Real Number Arithmetic

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

April 9, 2021

Arithmetic with Real Numbers

- So far we have seen how processors carry out arithmetic operations with signed and unsigned integers.
- A large number of applications need to operate on real numbers with an integer part and a fractional part.
- One problem with real numbers is that unlike integers, these are densely packed – there are an infinite number of real numbers between any two real numbers.
- Thus, only an approximate representation is possible in order to deal with real numbers using digital words of finite size.
- To take a decimal equivalent, let us say we represent real numbers with 3 digits after the decimal point.
- Now all numbers within .001 of each other will have the same representation. Thus, if two numbers are different but within .001 of each other, their difference will be zero using this representation!

Arithmetic with Real Numbers

Suppose we agree to live with the inherent loss of precision when using finite sized words to represent real numbers.

How do we go about representing real numbers then?

- One possible way is to scale the real numbers up by some convenient constant and to round off the scaled value to the nearest integer.
  Now we can carry out arithemetic operations on these (scaled up) integers.
- In fact, to take the example of representing real numbers with three digits after the decimal point, this can be done by scaling up the nubers by 1000.
- the final result of arithmetic operations will need to be scaled down appropriately to interpret it correctly as the real number result.
- This kind of representation is called fixed point representation.

Fixed point Representation

- In fixed point arithemetic, we assume that there is an invisible point at a fixed position, which separates the integral part from the fractional part.
- In digital logic, we use binary representation, such that each bit has a place value associated with it. This place value is a power of 2 in binary representation.
- Analogous to decimal point usage, when we use binary fixed point arithmetic, place values are unity and positive powers of 2 to the left of the invisible binary point and negative powers of 2 to the right of it.

Fixed point Representation

To take an example, suppose the binary point is assumed to lie after the 3 least significant bits of an 8 bit value.
(The binary point lies between b2 and b3 in an 8 bit value represented by b7-b0).

Now the byte 01001101 is treated as 01001.101 and represents:
$0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

This is $8 + 1 + 0.5 + 0.125 = 9.625$

Notice that the byte, interpreted as an unsigned integer, is 77, which is 8 times the value it represents as a fixed point number.

Thus, by assuming a binary point after the 3 least significant bits, we are scaling up by 8.

Fixed point Representation

How do we represent negative numbers?

- We use the same logic of scaling up and follow the convention used for representing negative integers – That the most significant bit has a negative place value.
- The signed integer 10100101 is interpreted as: $-1\times2^7+0\times2^6+1\times2^5+0\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0 = -128+37=-91$
- Similarly, with 3 bits allocated to fractional part, this byte will be interpreted as $-1\times2^4+0\times2^3+1\times2^2+0\times2^1+0\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}=-16+4.625=-11.375$
- Effectively, the represented real number is -91 scaled down by 8.

Fixed point Arithmetic

Now that we have a way of representing real numbers, how do we carry out elementary operations like addition, subtraction, multiplication and division?

- Addition and subtraction require no additional effort. Numbers represented using fixed point can just be added like integers and the binary point remains where it is.
- To take the example of the fixed point after 3 least significant bits, $n1/8 \pm n2/8 = (n1 \pm n2)/8$.
- So we add/subtract the words representing fixed point numbers as if these are integers and the implied binary point remains at the same position.
- Multiplication and division are not so trivial!

Fixed Point Multiplication

- When we multiply two fixed point real numbers, we begin by multiplying these as if these are integers.
- Let us continue with our example of the implied binary point being after the 3 least significant bits. $(n1/8) \times (n2/8) = (n1 \times n2)/64$, so the effective binary point has shifted to a position after 6 least significant bits, as expected.
- We need to round off the extra bits after the implied binary point. In our example of the implied binary point being after the 3 least significant bits, the last 3 bits of the product need to be rounded off.
- This is equivalent to a right shift by 3 bits, with rounding off of the least significant bit retained, based on the last bit shifted out.

Fixed Point Multiplication

- The overall width of the product is twice as much as the width of each number. This is handled in much the same way as integer multiplication is, because there also the size increases.
- So we either allocate two registers for the product (recall DX,AX in 8086) or truncate to single width with overflow warnings.
- Since some of the least significant bits were rounded off, we can accommodate a few extra bits over the original width without causing an overflow.

Fixed Point Division

Let us continue with our example of the implied binary point being after the 3 least significant bits.

- $(n1/8)/(n2/8) = n1/n2$. Thus, we can begin division as if we are dividing the corresponding integers.
- However, the result should not be an integer but should have 3 fractional bits. Thus, the remainder should be further divided by n2 to a precision of 3 fractional bits.
- We can scale up the remainder by 8 in this example and divide by n2 to get the three fractional bits.

Fixed Point Arithmetic

- The advantage of fixed point arithmetic is that it needs no additional hardware.
- Addition and subtraction can be carried out without any additional effort.
- Multiplication and division can use the hardware meant for integers and need minimal additional software to adjust the results.
- However, the range of numbers which can be represented is somewhat limited.
- There is a trade off between precision and range of representable numbers depending on where we place the invisible binary point.
- Precision is improved by including more bits in the fractional part. However, this limits the larges integral part which can be implemented.

Fixed Point Arithmetic

- Fixed point provides a fixed amount of resolution irrespective of the value of the quantity.
- With the binary point assumed to be after 3 least significant bits, the resolution is 1/8, irrespective of the integral value.
- This may be adequate if the integral part is about 100. However, this resolution is poor for quantities whose value is of the order of 1.
- Ideally, we would like a resolution which varies with the magnitude of the stored number.
- The *ratio* of the magnitude of the number to the resolution *relative resolution* should have the same order of magnitude for large and small numbers.
- This is provided by floating point representation.

Floating point Representation

- A number like 12345.678 can be written in many ways – as 12345678×10^{-3} or 12345.678×10^{0} or 12.345678×10^{3} or 0.12345678×10^{5}.
- The decimal point is not at a fixed location, and its position depends on the separately specified power of 10. Therefore, this representation is termed as floating point.
- We can choose one of these forms – say the one with one digit to the left of the decimal point as the *canonical* form of representing the number.
- Since the least significant digit and the integral part of the number are scaled by the same exponent, the *ratio* of the magnitude of the number and the least significant digit (resolution) remains the same as the exponent is changed.
- Thus numbers like 1.2345678×10^{n} have the same relative resolution even though their absolute values span a large range as n is changed.

Floating point Representation

- Floating point representaion has the following components:
  - Sign of the number
  - Integral part with range $<$ base of the exponent.
  - Fractional part known as manitssa
  - Signed exponent with an implicit base.
- In computers, we use 2 as the radix or the base of the exponent.
- This means the integral part can only be 1. If it is 0, we need to shift the binary point and change the exponent till it is 1.
- The operation of shifting the binary point and adjusting the exponent to keep the value of the number the same is called normalization.
- Since the integral part of a normalized number is always 1, we need not store it.

Floating point Representation

- In early computers, different manufacturers chose different parameters for storing the fractional part or the exponent for floating point numbers.
- This led to difficulties when numbers had to be sent from one computer to another.
- IEEE established a standard (IEEE std. 754) in 1980's for representation of floating point numbers to solve this problem.
- This standard covers different data sizes –
  - Single precision (32 bits). Known as float type in C.
  - Double precision (64 bits). Known as type double in C.
  - Extended precision (80 bits) or Ten Byte floats.
- The standard has been revised in 2008.

IEEE 754 Single precision Floating point

Single precision floating point numbers are 32 bits wide.

- Numbers use 1 bit for sign of the number, 8 bits for a biased exponent and 23 bits for significand or mantissa.
- A biased exponent is used rather than sign and magnitude for the exponent. In case of a single precision number, 127 is added to the actual exponent so the stored exponent is never negative within its range of representation.
- Numbers are shifted left or right and the exponent adjusted to keep the value of the number the same. This is repeated till there is a single 1 to the left of the binary point. This is called normalization.
- This '1' to the left of the binary point is not stored. Thus the 23 bit mantissa represents a resolution of 24 bits.

A stored floating point number is interpreted as:

$$(-1)^{\text{sign bit}} \times 1.(\text{23 bit mantissa}) \times 2^{\text{stored exponent}-127}$$

IEEE 754 Single precision Floating point

Let us see a few examples. Consider the number +1.0
The sign bit is 0 and the number is already normalized with a 1 to the left of binary point. The 1 is not stored, so the mantissa is 23 zero bits.
Actual exponent is 0. So the stored exponent is
$0 + 127 = 127 = 01111111$
Order of these components as stored is: sign, stored exponent, mantissa.
so 1.0 is represented as $0011111110 \cdots 0$ or 3F800000H.

The number 0.0 can never be normalized. So a stored exponent of zero and stored mantissa of 23 0s is used for representing it.

Stored exponent of 0 and 255 are used for special cases. Thus normalizable numbers will have a magnitude between 1.0×2^{-126} and $1.1 \cdots 1 \times 2^{127}$

Order of fields in floating point representation

- Different fields of the floating point number are stored in the order: sign, stored exponent, mantissa.
- The advantage of this is that while comparing two floating point numbers, we can compare these as if we are comparing two signed integers.
- If we have two floating point numbers f1 and f2 and if their representations, interpreted as signed integers give I1 and I2, then comparison of f1 and f2 is equivalent to signed comparison of I1 and I2.
- Since the exponent is stored in a more significant position, the integer corresponding to the higher magnitude floating point number has a higher value. Only if the magnitudes of the exponents are equal, the mantissa value is compared.
- Thus we can use signed integer comparison functions directly on the floating point representations. Values of the floating point numbers need not be computed at all if we just want a comparison.

Numbers with special representations

- Let us consider the number 0.(125zeros)$10 \cdots$.
- To normalize it, we shift left (which multiplies it by 2) and decrement the exponent to keep the value unchanged.
- After doing it 125 times, all the zeros to the right of binary point have been shifted out and we get $0.10 \cdots \times 2^{-125}$
- Doing it one more time gives $1.0 \cdots \times 2^{-126}$ and this is in canonical form (a 1 to the left of binary point).
- The stored exponent is $-126 + 127 = 1$, which is permitted. so this is the smallest magnitude number which can be normalized.

The number with a magnitude just smaller than this is: 0.(126zeros)$11 \cdots$.
This cannot be normalized because after 126 left shifts, we reach $0.11 \cdots \times 2^{-}126$. We still do not have a 1 to the left of binary point. Any further left shift will require a stored exponent of 0 or less – which is not allowed.

Numbers with special representations

- The smallest number which can be normalized is 1.0×2^{-126}, with a stored exponent of $-126 + 127 = 1$.
- Stored exponent of 0 is reserved for special numbers.
- If stored exponent $= 0$ and mantissa is also zero, this represents the number zero.
- If stored exponent is 0, but mantissa $\neq 0$, this represents a number which cannot be normalized, but can be represented as a ‘denormal’ number.

A denormal floating point number is interpreted as:

$$(-1)^{\text{sign bit}} \times 0.(\text{23 bit mantissa}) \times 2^{-126}$$

Numbers with special representations

At the other end of the spectrum, stored exponent of 255 is also reserved for special cases.

- If the stored exponent is 255 and the mantissa is all zeros, it represents ∞.
- Depending on the sign bit, this will represent $+\infty$ or $-\infty$.
- If the stored exponent is 255 and the matissa is not zero, it represents NaN (Not a Number).
- Notice that ∞ is a valid quantity and it is meaningful to compute, for example, atan(∞) – which gives $\pi/2$.
- NaN is not a valid quantity and results from expressions whose value is indeterminate like 0.0/0.0, $(\pm\infty)/(\pm\infty)$ etc.

Numbers with special representations

- There are two types of NaNs – quiet Nan and signaling NaN.
- If the stored exponent is 255 and the leading bit of mantissa is a 1, this is a quiet NaN.
- If the stored exponent is 255 and the leading bit of mantissa is a 0, this is a signaling NaN.
- The occurence of a signaling NaN can lead to an interrupt in the system.
- A quiet NaN is allowed to propagate through intermediate stages of expression evaluation.
- Some implementations of floating point math evaluate $x^0 = 1$ even if x is a quite NaN.

Double Precision (64 bit) Real Numbers

- Double precision real numbers are very similar to single precision.
- These use 1 bit for sign, 11 bits for biased exponent and 52 bits for mantissa.
- The stored exponent is biased with the value 1023.
- Numbers are normalized to canonical form (a 1 to the left of binary point).
- Stored exponent of zero and 2047 (all 1's) indicate special numbers just like single precision format.

Extended Precision (80 bit) Real Numbers

- This format is used for intermediate values during chain calculations to reduce round off errors.
- The integral part of the number is not implicitly assumed to be 1, but is stored explicitly as a 1 or a zero. Since there is adequate precision available through the large number of bits, the need for getting an extra bit of resolution is not that important.
- The format uses 1 sign bit, a 15 bit exponent, 1 bit integer part and 63 bit fractional part for the stored number.
- The extra time needed for re-normalizing a number after every computation is saved.

Machine Precision

- All real number formats approximate the infinitely dense packing of real numbers with a discrete representaion possible with finite width digital numbers.
- As a result, two real numbers which are actually different may be represented by the same bit combination.
- Machine precision for a representation is defined as the smallest number ϵ such that $1.0 + \epsilon \neq 1.0$.
- If we subtract a real number from another real number, we may get zero even though the two numbers are distinct, though close in magnitude.
- If the ratio of the two numbers is within $1.0 \pm \epsilon$, these will give a zero result when one is subtracted from another. This can lead to artificial singularities – for example when we divide a small number by (a - b) where a and b are distinct but close to each other.
- For single precision numbers, $\epsilon \approx 2^{-23}$. Notice that this is much larger than the smallest representable number.