MUHAMMAD HAMAD
ASSIGNMENT NO. 3
MS-IC-21354

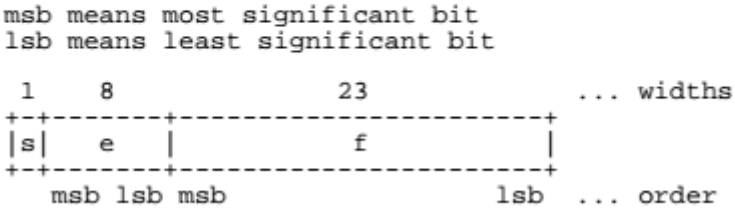
Questions 01. How does IEEE 754 standard store single and double precision numbers?

Ans : Single : A 32-bit single format number X is divided as shown in Fig 1. The value v of X is inferred from its constituent fields thus

1. If $e = 255$ and $f != 0$, then v is NaN regardless of s
2. If $e = 255$ and $f = 0$, then $v = (-1)^s$ INFINITY
3. If $0 < e < 255$, then $v = (-1)^s 2^{e-127} (1 . f)$
4. If $e = 0$ and $f != 0$, then $v = (-1)^s 2^{-126} (0 . f)$ (denormalized numbers)
5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

Figure 1.

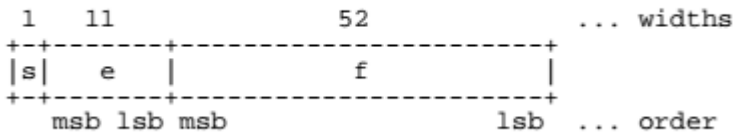
Single Format



Double : A 64-bit double format number X is divided as shown in Fig 2. The value v of X is inferred from its constituent fields thus :

1. If $e = 2047$ and $f != 0$, then v is NaN regardless of s
2. If $e = 2047$ and $f = 0$, then $v = (-1)^s$ INFINITY
3. If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023} (1 . f)$
4. If $e = 0$ and $f != 0$, then $v = (-1)^s 2^{-1022} (0 . f)$ (denormalized numbers)
5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

Double Format



Questions 02. Give some examples of the floating point numbers that cannot be stored exactly using IEEE 754? and why?

Ans : Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

`0.125`

`has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction`

`0.001`

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

0.3 or, better, 0.33 or, better, 0.333 and so on……...

No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

0.0001100110011001100110011001100110011001100110011...

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of 1/10, the binary fraction is 3602879701896397 / 2 ** 55 which is close to but not exactly equal to the true value of 1/10.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

>>> 0.1
0.1000000000000000055511151231257827021181583404541015625

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead

>>> 1 / 10
0.1

Just remember, even though the printed result looks like the exact value of 1/10, the actual stored value is the nearest representable binary fraction. Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers 0.1 and 0.10000000000000001 and 0.1000000000000000055511151231257827021181583404541015625 are all approximated by 3602879701896397 / 2 ** 55.

Another example of something that should be true but isn't in IEEE floating-point arithmetic is the following equation.

$$1/6+1/6+1/6+1/6+1/6+1/6=1.$$

The reason this fails is that 1/6 has no precise representation with a fixed number of mantissa bits, and so it must be approximated with the IEEE format. As it happens, it underestimates slightly. When we add this underestimation to itself 6 times, the total ends up being less than 1.

Question 03. Assume three floating point numbers a, b and c. Does Associative or Commutative law hold for floating point numbers? i.e. a + (b + c) == (a + b) + c, and a(bc) = (ab)c or a + b = b + a and ab = ba

Ans : **Associativity law for addition: $a + (b + c) = (a + b) + c$**

Let $a = -2.7 \times 10^{23}$, $b = 2.7 \times 10^{23}$, and $c = 1.0$

$a + (b + c) = -2.7 \times 10^{23} + (2.7 \times 10^{23} + 1.0) = -2.7 \times 10^{23} + 2.7 \times 10^{23} = 0.0$

$(a + b) + c = (-2.7 \times 10^{23} + 2.7 \times 10^{23}) + 1.0 = 0.0 + 1.0 = 1.0$

Beware – Floating Point addition not associative!

The result is approximate…

With IEEE floating-point numbers, however, many laws fall apart. The commutative law still holds for both addition and multiplication.

Question 04. What's the good way to compare floating point numbers?

Due to rounding errors, most floating-point numbers end up being slightly imprecise. As long as this imprecision stays small, it can usually be ignored. However, it also means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ slightly, and a simple equality test fails. For example:

```
float a = 0.15 + 0.15
float b = 0.1 + 0.2
if(a == b) // can be false!
if(a >= b) // can also be false!
```

Epsilon comparisons If comparing floats for equality is a bad idea then how about checking whether their difference is within some error bounds or epsilon value, like this:

bool isEqual = fabs(f1 – f2) <= epsilon;

With this calculation we can express the concept of two floats being close enough that we want to consider them to be equal. But what value should we use for epsilon?

Relative epsilon comparisons The idea of a relative epsilon comparison is to find the difference between the two numbers, and see how big it is compared to their magnitudes. In order to get consistent results you should always compare the difference to the larger of the two numbers. In English:To compare f1 and f2 calculate diff = fabs(f1-f2). If diff is smaller than n% ofmax(abs(f1),abs(f2)) then f1 and f2 can be considered equal.

Comparing floating-point values as integers

There is an alternative to heaping conceptual complexity onto such an apparently simple task: instead of comparing a and b as real numbers, we can think about them as discrete steps and define the error margin as the maximum number of possible floating-point values between the two values.

This is conceptually very clear and easy and has the advantage of implicitly scaling the relative error margin with the magnitude of the values. Technically, it's a bit more complex, but not as much as you might think, because IEEE 754 floats are designed to maintain their order when their bit patterns are interpreted as integers.

Questions 05. Does adding Big floating point number with small one affect final result?

Ans : `A seemingly innocuous operation like addition can greatly increase the amount of precision needed to represent the resulting number. The reason for this is that two numbers represented in scientific notation cannot be added unless they have the same exponent. Consider the following illustration of the computation 192 + 3 = 195 :`

The binary representation of 192 is $1.5*2^7$ = 0 10000110 100 … 0

The binary representation of 3 is $1.5*2^1$ = 0 10000000 100 … 0

Note that both these numbers have very simple mantissas requiring only 1 bit of precision. Their sum, however requires 7 bits to represent the mantissa:

The binary representation of 195 is $1.5234375*2^7$ = 0 10000110 1000011 … 0

This occurs because the number 3 is shifted from $1.5*2^1$ to $0.0234375*2^7$ first, so that the addition can be accomplished:

3 : $0.0234375*2^7$ = 0 10000110 0000011 … 0
192 : $1.5 *2^7$ = 0 10000110 1000000 … 0
195 : $1.5234375*2^7$ = 0 10000110 1000011 … 0

In general, the greater the discrepancy in size between the exponents of two numbers, the greater the precision needed to represent their sum. Should the discrepancy be too large, the smaller number will be lost entirely and the calculated sum will simply equal the larger number.