

Bits, Bytes and Numbers

THIS DOCUMENT COVERS

- ◆ [Bit Operators](#)
 - ◆ [Bit Properties](#)
 - ◆ [Bit Manipulation](#)
 - ◆ [Bit Representations](#)
 - ◆ [Numeric Bit Representations](#)
 - ◆ [Integers](#)
 - ◆ [Real Numbers](#)
 - ◆ [Numerical Operations](#)
-

Risk and Pricing Solutions

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
<hr/>	
a&b	00001001

|

a	00001101
b	11101011
<hr/>	
a b	11101111

^

a	00001101
b	11101011
<hr/>	
a^b	11100110

Risk and Pricing Solutions

Bit Properties

$$a \wedge 0s = a$$

a	00001101
0s	00000000
<hr/>	
$a \wedge 0s$	00001101

$$a \wedge 1s = \sim a$$

a	00001101
1s	11111111
<hr/>	
$a \wedge 1s$	11110010

$$a \wedge a = 0$$

a	00001101
0s	00000000
<hr/>	
$a \wedge 0s$	00000000

$$a \wedge 0s = 0$$

a	00001101
0s	00000000
<hr/>	
$a \wedge 0s$	00000000

$$a \wedge 1s = a$$

a	00001101
1s	11111111
<hr/>	
$a \wedge 1s$	00001101

$$a \wedge a = a$$

a	00001101
a	00001101
<hr/>	
$a \wedge a$	00001101

$$a \mid 1s = 1s$$

a	00001101
1s	11111111
<hr/>	
$a \mid 1s$	11111111

$$a \mid a = a$$

a	00001101
a	00001101
<hr/>	
$a \mid 1s$	00001101

$$a \wedge \sim a = 1s$$

a	00001101
$\sim a$	11110010
<hr/>	
$a \wedge \sim a$	11111111

Risk and Pricing Solutions

Bit Manipulation

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

Risk and Pricing Solutions

Bit Operations

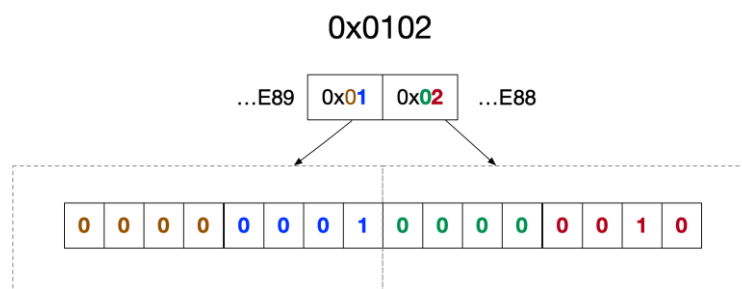
$(1 \ll i) \mid a$	$(1 \ll 2) \mid 0b00101000$	00101100	Set bit i to 1
$\sim(1 \ll i) \& a$	$\sim(1 \ll 2) \& 0b00101100$	00101000	Set bit i to 0
$(a \gg i) \& 1$	$(0b00101100 \gg 3) \& 1$	1	Get the value of bit i
$(\sim 0 \ll i) \& a$	$(\sim 0 \ll 3) \& 0b10101111$	10101000	Clear i least significant bits
$((1 \ll i)-1) \mid a$	$((1 \ll 3)-1) \mid 0b10100000$		Set i least significant 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

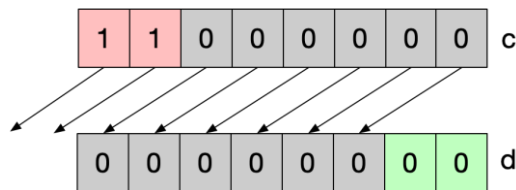
<< Left Shift

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.

Risk and Pricing Solutions

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

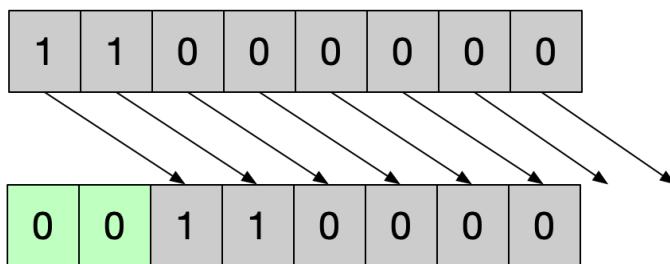
Figure 2 Left Shift



>> Right Shift

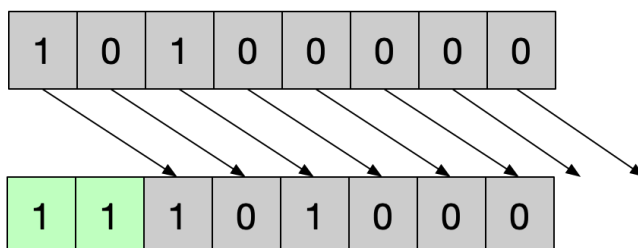
The right shift operator shifts all bits n places to the right. The rightmost n digits are dropped and the leftmost n digits are filled as follows. If the operand is unsigned the left n bits are filled with zeros

```
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 dividing by two when the operand is negative.

~ bitwise complement

Inverts all the bits

```
a      00001101  
~a     11110010
```

Risk and Pricing Solutions

& 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 shifting the multiplicand left by a number of digits equal to the multiplier.

```
      00001101
*     00000011
-----
      01101000
```

Risk and Pricing Solutions

Positional Number Systems

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

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

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 coefficient of zero. Our form becomes.

$$\pm(d_{\infty} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-\infty})_{\beta}$$

The following are some examples

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

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

$$(d_{n-1} \dots d_1 d_0)_2 = (d_{n-1}2^{n-1} + \dots d_12^1 + d_02^0) = \left(\sum_{k=0}^{n-1} d_k 2^k\right)$$

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.

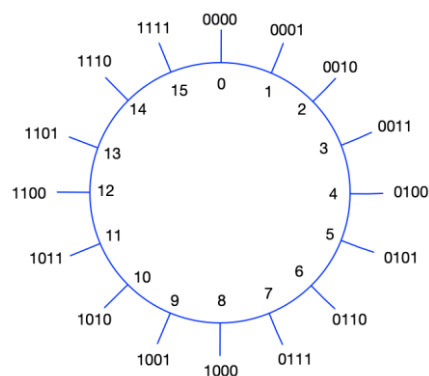
Such a representation can distinguish between 2^n different values which we can use to represent positive integers in the range $[0, 2^n - 1]$ To highlight the approach consider the specific case of $n = 4$

Risk and Pricing Solutions

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

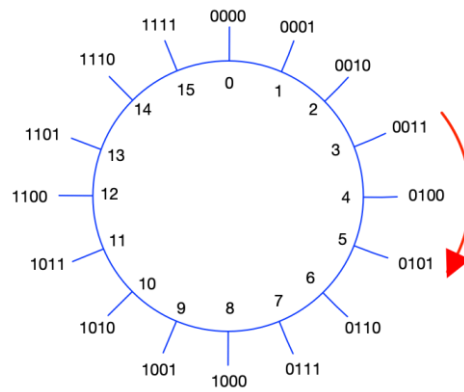
Figure 3 Unsigned Integers



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

Risk and Pricing Solutions



This is very simple binary addition

$$\begin{array}{r} 0011 \\ + 0010 \\ \hline 0101 \end{array}$$

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

Risk and Pricing Solutions

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.

$$\begin{array}{r} 1111 \\ + 0001 \\ \hline \mathbf{1}0000 \end{array}$$

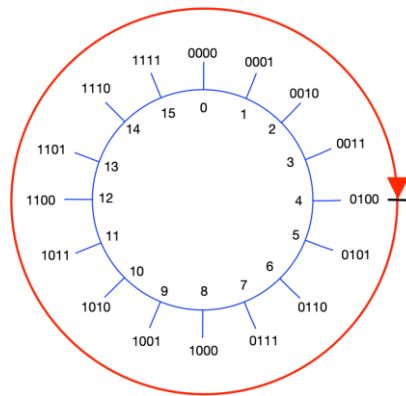
By implementing add in this way we have created a modulo number system. If there are n bits in our unsigned integer, then addition is mod_{2^n} . For any unsigned integers a and m we have

$$(a + 2^n)_{\text{mod}_{2^n}} = a_{\text{mod}_{2^n}}$$

$$(a + m \cdot 2^n)_{\text{mod}_{2^n}} = a_{\text{mod}_{2^n}} \text{ for }$$

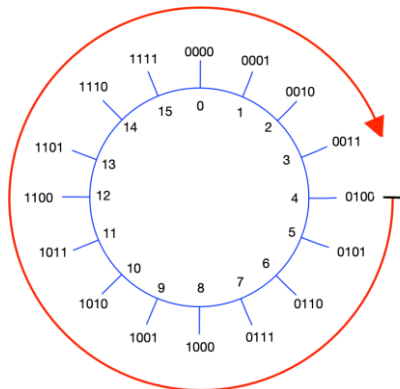
Risk and Pricing Solutions

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 $(a + 2^n)_{\text{mod}_2 n} = a_{\text{mod}_2 n}$ and so it is self-evident that

$$(a + [2^n - b])_{\text{mod}_2 n} = (a - b)_{\text{mod}_2 n}$$

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.

$$b + \sim b = 11 \dots 1$$

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

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

Risk and Pricing Solutions

If we substitute this into the expression

$$(a + [2^n - b])_{\text{mod}_2n} = (a - b)_{\text{mod}_2n}$$

We get

$$(a - b)_{\text{mod}_2n}(a + \sim b + 1)_{\text{mod}_2n}$$

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 $(a + \sim b + 1)_{\text{mod}_2n} = (a - b)_{\text{mod}_2n} \therefore$

From properties of modulo numbers we know that

$$(a + 2^n)_{\text{mod}_2n} = a_{\text{mod}_2n} \quad \text{and hence}$$

$$(a - b + 2^n)_{\text{mod}_2n} = (a - b)_{\text{mod}_2n} \quad \text{rearranging}$$

$$(a + [2^n - b])_{\text{mod}_2n} = (a - b)_{\text{mod}_2n} \quad \text{adding and subtracting 1}$$

$$(a + [2^n - 1 - b] + 1)_{\text{mod}_2n} = (a - b)_{\text{mod}_2n}$$

Risk and Pricing Solutions

2s Complement Signed Integers

In the previous section we showed that an unsigned integer can be specified as the polynomial $w = (d_{n-1}2^{n-1} + \dots d_12^1 + d_02^0) = (\sum_{k=0}^{n-1} d_k2^k)$ and that in this system the three-bit binary number 111_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.

$$\pm(d_{n-1}\dots d_1d_0)_2 = \pm(d_{n-1}2^{n-1} + \dots d_12^1 + d_02^0) = \pm\left(\sum_{k=0}^{n-1} d_k2^k\right)$$

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 $-1 \times d_n2^{n-1}$ giving us

$$w = -1 \times d_n2^{n-1} + \sum_{k=0}^{n-2} d_k2^k$$

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	$(-1 \times 1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

Risk and Pricing Solutions

In the previous section on unsigned integers we saw that the maximum value than can be represented with a n-bit unsigned integer is $(2^n - 1) = 11\dots1$. 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 \sim 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 $2^{n-1} - 1$

Risk and Pricing Solutions

Real Numbers

Real numbers from the set \mathbb{R} form the basis of most scientific calculations. Any real number can be written in normalized scientific form.

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

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 β and have p digits of precision in the mantissa/significand we can **represent** a real number using the representation.

$$\mp(d_0.d_1d_2\dots d_{p-1}) \times \beta^e$$

This is the same as the following.

$$\mp(d_0\beta^0 + d_1\beta^{-1} + \dots + d_{p-1}\beta^{-(p-1)}) \times \beta^e, (1 \leq d_0 < \beta, 0 \leq d_{i=0..(p-1)} < \beta, e \in \mathbb{Z})$$

We also need to restrict the values of the exponent in β^e such that

$$(e \in \mathbb{Z}, e_{\min} \leq e \leq e_{\max})$$

Properties of the finite representation

We now consider the important properties of our finite representation

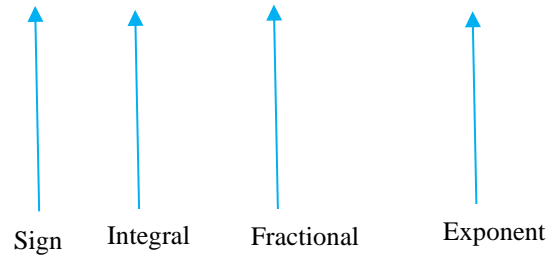
$$\mp(d_0.d_1d_2\dots d_{p-1}) \times \beta^e, (1 \leq d_0 < \beta, 0 \leq d_{i=1..(p-1)} < \beta, e \in \mathbb{Z}, e_{\min} \leq e \leq e_{\max})$$

NUMBER OF DISTINCT VALUES

How many different numbers can a normalised finite form represent? One key point of the standard form is that the digit d_0 before the decimal point must be in the set $[1, \beta - 1]$ where the digits $d_2\dots d_{p-1}$ can be zero $[0, \beta - 1]$. The total number of different representable values is given as

Risk and Pricing Solutions

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



LARGEST AND SMALLEST VALUES

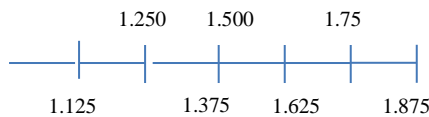
The largest representable value is $(\beta - \beta^{-(p-1)})\beta^{e_{max}}$. The smallest representable value is $-(\beta - \beta^{-(p-1)})\beta^{e_{max}}$. The smallest non-negative value is $1.0 \times \beta^{e_{min}}$.

NEAREST VALUES

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

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

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 $2^e \times 2^{-(p-1)}$

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

Risk and Pricing Solutions

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

Risk and Pricing Solutions

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 β 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 $2^e \times 2^{-3}$

Risk and Pricing Solutions

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 $\beta^e \times 10^{-2}$

Risk and Pricing Solutions

Representation error

ABSOLUTE ERROR

Most real numbers can only be approximated by a finite representation. As such we need to be able to measure the error in approximation. If we are approximating some real number x with a floating point representation $\text{float}(x)$ the absolute error in the approximation is given by.

$$\text{Absolute error} = \text{float}(x) - x$$

RELATIVE ERROR

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

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

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 mean the error in units of the last place of the mantissa. If our mantissa has 3 decimal digits in the fractional part our floating point representation of 5.413298 is given by 5.413. The error in units of the last place would be .298. The unit of the last place itself has value $\beta^{-(p-1)}\beta^e$

In our general form where we approximate a number x by a floating point number $\text{float}(x)$ with base β and p digits, the error in units in the last place becomes.

$$\left| d_0.d_1d_2\dots d_{p-1} - \frac{x}{\beta^e} \right| \frac{1}{\beta^{-(p-1)}}$$

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

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

Risk and Pricing Solutions

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 $\frac{1}{2}$ times the value of the unit of the last place

$$\frac{1}{2} \times \beta^{-(p-1)} = \frac{1}{2} \times \beta^{-p+1} = \frac{\beta}{2} \times \beta^{-p}$$

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

FROM UNITS OF THE LAST PLACE TO RELATIVE ERROR

For any chosen value of e a number of the form $\mp(d_0.d_1d_2\dots d_{p-1}) \times \beta^e$ can vary in value from $\mp(1.00\dots 0) \times \beta^e$ all the way to $\mp((\beta-1).(\beta-1)(\beta-1)\dots(\beta-1)) \times \beta^e \approx \beta^e \times \beta$. As such for any chosen value of e , where the error in terms of ulps is fixed the relative error will vary from $\frac{ulp \times \beta^e}{\beta^e \beta}$ up to $\frac{ulp \times \beta^e}{\beta^e}$. 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 due to the fact that 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 0.5×10^{-3} by 10^3 giving us an absolute error of 0.5 units. If we consider the value 1.000×10^3 our 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 9.999×10^3 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.

Risk and Pricing Solutions

Numerical Operations

Integer Division

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

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

$$\text{remainder} = \text{dividend} - (\text{quotient} \times \text{divisor}) = \text{dividend} \% \text{divisor} \therefore$$

$$\text{quotient} = (\text{dividend} - \text{remainder}) / \text{divisor}$$

Division Example

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

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

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

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

Risk and Pricing Solutions

```
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);
    }

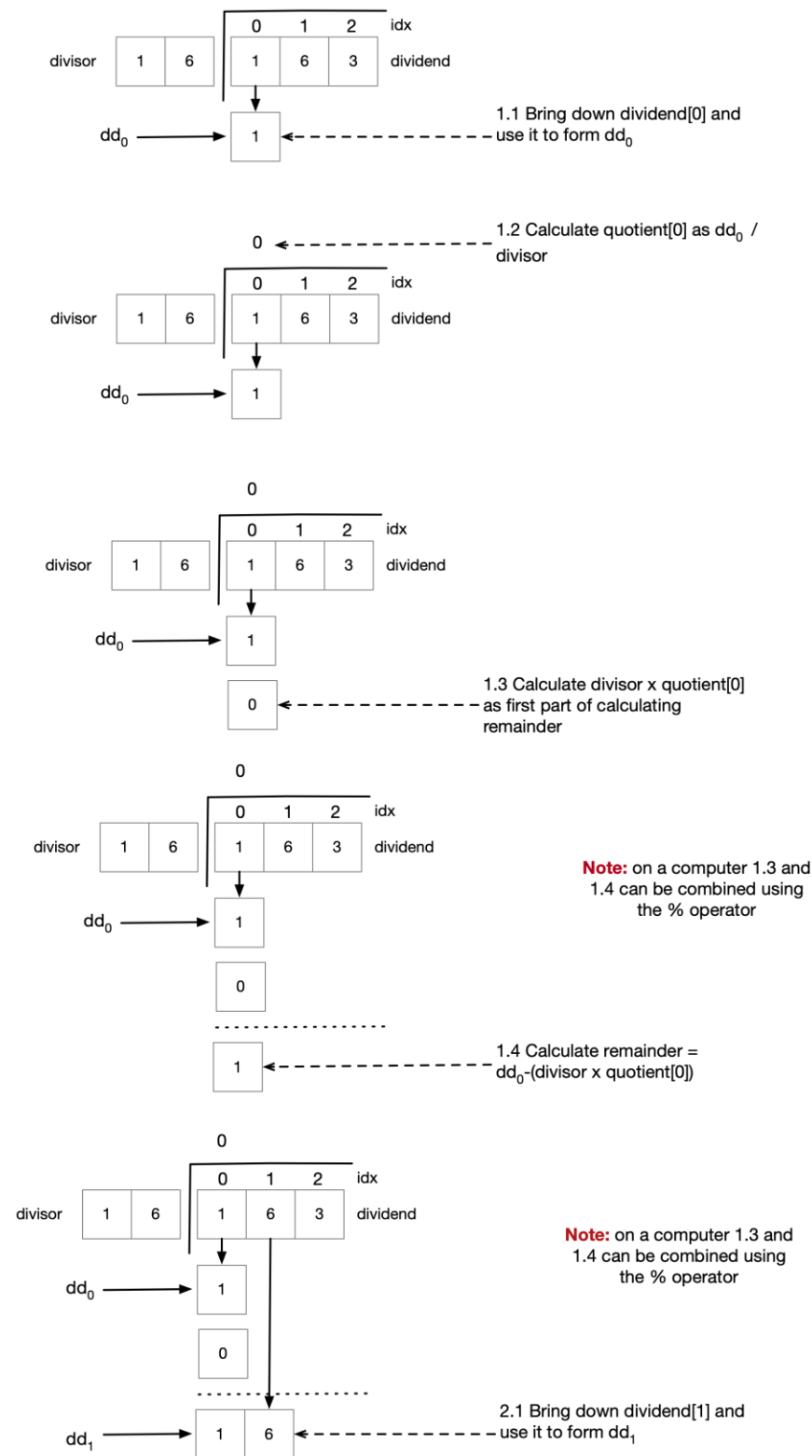
    if (divisor < 0)
    {
        (int q, int r) = UnsignedDivide(dividend, -divisor);
        return (-q, r);
    }

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


Risk and Pricing Solutions

LONG DIVISION ALGORITHM (ANY BASE)

Consider $\frac{163}{16}$



The following algorithm performs long division in any base.

Risk and Pricing Solutions

```
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);
    }
}
```

Risk and Pricing Solutions

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 from the dividendTemp
        if ( q_i > 0 )
            remainder -= divisor;
    }

    return (quotient,remainder);
}
```

Risk and Pricing Solutions

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();
}
```

Risk and Pricing Solutions

Converting Between Bases

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

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

we want to find the coefficients c_i and d_i such that

$$N = \pm(c_n\beta^n + \dots + c_2\beta^2 + c_1\beta^1 + d_1\beta^{-1} + d_2\beta^{-2} + \dots d_\alpha\beta^{-\alpha})_\beta$$

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

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

We want to convert it to base β such that

$$N = (c_m c_{m-1} \dots c_1 c_0)_\beta$$

We can rewrite this as

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

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

$$c_2 + \beta(c_3 + \beta(c_4 + \dots + \beta(c_m)) \dots)_\beta$$

If we repeat this until the quotient is zero we can read off the value of c_1 to c_m giving us the required number in the new base $(c_m c_{m-1} \dots c_1 c_0)_\beta$

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

Risk and Pricing Solutions

$$\begin{array}{r} 171 \\ 16 \overline{) 2748} \\ \underline{1600} \\ 1148 \\ \underline{112} \\ 28 \\ \underline{16} \\ 12 \end{array}$$

So after this first division we know that

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

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

$$\begin{array}{r} 10 \\ 16 \overline{) 171} \\ \underline{160} \\ 11 \end{array}$$

So now we know that

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

Inserting ii) into i) we get

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

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

Risk and Pricing Solutions

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

We first note that

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

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

EXAMPLE 1 CONVERT 0.526_{10} TO BASE 8

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

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

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

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

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

