Unit 2: Introduction to floating-point numbers

Binary Format

Digital computers store data using bits, that is, a combination of zeros and ones are stored. However, real-world data does not naturally fall into a combination of zeros and ones. For example, we normally use a decimal basis for numbers e.g. 3.14159. There used to be a diverse range of methods for storing real numbers with various problems, which made it difficult to use them reliably. In 1985, the first version of the IEEE Standard for Floating-Point Arithmetic (IEEE 754) was established and the vast majority of hardware floating point units now use this standard.

The standard defines

- Arithmetic formats;
- Interchange formats;
- Rounding rules;
- Operations;
- Exception handling.

Floating-point numbers

The IEEE 754 standard gives a formulaic representation of real numbers. For any representation, there is a trade-off between range and precision. For a given real number, *r*, a digital computer (using the IEEE 754 standard) stores the closest approximation to *r* of the form

$$(-1)^s \left(1.0 + \sum_{n=1}^{p-1} a_n 2^{-n}\right) 2^{e-bias},$$

where s and $a_1, a_2, \ldots, a_{p-1}$ are equal to 0 or 1. The array $a = [a_1, a_2, \ldots, a_{p-1}]$ is known as the *significand*; e is called the *exponent* and is a positive integer, which is stored using binary format with a fixed number of bits. The *unit roundoff* is defined as $u = 2^{-p}$. The values of p and $bias$, and the number of bits used to store e are defined by the format being used:

Name	Common name	p	$bias$	Exponent bits	Decimal digits
binary16	Half precision	11	15	5	3.31
binary32	Single precision	24	127	8	7.22
binary64	Double precision	53	1023	11	15.95
binary128	Quadruple precision	113	16383	15	34.02
binary256	Octuple precision	237	262143	19	71.34

The approximation to *r* is then stored as a binary array by concatenating s, e and $a_1, a_2, \ldots, a_{p-1}$:

s	e	a_1	a_2	...	a_{p-1}

Example

Let $b = 3.14159$. What is the half-precision representation of b and how is it stored in a binary array? Calculate the difference between b and its half-precision representation.

We need to find the closest approximation to b of the form

$$(-1)^s \left(1.0 + \sum_{n=1}^{10} a_n 2^{-n}\right) 2^{e-15}.$$

The value of b is positive and, hence, $s = 0$. Normalising b to have the form $(1.0 + \beta)2^{\alpha}$, where α is an integer and $\beta < 1.0$, we obtain $\alpha = 1$ and $\beta = 0.570795$. Therefore, $e - 15 = 1$, which gives $e = 16 = 10000_2$.

It remains to find the closest approximation to β of the form $\sum_{n=1}^{10} a_n 2^{-n}$.

Now $2^{-1} \leq \beta < 2^0$, which implies $a_1 = 1$.

Let $\beta \leftarrow \beta - 2^{-1} = 0.070795$. Now, $2^{-4} \leq \beta < 2^{-3}$, which implies that $a_2 = a_3 = 0$ and $a_4 = 1$.

Let $\beta \leftarrow \beta - 2^{-4} = 0.008295$. Now, $2^{-7} \leq \beta < 2^{-6}$, which implies that $a_5 = a_6 = 0$ and $a_7 = 1$.

Let $\beta \leftarrow \beta - 2^{-7} = 0.0004825$. Now, $2^{-12} \leq \beta < 2^{-11}$, which implies that $a_8 = a_9 = a_{10} = 0$.

Therefore, $s = 0$, $e = 10000_2$ and $a = 1001001000$.

The binary array is 0100001001001000.

The difference between *b* and its half-precision representation is 0.0004825.

The stored exponents $e = 00 \ldots 0_2$ and $e = 11 \ldots 1_2$ have special interpretations:

	$a_1 = a_2 = \cdots = a_{p-1} = 0$	Otherwise
$e = 00 \ldots 0_2$	Signed zero	Sub-normal number or zero
$e = 11 \ldots 1_2$	$\pm\infty$	NaN

The sub-normal numbers are defined as

$$(-1)^s \left(\sum_{n=1}^{p-1} a_n 2^{-n}\right) 2^{1-bias}$$

and expand the range of numbers that can be used by the chosen format. Most compilers can flush sub-normal numbers to zero via appropriate use of flags and compiler options. As a software developer, it is important to check the behaviour of the compiler being used and make sure that it is as desired.

Real numbers and software

In software codes, real numbers are usually stored using IEEE 754 double-precision format. Sometimes IEEE 754 single-precision format is used and, very occasionally, half-precision or quadruple precision is used. Normally, the same storage format is used throughout the code. For some codes, it may be advantageous to use multiple storage formats within a code e.g., some parts of the code use real numbers stored using double precision and other parts of the code use real numbers that have been stored using single. We classify such codes as *mixed-precision* codes.