Chapter Three: Data Representation

The organization of any computer depends considerably on how it represents numbers, characters, and control information. The converse is also true: Standards and conventions established over the years have determined certain aspects of computer organization. There are various ways in which computers can store and manipulate numbers and characters.
The most basic unit of information in a digital computer is called a *bit*, which is a contraction of *binary digit*. In the concrete sense, a bit is nothing more than a state of "on" or "off " (or "high" and "low") within a computer circuit.
We enter data into a computer or review (see) output data from a computer using the letter of alphabet, various special symbols, and the numerals in the decimal number system. But, since computer is an electronic device, which understands electrical flow (signal): there is no letter, symbol or number inside the computer. Computer works with binary numbers. As a semiconductor is conducting or isn't conducting; a switch is closed or opened. So data are represented in the form of a code that can have a corresponding electrical signal.

3.1 The Number System

A number system is a set of symbols used for counting. There are various number systems: Decimal, Binary, octal, hexadecimal etc. For the purpose of understanding how data are represented, stored and processed in computer, let us discuss the decimal and binary number systems.

The Decimal Number System: The Decimal number system is based on the ten different digits (or symbols) 0,1,2,3,4,5,6,7,8,9. We say that it is a base ten number. Though it is widely used, it is inconvenient for computer to represent data. So we need another number system called the Binary Number System.
Binary number system is based on the two different digits: 0 and 1. It is important to note that every decimal number system has its equivalent binary number. Conversion from binary to its equivalent decimal and from decimal to its equivalent binary is possible. Operation on binary number system is also possible.

Binary	**Decimal**	1	1
0	0	10	2

11	3
100	4

Binary	Decimal
101	5
110	6
111	7
1000	8
1001	9 etc.

The most elementary form to organize data within a computer (an electronic device) is in the form of a code which utilizes the "ON" and "OFF" states of electric switches or there is "current" and "no current" condition of the electronic components.

We see that the nature of the electronic devices has similarity with the binary number system in that both represent only two elementary states. It is therefore convenient to use binary number system to represent data in a computer. An "ON" corresponds to a 1. An "OFF" corresponds to a 0. In the computer "ON" is represented by the existence of a current and "OFF" is represented by non existence of current. On a magnetic disk, the same information is stored by changing the polarity of magnetized particles on the disk's surface.

Coding Methods

It is possible to represent any of the character in our language in a way as a series of electrical switches in arranged manner. These switch arrangements can therefore be coded as a series of equivalent arrangements of bits. There are different coding systems that convert one or more character sets into computer codes. Some are: **EBCDIC, BCD, ASCII-7 & ASCII-8.**

In all cases, binary coding schemes separate the characters, known as character set, in to zones. Zone groups characters together so as to make the coding scheme to decipher and the data easier to process. Within each zone, the individual characters are identified by digit code.

EBCDIC: Pronounced as "Eb-see-dick" and stands for Extended Binary Coded Decimal Interchange Code. Used usually for IBM Main frame model and in similar machines produced by other manufacturer.

- It is an 8-bit coding scheme: (00000000 – 11111111).
- It accommodates to code 2^8 or 256 different characters.
- It is a standard coding scheme for the large computers.

Binary Coded Decimal (BCD)

There were two types of **BCD** coding techniques used before. The **4 bit BCD**, which represent any digit of decimal number by four bits of binary numbers. If you want to represent 219 using 4 bit BCD you have to say 0010 0001 1001

- 4 bits BCD numbers are useful whenever decimal information is transferred into or out of a digital system. Examples of BCD systems are digital voltmeter, and digital clocks; their circuits can work with BCD numbers.
- BCD's are easy for conversion but slower for processing than binary. And they have limited numbers because with

BCD we can represent only numbers 0000 for 0 and 1001 for 9 and ,1010,1011,1100,1101,1110, 1111 can't be used because 1010 represent 10 in decimal and 10 in decimal is 1010 0000 in BCD.

BCD (6-bits): It uses 6-bits to code a Character (2 for zone bit and 4 for digit bit) it can represent 2^6 = 64 characters (10 digits, 26 capital characters and some other special characters).

The ASCII System

ASCII-7: ASCII stands for American Standard Code for Information Interchange. Used widely before the introduction of ASCII-8 (the Extended ASCII). It uses 7 bits to represent a character. With the seven bits, 2^7(128) different characters can be coded (0000000-1111111)

Also referred as ASCII-8 or Extended ASCII .It is the most widely used type of coding scheme for Micro Computer system. It uses 8-bits to represent alphanumeric characters (letters, digits and special symbols).

With the 8-bits, ASCII can represent 2^8 or 256 different characters (00000000-11111111).

Units of Data Representation

When data is stored, processed or communicated within the computer system, it is packed in units. Arranging from the smallest to the largest, the units are called **bit, byte** and **word**.

These units are based on the binary number system.

1. **Bit:** stands for binary digits.

A bit is a single element in the computer, on a disk that stands for either "ON" indicating 1 or "OFF" indicating 0. In the computer "ON" is represented by the existence of current and "OFF" is represented by the non-existence of current.

2. **Byte:** Bits can be organized into large units to make them represent more and meaningful information.

This large unit is called a byte and is the **basic "unit of data representation**" in a computer system. The IBM designers established a convention of using groups of 8 bits as the basic unit of addressable computer storage. They called this collection of 8 bits a **byte.**

The commonly used byte contains 8 bits. Since each bit has two states and there are 8 bits in a byte, the total amount of data that can be represented is 2^8 or 256 possible combinations. Each byte can represent a character (a character is either a letter, a number or a special symbol such as +,-,?,*, $, etc)
A byte is then used as a unit of measurement in the computer memory, processing unit, external storage and during communication.

1 Kilobyte (1KB) is 2^{10} or 1024 bytes

1 Megabyte (MB) is 2^{20} bytes or 2^{10} kilobytes

1 Gigabyte (GB) is 2^{30} bytes or 2^{20} kilobytes or 2^{10} megabytes

3. Word: refers the number of bits that a computer process at a time or a transmission media transmits at a time. Although bytes can store or transmit information, the process can be faster if more than one byte is processed at a time. Computer ***words*** consist of two or more adjacent bytes that are sometimes addressed and almost always are manipulated collectively. The **word size** represents the data size that is ***handled most*** efficiently by a particular architecture.

- A combination of bytes, then form a "**Word"**
- A word can contain one, two, three or four bytes based on the capacity of the computer.
- Word length is usually given in bits
- We say that a computer is an 8-bits, a 16 bit, a 32 bit a 64 bit or above to indicate that the amount of data it can process at a time
- The large the word length a computer has the more powerful and faster it is.
- Eight-bit bytes can be divided into two 4-bit halves called ***nibbles*** **(or *nybbles*).**
- Because each bit of a byte has a value within a positional numbering system, the nibble containing the least-valued binary digit is called the low-order nibble, and the other half the high-order nibble.
- Bits are the smallest units and can convey only two possible states 0 or 1

Conversion from Decimal to Any System

Number systems

There are different number systems. Some of are:

- Decimal number systems
- Binary number systems

- Octal number systems
- Hexadecimal number systems

Generally to convert a decimal number X to a number in base b, divide X by b, store the remainder, again divide the quotient by b, store the remainder, and continue until the quotient is 0. And concatenate (collect) the remainders starting from the last up to the first.

Examples: Convert

1. $(56)_{10}$ to base two (binary): X=56 b=2 Therefore $(56)_{10} = (111000)_2$
2. $(78)_{10}$ to base eight (Octal): X=78 b=8 Therefore $(78)_{10} = (116)_8$

The Octal Number System (Base 8)/Oct

It uses 8 symbols 0-7 to represent numbers: Like binary number system it is complete number system. **Example**: 77 in octal equals 49 in decimal and 111111 in binary.

When we compare the octal with the decimal, 0-7 in octal is the same as 0-7 in decimal but 10 in octal is not the same as 10 in decimal because 10 in octal holds the position of 8 in decimal, off course 10 in octal is the same as 8 in decimal.

Hexadecimal Number System (base16)/ hex

It uses 16 symbols to represent numbers. But for the numbers greater than 15 they represented in terms of the 16 symbols. For example the decimal number 16 represented as 10, 20 as 14, 30 as 1E and so on. These symbols are 0, 1,…, 9, A,B,C,D,E,F. The number of values that can be expressed by a single digit in any number system is called the system radix, and any value can be expressed in terms of its system radix.

Octal to Decimal: For example the system radix of octal is 8, since any of the 8 values from 0 to 7 can be written as a single digit.

Convert $(126)_8$ to decimal: Using the values of each column, (which in an octal integer are powers of 8) the octal value $(126)_8$ can also be written as:

$(1x8^2) + (2x8^1) + (6 \times 8^0)$, As $(8^2 = 64)$, $(8^1 = 8)$ and $(8^0 = 1)$, this gives a multiplier value for each column.

Multiply the digit in each column by the column multiplier value for that column to give:

$1x64 = 64 \quad 2x8 = 16 \quad 6x1 = 6$

Then simply add these results to give the decimal value: 64 + 16 + 6 = $(86)_{10}$, **Therefore $(126)_8 = (86)_{10}$.**

Binary to Decimal

Convert $(1101)_2$ to decimal.

The same method can be used to convert binary number to decimal:

$= (1x2^3)+(1x2^2)+(0x2^1)+(1x2^0)$

$= 8 + 4 +0 +1 = (13)_{10}$, Therefore **$(1101)_2 = (13)_{10}$.**

Hexadecimal to Decimal

Convert $(B2D)_{16}$ to decimal.

Using the same method to convert hexadecimal to decimal.

$= (Bx16^2)+(2x16^1)+(Dx16^0)$

$= (11x16^2)+(2x16^1)+(13x16^0)$

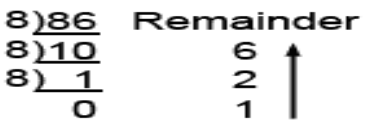
$= 2816 + 32 +13 =$ **$(2861)_{10}$, Therefore $(B2D)_{16} = (2861)_{10}$.**

The same method (multiplying each digit by it's column value) can be to convert any system to decimal.

EXERCISE: Try these conversions to decimal WITHOUT USING YOUR CALCULATOR FOR THE ACTUAL CONVERSION: A) $(110)_2$ B) $(67)_8$ C) $(AFC)_{16}$ D) $(FC)_{16}$

Converting from Decimal to any Radix: To convert a decimal integer number (a decimal number in which any fractional part is ignored) to any other radix, all that is needed is to continually divide the number by its radix, and with each division, write down the remainder. When read from bottom to top, the remainder will be the converted result.

Decimal to Octal

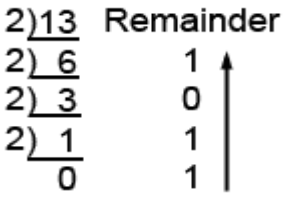


Eg: Decimal to Octal Conversion

For example, to convert the decimal number **$(86)_{10}$ to octal:** Divide $(86)_{10}$ by the system radix, which when converting to octal is 8. This gives the answer 10, with

a remainder of 6. Continue dividing the answer by 8 and writing down the remainder until the answer = 0. Now simply write out the remainders, starting from the bottom, to give $(126)_8$, **Therefore $(86)_{10} = (126)_8$**

Decimal to Binary

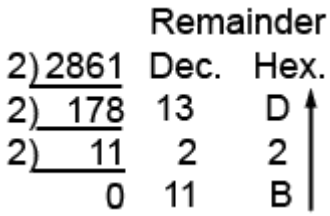


➔$(13)_{10}=(1101)_2$

Example: Decimal to Binary Conversion

This process also works to convert decimal to binary, but this time the system **radix is 2**. For example, to convert the decimal number $(13)_{10}$ to binary. **Therefore $(13)_{10} = (1101)_2$**

Decimal to Hexadecimal



Example: Decimal to Hexadecimal Conversion

It also works to convert decimal to hexadecimal, but now the **radix is 16**. As some of the remainders may be greater than 9 (and so require their alphabetic replacement), you may find it easier to use Decimal for the remainders, and then convert them to Hex. **Therefore $(2861)_{10} = (B2D)_{16}$**

Numbers with Fractions

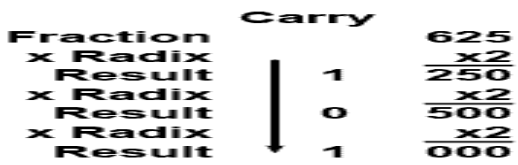
It is very common in the decimal system to use fractions; that is any decimal number that contains a decimal point, but how can decimal numbers, such as $(34.625)_{10}$ be converted to binary fractions?
In electronics this is not normally done, as binary does not work well with fractions. However as fractions do exist, there has to be a way for binary to deal

with them. The method used is to get rid of the **radix** (decimal) point by NORMALISING the decimal fraction using FLOATING POINT arithmetic. As long as the binary system keeps track of the number of places the radix point was moved during the normalisation process, it can be restored to its correct position when the result of the binary calculation is converted back to decimal for display to the user.
However, for the sake of completeness, here is a method for converting decimal fractions to binary fractions. By carefully selecting the fraction to be converted, the system works, but with many numbers the conversion introduces inaccuracies; a good reason for not using binary fractions in electronic calculations.

Converting the Decimal Integer to Binary

Example: **Converting the Integer to binary**: The radix point splits the number into two parts; the part to the left of the radix point is called the **INTEGER**. The part to the right of the radix point is the **FRACTION**. A number such as **$(34.625)_{10}$** is therefore split into **$(34)_{10}$ (the integer),** and **$(.625)_{10}$ (the fraction)**.To convert such a fractional decimal number to any other radix, the method described above is used to covert the integer. **So $(34)_{10} = (100010)_2$**

Converting the Decimal Fraction to Binary



Example : Converting the Fraction to Binary

✓To convert the fraction, this must be **MULTIPLIED** by the radix (in this case 2 to convert to binary). Notice that with each multiplication a **CARRY** is generated from the third column. The Carry will be either 1 or 0 and these are written down at the left hand side of the result. However when each result is multiplied the carry is ignored (don't multiply the carry). Each result is multiplied

in this way until the result (ignoring the carry) is 000. Conversion is now complete.

✓For the converted value just read the carry column from top to bottom.

So $(0.625)_{10} = (.101)_2$. Therefore the complete conversion shows that $(34.625)_{10} = (100010.101)_2$

❖ However, with binary, there is a problem in using this method, .625 converted easily but many fractions will not. For example if you try to convert .626 using this method you would find that the binary fraction produced goes on to many, many places without a result of exactly 000 being reached. With some decimal fractions, using the above method will produce carries with a repeating pattern of ones and zeros, indicating that the binary fraction will carry on infinitely. Many decimal fractions can therefore only be converted to binary with limited accuracy. The number of places after the radix point must be limited, to produce as accurate an approximation as required.

Quick Conversions: The most commonly encountered number systems are binary and hexadecimal, and a quick method for converting to decimal is to use a simple table showing the column weights, as shown in Tables 1.2.1a and 1.2.1b.

Converting Binary to Decimal

Table 1.2.1a

Bit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value (weighting) of each bit	128	64	32	16	8	4	2	1
8 Bit Binary	0	1	0	0	0	0	1	1

To convert from binary to decimal, write down the binary number giving each bit its correct 'weighting' i.e. the value of the columns, starting with a value of one for the right hand (least significant) bit. Giving each bit twice the value of the previous bit as you move left.

Example: To convert the binary number $(01000011)_2$ to decimal. Write down the binary number and assign a 'weighting' to each bit as in Table 1.2.1a . Now simply add up the values of each column containing a 1 bit, ignoring any columns containing 0.

➔Applying the appropriate weighting to 01000011 gives 64 + 2 + 1 = 67

Therefore: $(01000011)_2 = (67)_{10}$

Table 1.2.1b				
Column	16^3	16^2	16^1	16^0
Value (weighting) of each column	4096	256	16	1
Hex value	2	5	C	B

$$
\begin{aligned}
2 \times 4096 &= 8192 \\
5 \times 256 &= 1280 \\
C\ (12_{10}) \times 16 &= 192 \\
B\ (11_{10}) \times 1 &= 11 \\
&\underline{\quad 9675}
\end{aligned}
$$

Converting Hexadecimal to Decimal

A similar method can be used to quickly convert hexadecimal to decimal, using Table 1.2.1b. The hexadecimal digits are entered in the bottom row and then multiplied by the weighting value for that column. Adding the values for each column gives the decimal value. **Therefore: $(25CB)_{16} = (9675)_{10}$**

Exercise 1: Convert:

1. **$(11010011)_2$ to decimal.**
2. **$(10111011)_2$ to decimal.**
3. **$(34F2)_{16}$ to decimal.**
4. **$(FFFF)_{16}$ to decimal.**

Exercise 2: Check your answer by converting the decimal back to binary or hexadecimal.

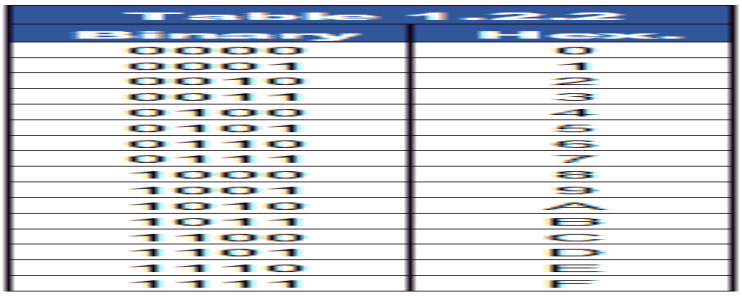


Table 1.2.2	
Binary	Hex.
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Binary and Hexadecimal

Converting between binary and hexadecimal is a much simpler process; hexadecimal is really just a system for displaying binary in a more readable form. Binary is normally divided into Bytes (of 8 bits) it is convenient for machines but quite difficult for humans to read accurately. Hexadecimal groups each 8-bit byte into two 4-bit nibbles, and assigns a value of between 0 and 15 to each nibble.

Therefore each hexadecimal digit (**also worth 0 to 15**) can directly represent one binary nibble. This reduces the eight bits of binary to just two hexadecimal characters.

For example: $(11101001)_2$ is split into 2 nibbles $(1110)_2$ and $(1001)_2$ then each nibble is assigned a hexadecimal value between 0 and F.

The bits in the most significant nibble (1110_2) add up to $8+4+2+0 = 14_{10} = (E)_{16}$
The bits in the least significant nibble (1001_2) add up to $8+0+0+1 = 9_{10} = (9)_{16}$
Therefore $(11101001)_2 = (E9)_{16}$, Converting hexadecimal to binary of course simply reverses this process.

3.2 Binary Arithmetic

Computer understands only the language of binary numbers. Therefore, the machine performs what is called binary arithmetic (binary computation).

3.2.1. Binary Addition:

Binary addition operates by the same rule as decimal addition, except that it is simpler. A carry to the next higher order (or more significant) position occurs when the sum is decimal 2, that is, binary 10. Therefore, the binary addition rules may be written as follows:

0+0=0, 0+1=1, 1+0=1, 1+1=0 plus a carry of 1 into the next position.
1+1+1=1 plus a carry of 1 into the next position. The last case occurs when the two binary digits in a certain position are 1s and there is a carry from the previous position

Example 1: 6+7 =13 ➔110+111=1101

Example 2: 19+31+10=60 ➔10011 +11111+1010=111100

3.2.2. Binary Subtraction

It operates by the same rule as decimal subtraction. The rule is as follows;
0-0=0, 1-0=1, 1-1=0, 10-1=1

Example:

11100	28	101101	45	25.375	11001.011
-11010	-26	- 111	-7	-7.750	111.110
00010 =	2	101100 =	38	38	10001.101

3.2.3. Binary Multiplication

It is a very simple process that operates by the following obvious rulers:

A. Multiplying any number by 1 rules the multiplicand unchanged: **1) 0x1=0 and 2) 1x1=1**

B. Multiplying any number by 0 produces 0: **1) 0x0=0 and 2) 1x0=0**

3.2.4. Binary division

That is the process of dividing one binary number by another.
is based on binary subtraction and multiplication like decimal

e.g.1111101 ÷11001

11001 ÷101
11001
11001
00000
1111101 ÷ 11001 = 101

3.2.5. Representation of Negative numbers

There are different ways of representing negative numbers in a computer.

I. Sign- Magnitude Representation

In signed binary representation, the left-most bit is used to indicate the sign of the number. Traditionally, **0** is used to denote a positive number and **1** is used to denote a negative number. But the magnitude part will be the same for the negative and positive values. For example, **1**1111111 represents **-127** while **0**1111111 represents **+127**. We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127.

In a 5- bit representation we use the first bit for sign and the remaining 4- bits for the magnitude. So using this 5 bit representation the range of number that can be represented is from -15 (11111) to 15(01111)

e.g.1. Represent-12 using 5-bi sign magnitude representation. First we convert 12 to binary i. e 1100, Now -12 = 11100

e.g.2. Represent –24 using 8-bits

24=00011000 and **-24** = 1001100

In general for n bits the sign-magnitude representation the range of values that can be represented are-**$(2^{n-1}-1)$ to $(2^{n-1}-1)$. i. e $2^{n-1}+1$ to $2^{n-1}-1$.**

Note: In sign magnitude representation zero can be represented as 0 or -0. This representation has two problems one is it reduces the maximum size of magnitude, and the second one is speed efficiency to perform arithmetic and other operations. For sign magnitude representation, correct addition and subtraction are relatively complex, involving the comparison of signs and relative magnitude of the two numbers. The solution to this problem is called the two's complement representation.

3.2.6. One's Complement

In one's complement representation, all positive integers are represented in their correct binary format. For example +3 is represented as usual by 00000011 using 8 bits. However, its complement, -3, is obtained by complementing every bit in the original representation. Each 0 is transformed into 1 and each 1 into 0. In our example, the one's complement representation of -3 is 11111100.

e.g. +2 is 00000010, -2 is 11111101

Note that in this representation positive numbers start with a 0 on the left, and negative numbers start with a 1 on the left most bit.

Example: add

1. –3 and 3 with word size 4 bits

3 = 0011

-3=1100

sum =1111 (=0)

2. -4 and +6

- 4 is 11111011

+ 6 is 00000110

the sum is (1) 00000001 the one in the parenthesis is the external carry; where 1 indicates a carry.

The correct result should be 2 or 00000010. In one's complement addition and subtraction, if there is an external carry it should be added to get the correct result. This indicates it requires additional circuitry for implementing this operation.

3.2.7. Two's Complement Representation

In two's complement representation, positive numbers are represented, as usual, in singed binary, just like in one's complement. The difference lies in the

representation of negative numbers. A negative number represented in two's complement is obtained by first computing the one's complement and then add one.

e.g.: **+3** is represented in signed binary as 00000011. Its one's complement representation is 11111100. The two's complement is obtained by adding one. It is 11111101.

e.g 1.: let's try addition.

(3) 00000011
+ (5) +00000101
(8) 0001000 The result is correct

e.g. 2. Let's try subtraction

(3) 00000011
(-5) + 111111011
11111110

e.g. 3. add +4 and -3(the subtraction is performed by adding the two's complement).

+4 is 00000100
-3 is 111111101
The result is [1] 000000001

If we ignore the external carry the result is 00000001 (i. e 1 In decimal). This is the correct result. In two's complement, it is possible to add or subtract signed numbers, regardless of the sign. Using the usual rules of binary addition, the result comes out.

3.2.8. Floating-Point Representation

In this representation decimal numbers are represented with a fixed length format. In order not to waste bits, the representation will normalize all the numbers. For example, 0.000123 wastes three zeroes on the left before non -zero digits. These zeroes have no meaning except to indicate the position of the Decimal point. Normalizing this number result in $.123 \times 10^{-3}$ .123 is the normalized mantissa; -3 is the exponent. We have normalized this by eliminating all the meaningless zeroes to the left of the first non-zero digit and by adjusting the exponent.

e.g. 1: 22.1 is normalized as $.221 \times 10^{2}$.

The general form of floating point representation is $\pm M \times 10^{\pm E}$ where M is the mantissa, and E is the exponent. It can be seen that a normalized number is

characterized by a mantissa less than 1 and greater than or equal to.1 all cases when the number is not zero. To represent floating numbers in the computer system it should be normalized after converting to binary number representation system.

e.g.2: 111.01 is normalized as .11101x2^{3}. The mantissa is 11101. The exponent is 3.

The general structure of floating point is Sign Exponent Mantissa

In representing a number in floating point we use 1 bit for sign, some bits for exponent and the remaining bit for mantissa. In floating point representation the exponent is replaced by biased exponent (Characteristics).

Biased exponent = true exponent + excess 2^{n-1}, where n is the number of bits reserved for the exponent.

e.g. Represent –234.375 in floating point using 7 bit for exponent and 16 bit for Mantissa.

First we have to change to normalized binary i. e 234 = 11100010 and 0.375= 0.011

234.375 = 11100010.011 = 0.11100010011x2^{8} with the true exponent = 8, excess $2^{n-1} = 2^{7-1} = 2^{6} = 64$

➔Biased Exponent = 8+64 = 72 = $(1001000)_2$ .Therefore, –234.375 is represented as

1	1001000	1110001001100000
Sign	7 bits (Exponent)	16(Mantissa)