Computer arithmetic

Floating point numbers

- Fractions representation
- Floating point numbers – decimal to binary
  - binary to decimal

- Binary to decimal

Let's take an example for n = 110.101

Step 1: Conversion of 110 to decimal

=> $110_2 = (1*2^2) + (1*2^1) + (0*2^0)$

=> $110_2 = 4 + 2 + 0$

=> $110_2 = 6$

So equivalent decimal of binary integral is 6.

Step 2: Conversion of .101 to decimal

=> $0.101_2 = (1*1/2) + (0*1/2^2) + (1*1/2^3)$

=> $0.101_2 = 1*0.5 + 0*0.25 + 1*0.125$

=> $0.101_2 = 0.625$

So equivalent decimal of binary fractional is 0.625

Step 3: Add result of step 1 and 2.

=> 6 + 0.625 = 6.625

Decimal to binary

A) Convert the integral part of decimal to binary equivalent

- Divide the decimal number by 2 and store remainders in array.
- Divide the quotient by 2.
- Repeat step 2 until we get the quotient equal to zero.
- Equivalent binary number would be reverse of all remainders of step 1.

B) Convert the fractional part of decimal to binary equivalent

To get the binary of the fractional part we have to multiple the fractional part by 2 and take the integer part before the decimal point as result and multiple the remaining fractional part by 2 again. We perform this process till the fractional part becomes 0. In some cases the fractional part will not become 0 so, for those scenarios we will stop after N digits, where N will be sufficiently large or given in the question.

Decimal to binary:

Let's take an example for n = 4.47 k = 3

Step 1: Conversion of 4 to binary

1. 4/2 : Remainder = 0 : Quotient = 2
2. 2/2 : Remainder = 0 : Quotient = 1
3. 1/2 : Remainder = 1 : Quotient = 0

So equivalent binary of integral part of decimal is 100.

Step 2: Conversion of .47 to binary

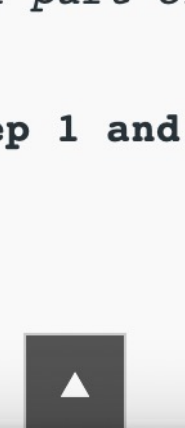
1. 0.47 * 2 = 0.94, Integral part: 0
2. 0.94 * 2 = 1.88, Integral part: 1
3. 0.88 * 2 = 1.76, Integral part: 1

So equivalent binary of fractional part of decimal is .011

Step 3: Combined the result of step 1 and 2.

Final answer can be written as:

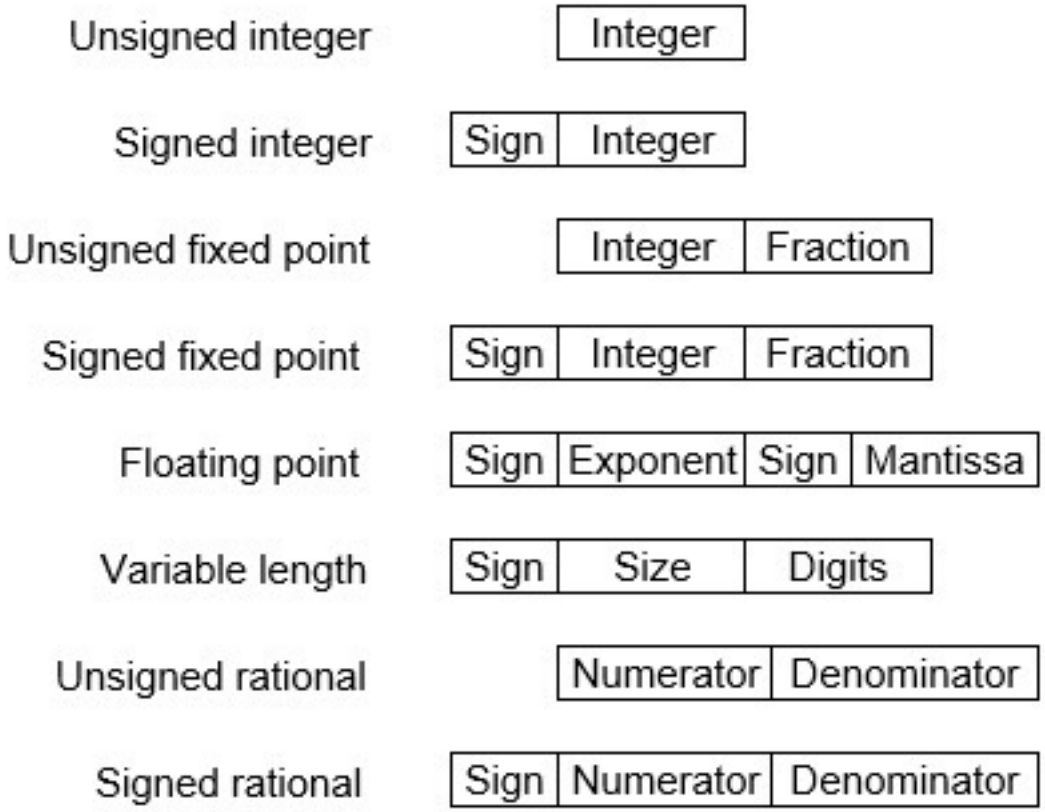
100 + .011 = 100.011



- Convert 0.125 to binary
  1. 0.125 * 2 = 0.250, Integral part: 0 , fractional part not equal to 0
  2. 0.250 * 2 = 0.500, Integral part: 0 , fractional part not equal to 0
  3. 0.500 * 2 = 1.00, Integral part: 1 , fractional part is 0.

Floating- and fixed-point representation

- Storing real numbers

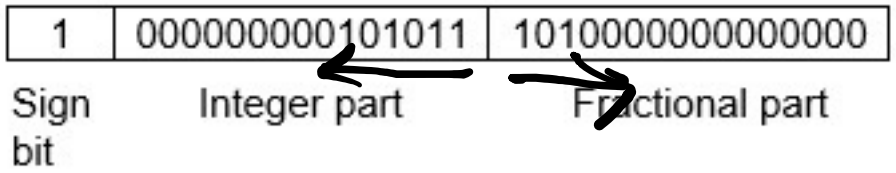


Fixed point representation

- There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing.
- These are (i) Fixed Point Notation and (ii) Floating Point Notation.
- In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

- **Example –**Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

  Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent.
000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

Floating point representation

- Scientific notation – binary point floats to the right of most significant bit and exponent is used to represent.
- For example –

  0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

  $3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

The alternative notation for the last two numbers is called scientific notation, which has a single digit to the left of the decimal point.

- Computer arithmetic that supports such numbers is called floating point because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name float for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

- Has three parts –
  - ~~Base~~
  - Exponent
  - Mantissa / Fraction / Significand
- Easy to represent small and big number in a uniform notation

- In general floating point numbers are represented as $(-1)^S \times F \times 2^E$

126 or 127

- This tradeoff is between precision and range: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented.

0. . . . . .

- Overflow and underflow :

Overflow - A situation in which a positive exponent becomes too large to fit in the exponent field.

Underflow - A situation in which a negative exponent becomes too large to fit in the exponent field

- One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent. In C this number is called *double*, and operations on doubles are called *double precision floating-point arithmetic*; single precision floating point is the name of the earlier format.

~~long~~ float, double, long double ↓ high

- IEEE 754 standards :

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation. According to IEEE 754 standard, the floating-point number is represented in following ways:

- Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
- Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
- Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
- Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Biased floating-point representation

- The 8-bit exponent shows the power of the number. To make the calculations easy, the sign of the exponent is not shown, but instead excess 128 numbering system is used. Thus, to find the real exponent, we have to subtract 127 from the given exponent. For example, if the mantissa is “10000000,” the real value of the mantissa is 128 − 127 = 1.
- The biased exponent is used for the representation of negative exponents. The biased exponent has advantages over other negative representations in performing bitwise comparing of two floating point numbers for equality.

- A ***bias*** of ($2^{n-1} - 1$), where **n** is # of bits used in exponent, is added to the exponent (e) to get biased exponent (*E*). So, the biased exponent (*E*) of *single precision* number can be obtained as

  E = e + 127. ***(E=e+1023. for double)***

- The exponent is an 8-bit unsigned integer from 0 to 255, in biased form: an exponent value of 127 represents the actual zero. The range of exponent in single precision format is -126 to +127. Exponents of −127 and +128 are reserved for special numbers.

- The scientific notation for biased exponent single/double precision is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- Example – $(1460.125)_{10}$ to binary

  $1460.125 = (10110110100.001)_2$

Normalise into scientific notation, we get $1.011011010001 \times 2^{10}$

S=0,e=10, M=011011010000

E=e+127 = 137 = $(10001001)_2$

Single precision representation is

01000100101101101000000000000000

Single precision scientific notation would be

(1+. 0110 1101 0000 0000 0000 000) X $2^{(137-127)}$

- Convert binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$