Chapter 3 — Arithmetic for Computers

3 Floating Point

3 Floating Point

3.1 Representing Large Integers

The general formula for the range of signed integers (in two's complement notation) using n-bits is,

$range_n = [-2^{n-1}, 2^{n-1} - 1]$ Eqn. 1

For an $n = 32$-bit signed integer defined using the C/C++ **int** data type, $range_{32} = [-2^{31}, 2^{31} - 1]$ which is [2,147,483,648; 2,147,483,647], or approximately ±2.1 billion. In C and C++, the **long int** data type can be used to represent 64-bit signed integers[1] for $range_{64} = [-2^{63}, 2^{63} - 1]$ which is [-9,223,372,036,854,775,808; 9,223, 372,036,854,775,807], or approximately ±9.2 quintillion.

However, there are situations where one may need to represent even larger numbers, say 602,214,086,000,000, 000,000,000 which is approximately 602 sextillion[2].

In general, the approximate range of decimal integers that can be represent using n-bits in two's complement can be determined by solving Eqn. 2 for x,

$2^{n-1} = 10^x$ Eqn. 2

Consequently, $range_n = \pm 10^{(log\ 2)(n-1)}$. For example, if $n = 16$, then $range_{16} = \pm 10^{(log\ 2)(16-1)}$ which is approximately $\pm 10^{4.515}$ which is approximately [-32,734; 32,734]. But, that is only an approximation since we rounded (log 2)(16 - 1) to three digits after the decimal point. Of course, we know the correct answer is $[-2^{15}, 2^{15} - 1]$ which is [-32,768; 32,767].

Now, suppose we want to represent integers as large as $\pm 10^{40}$? We can solve Eqn. 2 for n,

$$2^{n-1} = 10^x$$
$$lg\ 2^{n-1} = lg\ 10^x$$

Plugging in $x = 40$ we get $n \approx 16.735$, which means we would need to allocate 17-bytes for an integer. Here are a few more examples:

Bytes	Range	Desired Range	Bytes
1	$\pm 10^{2.104}$	$\pm 10^{50}$	21
2	$\pm 10^{4.515}$	$\pm 10^{75}$	32
4	$\pm 10^{9.332}$	$\pm 10^{100}$	42
8	$\pm 10^{18.965}$	$\pm 10^{150}$	63
16	$\pm 10^{38.231}$	$\pm 10^{300}$	125

There is nothing preventing the authors of a high-level language compiler from making an integer 16- or 32-bytes instead of 4-bytes or 8-bytes. However, the reason they do not is because the CPU only supports 4- or

1 The C and C++ languages do not specify the sizes of the basic data types such as **int** (Java does: an **int** is 4-bytes). It is generally true that on a 32-bit computer system, using a 32-bit compiler, an **int** is 4-bytes and a **long int** will typically be either 4-bytes (the size of an **int**) or 8-bytes (twice the size of an **int**). In a 64-bit system, an **int** will be either 4-bytes (using a 32-bit compiler) or 8-bytes (using a 64-bit compiler). However, in some systems, a **long int** may still be only 8-bytes.

2 Avogadro's Constant is $6.022140857 \times 10^{23}$ mol^{-1}. It's chemistry; don't ask me to explain because I don't understand it.

8-bytes integers (4-byte in a 32-bit system and 8-byte in 64-bit systems) and to perform arithmetic on 16-byte integers would require us to implement the arithmetic algorithms such as addition, multiplication, etc in software, as opposed to doing the arithmetic in hardware. Doing something in hardware is almost—perhaps always—faster than doing that same thing in software.

To summarize, the main drawback with increasing the range of integers by allocating more bytes is that the arithmetic computations, especially multiplication and division, become more computationally expensive, i.e., they take longer. It also becomes more expensive to move the integers back and forth between the CPU and the main memory as more bits means more time to send those bits over the CPU-to-memory bus. So, the question is: is there a better way to represent very large numbers and still be able to perform efficient mathematical computations on them?

Additionally, so far we have only discussed integers, but what about real numbers? Real numbers are ubiquitous in nature and we must be able to represent and perform arithmetic on them as well.

3.2 Converting Decimal Fractions to Binary

We know how to convert a decimal integer into binary and into two's complement notation. How do we convert a decimal fraction into binary? For example, what is the binary representation of 0.1? The basic algorithm is similar to the algorithm for converting a decimal integer into binary, which is: repeatedly divide by two and write the remainders right to left and then write the last non-zero quotient. However, to convert a fraction we repeatedly multiply by two instead. For example, what is the binary representation of 0.4?

3.3 Floating Point Data Types and Internal Representation of Floating Point Numbers

You have seen real numbers represented in scientific notation before.

$4{,}389{,}324{,}849{,}821{,}329{,}188{,}312{,}338{,}823 \approx 4.38 \times 10^{27}$
$0.00000000000000000000000123 = 1.23 \times 10^{-24}$

A floating point number is a finite representation of a real number consisting of a *sign*, *mantissa* (also referred to as the significand), and exponent: $s\ m \times 10^{e}$.

In floating point, the sign is always stored using 1-bit, with 0 being used to represent positive numbers, and 1 representing negative numbers. Of course, the mantissa and exponent are stored in binary format, and the multiplier is a power of 2 rather than 10. To see how a real number could be stored in floating point format, consider the real number 313.5_{10}

What is the floating point representation of: 313.5_{10}?
Convert 313.5 to binary fixed point form: 100111001.1_2
Normalize: $1.001110011_2 \times 2^8$ (Note that there is no need to store the leading 1 bit since we know it is 1)
Binary: 0 1000 001110011

Here is another example,

What is the floating point representation of: -1018.37692_{10}?
Convert to binary fixed point form: $-1111111010.0110000001111101110101000100000100110101011..._2$
Normalize: $-1.1111110100110000001111101110101000100000100110101011..._2 \times 2^9$
Binary: 1 1001 1111110100110000001111101110101000100000100110101011... (requires more than 57 bits)

Note that .37692 requires more than 43 binary digits to represent exactly. In fact, it may never terminate.

In floating point, the number of bits that are allocated to the mantissa and exponent are **finite** and determine the **range** and **precision** of floating point numbers that can be represented.

For example, consider using 3-bits to represent a floating point number where the sign is 1-bit, the mantissa is 1-bit, and the exponent is 1-bit. The mantissa bit may be either 0 or 1, and the exponent bit may also be either 0 or 1. Due to normalization, there is an implicit leading 1 bit before the mantissa bit. Thus, the only numbers that can be represented by this scheme are shown in the table below (the number 0.0 is represented by 0 or 1 for the sign and all 0-bits for the mantissa and exponent).

Sign	Mantissa	Exponent	Real Number
0	0	0	$+0.0$
0	0	1	$+1.0 \times 2^1 = +2.0$
0	1	0	$+1.1 \times 2^0 = +1.5$
0	1	1	$+1.1 \times 2^1 = +3.0$
1	0	0	-0.0
1	0	1	$-1.0 \times 2^1 = -2.0$
1	1	0	$-1.1 \times 2^0 = -1.5$
1	1	1	$-1.1 \times 2^1 = -3.0$

Thus, we can represent 7 unique numbers and the smallest and largest numbers are -3.0 and 3.0.

In general, the number of unique numbers will be $2^n - 1$, where n is the total number of bits allocated to the sign, mantissa, and exponent. The minus 1 term occurs because there are two representations for 0.0: +0.0 and -0.0.

Now consider what happens if we increase the number of bits for the mantissa to 2, but keep the number of bits for the exponent at 1 (the total number of bits is $n = 4$),

$s\ e\ mm$

The largest floating point number is 0 1 11 which is $+1.11_2 \times 2^1 = 3.5$, and the smallest floating point number is 1 1 11 which is $-1.11_2 \times 2^1 = -3.5$. Thus, we can represent 15 unique numbers. Going to 3-bits for the mantissa gives us 31 unique numbers and the smallest and largest numbers are -3.75 and 3.75. Going to 6-

bits for the mantissa gives us 255 unique numbers and the smallest and largest numbers are -3.96875 and 3.96875. Going to 12-bits for the mantissa gives us 16,383 unique numbers and the smallest and largest numbers are -3.99951172 and 3.99951172. In general, as we increase the number of bits for the mantissa, the number of digits we can accurately represent after the decimal point—the *precision*—increases substantially more than the range of numbers, i.e., the **density** of the numbers increases within the range.

Now what happens if we consider the other situation, where we keep the number of bits for the mantissa fixed, and vary the number of bits for the exponent. Consider the case where the number of bits for the mantissa is 1 and the size of the exponent is 2-bits ($n = 4$),

$s\ ee\ m$

The largest floating point number is 0 11 1 which is $+1.1 \times 2^3 = 12$, so we can represent 15 unique numbers and the smallest and largest numbers are -12.0 and 12.0. Consider 3-bits for the exponent. The number of unique numbers is 31 and the smallest and largest numbers are -192.0 and 192.0. Going to 6-bits for the exponent gives us 255 unique numbers and the smallest and largest numbers are $-1.38350580552822 \times 10^{19}$ and $1.38350580552822 \times 10^{19}$. As we increase the number of bits for the exponent, the range of numbers increases substantially more than the number of digits after the decimal point we can accurately represent, i.e, the density decreases.

Ultimately, what one would like to achieve is a compromise between range and precision. For example, 12-bits for the mantissa and 8-bits for the exponent would give a range of $\approx -1.158 \times 10^{77}$ to 1.158×10^{77} which is more than sufficient for most applications. The gap between numbers will be $2^{-12} \approx 0.000244$ which means our precision will be three digits after the decimal point. In general, the gap between pairs of floating point numbers is 2^M where M is the number of bits allocated to the mantissa.

3.4 Floating Point Representation of Very Small Numbers

With our current scheme, there is still another problem. How do we represent really small numbers such as 2.138×10^{-35} or $-189.123123 \times 10^{-8}$? We clearly need a way to represent **negative exponents** as well as positive exponents.

Two possible approaches to storing negative exponents—they are signed integers—would be to use signed magnitude[3] or two's complement representation for the exponent. However, in practice the approach that is used is to add a value, called a **bias**, to the exponent, giving us what is called **biased form** (which is also called **excess-n form**). The bias is simply a positive integer value.

What the bias does is it permits us to represent both negative and positive integers within a certain range. For example, suppose $n = 8$ bits are allocated for an integer; with 8-bits we can represent integers in [0, 255]. If we choose the bias to be 127, then we can represent decimal exponents in [-127, 128], e.g.,

Stored Exponent	Represents this Decimal Exponent
$0000\_0000_2 = 0_{10}$	0 - 127 = -127
$0000\_0001_2 = 1_{10}$	1 - 127 = -126
$0000\_0010_2 = 2_{10}$	2 - 127 = -125
$0111\_1111_2 = 127_{10}$	127 - 127 = 0
$1000\_0000_2 = 128_{10}$	128 - 127 = 1
$1111\_1111_2 = 255_{10}$	255 - 127 = 128

3 https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation

The digital circuits that perform arithmetic on floating point numbers are constructed knowing that the bias is 127 and therefore, any integer where the msb is 0 is an integer that is less than or equal to 0, and any integer with an msb of 1 is an integer that is greater than 0.

Let's look at how this applies to floating point numbers. Let us assume the mantissa is stored using m_{23} = 23-bits, the exponent is e_8 = 8-bits, and the bias b = 127. I will use the notation $e_{8(127)}$ to specify that the exponent is allocated 8-bits and the bias is 127. The format of our floating point numbers will be: $s\ e_{8(127)}\ m_{23}$. What is the floating point representation of: $2.1 \times 10^{-3} = 0.0021$?

Convert 0.0021 to binary fixed point form: 0.0021 = 0.0000_0000_1000_1001_1010_0000_0010_0111...
Normalize: 1.0001_0011_0100_0000_0100_111 × 2^{-9}
Add bias to exponent: -9 + 127 = 118_{10} = 0111_0110_2
Binary Representation: 0 01110110 00010011010000000100111
Hex Representation: 0011 1011 0000 1001 1010 0000 0010 0111 = 0x3B09_A027

3.5 IEEE-754 Floating Point Standard

The representation $s\ m_{23}\ e_{8(127)}$ described above conforms to the IEEE 754 floating point standard and is used by virtually all microprocessors today[4]. The IEEE 754 standard actually defines various **sizes** for floating point numbers. The **single precision** format (which conforms to the C/C++ **float** data type) is $s\ m_{23}\ e_{8(127)}$ consisting of 32-bits. A single precision float can accurately represent around 6-7 digits after the decimal point[5].

The **double precision** format (which conforms to the C/C++ **double** data type) is $s\ m_{52}\ e_{11(1023)}$ consisting of 64-bits.

A double precision float can represent around 15-16 digits of precision. There is also a quadruple-precision format: $s\ m_{112}\ e_{15(16383)}$. The IEEE 754 standard also defines representations for some special values,

+0 (positive zero)	$s = 0,\ m = 0,\ e = 0$	= 0 00000000 000...00
-0 (negative zero)	$s = 1,\ m = 0,\ e = 0$	= 1 00000000 000...00
+∞ (positive infinity)	$s = 0,\ m = 0,\ e = 255$	= 0 11111111 000...00
-∞ (negative infinity)	$s = 1,\ m = 0,\ e = 255$	= 1 11111111 000...00
NaN (not a number)	$s = 0$ or 1, $m \neq 0,\ e = 255$	= x 11111111 xxx...xx

NaN is generated when the result of an operation is undefined. For example, attempting to compute the square root of a number less than 0 or dividing 0/0. Infinity is similar to NaN; it results from operations such as x/0 when x ≠ 0 or when overflow occurs. **Underflow** occurs when the number is too small to be represented, for example, 1.0×10^{-1000}. In this case 0 is used.

4 At least those that support floating pointer numbers. Not all microprocessors include such support.
5 I say around 6-7 because the actual number of digits that are accurate depends on the real number being represented. With some, you get 6 accurate digits and with others, you get 7 digits of accuracy.

3.6 Rounding and Roundoff Error

Suppose we are converting a decimal real number d into its binary floating point representation equivalent f and our unnormalized mantissa ends up being $0.\underline{01001000100001000001010}11000...0 \times 2^{10}$, where bits -1:-23 following the binary point are underlined. If we **truncate** the mantissa by discarding the two 1-bits in bit positions -24:-25, then the floating point number f that is stored will be $2^{10}(2^{-24} + 2^{-25}) \cong 9.15527 \times 10^{-5}$ less than d. Ideally, differences between d and f would be 0, meaning that every decimal real number is stored exactly in floating point format. When that is not possible, we should attempt to minimize $|d - f|$. Since bits -23:-25 are 011_2 which we can interpret as $0.11_2 = 0.75_{10}$, perhaps we should store a 1-bit in bit position -23 rather than a 0-bit, i.e., what is f if we **round up**? In this case the number that is stored is $2^{10}(2^{-23} - 2^{-24} - 2^{-25}) \cong 3.051768 \times 10^{-5}$ larger than d. So, when truncating, $\varepsilon_{\text{trunc}} = 9.15527 \times 10^{-5}$ and when **rounding**, $\varepsilon_{\text{round}} = 3.051768 \times 10^{-5}$. Clearly, rounding results in less error (approximately 2/3 less), but it is unavoidable in most cases. This small amount of error is referred to as **roundoff error**.

Because minimizing roundoff error is crucial to obtaining accurate results when performing floating point arithmetic, IEEE 754 defines several different rounding modes that one may select to use in an application. The default mode is called **round to even, ties to nearest** mode.

In this mode, the default rounding mode is to round the number down or up to the nearest value, e.g., $1.01_2 = 1.25_{10}$ would be rounded down to $1_2 = 1_{10}$; $1.11_2 = 1.75_{10}$ would be rounded up to $10_2 = 2_{10}$. If the number falls exactly in the middle, e.g., 1.1_2 which we could round *down* to 1_2 or *up* to 10_2, then we can say we have a **tie**, and ties are rounded down or up to the nearest *even* number, e.g., 1.1_2 would be rounded *up* to 10_2 since 10_2 is even and 1_2 is not. On the other hand, $10.1_2 = 2.5_{10}$ would be rounded *down* to $10_2 = 2_{10}$ since 10_2 is even and 11_2 is not. Here are some other examples showing the differences between truncation and round to nearest, ties to even modes,

Truncate Mode	**Round to Nearest, Ties to Even Mode**	
$1.25_{10} = 1.01_2$	1_2	round down to nearest
$1.50_{10} = 1.10_2$	10_2	tie; 1.10_2 is rounded *up* to the *even* number 10_2
$1.75_{10} = 1.11_2$	10_2	round up to to nearest
$2.25_{10} = 10.01_2$	10_2	round down to nearest
$2.50_{10} = 10.10_2$	10_2	tie, 10.10_2 is rounded *down* to the *even* number 10_2
$2.75_{10} = 10.11_2$	11_2	round up to nearest
$3.25_{10} = 11.01_2$	11_{20}	round down to nearest
$3.50_{10} = 11.10_2$	100_2	tie, 11.10_2 is rounded *up* to the *even* number 100_2
$3.75_{10} = 11.11_2$	100_2	round up to nearest

Following the 23rd mantissa bit, there are four possible values for the 24th and 25th bits (which are going to be eliminated): 00_2, 01_2, 10_2, or 11_2. The following examples (on the next page) illustrate how we handle the last few mantissa bits. The underlined bits in column 1 are the first 23 bits following the binary point, designated $m_{-1:-23}$. The bold bits in column 1 are the bits we examine to determine how to round.

Binary Mantissa	Truncate Mode	Round to Nearest, Ties to Even Mode	
$1.\underline{001001001001110101000\mathbf{11}}\mathbf{00}$	1.00100100100111010100011	$1.001001001001110101000\mathbf{11}$	round 11.00_2 down to nearest 11_2
$1.\underline{001001001001110101000\mathbf{11}}\mathbf{01}$	1.00100100100111010100011	$1.001001001001110101000\mathbf{11}$	round 11.01_2 down to nearest 11_2
$1.\underline{001001001001110101000\mathbf{11}}\mathbf{10}$	1.00100100100111010100011	$1.00100100100111010100\mathbf{100}$	tie 11.1_2 round to nearest even 100_2
$1.\underline{001001001001110101000\mathbf{11}}\mathbf{11}$	1.00100100100111010100011	$1.00100100100111010100\mathbf{100}$	round 11.11_2 up to nearest 100_2
$1.\underline{001001001001110101000\mathbf{10}}\mathbf{00}$	1.00100100100111010100010	$1.001001001001110101000\mathbf{10}$	round 10.00_2 down to nearest 10_2
$1.\underline{001001001001110101000\mathbf{10}}\mathbf{01}$	1.00100100100111010100010	$1.001001001001110101000\mathbf{10}$	round 10.01_2 down to nearest 10_2
$1.\underline{001001001001110101000\mathbf{10}}\mathbf{10}$	1.00100100100111010100010	$1.001001001001110101000\mathbf{10}$	tie 10.10_2 round to nearest even 10_2
$1.\underline{001001001001110101000\mathbf{10}}\mathbf{11}$	1.00100100100111010100010	$1.001001001001110101000\mathbf{11}$	round 10.11_2 up to nearest 11\

Because the number of bits for the mantissa is finite, not all real numbers can be represented exactly in floating point form. For example, consider the real number $1/3 \cong 0.333333333...$ What is the single precision floating point representation of this number?

Convert to binary fixed point form: $0.0101010101010101010101010101010101010101010101010100101..._2$
Normalize: $1.\underline{01010101010101010101010}1010101010101010101010101010100101..._2 \times 2^{-2}$
Add bias to exponent: $-2 + 127 = 125_{10} = 0111\_1101_2$
Binary: 0 01111101 01010101010101010101010 (note bits -23:-26 were 1010 so 10.10_2 rounded down to 10_2)
Hex: 0x3EAA_AAAA

Which is ≈ 0.333333253860474. As you can see, only six digits after the decimal point are correct.

3.7 Range of Single- and Double-Precision Floating Point Numbers

What are the most positive and most negative single-precision floating point numbers we can represent? You might think the largest would be the number with the largest exponent and the largest mantissa, i.e.,

0 11111111 111..11 = 0x7FFF_FFFF

but it is not. A biased exponent of 255 is used to represent ±∞ and NaN, so the largest possible biased exponent is $254 = 1111\_1110_2$. The unbiased value would 254 – 127 = 127. The largest possible mantissa would be $1.11111111111111111111111_2$. So in binary we would have,

$0\ 11111110\ 11111111111111111111111_2 = \text{0x7F7F\_FFFF}$.

Now to figure out what this is in decimal,

$$= +1.11111111111111111111111_2 \times 2^{127}$$
$$= +1.99999988079071044921875_{10} \times 2^{127}$$
$$\cong +3.4028234663852885981170 42 \times 10^{38}$$

The most negative number is $-3.40282346638528859811704 2 \times 10^{38}$. A rule of thumb is that the range for single-precision floating point numbers is $\pm 3.4 \times 10^{38}$. The range for double-precision floating point numbers is approximately $\pm 1.8 \times 10^{308}$.

Now, what are the smallest positive and negative numbers, i.e., closest to 0.0, that we can represent? You might think it would be,

0 00000000 000...00 = 0x0000_0000

but it is not. A biased exponent of 0 with mantissa equal to 0 are the representations of ±0. If the biased exponent is 0, and the mantissa is nonzero, then this represents a valid floating point number. So, the smallest biased exponent is 00000000. The smallest mantissa cannot be all 0's but rather twenty-two 0's followed by a 1 or 00000000000000000000001. So in binary we would have:

$0\ 00000000\ 00000000000000000000001_2 = \text{0x0000\_0001}.$

Convert to decimal,

$= +1.00000000000000000000001_2 \times 2^{-127}$
$= +1.00000011920928955078125_2 \times 2^{-127}$
$\cong +5.877472454760669702252 2 \times 10^{-39}$

A rule of thumb is that the smallest single-precision floating point numbers are $\pm 5.9 \times 10^{-39}$ and for doubles is $\pm 2.2 \times 10^{-308.}$