 2 = 0
l x 1 = 1
13

In summary, the hexadecimal word 8W517F36 is treated as a B5813 instruction, if it is in the instruction register.