Data Representation

BHARAT SCHOOL OF BANKING-VELLORE

1. Number Systems

1.1 Decimal (Base 10) Number System

Decimal number system has ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, called *digits*. It uses *positional notation*. That is, the least-significant digit (right-most digit) is of the order of `10^0` (units or ones), the second right-most digit is of the order of `10^1` (tens), the third right-most digit is of the order of `10^2` (hundreds), and so on. For example,

```
735 = 7×10^2 + 3×10^1 + 5×10^0
```

We shall denote a decimal number with an optional suffix `D` if ambiguity arises.

1.2 Binary (Base 2) Number System

Binary number system has two symbols: 0 and 1, called *bits*. It is also a *positional notation*, for example,

```
10110B = 1×2^4 + 0×2^3 + 1×2^2 + 1×2^1 + 0×2^0
```

We shall denote a binary number with a suffix `B`. Some programming languages denote binary numbers with prefix `0b` (e.g., `0b1001000`), or prefix `b` with the bits quoted (e.g., `b'10001111'`).

A binary digit is called a *bit*. Eight bits is called a *byte* (why 8-bit unit? Probably because `8=2`^{3}).

1.3 Hexadecimal (Base 16) Number System

Hexadecimal number system uses 16 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F, called *hex digits*. It is a *positional notation*, for example,

```
A3EH = 10×16^2 + 3×16^1 + 14×16^0
```

We shall denote a hexadecimal number (in short, hex) with a suffix `H`. Some programming languages denote hex numbers with prefix `0x` (e.g., `0x1A3C5F`), or prefix `x` with hex digit quoted (e.g., `x'C3A4D98B'`).

Each hexadecimal digit is also called a *hex digit*. Most programming languages accept lowercase `'a'` to `'f'` as well as uppercase `'A'` to `'F'`.

Computers uses binary system in their internal operations, as they are built from binary digital electronic components. However, writing or reading a long sequence of binary bits is cumbersome and error-prone. Hexadecimal system is used as a *compact* form or *shorthand* for binary bits. Each hex digit is equivalent to 4 binary bits, i.e., shorthand for 4 bits, as follows:

`0H (0000B) (0D)`	`1H (0001B) (1D)`	`2H (0010B) (2D)`	`3H (0011B) (3D)`
`4H (0100B) (4D)`	`5H (0101B) (5D)`	`6H (0110B) (6D)`	`7H (0111B) (7D)`
`8H (1000B) (8D)`	`9H (1001B) (9D)`	`AH (1010B) (10D)`	`BH (1011B) (11D)`
`CH (1100B) (12D)`	`DH (1101B) (13D)`	`EH (1110B) (14D)`	`FH (1111B) (15D)`

1.4 Conversion from Hexadecimal to Binary

Replace each hex digit by the 4 equivalent bits, for examples,

```
A3C5H = 1010 0011 1100 0101B
102AH = 0001 0000 0010 1010B
```

1.5 Conversion from Binary to Hexadecimal

Starting from the right-most bit (least-significant bit), replace each group of 4 bits by the equivalent hex digit (pad the left-most bits with zero if necessary), for examples,

```
1001001010B = 0010 0100 1010B = 24AH
10001011001011B = 0010 0010 1100 1011B = 22CBH
```

It is important to note that hexadecimal number provides a *compact form* or *shorthand* for representing binary bits.

1.6 Conversion from Base *r* to Decimal (Base 10)

Given a *n*-digit base *r* number: `dn-1 dn-2 dn-3 ... d3 d2 d1 d0` (base r), the decimal equivalent is given by:

```
dn-1 × r^(n-1) + dn-2 × r^(n-2) + ... + d1 × r^1 + d0 × r^0
```

For examples,

```
A1C2H = 10×16^3 + 1×16^2 + 12×16^1 + 2 = 41410 (base 10)
10110B = 1×2^4 + 1×2^2 + 1×2^1 = 22 (base 10)
```

1.7 Conversion from Decimal (Base 10) to Base *r*

Use repeated division/remainder. For example,

```
To convert 261D to hexadecimal:
  261/16 => quotient=16 remainder=5
  16/16  => quotient=1  remainder=0
  1/16   => quotient=0  remainder=1 (quotient=0 stop)
  Hence, 261D = 105H
```

The above procedure is actually applicable to conversion between any 2 base systems. For example,

```
To convert 1023(base 4) to base 3:
  1023(base 4)/3 => quotient=25D remainder=0
  25D/3          => quotient=8D  remainder=1
  8D/3           => quotient=2D  remainder=2
  2D/3           => quotient=0   remainder=2 (quotient=0 stop)
  Hence, 1023(base 4) = 2210(base 3)
```

1.8 General Conversion between 2 Base Systems with Fractional Part

1. Separate the integral and the fractional parts.
2. For the integral part, divide by the target radix repeatably, and collect the ramainder in reverse order.
3. For the fractional part, multiply the fractional part by the target radix repeatably, and collect the integral part in the same order.

Example 1:

```
Convert 18.6875D to binary
Integral Part = 18D
  18/2 => quotient=9 remainder=0
  9/2  => quotient=4 remainder=1
  4/2  => quotient=2 remainder=0
  2/2  => quotient=1 remainder=0
  1/2  => quotient=0 remainder=1 (quotient=0 stop)
  Hence, 18D = 10010B
Fractional Part = .6875D
  .6875*2=1.375 => whole number is 1
  .375*2=0.75   => whole number is 0
  .75*2=1.5     => whole number is 1
  .5*2=1.0      => whole number is 1
  Hence .6875D = .1011B
Therefore, 18.6875D = 10010.1011B
```

Example 2:

```
Convert 18.6875D to hexadecimal
Integral Part = 18D
  18/16 => quotient=1 remainder=2
  1/16  => quotient=0 remainder=1 (quotient=0 stop)
  Hence, 18D = 12H
Fractional Part = .6875D
  .6875*16=11.0 => whole number is 11D (BH)
  Hence .6875D = .BH
```

```
Therefore, 18.6875D = 12.BH
```

1.9 Exercises (Number Systems Conversion)

1. Convert the following *decimal* numbers into *binary* and *hexadecimal* numbers:
    a. `108`
    b. `4848`
    c. `9000`

    Convert the following binary numbers into hexadecimal and decimal numbers:

    . `1000011000`
    a. `10000000`
    b. `101010101010`

    Convert the following hexadecimal numbers into binary and decimal numbers:

    . `ABCDE`
    a. `1234`
    b. `80F`

Answers: You could use the Windows' Calculator (`calc.exe`) to carry out number system conversion, by setting it to the scientific mode. (Run "calc" ⇒ Select "View" menu ⇒ Choose "Programmer" or "Scientific" mode.)

1. `1101100B`, `1001011110000B`, `10001100101000B`, `6CH`, `12F0H`, `2328H`.
2. `218H`, `80H`, `AAAH`, `536D`, `128D`, `2730D`.
3. `10101011110011011110B`, `1001000110100B`, `100000001111B`, `703710D`, `4660D`, `2063D`.

2. Integer Representation

Integers are *whole numbers* or *fixed-point numbers* with the radix point *fixed* after the least-significant bit. They are contrast to *real numbers* or *floating-point numbers*, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers. They have different representation and are processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor). Floating-point numbers will be discussed later.

Computers use *a fixed number of bits* to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

1. *Unsigned Integers*: can represent zero and positive integers.
2. *Signed Integers*: can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
    a. Sign-Magnitude representation
    b. 1's Complement representation
    c. 2's Complement representation

You, as the programmer, need to decide on the bit-length and representation scheme for your integers, depending on your application's requirements. Suppose that you need a counter for counting a small quantity from 0 up to 200, you might choose the 8-bit unsigned integer scheme as there is no negative numbers involved.

Data Representation

BHARAT SCHOOL OF BANKING-VELLORE

2.1 *n*-bit Unsigned Integers

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "*the magnitude of its underlying binary pattern*".

Example 1: Suppose that `n=8` and the binary pattern is `0100 0001B`, the value of this unsigned integer is `1×2^0 + 1×2^6 = 65D`.

Example 2: Suppose that `n=16` and the binary pattern is `0001 0000 0000 1000B`, the value of this unsigned integer is `1×2^3 + 1×2^12 = 4104D`.

Example 3: Suppose that `n=16` and the binary pattern is `0000 0000 0000 0000B`, the value of this unsigned integer is `0`.

An *n*-bit pattern can represent `2^n` distinct integers. An *n*-bit unsigned integer can represent integers from `0` to `(2^n)-1`, as tabulated below:

N	Minimum	Maximum
8	0	(2^8)-1 (=255)
16	0	(2^16)-1 (=65,535)
32	0	(2^32)-1 (=4,294,967,295) (9+ digits)
64	0	(2^64)-1 (=18,446,744,073,709,551,615) (19+ digits)

2.2 Signed Integers

Signed integers can represent zero, positive integers, as well as negative integers. Three representation schemes are available for signed integers:

1. Sign-Magnitude representation
2. 1's Complement representation
3. 2's Complement representation

In all the above three schemes, the *most-significant bit* (msb) is called the *sign bit*. The sign bit is used to represent the *sign* of the integer - with 0 for positive integers and 1 for negative integers. The *magnitude* of the integer, however, is interpreted differently in different schemes.

2.3 *n*-bit Sign Integers in Sign-Magnitude Representation

In sign-magnitude representation:

- The most-significant bit (msb) is the *sign bit*, with value of 0 representing positive integer and 1 representing negative integer.
- The remaining *n*-1 bits represents the magnitude (absolute value) of the integer. The absolute value of the integer is interpreted as "the magnitude of the (*n*-1)-bit binary pattern".

Example 1: Suppose that `n=8` and the binary representation is `0 100 0001B`.

Sign bit is `0` ⇒ positive
Absolute value is `100 0001B = 65D`
Hence, the integer is `+65D`

Example 2: Suppose that `n=8` and the binary representation is `1 000 0001B`.

Sign bit is `1` ⇒ negative
Absolute value is `000 0001B = 1D`
Hence, the integer is `-1D`

Example 3: Suppose that `n=8` and the binary representation is `0 000 0000B`.

Sign bit is `0` ⇒ positive
Absolute value is `000 0000B = 0D`
Hence, the integer is `+0D`

Example 4: Suppose that `n=8` and the binary representation is `1 000 0000B`.

Sign bit is `1` ⇒ negative

Absolute value is `000 0000B = 0D`
Hence, the integer is `-0D`

The drawbacks of sign-magnitude representation are:

1. There are two representations (`0000 0000B` and `1000 0000B`) for the number zero, which could lead to inefficiency and confusion.
2. Positive and negative integers need to be processed separately.

2.4 *n*-bit Sign Integers in 1's Complement Representation

In 1's complement representation:

- Again, the most significant bit (msb) is the *sign bit*, with value of 0 representing positive integers and 1 representing negative integers.
- The remaining *n*-1 bits represents the magnitude of the integer, as follows:
  - for positive integers, the absolute value of the integer is equal to "the magnitude of the (*n*-1)-bit binary pattern".
  - for negative integers, the absolute value of the integer is equal to "the magnitude of the *complement* (*inverse*) of the (*n*-1)-bit binary pattern" (hence called 1's complement).
-

Example 1: Suppose that `n=8` and the binary representation `0 100 0001B`.
Sign bit is `0` ⇒ positive
Absolute value is `100 0001B = 65D`
Hence, the integer is `+65D`

Example 2: Suppose that `n=8` and the binary representation `1 000 0001B`.
Sign bit is `1` ⇒ negative
Absolute value is the complement of `000 0001B`, i.e., `111 1110B = 126D`
Hence, the integer is `-126D`

Example 3: Suppose that `n=8` and the binary representation `0 000 0000B`.
Sign bit is `0` ⇒ positive

Absolute value is `000 0000B = 0D`
Hence, the integer is `+0D`

Example 4: Suppose that `n=8` and the binary representation `1 111 1111B`.
Sign bit is `1` ⇒ negative
Absolute value is the complement of `111 1111B`, i.e., `000 0000B = 0D`
Hence, the integer is `-0D`

Again, the drawbacks are:

1. There are two representations (`0000 0000B` and `1111 1111B`) for zero.
2. The positive integers and negative integers need to be processed separately.

2.5 *n*-bit Sign Integers in 2's Complement Representation

In 2's complement representation:

- Again, the most significant bit (msb) is the *sign bit*, with value of 0 representing positive integers and 1 representing negative integers.
- The remaining *n*-1 bits represents the magnitude of the integer, as follows:
  - for positive integers, the absolute value of the integer is equal to "the magnitude of the (*n*-1)-bit binary pattern".
  - for negative integers, the absolute value of the integer is equal to "the magnitude of the *complement* of the (*n*-1)-bit binary pattern *plus one*" (hence called 2's complement).

Example 1: Suppose that `n=8` and the binary representation `0 100 0001B`.
Sign bit is `0` ⇒ positive
Absolute value is `100 0001B = 65D`
Hence, the integer is `+65D`

Example 2: Suppose that `n=8` and the binary representation `1 000 0001B`.
Sign bit is `1` ⇒ negative

Absolute value is the complement of `000 0001B` plus `1`, i.e., `111 1110B + 1B = 127D`

Hence, the integer is `-127D`

Example 3: Suppose that `n=8` and the binary representation `0 000 0000B`.

Sign bit is `0` ⇒ positive

Absolute value is `000 0000B = 0D`

Hence, the integer is `+0D`

Example 4: Suppose that `n=8` and the binary representation `1 111 1111B`.

Sign bit is `1` ⇒ negative

Absolute value is the complement of `111 1111B` plus `1`, i.e., `000 0000B + 1B = 1D`

Hence, the integer is `-1D`

2.6 Computers use 2's Complement Representation for Signed Integers

We have discussed three representations for signed integers: signed-magnitude, 1's complement and 2's complement. Computers use 2's complement in representing signed integers. This is because:

1. There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

Example 1: Addition of Two Positive Integers: Suppose that `n=8`, `65D + 5D = 70D`

```
65D →    0100 0001B
 5D →    0000 0101B(+
          0100 0110B    → 70D (OK)
```

Example 2: Subtraction is treated as Addition of a Positive and a Negative Integers: Suppose that `n=8, 5D - 5D = 65D + (-5D) = 60D`

```
65D →    0100 0001B
-5D →    1111 1011B(+
          0011 1100B    → 60D (discard carry - OK)
```

Example 3: Addition of Two Negative Integers: Suppose that `n=8, -65D - 5D = (-65D) + (-5D) = -70D`

```
-65D →    1011 1111B
 -5D →    1111 1011B(+
           1011 1010B    → -70D (discard carry - OK)
```

Because of the *fixed precision* (i.e., *fixed number of bits*), an *n*-bit 2's complement signed integer has a certain range. For example, for *n*=8, the range of 2's complement signed integers is -128 to +127. During addition (and subtraction), it is important to check whether the result exceeds this range, in other words, whether *overflow* or *underflow* has occurred.

Example 4: Overflow: Suppose that `n=8, 127D + 2D = 129D` (overflow - beyond the range)

```
127D →    0111 1111B
  2D →    0000 0010B(+
           1000 0001B    → -127D (wrong)
```

Example 5: Underflow: Suppose that `n=8, -125D - 5D = -130D` (underflow - below the range)

```
-125D →    1000 0011B
  -5D →    1111 1011B(+
            0111 1110B    → +126D (wrong)
```

The following diagram explains how the 2's complement works. By re-arranging the number line, values from `-128` to `+127` are represented contiguously by ignoring the carry bit.

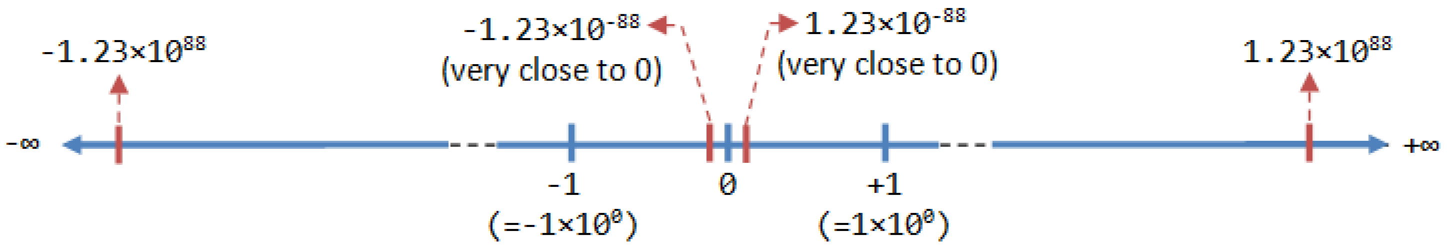
2.7 Range of *n*-bit 2's Complement Signed Integers

An *n*-bit 2's complement signed integer can represent integers from `-2^(n-1)` to `+2^(n-1)-1`, as tabulated. Take note that the scheme can represent all the integers within the range, without any gap. In other words, there is no missing integers within the supported range.

n	minimum	maximum
8	-(2^7) (=-128)	+(2^7)-1 (=+127)
16	-(2^15) (=-32,768)	+(2^15)-1 (=+32,767)
32	-(2^31) (=-2,147,483,648)	+(2^31)-1 (=+2,147,483,647)(9+ digits)
64	-(2^63) (=-9,223,372,036,854,775,808)	+(2^63)-1 (=+9,223,372,036,854,775,807)(18 digits)

3. Floating-Point Number Representation

A floating-point number (or real number) can represent a very large (`1.23×10^88`) or a very small (`1.23×10^-88`) value. It could also represent very large negative number (`-1.23×10^88`) and very small negative number (`-1.23×10^88`), as well as zero, as illustrated:



Floating-point Numbers (Decimal)

A floating-point number is typically expressed in the scientific notation, with a *fraction* (`F`), and an *exponent* (`E`) of a certain *radix* (`r`), in the form of `F×r^E`. Decimal numbers use radix of 10 (`F×10^E`); while binary numbers use radix of 2 (`F×2^E`).

Representation of floating point number is not unique. For example, the number `55.66` can be represented as `5.566×10^1`, `0.5566×10^2`, `0.05566×10^3`, and so on. The fractional part can be *normalized*. In the normalized form, there is only a single non-zero digit before the radix point. For example, decimal number `123.4567` can be normalized as `1.234567×10^2`; binary number `1010.1011B` can be normalized as `1.0101011B×2^3`.

It is important to note that floating-point numbers suffer from *loss of precision* when represented with a fixed number of bits (e.g., 32-bit or 64-bit). This is because there are *infinite* number of real numbers (even within a small range of says 0.0 to 0.1). On the other hand, a *n*-bit binary pattern can represent a *finite* `2^n` distinct numbers. Hence, not all the real numbers can be represented. The nearest approximation will be used instead, resulted in loss of accuracy.

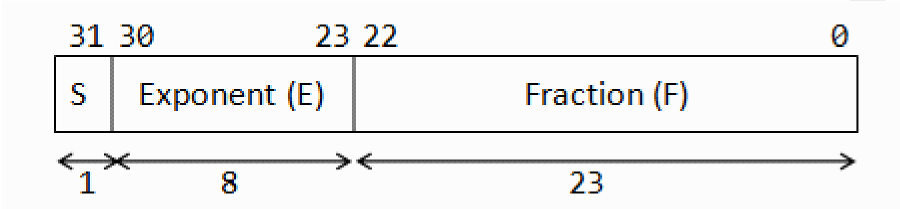
It is also important to note that floating number arithmetic is very much less efficient than integer arithmetic. It could be speed up with a so-called dedicated *floating-point co-processor*. Hence, use integers if your application does not require floating-point numbers.

In computers, floating-point numbers are represented in scientific notation of *fraction* (`F`) and *exponent* (`E`) with a *radix* of 2, in the form of `F×2^E`. Both `E` and `F` can be positive as well as negative. Modern computers adopt IEEE 754 standard for representing floating-point numbers. There are two representation schemes: 32-bit single-precision and 64-bit double-precision.

3.1 IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

- The most significant bit is the *sign bit* (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent *exponent* (E).
- The remaining 23 bits represents *fraction* (F).



32-bit Single-Precision Floating-point Number