Risk and Pricing Solutions

Bits, Bytes and Numbers

THIS DOCUMENT COVERS

Bit Operators

<<

```
a              **11**010101
a << 2         010101**00**
```

>> (Signed Integers)

```
a              110101**01**
a >> 2         **11**110101

a              010101**01**
a >> 2         **00**010101
```

~

```
a              00001101
~a             11110010
```

&

```
a              00001101
b              11101011
-----------------------
a&b            00001001
```

|

```
a              00001101
b              11101011
-----------------------
a|b            11101111
```

^

```
a              00001101
b              11101011
-----------------------
a^b            11100110
```

Bit Properties

a ^ 0s = a

```
a               00001101
0s              00000000
________________________
a ^ 0s          00001101
```

a ^ 1s = ~a

```
a               00001101
1s              11111111
________________________
a ^ 1s          11110010
```

a ^ a = 0

```
a               00001101
0s              00000000
________________________
a & 0s          00000000
```

a & 0s = 0

```
a               00001101
0s              00000000
________________________
a & 0s          00000000
```

a & 1s = a

```
a               00001101
1s              11111111
________________________
a & 1s          00001101
```

a & a = a

```
a               00001101
a               00001101
________________________
a & a           00001101
```

a | 1s = 1s

```
a               00001101
1s              11111111
________________________
a | 1s          11111111
```

a | a = a

```
a               00001101
a               00001101
________________________
a | 1s          00001101
```

a ^ ~a = 1s

```
a       00001101
~a      11110010
________________
a^~a    11111111
```

Bit Manipulation

1 << i	`1 << 3`	0001000	Create a mask with all zeros except a single 1 at bit location i
~(1 << i)	`~(1 << 3)`	11110111	Create a mask with all ones except a single 0 at bit location i
~0 << n	`~0 << 3`	11111000	Create a mask of all 1s except for 0s in the n least significant digits
(1 << i)-1	`(1 << 3)-1`	00000111	Create a mask of all 0s except for 1s in the n least significant digits
(1 << j-i+1)-1)<<i	`(1 << 4-2+1)-1)<<2`	00011100	Create a mask of all 0s except for digits i through j which contain 1s
~(((1 << i - j + 1) - 1) << i))	`~(((1 << i - j + 1) - 1) << i))`	111000111	Create a mask of all 1s except for digits i through j which contain 0s
a+(~b+1)	`5 + (~3+1)`	2	Perform subtraction without using the - key

Bit Operations

(1 << i) | a	`(1 << 2) | 0b00101000`	`00101100`	Set bit i to 1
~(1 << i) & a	`~(1 << 2) & 0b00101100`	`00101000`	Set bit i to 0
(a >> i) & 1	`(0b00101100 >> 3) & 1`	`1`	Get the value of bit i
(~0 << i) & a	`(~0 << 3) & 0b10101111`	`10101000`	Clear i least significant bits
((1 << i)-1) | a	`((1 << 3)-1) | 0b10100000`		Set i least signifcant bits

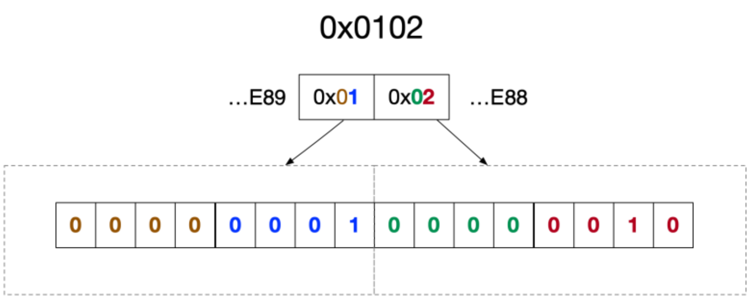
a & (a-1)	`a 00111100`
	`a-1 00111011`
	`a&(a-1) 00111000`
			Clear the rightmost (least significant) 1 digit

Binary representations

Most numeric types consist of multiple bytes. The order in which the bytes are arranged in memory is known as endianness. On a little-endian system, a numeric object's least to most significant bytes are arranged in order from lower memory addresses to higher memory addresses. Consider a .NET unsigned short which occupies 2 bytes or 16 bits

```
ushort a = 0x0102;
```

Figure 1 Endianness



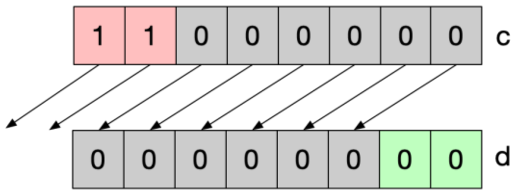
Bit Operators

<< Left Shift

A left shift moves everything n place to the left. The left most n bits are dropped and the rightmost n bits are filled with zeros.

```
byte c = byte.MaxValue;
byte d = (byte)(a >> 2);
```

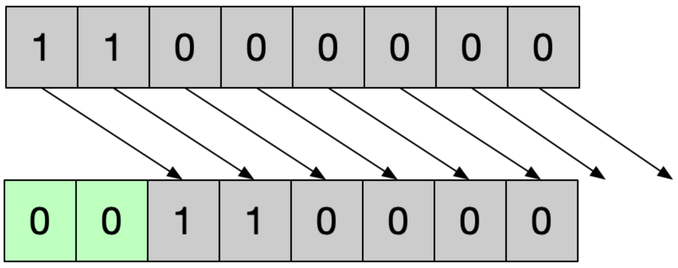
Figure 2 Left Shift



>> Right Shift

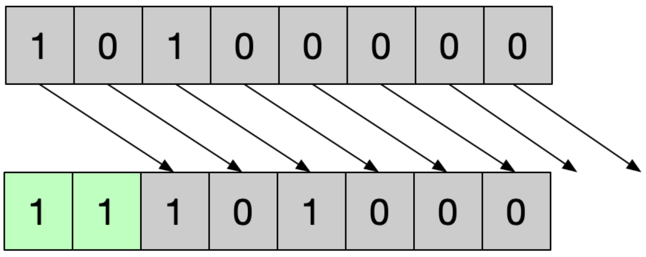
The right shift operator shifts all bits n places to the right. The rightmost n digits are dropped and the leftmost n digits are filled as follows. If the operand is unsigned the left n bits are filled with zeros

```
byte c = 128+64;
byte d = (byte)(c >> 2);
```



If the operand is signed the sign of the bits filled on the left matches the sign bit in the most significant position.

```
sbyte a = -96;
sbyte b = (sbyte)(a >> 2);
```



This extra complexity ensures that shifting right 1 place is equivalent to divinding by two when the operand is negative.

~ bitwise complement

Inverts all the bits

```
a      00001101
~a     11110010
```

& bitwise and

Copies a 1 into the result if the corresponding bits in each operand are 1

```
a      00001101
b      11101011
a&b    00001001
```

| bitwise or

```
a      00001101
b      11101011
a|b    11101111
```

^ exclusive or

```
a      00001101
b      11101011
a^b    11100110
```

Manipulating Binary

Tricks

Adding the same number

Performing integer addition where both operands are the same equal to multiplying by two which is equal to shifting left one place.

```
       00001101
+      00001101
       00011010
```

Multiplication

In binary multiplication is simply shiting the multiplicand left by a number of digits equal to the multiplier.

```
       00001101
*      00000011
       01101000
```

Positional Number Systems

A positional number system represents any real number $\Re$ as a polynomial in the base of the number system.

$$\pm(d_{\infty}\beta^{\infty}+\ldots+d_1\beta^1+d_0\beta^0+d_{-1}\beta^{-1}+d_{-2}\beta^{-2}+\ldots d_{-\alpha}\beta^{-\alpha})=\pm\left(\sum_{k=-\infty}^{\infty}d_k\beta^k\right)$$

When writing polynomial representations of numbers, we use a radix point to separate the whole and fractional parts. We can then drop the powers of the base β as the exponent is implicit in the position of the digit. If a power has no value, we still need to mark it with a co-efficient of zero. Our form becomes.

$$\pm(d_{\infty}\ldots d_1d_0.d_{-1}d_{-2}\ldots d_{-\infty})_{\beta}$$

The following are some examples

- $+34.15_{10}=(3\times10^1+4\times10^0+1\times10^{-1}+5\times10^{-2})_{10}$
- $-11.01_2=-(1\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2})_2=-3.25_{10}$

Integers

If we drop the fractional part of the representation and consider only positive values, we have what programming languages refer to as the ‘unsigned integers

$$(d_{n-1}\ldots d_1d_0)_2=(d_{n-1}2^{n-1}+\ldots d_12^1+d_02^0)=\left(\sum_{k=0}^{n-1}d_kn^k\right)$$

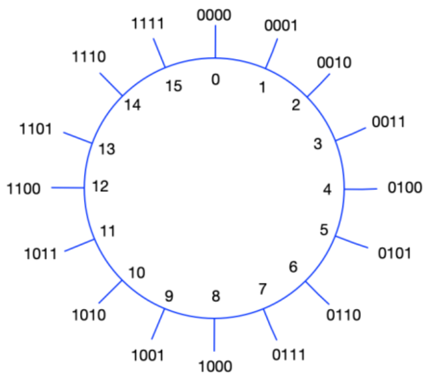
> **NOTE:** $\mathbb{N}$
>
> Mathematicians usually assume the natural numbers $\mathbb{N}$ exclude zero. We use the standard computer science convention that the set $\mathbb{N}$ includes zero.

Such a representation can distinguish between 2^n different values which we can use to represent positive integers in the range $[0,2^n-1]$ To highlight the approach consider the specific case of $n=4$

00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

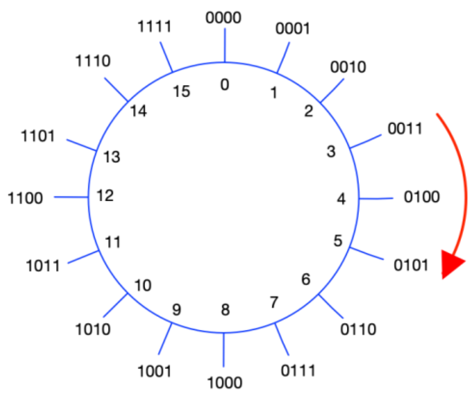
It is useful to visualize such a system as a circle

Figure 3 Unsigned Integers



Unsigned Addition

Addition $(a+b)$ is achieved by starting at a and moving b places clockwise around the wheel. Consider the specific case of $(3+2)_{10} = (0011+0010)_2$ We visualize this as follows



This is very simple binary addition

```
      0011
+     0010
----------
      0101
```

The following C# code shows how we might achieve such addition

Figure 4 Adding Unsigned Integers

```
public uint Add(uint a, uint b)
{
        uint carry = 0;
        uint result = 0;

        for (int bitIdx = 0; bitIdx < SizeInBits; bitIdx++)
        {
                // We deal in one binary digit at multiplicand time. By right
                // shifting multiplicand and bitIdx times we set the bit we want
                // into the least significant bit.
                uint aShifted = (a >> bitIdx);
                uint bShifted = (b >> bitIdx);

                // Now we make use of the fact that the number 1 has
                // in our unsigned representation consists of SizeInBits
                // zeros followed by multiplicand solitary one in the least significant
                // position. We can hence take our shifted valued and logically
                // and them with 1 to ensure we only have the digit values in the least
                // significant locations remaining.
                uint aDigit = aShifted & 1;
                uint bDigit = bShifted & 1;

                // We have three binary digits that feed into the current digit
                // {the multiplicand digit, the multiplier digit and the carry}
                //If one or all three
                // of these are one then the digit will be one, otherwise it will be
                // zero.
                uint sumBit = (aDigit ^ bDigit) ^ carry;

                // We now shift the bit into its correct location and add it into the
                // result
                result = result | (sumBit << bitIdx);

                // Finally calculate the carry for the next digit
                carry = (aDigit & bDigit) | (aDigit & carry) | (bDigit & carry);
        }

        return result;
}
```

Notice in our add method we do not deal with the overflow from the most significant bit. When we add one to the largest representable binary digit which consists of all ones the result is the smallest binary digit consisting of all zeros. In a four bit unsigned integer we would have as follows. Note the bold red overflow is discarded.

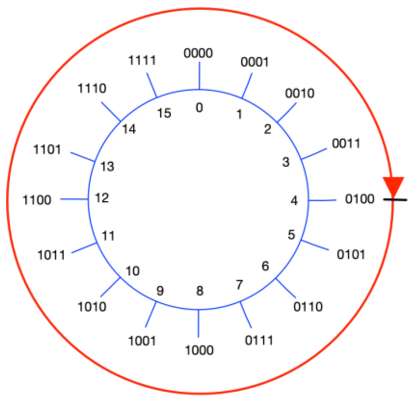
```
      1111
 +    0001
 ---------
     10000
```

By implementing add in this way we have created a modulo number system. If there are n bits in our unsigned integer, then addition is mod_{2^n}. For any unsigned integers a and m we have

$(a + 2^n)_{mod_{2^n}} = a_{mod_{2^n}}$

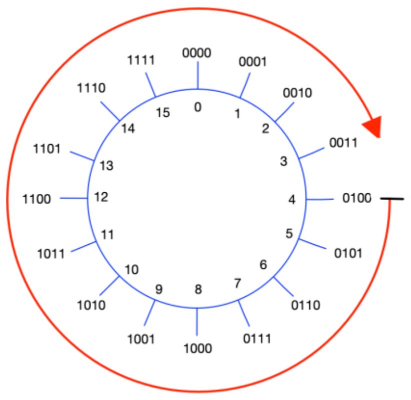
$(a + m.2^n)_{mod_{2^n}} = a_{mod_{2^n}}$ for

In our case adding $2^4 = 16$ to any value gets back to the same value. We show $4 + 2^4 = 4$



Unsigned Subtraction

In our 4-bit integer notice what happens if we add $2^4 - 1 = 15$ to 4. We only rotate to 3. So, adding $2^4 - 1$ is the same as adding -1.



Similarly, adding $2^4 - 2$ is the same as subtracting 2 and adding $2^4 - b$ is the same as subtracting b. We noted in the previous section that $(a + 2^n)_{mod_{2^n}} = a_{mod_{2^n}}$ and so it is self-evident that

$$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$$

This is a very useful result if we combine it with the following observation. Adding any binary number to its complement gives a number consisting solely of 1s.

$$b + {\sim}b = 11\ldots1$$

In our representation we have that $11\ldots1 = 2^n - 1$ hence it follows that

$$b + {\sim}b = 2^n - 1$$

If we substitute this into the expression

$$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$$

We get

$$(a - b)_{mod_{2^n}}(a + \sim b + 1)_{mod_{2^n}}$$

This means we can use our method for addition of unsigned integers to perform subtraction of unsigned integers. The following shows the simple C# code

```
public uint Subtract(uint a, uint b) => Add(a, Add(~b, 1));
```

b (Decimal)	B (Binary)	~b (Binary)	Adding (Clockwise)	Subtraction (Anticlockwise)
0	0000	1111	15	$15 - 2^4 = -1$
1	0001	1110	14	$14 - 2^4 = -2$
2	0010	1101	13	$13 - 2^4 = -3$
3	0011	1100	12	$12 - 2^4 = -4$

Proof that $(a + \sim b + 1)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ ∴

From properties of modulo numbers we know that

$(a + 2^n)_{mod_{2^n}} = a_{mod_{2^n}}$ and hence

$(a - b + 2^n)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ rearranging

$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ adding and subtracting 1

$(a + [2^n - 1 - b] + 1)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$

2s Complement Signed Integers

In the previous section we showed that an unsigned integer can be specified as the polynomial $w = (d_{n-1}2^{n-1} + \dots d_1 2^1 + d_0 2^0) = (\sum_{k=0}^{n-1} d_k n^k)$ and that in this system the three-bit binary number111_2 represents the decimal value 7_{10}.

If we allow both positive and negative values we obtain the set of integers $\mathbb{Z}$. In programming languages these are known as the signed integers.

$$\pm(d_{n-1}\dots d_1 d_0)_2 = \pm(d_{n-1}2^{n-1} + \dots d_1 2^1 + d_0 2^0) = \pm\left(\sum_{k=0}^{n-1} d_k n^k\right)$$

In order to provide support for signed integers most systems use a 2’s complement notation. Twos complement is a way of encoding negative numbers into ordinary binary such that addition still works. In a 2’s complement signed representation we change our most significant digits weighting to $-1 \times d_n 2^{n-1}$giving us

$$w = -1 \times d_n 2^{n-1} + \sum_{k=0}^{n-2} d_k 2^k$$

A n bit 2s complement representation supports the values from $-2^{n-1}\dots(2^{n-1} - 1)$ In this system the three-bit binary number 111_2 is interpreted as the decimal value $-4 + 2 + 1 = -1_{10}$ If all the coefficients are set to 1, i.e. $(1\dots 1_1)$ the value is interpreted as -1. The following shows the values of a 4-bit 2’s complement integer representation.

+00	0000	$(-1 \times 0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$
+01	0001	$(-1 \times 0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$
+02	0010	
+03	0011	
+04	0100	
+05	0101	
+06	0110	
+07	0111	$(-1 \times 0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
-08	1000	$(-1 \times 1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$
-07	1001	
-06	1010	
-05	1011	
-04	1100	
-03	1101	
-02	1110	
-01	1111	$(-1 \times 1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

In the previous section on unsigned integers we saw that the maximum value than can be represented with a n-bit unsigned integer is $(2^n - 1) = 11\ldots1$.We also proved that subtracting a value from $(2^n - 1)$is every bit is the same as flipping the bits

$$(2^n - 1) - b = \sim b$$

In our twos complement notation the binary value with a one in every bit is no longer $(2^n - 1)$but instead is -1. Our equation then becomes

$$-1 - b = \sim b \therefore -b = \sim b + 1$$

In order to negate a positive 2's complement number we flip its bits and add one. We complement it and add one $-b = \sim b + 1$ When adding a pair of twos complement numbers where one of them is negative we simply move around the wheel the number of places in the positive direction of the twos complement binary representation.

Figure 5 Addition of negative unsigned integers

```
short Negate( short x )
{
    short neg = binaryAdd(~x, 1);
    return neg;
}

short binarySubtract( short x, short y )
{
    short minusY = Negate(y);

    return binaryAdd( x, minusY);
}
```

Why twos complement is powerful

The most powerful aspect of 2's complement notation is that we can add positive and negative numbers. If we have n bits, we can represent 2^n values. If we move 2^n points around our modular system, we get back to the same number (overflow).

The algorithm to multiply a 2's complement number by -1 is to flip all its bits using the logical negation operator ~ and then add one. This is beautiful because we can use the same bitwise addition to perform addition and subtraction. To do subtraction just form the ones complement and then do normal addition

Summary

- The most significant bit represents the sign
- Negating a value requires switching all its bits and then adding one
- 1 is represented by 001 and -1 is represented by 111
- N-bit implementation can represent numbers from $-2^{n-1}$to$2^{n-1} - 1$

Real Numbers

> "The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use double and hope for the best"
>
> Bjarne Stroustrup – The C++ Programming Language

Real numbers from the set $\Re$ form the basis of most scientific calculations. Any real number can be written in normalized scientific form.

$$mantissa \times \beta^{e}, 1.0 \leq \text{mantissa} < \beta, \text{e} \in \mathbb{Z}$$

The mantissa is a real number whose value is greater than or equal to 1.0 and less than β. The exponent is an integer. An example of a number in this form is

$$(3.1456224 \times 10^{4})_{10}$$

Given an infinite number of digits in the fractional part of the mantissa any real number can be represented in this general form.

Restricting the representation size

In practice we do not have the luxury of an infinite number of digits in the mantissa and hence it is common to use a more restricted representation. In full generality, if we use a base $\boldsymbol{\beta}$ and have $\boldsymbol{p}$ digits of precision in the mantissa/significand we can **represent** a real number using the representation.

$$\mp\left(d_0.d_1d_{2\ldots}d_{p-1}\right) \times \beta^{e}, \left(1 \leq d_0 < \beta, 0 \leq d_{i:=1..(p-1)} < \beta, e \epsilon z, e_{min} \leq e \leq e_{max}\right)$$

This is the same as the following.

$$\mp\left(d_0\beta^{0} + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}\right) \times \beta^{e}, \left(1 \leq d_0 < \beta, 0 \leq d_{i:=0..(p-1)} < \beta, e \epsilon z\right)$$

DEFINITIONS

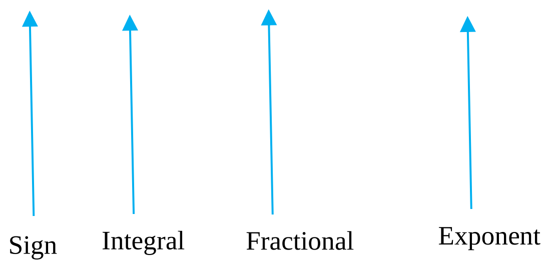
1. Mantissa/Significand	$d_0.d_1d_{2\ldots}d_{p-1}$
2. P	The number of digits in the mantissa/significand
3. e_{min} and e_{max}	The max and minimum exponents
4. β	The base

Properties of the finite representation

Number Of Distinct Values

How many different numbers can a normalised finite form represent? One key point of the standard form is that that the digit d_0 before the decimal point must be in the set $[1, \beta - 1]$ where the digits $d_{2....}d_3$can be zero$[0, \beta - 1]$. The total number of different representable values is given as

$$2 \times (\beta - 1) \times \beta^{(p-1)} \times (e_{max} - e_{min} + 1)$$



Largest and Smallest Values

The largest representable value is $\left(\beta - \beta^{-(p-1)}\right)\beta^{e_{max}}$. The smallest representable value is $-\left(\beta - \beta^{-(p-1)}\right)\beta^{e_{max}}$. The smallest non-negative value is $1.0 \times \beta^{e_{min}}$.

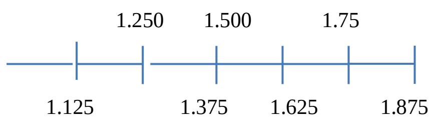
Nearest Values

Example 1

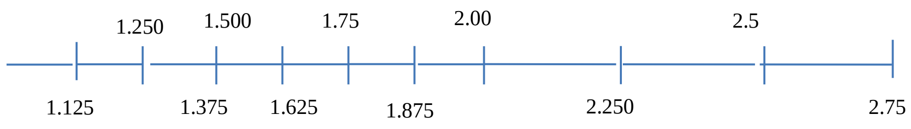
Consider the following case where we use base 2, 4 digits for the mantissa and 2 digits for the exponent.

$$\mp(d_0.d_1 d_2 d_3) \times 2^e$$

If e is 0 then the distance between the two nearest values is $2^{-3} \times 2^0 = 0.125_{10}$



If we increase e to 1 the distance between the two nearest values becomes $2^{-3} \times 2^1 = 0.250_{10}$



For a given exponent e and a given number of binary digits p in the mantissa the distance between the nearest two points in our representation is $2^e \times 2^{-(p-1)}$

Example 2

Consider a floating-point representation with base $\beta = 10, p = 3, e \in \{1,0,-1\}$ The difference between the nearest two numbers depends on the exponent e.

$1.01 \times 10^1 = 10.1$	0.1
$1.01 \times 10^1 = 10.2$	0.1
$1.01 \times 10^1 = 10.3$	0.1
$1.01 \times 10^0 = 1.01$	0.01
$1.01 \times 10^0 = 1.02$	0.01
$1.01 \times 10^0 = 1.03$	0.01
$1.01 \times 10^{-1} = 0.101$	0.001
$1.01 \times 10^{-1} = 0.102$	0.001
$1.01 \times 10^{-1} = 0.103$	0.001

Generalising

Generalizing, for any base β, precisision p and exponent e the distance between the nearest two values is $\beta^e \times \beta^{-(p-1)}$.

Summary Of Properties of finite representations

♦ Possible different values	$2 \times (\beta - 1) \times \beta^{(p-1)} \times (e_{max} - e_{min} + 1)$
♦ Smallest positive value	$1.0 \times \beta^{e_{\min}}$
♦ Largest value	$(\beta - \beta^{-(p-1)})\beta^{e_{max}}$
♦ Smallest value	$-(\beta - \beta^{-(p-1)})\beta^{e_{max}}$
♦ Difference between nearest 2 values	$\beta^e \times \beta^{-(p-1)}$

Examples

BASE 2

If we use base 2 with 4 digits for the mantissa and $e_{max} = 1, e_{min} = -1$ our finite normalized scientific representation becomes

$$\mp(d_0.d_1d_2d_3) \times 2^e$$

The leading integer digit of the mantissa must be non-zero in normalized notation and such a binary digit is in the set[0,1]the only valid value it can take is 1. All the other digits in the mantissa and exponent can be either 0 or 1 giving us a total number of representable values as the product of

1	values of the integer part of the mantissa
2^3	values of the fractional part of the mantissa
2^2	values of the exponent
2	positive and negative values of the exponent
2	positive and negative values of the mantissa

Giving a total number of representable values of

$$[1 \times 2^3 \times 2^2 \times 2 \times 2] = 128$$

We note an important point here. We used 4 bits for the mantissa and 2 bits for the exponent, one bit for the sign of the mantissa and one bit for the sign of the exponent coming to a total of 8 bits. However the total number of representable values is only $128 = 2^7$. This is because the leading integer digit of the mantissa has to be one. (remember in normalized scientific notation the integer digit must be greater than or equal to one and less than β. If $\beta$is 2 then only the integer digit 1 meets this criteria). We need one bit less in the representation. When we look at computer representation of floating point numbers later we will meet this again.

Smallest non-zero positive value	$1.0 \times 2^{-1} = 0.25_{10}$
Largest representable value	$(2 - 2^{-3})2^1 = 3.75_{10}$
Smallest representable value	$-(2 - 2^{-3})2^1 = -3.75_{10}$
Difference between nearest 2 values	$2^e \times 2^{-3}$

Base 10

If we use base 10 with 3 digits for the mantissa and $e_{max} = 99, e_{min} = -99$ our finite normalized scientific representation becomes

♦ Smallest non-zero positive value	1.0×10^{-99}
♦ Largest representable value	$(10 - 10^{-2})10^{99} = 9.99 \times 10^{99}$
♦ Smallest representable value	$-(10 - 10^{-2})10^{99} = -9.99 \times 10^{99}$
♦ Difference between nearest 2 values	$\beta^{e} \times 10^{-2}$

Representation error

Internally real numbers are stored in binary representation, i.e. our base is 2.

$$\mp(d_0.d_1d_{2\ldots}d_{p-1}) \times 2^e, (d_0 = 1, d_{i:=1..(p-1)} \in \{0,1\})$$

All floating-point numbers are **rational numbers** which means they have a terminating expansion in the relevant base. As such most real numbers cannot be expressed exactly. Any number with an infinite expansion cannot be represented.

Also a number which has a finite expansion in one base can have non-finite expansion in another base. If the base is 2, as in binary floating point only **rational** numbers whose denominators are powers of 2 can be represented.

$$\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$$

Converting a base 10 fraction such as 0.1 to binary floating point will result in an infinite expansion which can only be approximated with a finite number of digits. The following sections how to measure this error in approximation.

Absolute error

If we are approximating some real number x with a floating point representation float(x) the absolute error in the approximation is given by.

$$Absolute\ error = float(x) - x$$

We noted earlier that the difference between the two nearest values in a given representation is defined as

$$\beta^e \times \beta^{-(p-1)}.$$

If we need to round a real number to the nearest machine representable number, an upper bound on the maximum absolute error is half this space or $\frac{\beta^{e-(p-1)}}{2}$

Units of the last place (ULPs)

Often, we are interested in the absolute error in terms of the precision of the mantissa, ignoring the exponent part. We often talk of the error in "units of the last place" which means the error in units of the last place of the mantissa. If our mantissa has precision of 4 decimal digits our floating-point representation of 54.13298 is given by 54.13. The absolute difference is given by

z	$5.413298 \times 10^1 = 54.13298$
f	$5.413 \times 10^1 = 54.13$
z-f	$0.000298 \times 10^1 = 00.00298$

The error in units of the last place would be .298. Since one unit in the last place is 0.01 our absolute error of 0.00298 is equal to 0.298 units in the last place. In general we can calculate the value of 1ulp as

$$1ulp = \beta^{-(p-1)}\beta^e$$

In our case we had $1ulp = \beta^{-(p-1)}\beta^e = 10^{-3}10^1 = 10^{-2} = 0.01$

In our general form where we approximate a number x using a floating-point number float(x) with base β and p digits, the error in units in the last place becomes.

$$ulps = \left|d_0.d_1d_{2\ldots}d_{p-1} - \frac{x}{\beta^e}\right|\frac{1}{\beta^{-(p-1)}}$$

Of course, given a number in units of the last place we simply multiply by the exponent term to get the absolute error

$$absolute\ error = ulps \times \beta^{-(p-1)}\beta^e$$

If we have procedure that guarantees that the floating point number chosen to approximate our real number x is the closest floating point number then the error in terms of "units of the last place" can be at most ½ times the value of the unit of the last place

$$0.5ulp = \frac{\beta}{2}\beta^e\beta^{-p}$$

Relative Error

One problem with absolute error is that it does not consider the scale of the number being approximated. Relative error includes the magnitude of the value we are approximating.

$$Relative\ error = \left|\frac{float(x) - x}{x}\right|$$

Using the example from the previous section the relative error is given as

$$\frac{3.1459 - 3.14}{3.1459} = 0.000506$$

Now to see the relationship between absolute and relative error consider approximating the following two real numbers with the nearest floating point representatives; 9.995 and 0.005 (As we are looking at relative error the magnitude given by β^e can be ignored) The relative error of the two numbers are

$$\frac{9.995 - 9.99}{9.995} = \frac{0.005}{9.995} = 0.0005$$

$$\frac{1.005 - 1.00}{1.005} = \frac{0.005}{1.005} = 0.005$$

So, although all numbers with a given β^e have the same maximum absolute error, there relative error varies by a factor of β^e This is known as the wobble.

FROM UNITS OF THE LAST PLACE TO RELATIVE ERROR

For any chosen value of e a number of the form $\mp(d_0.d_1d_{2\ldots}d_{p-1}) \times \beta^e$ can vary in value from $\mp(1.00 \ldots 0) \times \beta^e$all the way to $\mp((\beta-1).(\beta-1)(\beta-1)\ldots(\beta-1)) \times \beta^e \approx \beta^e \times \beta$ As such for any chosen value of e, where the error in terms of ulps is fixed the relative error will vary from $\frac{ulps \times \beta^e}{\beta^e \beta}$ up to $\frac{ulps \times \beta^e}{\beta^e}$ If we have chosen the nearest floating point value to the real value then we can say that the error measured in ulps is 0.5 then our relative error will vary from $\frac{1}{2}\beta^{-p}$ up to $\frac{\beta}{2}\beta^{-p}$

Put another way. For a fixed value of error in ulps the relative error can vary by a factor of β because the mantissa can vary from 1.0 up to just under $\beta - B^{-p} \approx B$.

We now consider a numerical example to cement these formulas. Consider the special case where we use a base of 10 and mantissa of 4 digits. Assuming we always choose the correct nearest floating point number then the error in ulps will be 0.5. In our case the last place has value 10^{-3} . Our chosen value of e is 3 so from our ulp error to absolute error we multiply $0.5 \times 10^{-3}$by 10^3giving us an absolute error of 0.5 units. If we consider the value 1.000×10^3our relative error becomes $\frac{0.5 \times 10^{-3} \times 10^3}{1.000 \times 10^3} = 0.5 \times 10^{-3} = \frac{1}{2}\beta^{-p}$ On the other hand if we consider the value $9.999 \times 10^3$and our relative error becomes $\frac{0.5 \times 10^{-3} \times 10^3}{9.999 \times 10^3} =$ $0.5 \times 10^{-2} = \frac{\beta}{2}\beta^{-p}$ Both of these confirm what we expect. A fixed absolute ulp for a given exponent e gives a relative error that varies by a factor of B depending on the value of the mantissa

Single Precision Float

If we consider single precision number $x = \mp q \times 10^m$the valid values the 32 bits are allocated as

$$\mp(d_0.d_1 d_{2\ldots} d_{23}) \times 2^{e_8 e_7 \ldots e_1}$$

- 1 bit represents the sign of the number $\mp$
- 23 bits for the fractional part of the mantissa
- 8 bit signed number for the exponent

The storage however is a little peculiar. We might expect that using 8 bits for the exponent would allow use to have 256 different values. However four values are reserved for special values such as plus and minus zero and plus and minus infinity.

The representation is $(-1)^s \times 2^{c-127} \times (1.f)_2$ where $-126 \le c \le 127$ (0 and 255 are used for special values) and $1 \le (1.f)_2 \le (1.11111111111111111111111)_2 = 2 - 2^{-23}$ The largest possible value representable is hence $(2 - 2^{-23})2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$. The smallest positive number becomes $(1)2^{-126} \approx 1.2 \times 10^{-38}$

The binary machine number $\varepsilon = 2^{-23}$is the machine epsilon and is hence the smallest positive value such that$1 + \varepsilon \neq 1$. Because $2^{-23} \approx 1.2 \times 10^{-7}$we can infer that single precision floating point has accuracy to six significant decimal figures.

So the mantissa can represent from 1 to $2 - 2^{-23}$ in increments of 2^{-23} which in decimal in approximately from 1 to $2 - 1.2 \times 10^{-7}$ in increments of1.2×10^{-7} so since any single precision mantissa representation can be up to 1.2×10^{-7} from the real number. As such the precision of the mantissa is 6 significant figures.

Numerical Operations

Integer Division

Division is nothing but repeated subtraction. Integer division is defined using the following terms.

$$dividend = (quotient \times divisor) + remainder \therefore$$

$$remainder = dividend - (quotient \times divisor) = dividend \,\%\, divisor \therefore$$

$$quotient = (dividend - remainder) / \, divisor$$

Division Example

$$18 = (16 \times 1) + 2$$

$$2 = 18 - (16 \times 1) = 18 \,\%\, 2$$

$$1 = 18 - \frac{(18 - 2)}{16} = 18 \,/\, 2$$

Euclid's Division Algorithm

The simplest algorithm to perform integer division is to repeatedly subtract. The runtime of this operation is very slow O(q) where q is the quotient

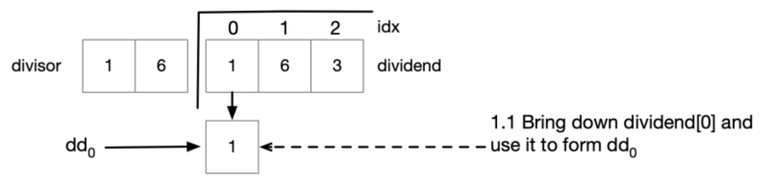
```
private (int q, int r) UnsignedDivide(int dividend, int divisor)
{
    int quotient = 0;
    int remainder = dividend;

    while (remainder >= divisor)
    {
          remainder -= divisor;
          quotient++;
    }

    return (quotient, remainder);
}
```

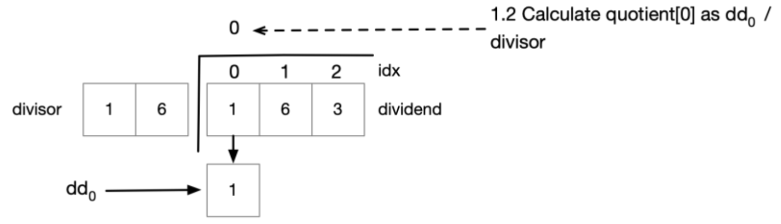
Signed divide is nothing more than a decorator of the unsigned divide method

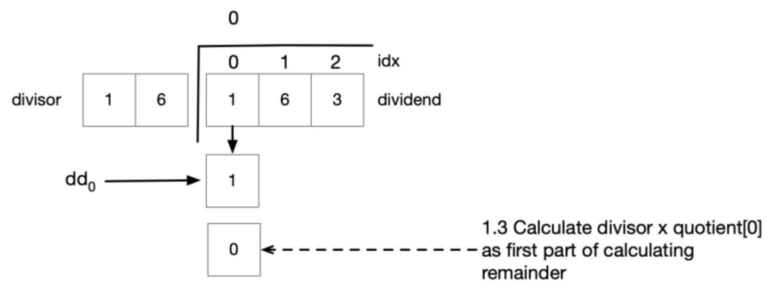
```
public (int q, int r) Divide(int dividend, int divisor)
{
    if (divisor == 0) throw new DivideByZeroException();

    if ( dividend < 0 && divisor < 0 )
          return UnsignedDivide(-dividend,-divisor);

    if (dividend < 0)
    {
          (int q, int r) =UnsignedDivide(-dividend, divisor);
          return (-q,r);
    }

    if (divisor < 0)
    {
          (int q, int r) = UnsignedDivide(dividend, -divisor);
          return (-q, r);
    }

    return UnsignedDivide(dividend,divisor);
}
```

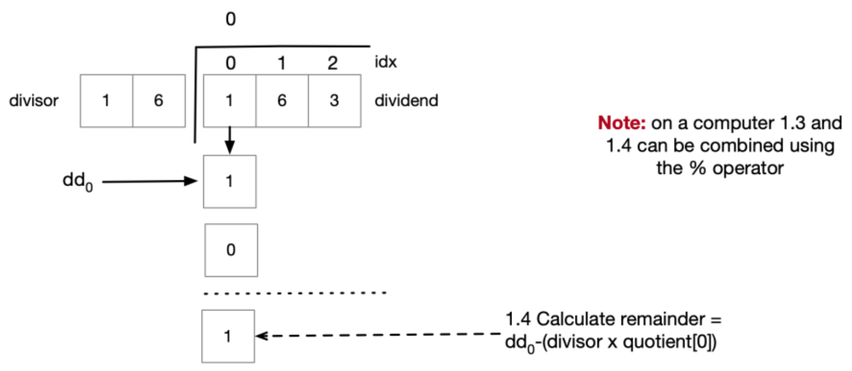
Long Division Algorithm (Any Base)

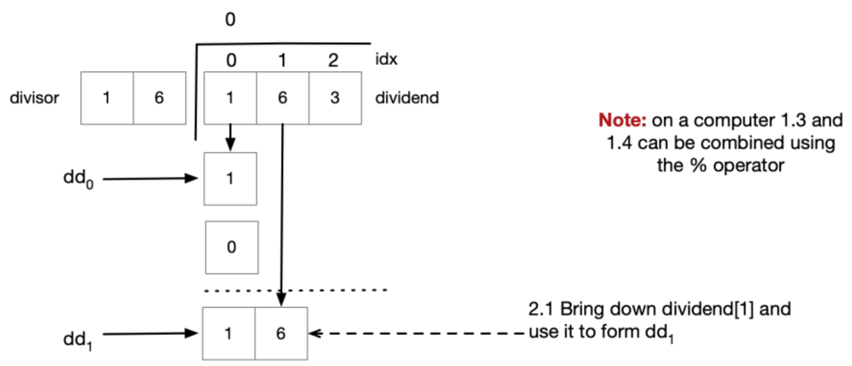
Consider $\frac{163}{16}$






0
0 1 2 idx
divisor 1 6
1 6 3 dividend
dd_0 1
0
1.3 Calculate divisor x quotient[0] as first part of calculating remainder


0
0 1 2 idx
divisor 1 6
1 6 3 dividend
Note: on a computer 1.3 and 1.4 can be combined using the % operator
dd_0 1
0
1
1.4 Calculate remainder = dd_0-(divisor x quotient[0])


0
0 1 2 idx
divisor 1 6
1 6 3 dividend
Note: on a computer 1.3 and 1.4 can be combined using the % operator
dd_0 1
0
dd_1 1 6
2.1 Bring down dividend[1] and use it to form dd_1

The following algorithm performs long division in any base.

```
public (string quotient, string remainder) IntegerLongDivision(string dividend,
int divisor,
     int b = 10)
{
     StringBuilder quotient = new StringBuilder();
     int remainder = 0;
     int dd = 0;

     for (int idx = 0; idx < dividend.Length; idx++)
     {
            // idx.1 copy in next digit into temporary dividend dd
            dd = (dd * b) + dividend[idx].ToIntDigit();

            // idx.2 calculate partial quotient and set into quotient[idx]
            int partialQuotient = dd / divisor;
            quotient.Append(partialQuotient.ToChar());

            // idx.3 calculate this temporary as part of calculating the remainder
            int temp = partialQuotient * divisor;

            // idx.4 Calculate the remainder
            remainder = dd % divisor;

            // the remainder will form the basis of dd[idx+1]
            dd = remainder;
     }

     return (quotient.ToString(), remainder.ToChar().ToString());
}

public static class Extensions
{
     public static int ToIntDigit(this char c)
     {
            if (char.IsNumber(c)) return (int)char.GetNumericValue(c);

            return char.ToLower(c) - 'a' + 10;
     }

     public static char ToChar(this int i)
     {
            if (i >= 0 && i < 10)
                   return (char)(i + '0');

            return (char)(i + 'a' - 10);
     }
}
```

Long Division Algorithm Binary

If we want to do long division in binary the algorithm is very simple

```
public (int quotient, int remainder) UnsignedDivide(int dividend, int divisor)
{
	int numBits = sizeof(byte) * 8;
	int quotient = 0;

	int remainder = 0;

	for (int i = numBits-1; i >= 0; i--)
	{
		// Get the value of the dividend's bit index i
		byte d_i = (byte)((dividend >> i) & 1);

		// Shift the remainder left by 1 bit and add in the
		// bit i from the dividend
		remainder = ((remainder << 1) | d_i);

		// The value of the quotient at index i can only be 1 or 0.
		// It is 1 if the divisor is greater than or equal to
		// remainder, otherwise it is zero
		int q_i = (((remainder >= divisor) ? 1 : 0) << i);

		// copy q_i into the quotient
		quotient |= q_i;

		// If the quotient digit q_i is non zero we subtract the
		// divisor fro, the dividendTemp
		if ( q_i > 0 )
			remainder -= divisor;
	}

	return (quotient,remainder);
}
```

Integer Long Division Algorithm Floating Point Result

```
public string IntegerDivisionWithFloatingPointResult(string dividend, int divisor,
        int b = 10, int maxDigits = 8)
{
        StringBuilder quotient = new StringBuilder();
        int remainder = 1;
        int dd = 0;

        for (int idx = 0; (idx < dividend.Length || remainder > 0)
                    && idx < maxDigits; idx++)
        {
                // Add in a decimal point
                if (idx == dividend.Length)
                        quotient.Append(".");

                // idx.1 copy in next digit into temporary dividend dd
                if (idx < dividend.Length)
                        dd = (dd * b) + dividend[idx].ToIntDigit();
                else
                        // The integer dividend has no more digits so we just increase
                        // by a factor of b as we move to the right side of the point
                        // point
                        dd = (dd * b);

                // idx.2 calculate partial quotient and set into quotient[idx]
                int partialQuotient = dd / divisor;
                quotient.Append(partialQuotient.ToChar());

                // idx.3 calculate this temporary as part of calculating remainder
                int temp = partialQuotient * divisor;

                // idx.4 Calculate the remainder
                remainder = dd % divisor;

                // the remainder will form the basis of dd[idx+1]
                dd = remainder;
        }

        return quotient.ToString();
}
```

Converting Between Bases

Give a number N in base λ we want to convert it to a new base β. Given

$$N = \pm(a_n\lambda^{\infty}+\ldots+a_2\lambda^1 + a_1\lambda^0 + b_1\lambda^{-1} + b_2\lambda^{-2}+\ldots b_\alpha\lambda^{-\alpha})_\lambda$$

we want to find the coefficients c_i and d_i such that

$$N = \pm(c_n\beta^{\infty}+\ldots+c_2\beta^1 + c_1\beta^0 + d_1\beta^{-1}db_2\beta^{-2}+\ldots d_\alpha\beta^{-\alpha})_\beta$$

When doing the conversion we consider the integral and fractional part separately.

Integral Part

Looking first at the integral part we have a number N

$$\boldsymbol{N = (a_n a_{n-1}\ldots a_2 a_1)_\lambda}$$

We want to convert it to base β such that

$$N = (c_m c_{m-1}\ldots c_1 c_1)_\beta$$

We can rewrite this as

$$N = c_1 + \beta\big(c_2 + \beta\big(c_3+\ldots+\beta(c_m)\big)\ldots\big)_\beta$$

If we divide it by β then the remainder is clearly c_1 and the quotient is

$$c_2 + \beta\big(c_3 + \beta\big(c_4+\ldots+\beta(c_m)\big)\ldots\big)_\beta$$

If we repeat this until the quotient is zero we can read off the value of $c_1$to $c_n$giving us the required number in the new base $(c_m c_{m-1}\ldots c_1 c_1)_\beta$

Let us consider the scenario where we want to convert the decimal number 2748 to hexadecimal. We first divide our decimal number by the new base 16

$$\begin{array}{r|l} & \;\;171 \\ \cline{2-2} 16 & 2748 \\ & 1600 \\ \cline{2-2} & 1148 \\ & 112 \\ \cline{2-2} & \;\;28 \\ & \;\;16 \\ \cline{2-2} & \;\;12 \end{array}$$

So after this first division we know that

1) $2748 = [(171 \times 16)] + 12$

We can’t represent 171 as we only have sixteen symbols so we to divide 171 by 16

$$\begin{array}{r|l} & \;\;10 \\ \cline{2-2} 16 & 171 \\ & 160 \\ \cline{2-2} & \;\;11 \end{array}$$

So now we know that

2) $171 = [(10 \times 16)] + 11$

Inserting ii) into i) we get

3) $2748 = [(\{[10 \times 16] + 11\} \times 16)] + 12 = (16^2 \times 10) + (16^1 \times 11) + (16^0 \times 12)$

Which we know is a positional number ABC_{16}

Similarly we can do the same for base 2

FRACTIONAL PART

Consider the situation where we have a fraction part $0 < x < 1$ in some base λ and we want to find the digits $d_k$in the representation

$$x = \sum_{k=1}^{\infty} d_h \beta^{-k} = (0.d_1 d_2 d_3 ...)_\beta$$

We first note that

$$\beta x = (d_1.d_2 d_3 ...)_\beta$$

So if we take our fractional part and multiply it by β then the resulting integral component is the $d_1$we can similarly repeate the process to find the digits $d_2..d_m$

EXAMPLE 1 CONVERT 0.526_{10} TO BASE 8

i) $8 \times 0.526_{10} = 4.208 \therefore 0.526_{10} = \frac{1}{8}4 + \frac{1}{8}(0.208)$

$$8 \times 0.208_{10} = 1.664 \therefore 0.208_{10} = \frac{1}{8}1 + \frac{1}{8}(0.664)$$

$$8 \times 0.664_{10} = 5.312 \therefore 0.664_{10} = \frac{1}{8}5 + \frac{1}{8}(0.312)$$

$$8 \times 0.312_{10} = 2.496 \therefore 0.312_{10} = \frac{1}{8}2 + \frac{1}{8}(0.496)$$

$$0.526_{10} \approx 0.4152_8$$