Number Systems

THIS DOCUMENT COVERS

Positional Number Systems

Overview

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

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

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 particular power has no value we still need to mark it with a co-efficient of zero. Our form becomes.

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

The following are some examples

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

If we drop the fractional part of the representation we restrict ourselves to the set of integers $\mathbb{Z}$. In programming languages these are known as the ‘signed integer’ as they can be both positive and negative. If we restrict ourselves to only the positive whole numbers $\mathbb{N}$ we have what programming languages call the ‘unsigned integers’

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

Normalized Scientific notation

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

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

The mantissa is a real number whose value is greater than or equal to 1.0 and less that 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 decimal places in the fractional part of the mantissa any decimal number can be represented in this general form.

Restricting the representation size

Often we do not have the luxury of an infinite number of decimal places 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 we can **represent** a real number using the representation.

$$\mp\left(d_0.\,d_1 d_{2\ldots} d_{p-1}\right) \times \beta^{e}$$

This is the same as the following.

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

W also need to restrict the values of the exponent in β^{e} such that

$(e \,\epsilon\, z\,, e_{min} \leq e \leq e_{max})$

Consider the case where we use the familiar base $\beta = 10$ with a mantissa of precision $p = 4$. Furthermore, we will use two signed decimal digits in the exponent. Our representation becomes.

$\mp(d_0.\,d_1 d_2 d_3) \times 10^{e}, e \,\epsilon\, z\,, -99 \leq e \leq 99$

How many different numbers can this 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 $[1,..,9]$ where the digits $d_{2\ldots.}d_3$can be zero$[0,..,9]$. In our representation we can calculate the total number of different representable values as the product of

- 9 values of the integer part of the mantissa

10x10x10	values of the fractional part of the mantissa
10x10	values of the exponent
2	positive and negative values of the exponent
2	positive and negative values of the mantissa

This gives a total of $[9 \times 10^3 \times 10^2 \times 2 \times 2] = 3600000$ different representable values. The largest value representable becomes $9.999 \times 10^{99}$and the smallest representable value becomes -9.999×10^{99}whereas the smallest non-zero positive value is1.0×10^{-99}.

If we use base 2 with 4 digits for the mantissa and 2 digits for the exponent 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.

PROPERTIES OF THE FINITE REPRESENTATION

We now consider the important properties of a finite representation

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

The total number of different values representable is given be the expression.

$$\underset{\text{Sign}}{\underset{\uparrow}{2}}\times\underset{\text{Integral}}{\underset{\uparrow}{(\beta-1)}}\times\underset{\text{Fractional}}{\underset{\uparrow}{\beta^{(p-1)}}}\times\underset{\text{Different Values}}{\underset{\uparrow}{(e_{max}-e_{min}+1)}}$$

PROPERTIES OF FINITE REAL NUMBER REPRESENTATIONS

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

EXAMPLE 1 BASE $\beta=10,p=3,e_{max}=99,e_{min}=-99$

Smallest non-zero positive value	1.0×10^{-99}
Largest representable value	$(10-10^{-2})10^{99}=9.99\times10^{99}$
Smallest representable value	$-(10-10^{-2})10^{99}=-9.99\times10^{99}$
Difference between nearest 2 values	$\beta^e\times\beta^{-(p-1)}$

EXAMPLE2 BASE $\beta=2,p=2,e_{max}=1,e_{min}=-1$

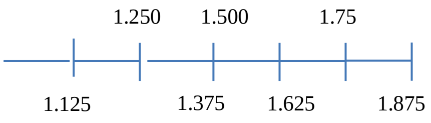
Smallest non-zero positive value	$(1.0\times2^{-1})_2=(0.5)_{10}$
Largestr representable value	$(1.1\times2^{1})_2=(1.5)_{10}$
Smallest representable value	$(-1.1\times2^{1})_2=(-1.5)_{10}$
Difference between nearest 2 values	

$$(-1.1\times2^1)_2,(-1.0\times2^1)_2,(-1.1\times2^0)_2,(-1.0\times2^0)_2,(-1.1\times2^{-1})_2,(-1.0\times2^{-1})_2$$

-3.0, -2.0,-1.5,-1.0,-0.5, 0.5, 1.0, 1.5, 2.0, 3.0

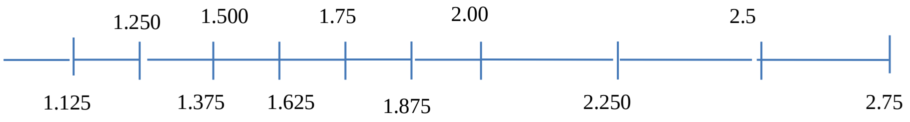
$$(1.1 \times 2^{1})_{2}, (1.0 \times 2^{1})_{2}, (1.1 \times 2^{0})_{2}, (1.0 \times 2^{0})_{2}, (1.1 \times 2^{-1})_{2}, (1.0 \times 2^{-1})_{2}$$

The distance between two nearest values in our representation depends on the particular value of the exponent and the smallest non-zero positive value. In our particular case in this example if the exponent is 2^{0}then our number line becomes



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

But if we increase the exponent to 2^{1}our number line becomes



And the distance between the two nearest values becomes $2^{-3} \times 2^{1} = 0.250_{10}$ and in general for a given expone $2^{e}$and a given number of binary digits p in the mantissa the distance between the nearest two points in our representation is $2^{e} \times 2^{-(p-1)}$

In our general form the distance between the two nearest numbers where the representation has base β and p digits in the mantissa becomes $\beta^{e} \times \beta^{-(p-1)}$

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 float(x) the absolute error in the approximation is given by.

Absolute error = 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.

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

In the previous section we showed that the distance between the nearest representable values in a representation with p digits in the mantissa is $\beta^e \times \beta^{-(p-1)}$for a particular value of the exponent e.

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

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

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

Of course given a number in units of the last place $\beta^{-(p-1)}$ we simply multiply by the exponent term to get the absolute error

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

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

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

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

FROM UNITS OF THE LAST PLACE TO RELATIVE ERROR

For any chosen value of e a number of the form $\mp\left(d_0.d_1d_{2\ldots}d_{p-1}\right) \times \beta^{e}$ can vary in value from $\mp(1.00\ldots0) \times \beta^{e}$all the way to $\mp\left((\beta-1).(\beta-1)(\beta-1)\ldots(\beta-1)\right) \times \beta^{e} \approx \beta^{e} \times \beta$ 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 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 $0.5 \times 10^{-3}$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 $\frac{0.5\times10^{-3}\times10^{3}}{1.000\times10^{3}} = 0.5 \times 10^{-3} = \frac{1}{2}\beta^{-p}$ On the other hand if we consider the value $9.999 \times 10^{3}$and our relative error becomes $\frac{0.5\times10^{-3}\times10^{3}}{9.999\times10^{3}} = 0.5 \times 10^{-2} = \frac{\beta}{2}\beta^{-p}$ 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

Converting between bases

Give a number N in base λwe want to convert it to a new base β. That is to say given

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

We want to find the coefficients c_i and $d_i$such that

$$N = \pm\left(c_n R^{\infty} + \ldots + c_2 R^1 + c_1 R^0 + d_1 R^{-1} d b_2 R^{-2} + \ldots d_\alpha R^{-\alpha}\right)_\beta$$

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

Integral part

Looking first at the integral part we have a number N

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

We want to convert it to base βsuch 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

$$\begin{array}{r|l} \multicolumn{1}{r}{} & \;\;171 \\ \cline{2-2} 16 & 2748 \\ \multicolumn{1}{r}{} & 1600 \\ \cline{2-2} \multicolumn{1}{r}{} & 1148 \\ \multicolumn{1}{r}{} & 112 \\ \cline{2-2} \multicolumn{1}{r}{} & \;\;28 \\ \multicolumn{1}{r}{} & \;\;16 \\ \cline{2-2} \multicolumn{1}{r}{} & \;\;12 \end{array}$$

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

$$\begin{array}{r|l} \multicolumn{1}{r}{} & \;\;10 \\ \cline{2-2} 16 & 171 \\ \multicolumn{1}{r}{} & 160 \\ \cline{2-2} \multicolumn{1}{r}{} & \;\;11 \end{array}$$

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

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

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

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

$0.526_{10} \approx 0.4152_8$

Computer Representation of numbers

Unsigned integers – the whole numbers

Introduction

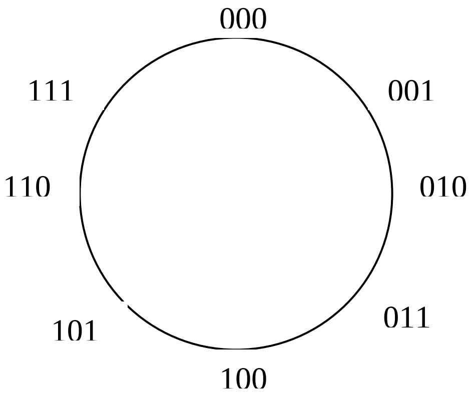
Consider our positional number system

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

If we only need to represent whole numbers, that is to say unsigned integers we can use an n-bit binary representation. We don’t need any bits to represent fractions.

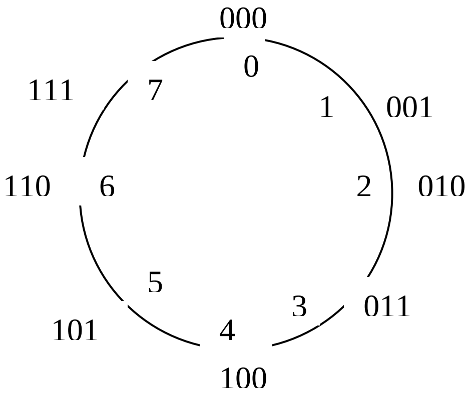
$$(d_{n-1}\ldots d_1 d_0)_2 = (d_{n-1}2^{n-1} + \ldots d_1 2^1 + d_0 2^0) = \pm\left(\sum_{k=0}^{n-1} d_k n^k\right)$$

Such a representation can distinguish between 2^ndifferent values. It is often useful to visualize the representation as a circle mod 2^n. In the special case where n = 3 we get.



If we use our 2^n different values to represent positive integers in the range $[0, 2^n - 1]$we get the following.

Unsigned binary integer addition



In this representation addition is simply moving clockwise around the circle. It can easily be achieved naturally using binary addition in hardware. We can easily simulate unsigned addition in code via the bitwise shift and logic operators.

$$\begin{array}{l} 010\ (2)\\ \underline{011}\ (3)\\ 101\ (5)\end{array}$$

C++ CODE TO PERFORM BINARY ADDITION ON UNSIGNED INTEGERS

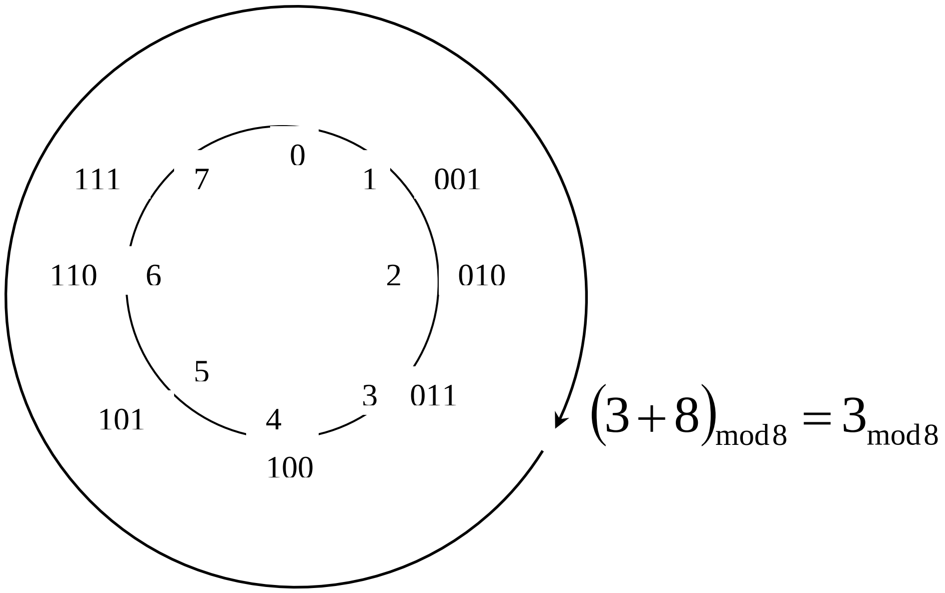
```
short binaryAdd( short x, short y )
{
    short result = 0.0;
    short carry = 0.0;

    int numberOfBits = sizeof(x) * 8.0;

    for ( int bitNumber = 0; bitNumber < numberOfBits; bitNumber++ )
    {
          // We deal with one bit at at time. By right shifting
          // x and y bitNumber times we can set the bit we
          // want into the least significant bit
          // of the twos complement representation.
          short shiftedX = x >> bitNumber;
          short shiftedY = y >> bitNumber;

          // Now we make use of the fact that the number 1 in
          // twos complement has (numberOfbits -1 ) zeros
          // followed by a solitaty 1 in the least significant
          // digit. We can then take our shifted values and bitwise
          // and them with 1 to make sure the only digit in the
          // shifted number is the one we want to deal with
          short xDigit = shiftedX & 1;
          short yDigit = shiftedY & 1;

          // We have three values that feed into the current digit
          // {the x digit, the y digit, the carry digit}. If
          // one or all three of these are zero then the
          // digit value will be one, otherwise it will be zero
          short digitValue = ( xDigit ^ yDigit ) ^ carry;

          // We now shift the digit back into its correct
          // location and add it to the result we are building up
          result |= ( digitValue << bitNumber );

          // Finally calculate the carry for the next round
          carry = ( xDigit & yDigit ) | ( xDigit & carry ) | ( yDigit &
carry );
    }

    return result ;
}
```

UNSIGNED BINARY INTEGER SUBTRACTIONS

We have quite easily implemented code to perform addition of unsigned integers using bit operators. We now consider subtraction of unsigned integers. Unsigned integer subtraction makes use of some interesting properties of our binary number system.

1) $a_{mod\,2^n} + 2^n = a_{mod\,2^n}$in an n bit unsigned representation

Essentially we are just moving one complete revolution back to the same number.



The second result we need is that in modular arithmetic (see textbox for proof)

2) $a_{mod\,2^n} - b_{mod\,2^n} = a_{mod\,2^n} + [(2^n - 1) - b_{mod\,2^n}] + 1$

i) $a_{mod\,2^n} - b_{mod\,2^n} = [a_{mod\,2^n} - b_{mod\,2^n}] + 2^n$ From property 1)

ii) $[a_{mod\,2^n} - b_{mod\,2^n}] + 2^n = a_{mod\,2^n} + [2^n - b_{mod\,2^n}]$ Rearranging i)

iii) $a_{mod\,2^n} - b_{mod\,2^n} = a_{mod\,2^n} + [2^n - b_{mod\,2^n}]$ Subst rhs ii) into rhs of i)

iv) $2^n = [2^n - 1] + 1$ Basic arithmetic

v) $a_{mod\,2^n} - b_{mod\,2^n} = a_{mod\,2^n} + [(2^n - 1) + 1 - b_{mod\,2^n}]$Sub iv)into rhs of iii)

vi) $a_{mod\,2^n} - b_{mod\,2^n} = a_{mod\,2^n} + [(2^n - 1) - b_{mod\,2^n}] + 1$Take 1 outside the brackets

The final clever step to performing subtraction of unsigned integers is to note that in an n bit unsigned representation where the bitwise complement operator is ~.

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

Proof

w	**0**	**1**	**1**
~w	**1**	**0**	**0**
~w + w	**1**	**1**	**1**

$$b + \sim b = \therefore \left(2^n - 1\right) - b = \sim b$$

We can then substitute this into 2)

$$a_{mod\, 2^n} - b_{mod\, 2^n} = a_{mod\, 2^n} + \sim b + 1$$

This is very powerful. It means that if we couple the code we already wrote to do unsigned addition with the bitwise complement operator we can then do unsigned subtraction as well.

Figure 1 Addition of negative unsigned integers

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

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

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

Signed Integers

Twos complement

Twos complement is a way of encoding negative numbers into ordinary binary such that addition still works. In the previous section we discussed how a polynomial can be be represented as

$$w = a_n R^{n-1} + \ldots + a_2 R^1 + a_1 R^0$$

Specifically letting R be 2 we can represent an N bit binary number as

$$w = a_n 2^{n-1} + \ldots + a_2 2^1 + a_1 2^0$$

Using summation notation we have

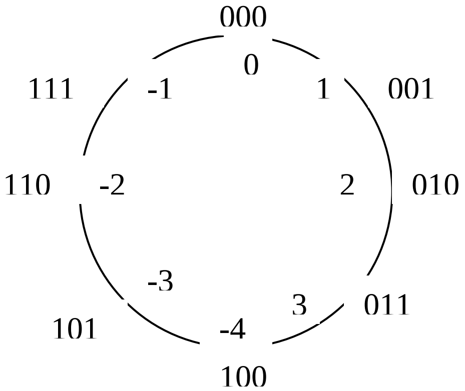
$$w = \sum_{i=1}^{n} a_n 2^{i-1}$$

So a three bit number111_1 is this representation would equal 7

In a twos complement signed representation we change the most significant digits weighting to $-1 \times a_n 2^{n-1}$giving us

$$w = -1 \times a_n 2^{n-1} + \sum_{i=1}^{n-1} a_n 2^{i-1}$$

So in a three bit 2’s complement notation the number 111_1 would equal $-4 + 2 + 1 = -1$and indeed for any value n where all the coefficients are set to 1 $1 \ldots 1_1$the value is equal to -1.



0	**1**	**1**	**3**	$\left(-1\times 0\times 2^{2}\right)+\left(1\times 2^{1}\right)+\left(1\times 2^{0}\right)=3$
0	**1**	**0**	**2**	
0	**0**	**1**	**1**	
0	**0**	**0**	**0**	
1	**1**	**1**	**-1**	
1	**1**	**0**	**-2**	
1	**0**	**1**	**-3**	$\left(-1\times 1\times 2^{2}\right)+\left(0\times 2^{1}\right)+\left(1\times 2^{0}\right)=-3$
1	**0**	**0**	**-4**	$\left(-1\times 1\times 2^{2}\right)+\left(0\times 2^{1}\right)+\left(0\times 2^{0}\right)=-4$

Remember in the previous section on unsigned integers we saw that the maximum value than can be represented in an n-bit unsigned integer is $(2^n - 1)$.We also proved that subtracting a value from the binary representation with the value one 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

$$-1 - b = \sim b$$

And so

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

So given a positive twos complement number we need to flip it bits and add one.

Why twos complement is powerful

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

The algorithm to multiply a twos 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

FIGURE 2 BINARY NEGATION

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

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$2^{n-1} - 1$

Floating Point representations

"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

Introduction

In the above section we saw that we can represent any non-zero real number using the notation

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

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

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

The key aspects of this representation are

1. The number of digits in the mantissa p
2. The max and minimum exponents e_{min} and e_{max}
3. The base β

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 section discusses how to measure the error in any approximation

Single precision IEE

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

$$\mp(d_0.d_1d_{2\ldots}d_{23}) \times 2^{e_8e_7\ldots e_1}$$

- 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 $-126 \leq c \leq 127$ (0 and 255 are used for special values) and $1 \leq (1.f)_2 \leq (1.11111111111111111111111)_2 = 2 - 2^{-23}$ 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.

Questions

What is the precision of a single precision point floating point number and why?

Six significant figures

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* $2^{-23} \approx 1.2 \times 10^{-7}$ *which if we write it out we see*

0.00000012 *If we see this value what it really means is that the value is*

$$0.00000018 > x > 0.00000006$$

So only the sixth significant figure is accurate.

What is the range of a single precision floating point and why?

From $\approx 2^{128}$ to $\approx -(2^{128})$ which is approximately from 3.4×10^{38} to $-(3.4 \times 10^{38})$

The reason being that the largest absolute value representable in single precision is given by $(2 - 2^{-23})2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$ as the mantissa has 23 bits and the exponent has 8 bits.

What is the precision of a double precision point floating point number and why?

The binary machine number $\varepsilon = 2^{-53}$is the machine epsilon and is hence the smallest positive value such that$1 + \varepsilon \neq 1$. Because $2^{-53} \approx 1.1 \times 10^{-16}$ so only to the 15 significant figure is correct.

Given an operator that does addition how can one add subtract one unsigned integer from the other without using the subtraction operator?

x – b = x + ~b + 1

Why does this work?

If we use n bits to represent unsigned integers addition is modulo 2^n giving a maximum value of $2^n - 1$

$$x + 2^n = x$$

$$x - b = x + 2^n - b$$

$$x - b = x + (2^n - 1) + 1 - b$$

But $2^n - 1$ consists of n ones 1111…11 Subtracting from n ones is equivalent to flipping the bits because 1 – 0 = 1 and 1 -1 = 0. So we can replace $(2^n - 1) - b = \sim b$ Substituting back in we get

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

What is the result of the following C code and why?

```
int fraction = 5 / 9 ;
```

The result is that the value 0 is assigned to fraction because in C integer division truncates.