CHAPTER 2

Data Representation in Computer System

Learning objectives

- Introduction
- Positional Numbering Systems
- Conversion Between Bases
- Signed Integer Representation
- Floating-point Representation
- Character Codes

INTRODUCTION

The computer organization is depends considerably on how the data representation of numbers, characters and control information. This chapter describes the various methods in which computers can store and manipulate numbers and characters.

The most basic unit of information in digital computer is called a bit, which is a contraction of binary digit. The binary number system and digital codes are fundamental to computers. In this chapter, the binary number system and its relationship to other systems such as decimal, hexadecimal, and octal are introduced. Arithmetic operations with binary numbers are discussed to provide a basis for understanding how computers and many other types of digital systems work.

2.3. SIGNED INTEGER REPRESENTATION

The unsigned integer representing numbers from 0 to upwards and signed integers allow for negative number representation. In the computer have we only binary digits, so to representation of negative integers we need some sort of convention. There are four conventions in use for representing negative integers are:

1. Sign Magnitude
2. 1's Complement
3. 2's Complement
4. Excess (or Bias) Representation

2.3.1. Sign Magnitude

In this representation, in an n bit word, the rightmost n – 1 bits hold the magnitude of the integer number, and the most significant bit is the sign bit. The value of the sign bit is 0 for positive numbers and 1 for negative numbers.

Some sample representations follows for binary numbers.

EXAMPLE 2.37.

- +6 in 8-bit representation is: 00000110
- –6 in 8-bit representation is: 10000110
- +7 in 8-bit representation is: 00000111
- –7 in 8-bit representation is: 10000111
- +15 in 8-bit representation is: 00001111
- –15 in 8-bit representation is: 10001111

This representation also can be implements on the other number systems, if number uses n digits to represent a number, than MSD (most significant digit) is the sign digit and the remaining n-1 digits are magnitude digits. The value of the sign digit is 0 for positive numbers and r – 1 for negative numbers, where r is the radix of the number system. Some sample representations follow.

EXAMPLE 2.38.

Here, we assume that five digits are available to represent each number. The sign and magnitude portions of the number are separated by "," for illustration purposes only. The "," is not used in the actual representation.

Number	Represetation
$(-2)_2$	1,0010
$(+56)_8$	0,0056
$(-56)_8$	7,0056
$(-1F)_{16}$	0,001F
$(-1F)_{16}$	F,001F

Sign Magnitude

All numbers are shown as five-digit numbers

The sign and magnitude portions are handled separately in arithmetic using sign-magnitude numbers. The magnitude of the result is computed and then the appropriate sign is attached to the result, just as in decimal arithmetic. The sign magnitude system has been used in such small digital systems as digital meters and typically when the decimal mode of arithmetic is used in digital computers. Complement number representation is the most prevalent representation mode in modern-day computer systems.

Disadvantages of sign magnitude system are:

- Arithmetic operations are difficult in this system
- Sign magnitude system has two representations for zero, which is wrong.
  - o +0 in 8-bit representation is: 00000000
  - o –0 in 8-bit representation is: 10000000

2.3.2. One's Complement

Computers generally use a system called "complementary representation" to store negative integers. Two basic types -

- One's complement and
- Two's complement,

where 2's complement is the most widely used.

The number range is split into two halves, to represent the positive and negative numbers. Negative numbers begin with 1, positive with 0.

To perform 1's complement operation on a binary number, replace 1's with 0's and 0's with 1's (ie Complement it!) for example:

- +6 in 8-bit representation is: 00000110
- -6 in 8-bit representation is: 11111001 (which is complement of +6)
- +7 in 8-bit representation is: 00000111
- -7 in 8-bit representation is: 11111000
- +15 in 8-bit representation is: 00001111
- -15 in 8-bit representation is: 11110000

Advantages of 1's complement:

- Arithmetic is easier (cheaper/faster electronics)
- Fairly straightforward addition
  - – Add any carry from the Most Significant (left-most) Bit to Least Significant (right most) Bit of the result
- For subtraction
  - – form 1's complement of number to be subtracted and then add

Disadvantages of 1's complement:

- still 1's complement has two representations for zero
  - – 00000000 and 11111111 (in 8-bit representation)

2.3.3. Two's Complement

Another form of negative representation is 2's complement. To perform the 2's complement operation on a binary number, replace 1's with 0's and 0's with 1's (i.e. the one's complement of the number) and than add 1, for example:

+6 represented by: 00000110

–6 represented by: 11111010 (2's complement of +6)

Small Concept

2's complement used for negative notations.

To understand how 2's complement has been performed see next example:

EXAMPLE 2.38.

$(-6)_{10} = (?)_2$

+6 represented b	:	00000110
1's complement	:	11111001
Add 1	:	+1
2's complement	:	11111010

Advantages of 2's complement:

- Arithmetic is very straightforward
- End Around Carry is ignored
- only one representation for zero (00000000)

2.3.4. Excess or Bias Representation

The excess-128 is a representation for 8-bit signed numbers; excess-8 is a representation for 4-bit signed numbers. Therefore we can say excess 2m-1 is representation used for m-bit numbers.

The excess-128 is stored as the true value with addition of 128, for example:

EXAMPLE 2.39.

$(-3)_{10} = (?)_2$ using excess-128

$= -3 + 128 = 125 = (01111101)_2$

$(-3)_{10} = (01111101)_2$

EXAMPLE 2.40.

$(26)_{10} = (?)_2$ using excess-128

$= 26+128 = 154 = (10011010)$

$(26)_{10} = (10011010)_2$

The excess-8 is stored as the true value in 4 bits ($2^{3-1=2} = 2^2$ bits) with addition of 8, for example

EXAMPLE 2.41.

$$(-3)_{10} = (?)_2 \text{ using excess-8}$$
$$= -3 + 8$$
$$= 5$$
$$= (1101)_2$$
$$(-3)_{10} = (1101)_2$$

EXAMPLE 2.42.

$$(-11)_{10} = (?)_2 \text{ using excess-8}$$
$$= -11 + 8$$
$$= -3$$
$$= (0101)$$
$$(-11)_{10} = (0101)_2$$

2.4. FLOATING-POINT REPRESENTATION

Fixed-point representation is convenient for representing numbers with bounded orders of magnitude. For instance, in a digital computer that uses 32 bits to represent numbers, the range of integers that can be used is limited to ± (231-1), which is approximately ±1012. Integer formats are not useful in scientific or business applications that deal with real (fractional) number values.

In scientific computing environments, a wider range of numbers may be needed, and fraction numbers are also has more importance. Fractional numbers are very large or very small numbers can be represented with only a few digits by using scientific notation. For example:

$$976{,}000{,}000{,}000{,}000 = 9.76 \times 10^{14}$$
$$0.0000000000000976 = 9.76 \times 10^{-14}$$

The fraction numbers are used as a floating-point representation in digital computer systems. The general form of a floating-point representation number N is

$$N = \pm F \times R \pm E,$$

where

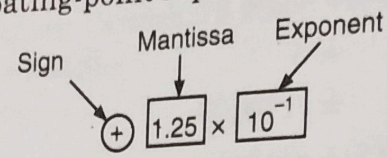
- Sign - plus or minus
- Mantissa or Fraction F (often called the significand)
- Exponent E (includes exponent sign)
- The radix R.

Consider the number

$$N = 7860000$$
$$= (0.786) \times 10^7 = (0.0786) \times 10^8$$
$$= (7.86) \times 10^6$$

The last three forms are valid floating-point representations. Among them, the first two forms are preferred; however, since with these forms there is no need to represent the integer

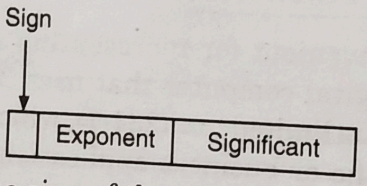
portion of the mantissa, which is 0. The first form requires the fewest digits to represent the mantissa, since all the significant zeros have been eliminated from the mantissa. This form is called the normalized form of floating-point representation.



Note from the example above that the radix point floats within the mantissa incrementing the exponent by 1 for each move to the left and decrementing the exponent by 1 for each move to the right. This shifting of the mantissa and scaling of the exponent is frequently done in the manipulation of floating-point numbers.

In digital computers floating-point numbers consist of three parts: a sign bit an exponent part (representing the exponent on power of 2), and a fractional part called significand (mantissa). The number of bits used for the exponent and significand depends on whether we would like to optimize for range (more bits in the exponent) or precision (more bits in the significand).

Computer representation of a floating-point number consists of three fixed-size fields:



- The one-bit sign field is the sign of the stored value.
- The size of the exponent field, determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

Such as The IEEE-754 *single precision* floating point standard uses an 8-bit exponent and a 23-bit significand. The IEEE-754 *double precision* standard uses an 11-bit exponent and a 52-bit significand.

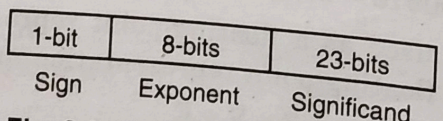


Fig. 2.2. IEEE-754 single precision

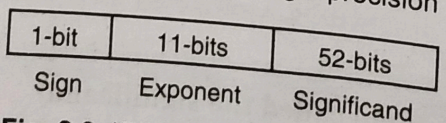


Fig. 2.3. IEEE-754 double precision

For illustrative purposes, (and to keep working and understanding VERY simple) we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

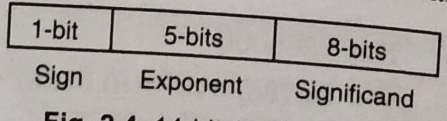


Fig. 2.4. 14-bit Simple model

The significand of a floating-point number is always preceded by an implied binary point. The significand contains a fractional binary value. The exponent is the power of 2 to which the significand is raised.

For Example:

- Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32_{10} is 2^5.
- So in (binary) scientific notation $32_{10} = 1.0 \times 2^5 = 0.1 \times 2^6$.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.

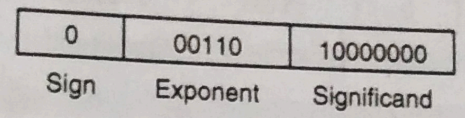


Fig. 2.5. 32_{10} in 14-bit Simple model

One obvious problem with this model is that we haven't provided for negative exponents. If we wanted to store 0.25 we would have no way of doing so because 0.25 is 2^{-2} and the exponent -2 cannot be represented.

One more larger problem with this model is that we do not have unique representation for each number. The illustrations given below are all equivalent representations for 32 using our simplified model.

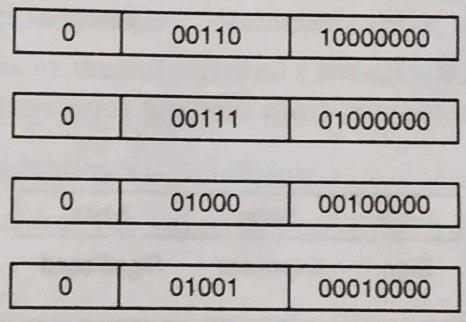


Fig. 2.6. Different representation of 3210 in14-bit Simple model

Not only do these synonymous representations waste space, but they can also cause confusion.

To resolve the issue of synonymous forms, we'll use a rule that the first digit of the significand must be 1. This gives a unique pattern for each number. In the IEEE-754 standard, this 1 is implied meaning that a 1 is assumed after the binary point. By using an implied 1, we increase the precision of the representation by a power of two.

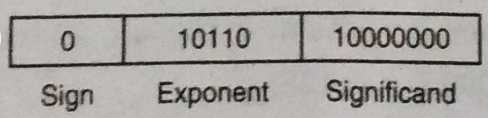
To provide for negative exponents, we'll use excess or bias representation for exponent (refer section 2.3.4). We have a 5-bit exponent. We'll use 16 for our bias. This is called excess-16 representation.

In our model, exponent values less than 16 are negative, meaning that the number we're representing is negative.

EXAMPLE 2.43.

Express 32_{10} in the revised 14-bit floating-point model.

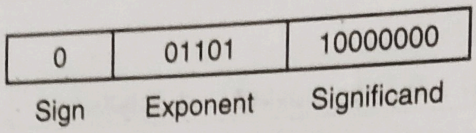
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} ($=10110_2$).



EXAMPLE 2.44.

Express 0.0625_{10} in the revised 14-bit floating-point model.

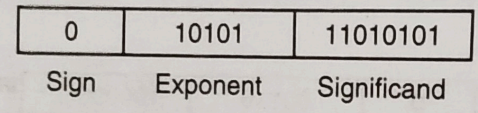
- We know that 0.0625 is 2^{-4}.
- So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2 - 3$.
- To use our excess 16 biased exponent, we add 16 to –3, giving 13_{10} (=01101_2).



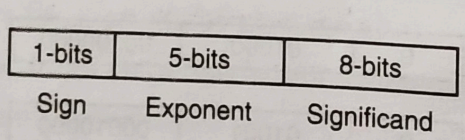
EXAMPLE 2.45.

Express -26.625_{10} in the revised 14-bit floating-point model.

- We find $26.625_{10} = 11010.101_2$.
- Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} (=10101_2).
- We also need a 1 in the sign bit.

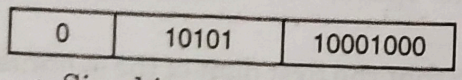


Floating Point to Decimal : As we know the format of our 14-bit simple is as given in figure 2.4, there for we can also convert from floating representation to decimal number, lets consider some examples:



EXAMPLE 2.46.

$$11010110001000_2 = (?)_{10}$$

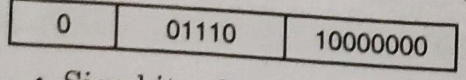


- Sign bit = 0 (positive)
- Exponent = 5 (10101 – 10000 = 5)
- Significand = .10001000
- We shift the decimal point 5 positions giving us 10001.0 = +17

$$11010110001000_2 = (+17)_{10}$$

EXAMPLE 2.47.

$$00111010000000_2 = (?)_{10}$$



- Sign bit = 0 (positive)
- Exponent = –2 (01110 – 10000 = – 2)
- Significand = .10000000
- We shift the decimal point 2 positions to the left, giving us 0.001 = +.125

$$00111010000000_2 = (+125)_{10}$$

EXAMPLE 2.48.

$$11001111010100_2 = (?)_{10}$$

1	10011	11010100

- Sign bit = 1 (negative)
- Exponent = 3 (10011 – 10000 = 3)
- Significand = .11010100
- We shift the decimal point 3 positions to the right, giving us 110.101 = –6.625

$$11001111010100_2 = (-6.625)_{10}$$

2.4.1. The IEEE-754 Floating-Point Standard

In the above section we have used 14-bit floating-point model for better conceptual understanding and for simplicity. In 1985, the Institute of Electrical and Electronic Engineers (IEEE) published a floating-point standard for both single- and double-precision floating-point numbers. This standard officially known as IEEE-754.

IEEE-754 Single Precision

The IEEE-754 single precision standard for floating point has following features:

- The IEEE-754 single precision uses bias of 127 for its 8-bit exponent.
- An exponent of 255 indicates a special value.
- If the significand is zero, the value is infinity.
- If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.

IEEE-754 double precision

The IEEE-754 double precision standard for floating point has following features:

- The double precision standard has a bias of 1023 for its 11-bit exponent.
- The "special" exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.
- If the significand is zero, the value is infinity.
- If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.

Floating-point Arithmetic

Addition & Subtraction : Floating-point addition and subtraction are done the same as using pencil and paper. The first thing that we do is express both operands in the same exponential power, then add the numbers, keeping the exponent in the sum.

If the exponent needs adjustment, fix it at the end.

EXAMPLE 2.49.

Find the sum of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

We find $12_{10} = 0.1100 \times 2^4$

And $1.25_{10} = 0.101 \times 2^1$

$= 0.000101 \times 2^4$

	0	10100	11000000
×	0	10001	10100000
	0	10101	01111000

Thus, our sum is 0.110101×2^4.

Multiplication: Floating-point multiplication is also the same as pencil and paper. Multiply the two operands and add exponents.

If the exponent requires adjustment, fix it at the end.

EXAMPLE 2.50.

Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

We find $12_{10} = 0.1100 \times 2^4$

And $1.25_{10} = 0.101 \times 2^1$

	0	10100	11000000
−	0	10001	10100000
	0	10101	10101100

Thus, our product is

$$0.0111100 \times 2^5 = 0.1111 \times 2^4.$$

The normalized product requires an exponent of $22_{10} = 10110_2$.

When discussing floating-point numbers, it is important to understand the following rules and *terms range, precision, and accuracy* etc.

- The range of a numeric integer format is the difference between the largest and smallest values that it can express.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value.
- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is infinite, so F.P. can give nothing more than an approximation of a real value.
- At some point, every model breaks down, introducing errors into our calculations.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

2.5. CHARACTER CODES

Digital computers store everything in the form of binary digits. So, how can we encode numbers, images, sound and text? Therefore we need standard encoding systems for each type of data. Some standards evolve from proprietary products which became very popular. Other standards are created by official industry bodies where none previously existed.

Alphanumeric data such as names and addresses are represented by assigning a unique binary code or sequence of bits to represent each character. As each character is entered from a keyboard (or other input device) it is converted into a binary code. Character code sets contain two types of characters:

- Printable (normal characters)
- Non-printable. Characters used as control codes.
  - CTRL G (beep)
  - CTRL Z (end of file)

There are three main coding schemes are in use:

1. ASCII
2. EBCDIC
3. Unicode

2.5.1. ASCII Characters

ASCII (American Standard Code for Information Interchange) code represents each character in a standard character set as a single byte binary code. The ASCII is a standard code form that most PCs use to allow for communication between systems. It usually uses a 7 bit binary code so it can store 128 different characters and simple communications protocols. ASCII codes are sufficient for all characters on a standard keyboard plus control codes. These codes can be extended (extended ASCII) to use 8 bits (so can store 256 characters) to encode Latin language characters.

The first 32 ASCII codes are used for simple communications protocols, not characters. For example:

- ACK - acknowledge and would be sent by a device to acknowledge receipt of data.
- 0110010 – 2
- 0110001 – 1
- .....
- 1000001 – A
- 1000010 – B

Note: Letters have increasing values from a-z or A-Z.

Table 2.1: EBCDIC and 8-bit ASCII codes

Character	EBCDIC Code	ASCII Code
blank	0100 0000	0010 000
-	0100 1011	0010 1110
(	0100 1101	0010 1000
+	0100 1110	0010 1011
$	0101 1011	0010 0100
*	0101 1100	0010 1010
)	0101 1101	0010 1001
	0110 0000	0010 1101

/	0110 0001	0010 1111
:	0110 1011	0010 0111
,	0111 1101	0010 1100
=	0111 1110	0011 1101
A	1100 0001	0100 0001
B	1100 0010	0100 0010
C	1100 0011	0100 0011
D	1100 0100	0100 0100
E	1100 0101	0100 0101
F	1100 0110	0100 0110
G	1100 0111	0100 0111
H	1100 1000	0100 1000
I	1100 1001	0100 1001
J	1101 0001	0100 1010
K	1101 0010	0100 1011
L	1101 0011	0100 1100
M	1101 0100	0100 1101
N	1101 0101	0100 1110
O	1101 0110	0100 1111
P	1101 0111	0101 0000
Q	1101 1000	0101 0001
R	1101 1001	0101 0010
S	1110 0010	0101 0011
T	1110 0011	0101 0100
U	1110 0100	0101 0101
V	1110 0101	0101 0110
W	1110 0110	0101 0111
X	1110 0111	0101 1000
Y	1110 1000	0101 1001
Z	1110 1001	0101 1010
0	1111 0000	0011 0000
1	1111 0001	0011 0001
2	1111 0010	0011 0010
3	1111 0011	0011 0011
4	1111 0100	0011 0100
5	1111 0101	0011 0101
6	1111 0110	0011 0110
7	1111 0111	0011 0111
8	1111 1000	0011 1000
9	1111 1001	0011 1001

2.5.2. EBCDIC Characters

In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC). EBCDIC was one of the first widely-used computer codes that supported upper and lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters. EBCDIC and BCD are still in use by IBM mainframes today.

2.5.3. Unicode Characters

The ASCII and EBCDIC codes basically supports the Latin character sets used in computers. There are many more character sets in the world, and a simple ASCII-to-language-X mapping does not work for the general case. So a new universal character standard was developed that supports a more character sets, called Unicode.

The Unicode Standard uses a 16-bit code set in which there is a one-to-one correspondence between 16-bit codes and characters. Like ASCII, there are no complex modes or escape codes. While Unicode supports many more characters than ASCII or EBCDIC, it is not the end-all standard. In fact, the 16-bit Unicode standard is a subset of the 32-bit ISO 10646 Universal Character Set (UCS-4).

POINTS TO REMEMBER

- The computer organization is depends considerably on how the data representation of numbers, characters and control information.
- A number system uses a specific radix (base). The base or radix of a number system represents the number of digits or basic symbols in that particular number system.
- There are four basic number system in the computer system:
  - (i) Binary number system
  - (ii) Octal number system
  - (iii) Decimal number system
  - (iv) Hexadecimal system
- The radices include binary (base 2), quaternary (base 4), octagonal (base 8), and hexagonal (base 16).
- The binary system has base 2 and dominant in computer systems.
- Octal number system has a base of 8 i.e., it has eight basic symbols. First eight decimal digits 0, 1,2,3,4,5,6,7 are used in this system.
- The hexadecimal number system has a base of 16. It has 16 symbols from 0 through 9 and A through F.
- There are four conventions in use for representing negative integers are:
  - (*i*) Sign Magnitude
  - (*ii*) 1's Complement
  - (*iii*) 2's Complement
  - (*iv*) Excess (or Bias) Representation
- In 1985, the Institute of Electrical and Electronic Engineers (IEEE) published a floating-point standard for both single- and double-precision floating-point numbers.
- The IEEE-754 single precision uses bias of 127 for its 8-bit exponent.
- The IEEE-754 double precision standard has a bias of 1023 for its 11-bit exponent.