**Arithmetic Operations on Binary Numbers**

Because of its widespread use, we will concentrate on addition and subtraction for Two's Complement representation.

The nice feature with Two's Complement is that addition and subtraction of Two's complement numbers works without having to separate the sign bits (the sign of the operands and results is effectively built-into the addition/subtraction calculation).

Remember: −2n−1 ≤ Two's Complement ≤ 2n−1 − 1

−8 ≤ x[4] ≤ +7

−128 ≤ x[8] ≤ +127

−32768 ≤ x[16] ≤ +32767

−2147483648 ≤ x[32] ≤ +2147483647

What if the result overflows the representation?

If the result of an arithmetic operation is to too large (positive or negative) to fit into the resultant bit-group, then arithmetic **overflow** occurs. It is normally left to the programmer to decide how to deal with this situation.

**Two's Complement Addition**

Add the values and discard any carry-out bit.

**Examples:** using 8-bit two’s complement numbers.

1. Add −8 to +3
2. (+3) 0000 0011
3. +(−8) 1111 1000
4. -----------------
5. (−5) 1111 1011
6. Add −5 to −2
7. (−2) 1111 1110
8. +(−5) 1111 1011
9. -----------------
10. (−7) 1 1111 1001 : discard carry-out

**Overflow Rule for addition**

If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

|  |  |
| --- | --- |
| i.e. | Adding two positive numbers must give a positive result |
|  | Adding two negative numbers must give a negative result |

Overflow occurs if

* (+A) + (+B) = −C
* (−A) + (−B) = +C

Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)

(−7) 1001

+(−6) 1010

------------

(−13) 1 0011 = 3 : Overflow (largest −ve number is −8)

**A couple of definitions:**

|  |  |
| --- | --- |
| Subtrahend: | what is being subtracted |
| Minuhend: | what it is being subtracted from |

Example: 612 - 485 = 127

485 is the subtrahend, 612 is the minuhend, 127 is the result

**Two's Complement Subtraction**

Normally accomplished by negating the subtrahend and adding it to the minuhend. Any carry-out is discarded.

Example: Using 8-bit Two's Complement Numbers (−128 ≤ x ≤ +127)

(+8) 0000 1000 0000 1000

−(+5) 0000 0101 -> Negate -> +1111 1011

----- -----------

(+3) 1 0000 0011 : discard carry-out

**Overflow Rule for Subtraction**

If 2 Two's Complement numbers are subtracted, and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend.

Overflow occurs if

* (+A) − (−B) = −C
* (−A) − (+B) = +C

Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)

Subtract −6 from +7

(+7) 0111 0111

−(−6) 1010 -> Negate -> +0110

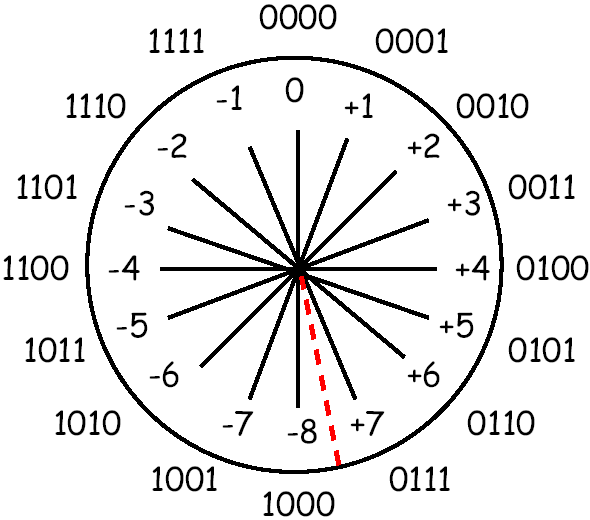
---------- -----

13 1101 = −8 + 5 = −3 : Overflow

**Number Circle for 4-bit Two's Complement numbers**

Numbers can be added or subtracted by moving round the number circle

* Clockwise for addition
* Anti-Clockwise for subtraction (addtion of a negative number)



Overflow occurs when a transition is made

* from +2n−1−1 to −2n−1 when adding
* from −2n−1 to +2n−1−1 when subtracting

**Two's Complement Summary**

**Addition**

* Add the values, discarding any carry-out bit

**Subtraction**

* Negate the subtrahend and add, discarding any carry-out bit

**Overflow**

* Occurs when adding two positive numbers produces a negative result, or when adding two negative numbers produces a positive result. Adding operands of unlike signs never produces an overflow
* Notice that discarding the carry out of the most significant bit during Two's Complement addition is a normal occurrence, and does not by itself indicate overflow
* As an example of overflow, consider adding (80 + 80 = 160)10, which produces a result of −9610 in 8-bit two's complement:
* 01010000 = 80
* + 01010000 = 80
* --------------
* 10100000 = −96 (not 160 because the sign bit is 1.)
* (largest +ve number in 8 bits is 127)

**Number Representation and Arithmetic Operations**

The most natural way to represent a number in a computer system is by a string of bits, called a binary number.

We will first describe binary number representations

* for integers as well as
* arithmetic operations on them.
* Then we will provide a brief introduction to the representation of floating-point numbers.

### Integers

Consider an *n*-bit vector

*B* = *bn*−1 *... b*1*b*0

where *bi* = 0 or 1 for 0 ≤ *i* ≤ *n* − 1. This vector can represent an unsigned integer value

*V(B)* in the range 0 to 2*n* − 1, where

*V(B)* = *bn*−1 × 2*n*−1 + ··· + *b*1 × 21 + *b*0 × 20

We need to represent both positive and negative numbers. Three systems are used for representing such numbers:

* Sign-and-magnitude
* 1’s-complement
* 2’s-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.

Fig:1(Binary, signed-integer representations.)

* all three representations using 4-bit numbers.
* Positive values have identical representations in all systems,
* but negative values have different representations.

In the *sign-and-magnitude* system, negative values are represented by changing the most significant bit (*b*3 in Figure 1) from 0 to 1 in the *B* vector of the corresponding positive value. For example, 5 is represented by 0101, and 5 is represented by 1101

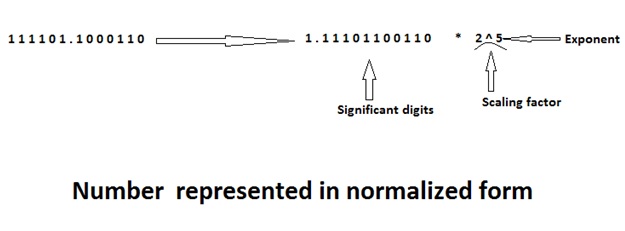
*B* Values represented

3 *b*2 *b*1 *b*0 Sign andmagnitude 1’s complement 2’s complement

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 1 | 1 | 1 | + 7 | + 7 | + 7 |
| 0 1 | 1 | 0 | + 6 | + 6 | + 6 |
| 0 1 | 0 | 1 | + 5 | + 5 | + 5 |
| 0 1 | 0 | 0 | + 4 | + 4 | + 4 |
| 0 0 | 1 | 1 | + 3 | + 3 | + 3 |
| 0 0 | 1 | 0 | + 2 | + 2 | + 2 |
| 0 0 | 0 | 1 | + 1 | + 1 | + 1 |
| 0 0 | 0 | 0 | + 0 | + 0 | + 0 |
| 1 0 | 0 | 0 | – 0 | – 7 | – 8 |
| 1 0 | 0 | 1 | – 1 | – 6 | – 7 |
| 1 0 | 1 | 0 | – 2 | – 5 | – 6 |
| 1 0 | 1 | 1 | – 3 | – 4 | – 5 |
| 1 1 | 0 | 0 | – 4 | – 3 | – 4 |
| 1 1 | 0 | 1 | – 5 | – 2 | – 3 |
| 1 1 | 1 | 0 | – 6 | – 1 | – 2 |
| 1 1 | 1 | 1 | – 7 | – 0 | – 1 |

# **Floating point representation**

* In **floating point representation**, the computer must be able to represent the numbers and can be operated on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds, for the accommodation of very large integers and very small fractions. In this case, the binary point is said to be the float, and the numbers are called the floating point numbers.
* The **floating point representation** has three fields:
  + Sign
  + Significant digits and
  + Exponents
* Let us consider the number **1 1 1 1 0 1. 1 0 0 0 1 1 0** to be represent in the floating point format.



* To represent the number in floating point format, the first binary point is shifted to the right of the first bit and the number is multiplied by the correct scaling factor to get the same value. The number is said to be in the normalized form.
* It is important to note that the base in the scaling factor is fixed 2.
* The string of the significant digits is commonly known as mantissa.
* In the above example, we can say that,

Sign = 0

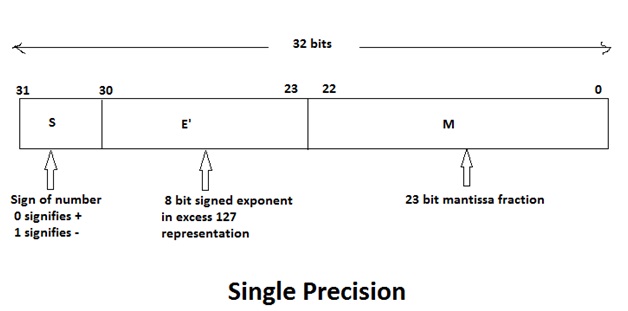
Mantissa = 1 1 1 0 1 1 0 0 1 1 0

Exponent =5

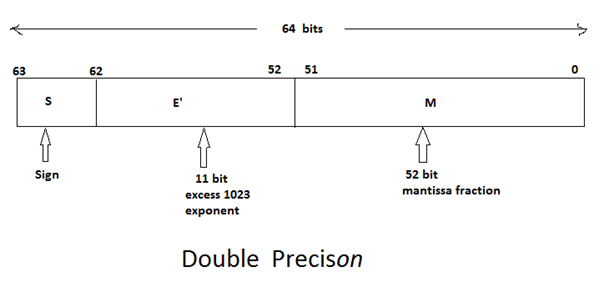
* In floating point numbers, the bias value is added to the true exponent. This solves the problem of representation of negative exponent.

### IEEE Standards for Floating Point Numbers

* The standards for the representation of floating point numbers in 32 bits and 64 bits have been developed by the Institute of Electrical and Electronics Engineers (IEEE), which is referred to as IEEE 754 standards.
* The standard representation of floating point number in 32 bits is called as a single precision representation because it occupies a single 32 bit word. The 32 bits divided into three fields:
  + (Field 1) Sign = 1 bit
  + (Field 2) Exponent = 8 bits
  + (Field 3) Mantissa = 23 bits



* Instead of the signed exponent E, the value actually stored in the exponent field is **E' = E (Scaling factor) + bias**.
* In the 32 bit floating point system (single precision), bias is **127**. Hence **E' = E (scaling factor) + 127**. This representation of exponent is called as the excess **127** format.
* In a single precision,the end values of **E'** respectively are used to indicate the floating point values of exact zero and infinity where the values of **E'** are namely the **0 and 255**.
* Thus range of **E'** for normal values in the single precision is **0 < E' < 255**. This means that for the representation of floating point number in 32 bits, the actual exponent **E** is in the range **-126 <= E <= 127**.
* The 64 bit standard representation is called a double precision representation because it occupies two 32 bit words.
* The 64 bits are divided into three fields:
  + (Field 1) Sign = 1 bit
  + (Field 2) Exponent = 11 bit
  + (Field 3) Mantissa = 52 bits
* In the double precision format value actually stored in the exponent field is given as **E' = E + 1023**
* Here, bias value is 1023 and hence it is also called excess 1023 format.
* The end values of **E'** namely 0 and 2047 are used to indicate the floating point exact values of exact zero and infinity, respectively.
* Thus the range of **E'** for normal values in double precision is **0 < E' < 2047**. This means that for 64 bit representation the actual exponent **E** is in the range **-1022 <= E <= 1023**.



* IEEE Standard 754
  + Established in 1985 as a uniform standard for floating point arithmetic It is supported by all major CPUs. Before 1985 there were many idiosyncratic formats.
* Fractional Binary Numbers: Examples

The binary number bibi− 1 b2 b 1 . . . b0.b − 1 b − 2 b − 3 . . . b −j represents a particular (positive) sum. Each digit is multiplied by a power of two according to the following chart: Bit: bi bi− 1 . . . b2 b 1 b0 · b − 1 b − 2 b − 3 . . . b − j Weight: 2 i 2 i− 1 . . . 4 2 1 · 1 / 2 1 / 4 1 / 8 . . . 2 − j Representation: Bits to the right of the binary point represent fractional powers of 2. This represents the rational number: Xi k = −j b k × 2 k The sign is treated separately