

Chapter Three

Data Representation and Computer Arithmetic

- Number Systems and Conversion
- Units of Data Representation and Coding Methods
- Binary Arithmetic and Complements
- Fixed and Floating points representation
- Character Representations and Coding
- Boolean Algebra and Logic Circuits *

Number systems and conversion

- Number Systems
 - Decimal
 - Binary
 - Octal
 - Hexadecimal
- Conversion

Decimal systems

- **The decimal system**

- Base 10 with ten distinct digits (0, 1, 2, ..., 9)
- Any number greater than 9 is represented by a combination of these digits
- The weight of a digit based on power of 10

Example:

- The number 81924 is actually the sum of:
 $(8 \times 10^4) + (1 \times 10^3) + (9 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$

Binary systems

Computers use the binary system to store and compute numbers.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0 or 1	0 or 1	0 or 1	0 or 1	0 or 1	0 or 1	0 or 1	0 or 1

To represent any decimal number using the binary system, each place is simply assigned a value of either 0 or 1. To convert binary to decimal, simply add up the value of each place.

Example:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	1	0	0	1
128	0	0	16	8	0	0	1

128 + 0 + 0 + 16 + 8 + 0 + 0 + 1 = 153

10011001 = 153

Binary systems

- **The binary system (0 & 1)**

- The two digits representation is called a binary system
- Two electrical states – **on (1) & off (0)**
- The position weights are based on the **power of 2**
- The various combination of the two digits representation gives us the final value

Examples :

- i) 1011011 in binary = 91 in decimal
- ii) 1101.01 in binary = 13.25 in decimal

Binary into Decimal conversion

5th 4th 3rd 2nd 1st 0th

$$\begin{aligned} 1\ 0\ 1\ 1\ 1\ 1_2 &= (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 32 + 8 + 4 + 2 + 1 \\ &= 47_{10} \end{aligned}$$

$$11001_2 = 25_{10}$$

Exercise :

Convert the following binary numbers into their decimal equivalent

- $11010_2 = (?)_{10}$
- $101.1101_2 = (?)_{10}$

Binary Fractions

Binary fractions can also be represented:

Position Value: 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} etc.

Fractions: $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$ $\frac{1}{16}$ $\frac{1}{32}$

Decimal: .5 .25 .125 .0625 .03125

Conversion of Decimal to Binary

Divide by 2 (**remainder division**) till the dividend is zero and read remainders in reverse order.

Read answer in this direction, **Write answer left to right**

mod 2	637
1	318
0	159
1	79
1	39
1	19
1	9
1	4
0	2
0	1
1	0

$$637_{10} = 1001111101_2$$

➤ Convert 789_{10} to base 2

To Binary Fractions - Conversions

Multiply by 2 till enough digits are obtained, say 8, or a product is zero.

Read answer
in this direction
write it left
to right

$$\begin{array}{r} \hline .637_{10} \\ 1.274 \\ 0.548 \\ 1.096 \\ 0.192 \\ 0.384 \\ 0.768 \\ 1.536 \\ 1.072 \end{array}$$

Ans= 0.10100011₂

➤ Convert 0.325_{10} to base 2

Octal system

- Octal system
 - Base 8 systems (0, 1, 2, ..., 7)
 - Used to give shorthand ways to deal with the long strings of 1 & 0 created in binary
 - Numbers 0 .. 7 can be represented by three binary digits

❖ Examples :-

i) $(3137)_8 = 1631$

ii) 134 in octal = 1011100 in binary

iii) $(6)_8 = (110)_2$

iv) 432.2 in octal = 282.25 in decimal

v) 123.45 in octal = 001010011.100101 in binary

Hexadecimal systems

- The Hexadecimal system
 - Base 16 system
 - 0 .. 9 and letters A .. F for sixteen place holders needed
 - A = 10, B = 11, ..., F = 15
 - Used in programming as a short cut to the binary number systems
 - Can be represented by four binary digits

❖ Examples :-

i) $(1D7F)_{16} = 7551_{10}$

ii) 6B2 in hexadecimal = 011010110010 in binary

iii) 101000010111 in binary = A7 in hexadecimal

➤ Exercise:-

- ✓ Convert Hexadecimal 0, 1, 2, ..., 9, A, ... F into Decimal, Octal and Binary equivalents.

Exercise – Convert ...

Decimal	Binary	Octal	Hexa- decimal
29.8			
	101.1101		
		3.07	
			C.82

Exercise – Convert ...

Answer

Decimal	Binary	Octal	Hexa- decimal
29.8	11101.110011...	35.63...	1D.CC...
5.8125	101.1101	5.64	5.D
3.109375	11.000111	3.07	3.1C
12.5078125	1100.10000010	14.404	C.82

Bits

- The two binary digits 0 and 1 are frequently referred to as bits.
- How many bits does a computer use to store an integer?
 - Intel Pentium PC = 32 bits
 - Alpha = 64 bits
- What if we try to compute or store a larger integer?
 - If we try to compute a value larger than the computer can store, we get an arithmetic overflow error.

Representing Unsigned Integers

- How does a 16-bit computer represent the value 14?

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

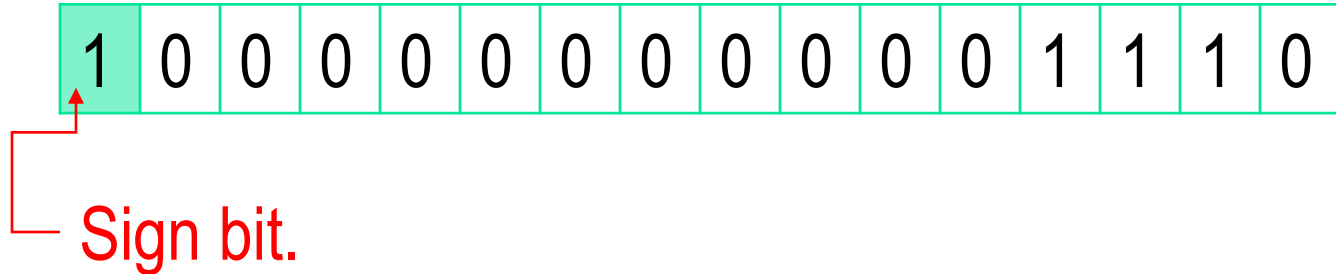
- What is the largest 16-bit integer?

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

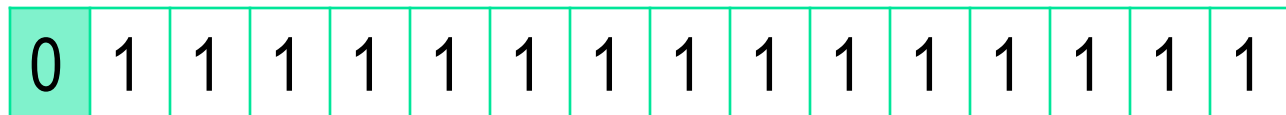
$$= 1 \times 2^{15} + 1 \times 2^{14} + \dots + 1 \times 2^1 + 1 \times 2^0 = 65,535$$

Representing Signed Integers

- How does a 16 bit computer represent the value -14?



- What is the largest 16-bit signed integer?



$$= 1 \times 2^{14} + 1 \times 2^{13} + \dots + 1 \times 2^1 + 1 \times 2^0 = 32,767$$

- Problem → the value 0 is represented twice!
 - Most computers use a different representation, called two's complement.

Signed-magnitude representation

- Also called, “sign-and-magnitude representation”
- A number consists of a magnitude and a symbol representing the sign
- Usually 0 means positive, 1 negative
 - Sign bit
 - Usually the entire number is represented with 1 sign bit to the left, followed by a number of magnitude bits

Machine arithmetic with signed-magnitude representation

- Takes several steps to add a pair of numbers
 - Examine signs of the addends
 - If same, add magnitudes and give the result the same sign as the operands
 - If different, must...
 - Compare magnitude of the two operands
 - Subtract smaller number from larger
 - Give the result the sign of the larger magnitude operand
 - If magnitudes are equal and sign is different; two representations of zero problem
- For this reason the signed-magnitude representation is not as popular as one might think because of its “naturalness”

Complement number systems

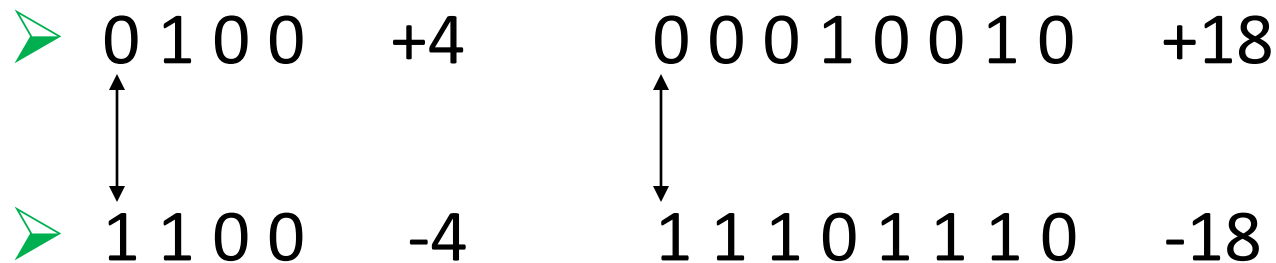
- Negates a number by *taking its complement* instead of negating the sign
- Not natural for humans, but better for machine arithmetic
- Will describe 2 complement number systems
 - *Radix complement – very popular in real computers*
 - Must first decide how many bits to represent the number – say n .
 - Complement of a number = $r^n - \text{number}$
 - **Example: 2's Complement**
 - *Diminished radix-complement – not very useful, may skip or not spend much time on it*
 - **Example: 1's Complement** = $r^n - \text{number} - 1$

Two's-complement representation

- Just radix-complement when $\text{radix} = 2$
- The most used representation of integers in computers and other digital arithmetic circuits
- 0 and positive numbers: leftmost bit = 0
- Negative numbers: leftmost bit = 1
- Representation of zero
 - i.e. 0 is represented as 0000 using 4-bit binary sequence.
- To find a number's 2's-complement — just flip all the bits and add 1

Properties of Two's Complement Notation

- Relationship between $+n$ and $-n$.



Two's Complement Notation with 4-bits

Binary Pattern

Value in 2's complement.

0 1 1 1

7

0 1 1 0

6

0 1 0 1

5

0 1 0 0

4

0 0 1 1

3

0 0 1 0

2

0 0 0 1

1

0 0 0 0

0

1 1 1 1

-1

1 1 1 0

-2

1 1 0 1

-3

1 1 0 0

-4

1 0 1 1

-5

1 0 1 0

-6

1 0 0 1

-7

1 0 0 0

-8

Advantages of Two's Complement Notation

- It is easy to add two numbers.

$$\begin{array}{r} 0001 \quad +1 \\ + 0101 \quad +5 \\ \hline 0110 \quad +6 \end{array}$$

$$\begin{array}{r} 1000 \quad -8 \\ + 0101 \quad +5 \\ \hline 1101 \quad -3 \end{array}$$

- Subtraction can be easily performed.
- Multiplication is just a repeated addition.
- Division is just a repeated subtraction
- Two's complement is widely used in *ALU*

Comparison of decimal and 4-bit numbers

Complements and other Notations

<i>Decimal</i>	<i>Two's Complement</i>	<i>Ones' Complement</i>	<i>Signed Magnitude</i>	<i>Excess 2^{m-1}</i>
-8	1000	—	—	0000
-7	1001	1000	1111	0001
-6	1010	1001	1110	0010
-5	1011	1010	1101	0011
-4	1100	1011	1100	0100
-3	1101	1100	1011	0101
-2	1110	1101	1010	0110
-1	1111	1110	1001	0111
0	0000	1111 or 0000	1000 or 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

Excess is $2^{4-1} = 8$; Thus,
retrieved value = stored value - 8

**Decimal numbers, their
two's complements,
ones' complements,
signed magnitude and
excess 2^{m-1} binary codes**

Existence of two
zeros!

EXPLAIN

Two's-Comp Addition and Subtraction Rules

- Starting from 1000 (-8) on up, each successive 2's comp number all the way to 0111 (+7) can be obtained by adding 1 to the previous one, ignoring any carries beyond the 4th bit position
- Since addition is just an extension of ordinary counting, 2's comp numbers can be added by ordinary binary addition!
- No different cases based on operands' signs!
- Overflow possible
 - Occurs if result is out of range
 - Happens if operands are the same sign but sum is a different sign of that of the operands

Storing an integer in two's complement format:

- Convert the integer to an n-bit binary.
- If it is **positive** or **zero**, it is stored as it is. If it is **negative**, take the two's complement and then store it.

Retrieving an integer in two's complement format:

- If the **leftmost bit** is 1, the computer applies the two's complement operation to the n-bit binary. If the leftmost bit is 0, no operation is applied.
- The computer changes the binary to decimal (integer) and corresponding sign is added.

1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7

Example:

Retrieve the integer that is stored as 11100110 in memory using two's complement format.

Solution:

The leftmost bit is 1, so the integer is negative. The integer needs to be two's complemented before changing to decimal.

Leftmost bit is 1. The sign is negative

1	1	1	0	0	1	1	0
↓	↓	↓	↓	↓	↓	↓	↓

Apply two's complement operation

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Integer changed to decimal

26

Sign is added

-26

Comparison

Summary of integer representations

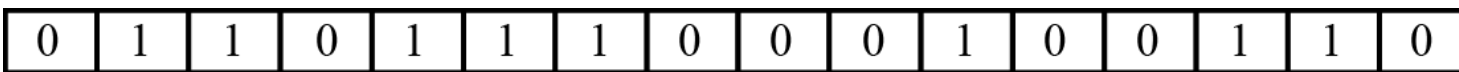
<i>Contents of memory</i>	<i>Unsigned</i>	<i>Sign-and-magnitude</i>	<i>Two's complement</i>
0000	0	0	+0
0001	1	1	+1
0010	2	2	+2
0011	3	3	+3
0100	4	4	+4
0101	5	5	+5
0110	6	6	+6
0111	7	7	+7
1000	8	−0	−8
1001	9	−1	−7
1010	10	−2	−6
1011	11	−3	−5
1100	12	−4	−4
1101	13	−5	−3
1110	14	−6	−2
1111	15	−7	−1

STORING REAL NUMBERS

A **number** is changed to binary before being stored in computer memory, as described earlier. There are two issues that need to be handled:

1. How to store the sign of the number (**we already know this**).
2. How to show the (decimal) point.

For the (decimal) point, computers use two different representations: **fixed-point** and **floating-point**. **The first is used to store a number as an integer, without a fraction part,** the second is used to store a number as a real number, with a fractional part.



Memory location



Decimal point / Radix point
(assumed position)

Fixed point representation of integers

An integer is normally stored in memory using fixed-point representation.

Applications of unsigned integers:

Counting- Addressing- storing other data types (text, images, audio and video)

Storing real numbers Continued

A real number is a number with an integral part and a fractional part. For example, 23.7 is a real number—the integral part is 23 and the fractional part is $7/10$. Although a fixed-point representation can be used to represent a real number, the result may not be accurate or it may not have the required precision. The next two examples explain why.

Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.

Example:

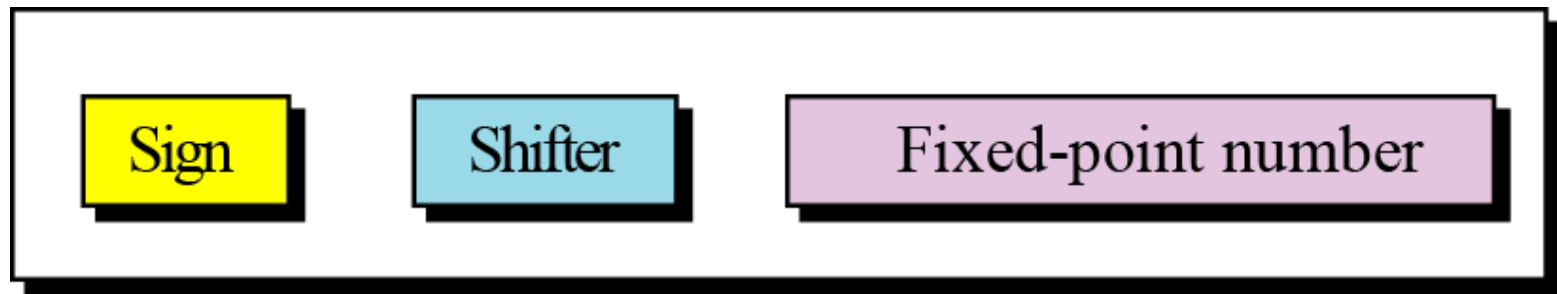
In the decimal system, assume that we use a fixed-point representation with **two digits at the right** of the decimal point and **fourteen digits at the left** of the decimal point, for a total of sixteen digits. The precision of a real number in this system is lost if we try to represent a decimal number such as 1.00234: the system stores the number as 1.00

Example:

In the decimal system, assume that we use a fixed-point representation with **six digits to the right** of the decimal point and **ten digits to the left** of the decimal point, for a total of sixteen digits. The accuracy of a real number in this system is lost if we try to represent a decimal number such as 236154302345.00. The system stores the number as 6154302345.00. The integral part is much smaller than it should be.

Floating-point representation

The solution for maintaining accuracy or precision is to use **floating-point representation**.



Floating-point representation

A floating point representation of a number is made up of three parts: a sign, a shifter and a fixed-point number.

Floating-point representation is used in science to represent very small or very large decimal numbers. In this representation called **scientific notation**, the fixed-point section has only one digit to the left of point and the shifter is the power of 10.

Example:

The following shows the decimal number

7425000000000000000000.00

in scientific notation (floating-point representation).

Actual number	→	+	7,425,000,000,000,000,000,000.00
Scientific notation	→	+	7.425×10^{21}

The three sections are the **sign** (+), the **shifter** (21) and the **fixed-point part** (7.425). Note that the shifter is the exponent.

Some programming languages and calculators shows the number as **+7.425E21**

Example:

Show the number -0.00000000000000232

in scientific notation (floating-point representation).

Solution

We use the same approach as in the previous example—we move the decimal point after the digit 2, as shown below:

Actual number	→	–	0.00000000000000232
Scientific notation	→	–	2.32×10^{-14}

The three sections are the **sign** (–), the **shifter** (-14) and the **fixed-point part** (2.32). Note that the shifter is the exponent.

Example:

Show the number,

$$(10100100000000000000000000000000.00)_2$$

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the radix (decimal) point.

Actual number $\rightarrow + (10100100000000000000000000000000.00)_2$

Scientific notation $\rightarrow + 1.01001 \times 2^{32}$

Example:

Show the number

[illegible]

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the decimal point.

Actual number → - $(0.0000000000000000000000000101)_2$

Scientific notation $\rightarrow -1.01 \times 10^{-24}$

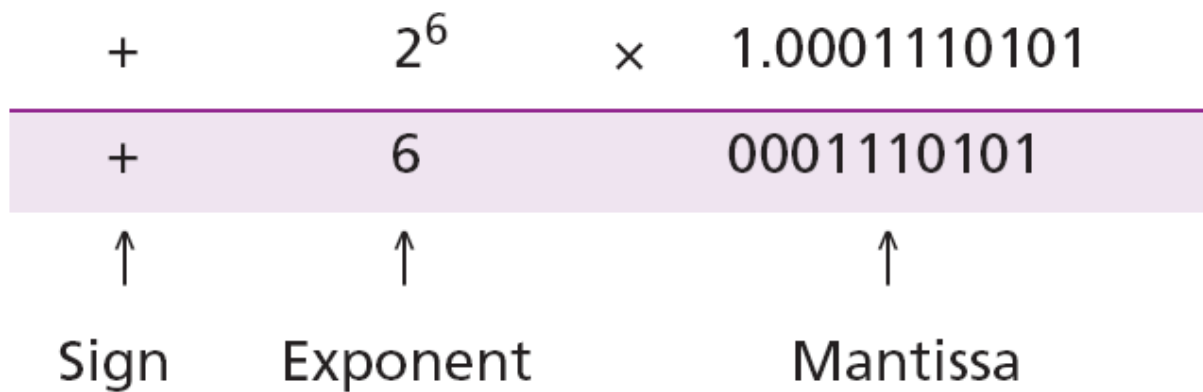
Normalization

To make the **fixed part** of the representation **uniform**, both the scientific method (for the decimal system) and the floating-point method (for the binary system) **use only one non-zero digit on the left of the decimal point**. This is called **normalization**. In the decimal system this digit can be 1 to 9, while in the binary system it can only be 1. In the following, d is a non-zero digit, x is a digit, and y is either 0 or 1.

Decimal	→	±	d.xxxxxxxxxxxxxxx
Binary	→	±	1.yyyyyyyyyyyyyyy

Note: d is 1 to 9 and each x is 0 to 9

Note: each y is 0 or 1



Note that the point and the bit 1 to the left of the fixed-point section are not stored; They are implicit.

The mantissa is a fractional part that, together with the sign (and the implicit **1), is treated like an integer stored in **sign-and-magnitude** representation.**

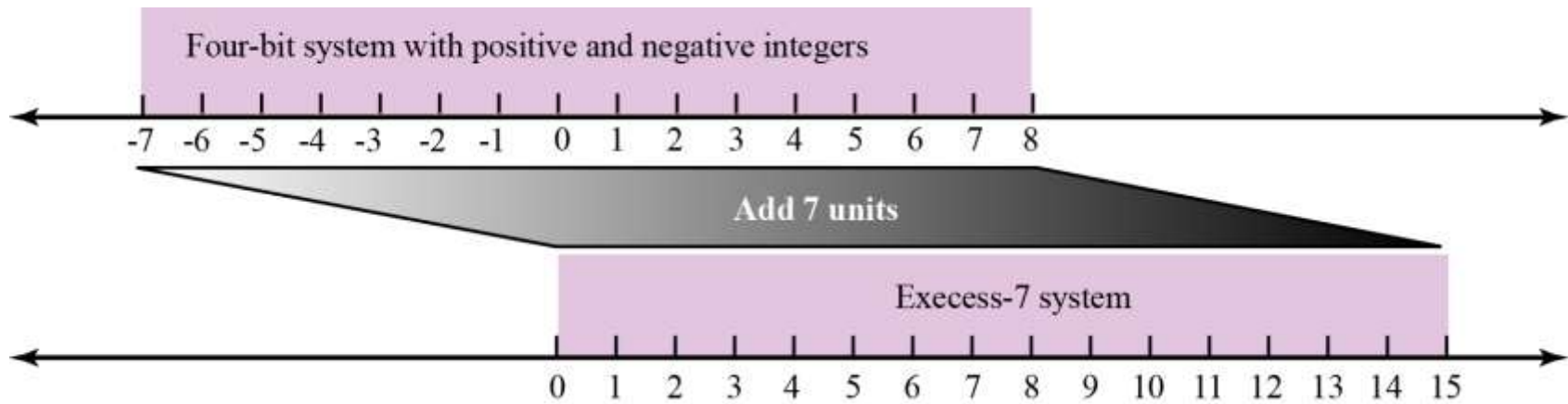
We need to remember that it is not an integer, it is a **fractional part** that is stored like an integer. If we insert **extra 0s to the right** of the number, **the value will not change**, whereas in a real integer if we insert **extra 0s to the left** of the number, the **value will not change**.

Excess Notation

- The **exponent**, the power that shows how many bits the decimal point should be moved to the left or right, is a signed number.
- Although this could have been stored using **two's complement representation**, a new representation, called the **Excess notation**, is used instead.
- In the Excess notation, **both positive and negative integers are stored as unsigned integers**.
- To represent a positive or negative integer, a positive integer (called a bias) is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where m is the size of the memory location to store the exponent.

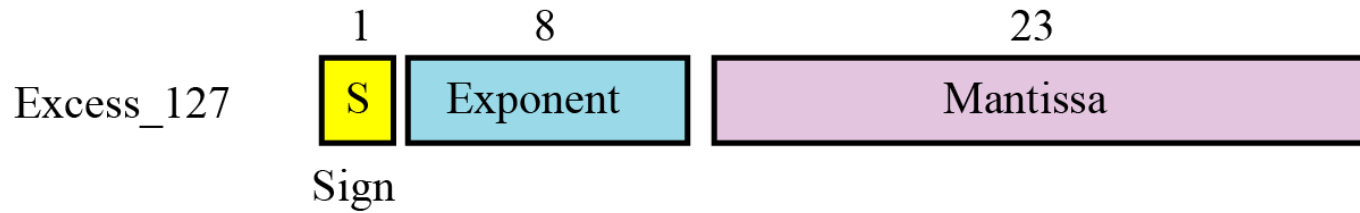
Example:

We can express **sixteen integers** in a number system with **4-bit allocation**. By adding seven units to each integer in this range, we can uniformly translate all integers to the right and make all of them positive without changing the relative position of the integers with respect to each other, as shown in the figure. The new system is referred to as **Excess-7**, or biased representation with biasing value of 7.

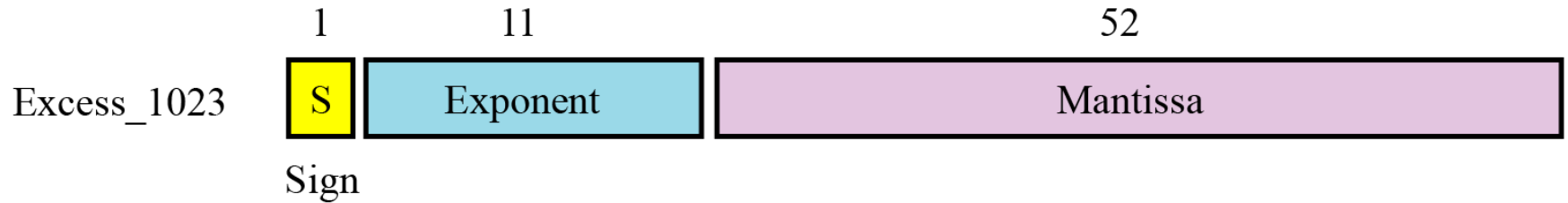


Shifting in Excess representation

IEEE Standard



a. Single precision (32 bits)



b. Double precision (64 bits)

IEEE standards for floating-point representation

IEEE Specifications

Specifications of the two IEEE floating-point standards

<i>Parameter</i>	<i>Single Precision</i>	<i>Double Precision</i>
Memory location size (number of bits)	32	64
Sign size (number of bits)	1	1
Exponent size (number of bits)	8	11
Mantissa size (number of bits)	23	52
Bias (integer)	127	1023

Storage of IEEE standard floating point numbers:

1. Store the sign in S (0 or 1).
2. Change the number to binary.
3. Normalize.
4. Find the values of E and M.
5. Concatenate S, E, and M.

Example 1:

Show the Excess_127 (single precision) representation of the decimal number 5.75

Solution

- The sign is positive, so $S = 0$.
- Decimal to binary transformation: $5.75 = (101.11)_2$.
- Normalization: $(101.11)_2 = (1.0111)_2 \times 2^2$.
- $E = 2 + 127 = 129 = (10000001)_2$, $M = 0111$. We need to add nineteen zeros at the right of M to make it 23 bits.
- The representation is shown below:

0	10000001	01110000000000000000000
S	E	M

The number is stored in the computer as

01000000101110000000000000000000

Example 2:

Show the Excess_127 (single precision) representation of the decimal number -161.875

Solution

- The sign is negative, so $S = 1$.
- Decimal to binary transformation: $161.875 = (10100001.111)_2$.
- Normalization: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.
- $E = 7 + 127 = 134 = (10000110)_2$ and $M = (0100001111)_2$.
- Representation:

1	10000110	010000111100000000000000
S	E	M

The number is stored in the computer as

11000011001000011110000000000000

Example 3:

Show the Excess_127 (single precision) representation of the decimal number -0.0234375

Solution

- $S = 1$ (the number is negative).
- Decimal to binary transformation: $0.0234375 = (0.0000011)_2$.
- Normalization: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.
- $E = -6 + 127 = 121 = (01111001)_2$ and $M = (1)_2$.
- Representation:

1	01111001	100000000000000000000000
S	E	M

The number is stored in the computer as

10111100110000000000000000000000

Retrieving numbers stored in IEEE standard floating point format:

1. Find the value of S, E, and M.
2. If $S=0$, set the sign to positive, otherwise set the sign to negative.
3. Find the shifter ($E-127$).
4. De-normalize the mantissa.
5. Change the de-normalized number and find the absolute value (in decimal).
6. Add sign.

Example 4:

The bit pattern $(1\mathbf{10010100}00000000111000100001111)_2$ is stored in Excess_127 format, in memory. Show the retrieved value in decimal.

Solution

- a. The first bit represents S, the next eight bits, E and the remaining 23 bits, M.

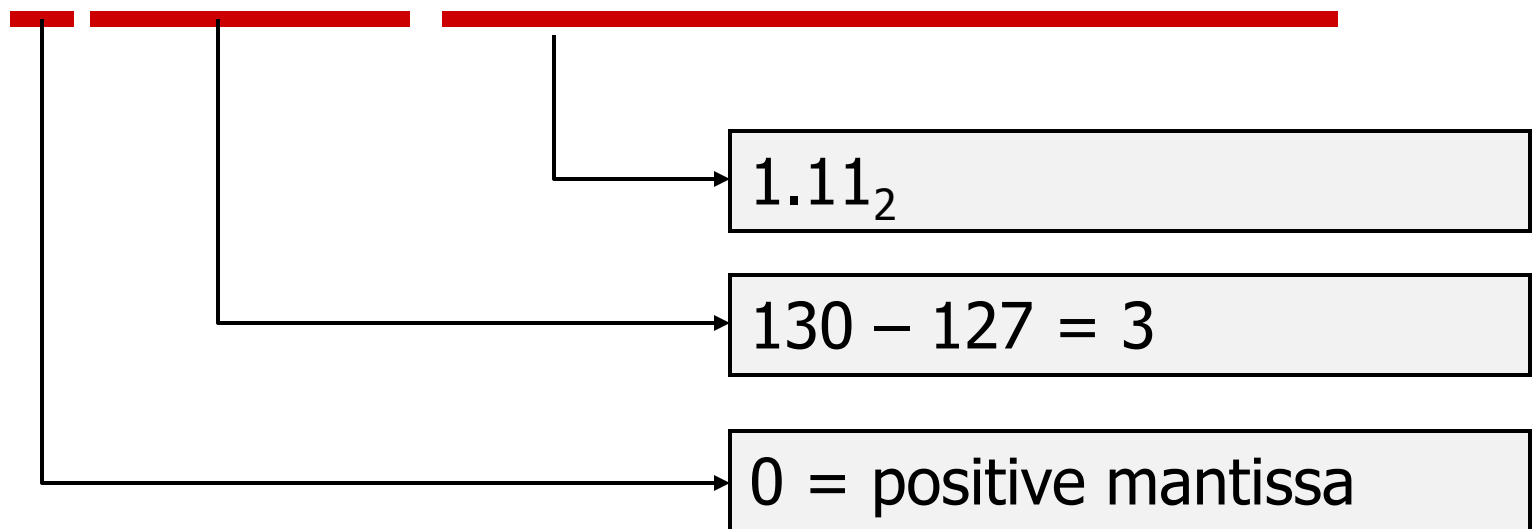
S	E	M
1	10010100	00000000111000100001111

- b. The sign is negative.
c. The shifter = $E - 127 = 148 - 127 = 21$.
d. This gives us $(1.00000000111000100001111)_2 \times 2^{21}$
e. The binary number is $(1000000001110001000011.11)_2$
f. The absolute value is 2,104,387.75
g. The number is $-2,104,387.75$

Example 5

- Single precision

0 10000010 110000000000000000000000



$$+1.11_2 \times 2^3 = 1110.0_2 = 14.0_{10}$$

Exercise

1. Represent $+0.8$ in the following floating-point representation:
 - 1-bit sign
 - 4-bit exponent
 - 6-bit normalised mantissa (significand).
2. Convert the value represented back to decimal.
3. Calculate the (relative) error of the representation.