CS1100 - Lecture 16

Instructor : Jasine Babu
Teaching Assistants : Nikhila K N, Veena Prabhakaran

In the previous class, we had studied that integer variables can be represented precisely. Now, let us consider the representation of real numbers. It is a well known fact that not every Real number can be precisely represented using finitely many digits in decimal representation. For example, the rational value `1/3` does not have a finite decimal representation. However, if we fix the number of digits after the decimal point as 3, then `1/3` can be approximated as 0.333. If we have more digits after the decimal point, the approximation becomes better. But we can not achieve 100% accuracy. In binary representation also, we have to handle similar issues.

Converting Real numbers from decimal to binary

Consider a Real number which can be represented using finitely many digits. For example, consider 52.375. Here, the places after the decimal point have their positional values as negative powers of 10. The first digit after the decimal point has positional value `1/10`, the next digit after the decimal point has positional value `1/100` and so on. We have

$$52.375 = 5 \times 10^1 + 2 \times 10^0 + 3/10 + 7/10^2 + 5/10^3$$

To convert this number to binary we will convert the integer part (52) and fractional part (.375) separately into binary and concatenate both the results. The conversion of integer part is using the same method we discussed in the previous class. The binary equivalent of 52 is 00110100. To convert the fractional part into binary we will repeatedly multiply the fractional part by 2 and note down the integer part of the result of multiplication, until the resultant fractional part becomes 0.

$2 \times 0.375 = 0.75$ —— integer part 0
$2 \times 0.75 = 1.5$ —— integer part 1
$2 \times 0.5 = 1.0$ —— integer part 1

Thus,

$0.375_{10} = 0.011_2$ and
$52.375_{10} = 00110100.011_2$

Consider the decimal to binary conversion of the number 0.3.

$2 \times 0.3 = 0.6$ —— integer part 0
$2 \times 0.6 = 1.2$ —— integer part 1
$2 \times 0.2 = 0.4$ —— integer part 0
$2 \times 0.4 = 0.8$ —— integer part 0
$2 \times 0.8 = 1.6$ —— integer part 1
$2 \times 0.6 = 1.2$ —— integer part 1
$2 \times 0.2 = 0.4$ —— integer part 0
$2 \times 0.4 = 0.8$ —— integer part 0
$2 \times 0.8 = 1.6$ —— integer part 1
$2 \times 0.6 = 1.2$ —— integer part 1

Therefore, the binary equivalent of 0.3 is 0.01001100110011.... It requires an infinite number of bits after the decimal point though, we have an exact finite decimal representation 0.3 for the same number.

Computer representation of Real numbers

As we saw in the previous section, not every Real number has an exact finite representation in binary. Since, in a computer the storage is definitely finite, some loss in precision while representing Real numbers is unavoidable.

In C language, the data types *float*, *double*, *long double* are used for representing Real numbers. These are called floating point number representations. The accuracy of the representation is the least for `float` and the best for *long double*. But none of these representations are precise, as we explained above. Therefore, while doing comparisons between two floating point numbers, a programmer should be cautious of errors because of the lack of precision in the representation. The following program demonstrates this fact.

```
#include<stdio.h>
int main()
{

   float x, y, z;

   x=1;
   y=5;

   if (x/y == 0.200000)
      printf("equality check 1 passed \n");
   else
     printf("equality check 1 failed \n");

   z=x/y;
   printf("%f\n", z);
   x=1.3333;
   y=52.2;
```

```
    z=x+y;
    printf("%f + %f = %f !!!\n",x, y, z);

    return 0;
}
```

The output of the above program, when it is executed in my machine is as follows.

```
equality check 1 failed
0.200000
1.333300 + 52.200001 = 53.533302 !!!
```

Though the value of 1/5 is equal to 0.200000, there could be lack of precision while representing the value 1 using the variable `x` and the value 5 using the variable `y`. Moreover, when `x/y` is computed, some inaccuracy would have occurred while storing the result. Any of the above can be the reasons of the equality check failing while evaluating the condition `(x/y == 0.200000)`.

Note that, after storing the value 52.2 in `y` and printing it back, the output printed is 52.200001. Moreover, after adding 1.333300 and 52.200001, we may expect the result to be 53.533301; but the result obtained is 53.533302.

One point to note here is that the results are not precise; but the error in the representation is not large. If we want to compare between two floating point values, we should allow some tolerance for imprecision.

IEEE-754 standard for floating point representations

A floating point number is usually represented using scientific notation. This notation has a mantissa or precision part and an exponent part. Let us consider a number like 52.25×10^5. This number can be also represented as 5.225×10^6. This form in which there is exactly one digit before the decimal point and that one digit is non-zero is known as a normalized representation. In this normalized decimal representation, 5.225 represents the precision (mantissa) and 6 is the exponent.

IEEE-754 standard defines the following three floating point data types for representing binary numbers. These standards are adopted by most of the devices and applications in use.

- single precision floating point (*float*) 32 bits
- double precision floating point (*double*) 64 bits
- extended double precision floating point (*long double*) 12 bytes

Single precision floating point format

Suppose `x` is a positive Real number that we want to represent using a single precision floating point format. This format uses 23-bits for storing the mantissa and 8-bits for storing the exponent and one bit for storing the sign.

First consider the normalized binary representation of `x` to the accuracy of 23-bits after the decimal point. Recall that in the normalized binary representation, there will be exactly one bit before the decimal point and that bit will be `1`. Suppose, this normalized representation of `x` is given by $\texttt{1} \cdot \texttt{m}_1\,\texttt{m}_2 \ldots \texttt{m}_{23} \times 2^{\texttt{k}}$. Since the bit before the decimal point is always `1`, there is no need to waste one bit for storing that part while representing the number in a computer. For this reason, in the single precision floating point format, the 23-bit mantissa part will store only $\texttt{m}_1\texttt{m}_2 \ldots \texttt{m}_{23}$. To represent the exponent `k`, the unsigned binary equivalent of `k+127` is computed and stored in bits $e_7 \cdots e_2 e_1 e_0$.

As per the standard, 32 bit floating point representation of real numbers is shown in the figure below.

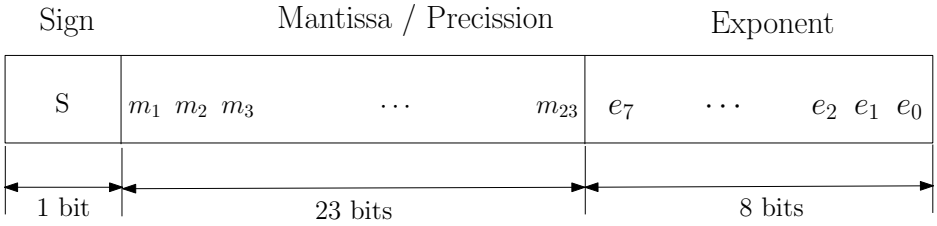

Figure 1: Single precision floating point representation

There are three parts in the representation; sign, mantissa and exponent. The sign bit can be either 0 or 1, where 1 represents a negative number and 0 represents a positive number. Then comes the mantissa part which represents the precision. The third part is the exponent.

All 1's bit pattern and all 0's bit pattern in the exponent are reserved for special purposes. Apart from these two, there are 254 different 8-bit patterns available for the exponent. These 254 possibilities are mapped to exponent values -126 to +127 as follows.

$e_7\, e_6\, e_5\, e_4\, e_3\, e_2\, e_1\, e_0$	Exponent Value
0 0 0 0 0 0 0 1	-126
0 0 0 0 0 0 1 0	-125
0 0 0 0 0 0 1 1	-124
⋯ ⋯ ⋯	⋯ ⋯ ⋯
1 1 1 1 1 1 0 1	+126
1 1 1 1 1 1 1 0	+127

The number represented by the pattern shown in Figure 1 is approximately

$$-1^S \times \left(\left(1 + \frac{m_1}{2} + \frac{m_2}{2^2} + \cdots + \frac{m_{23}}{2^{23}}\right) \times 2^k\right)$$

where `k` is obtained by subtracting 127 from the decimal equivalent of the unsigned binary number represented by the bit pattern $e_7 \cdots e_2\, e_1\, e_0$.

The largest possible number that can be represented using single precision floating point representation is represented by the following bit pattern

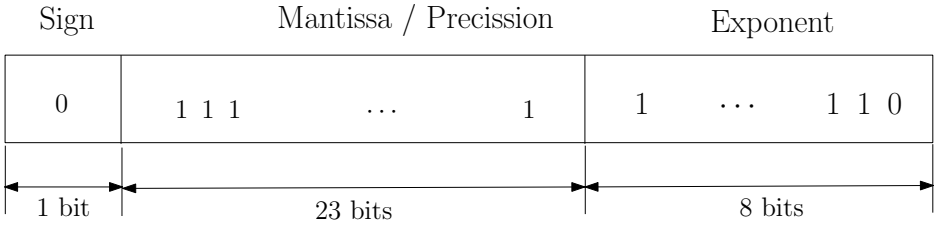

Figure 2: Largest positive number using single precision floating point representation

The value corresponding to this number is

$$\left(\left(1+\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^{23}}\right)\times 2^{127}\right)$$

$$\approx 2^{128}$$

When 2^{128} is converted to decimal, it is close to 10^{38}, because $128 \times log_{10}2 = 38$.

The lowest possible positive number corresponds to the bit pattern

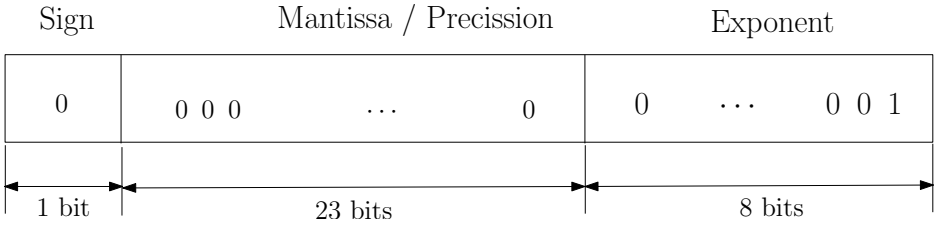

Figure 3: Smallest positive number in single precision floating point representation

The number represented by the above bit pattern is $1 \times 2^{-126} \approx 10^{-38}$.

Special values

A 32-bit pattern with both the mantissa and exponent parts equal to 0 represents the number 0. A 32-bit pattern with all exponent bits equal to 1 and all mantissa bits equal to 0 represents $infinity$. A 32-bit pattern with all exponent bits equal to 1 and a non zero mantissa represents NaN (Not a Number).

Epsilon of Single precision floating point

Epsilon of the single precision floating point representation is the smallest positive number x such that the representation of 1.0 + x can be precisely done using the single precision floating point representation and 1.0 + x $\neq$ 1.0. This number 1.0 + x corresponds to the following bit pattern.

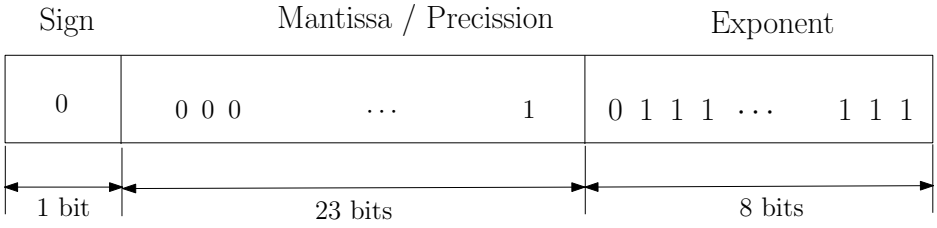

Figure 4: Epsilon of the single precision floating point representation

This bit pattern corresponds to the number $(1 + 2^{-23}) \times 2^0$. Therefore, epsilon is 2^{-23}, which is approximately equal to 10^{-7}. Epsilon is a measure of the precision of the representation. It can also be considered as the largest relative error that can occur while representing a real number within the range of values possible (relative error in the representation of a real number x is given by $|x - r|/|x|$ where r is the floating point value used to represent x approximately in the single precision floating point format).

Double precision floating point format

In double, 64 bits are used for representing a real number. In this 64 bits, 1 bit is used for sign, 52 bits are used for representing mantissa and the remaining 11 bits for representing exponents.

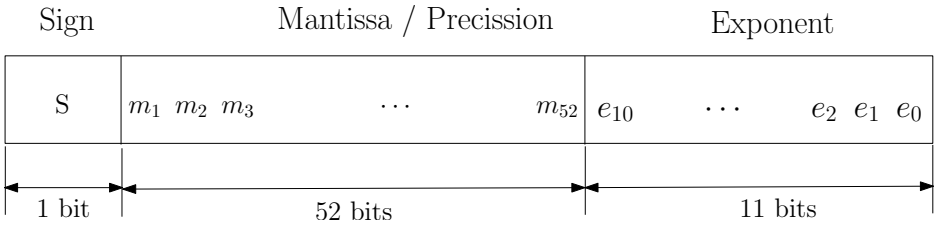

Figure 5: Double precision floating point representation

Double precision format can represent numbers more precisely because there are more bits in the mantissa. Since there are more bits in the exponent, it can also store larger numbers. Similar to what we have done in the case of single precision floating point numbers, we can compute the different parameters for double precision floating point numbers also. In this case, the possible values of the exponent ranges from -1022 to +1023. The highest number that can be represented using double is approximately $2^{1024} \approx 10^{308}$ and the smallest positive number that can be represented is $\approx 10^{-308}$. Epsilon of the representation is approximately 2^{-52} which is approximately 10^{-16}.

Floating point representations in C

In C, there are three basic floating point formats: `float, double` and `long double`. The data type `float` is exactly the same as the IEEE single precision floating point format. The data type `double` is exactly the same as IEEE double precision floating point format. Computations and storage using the above two formats are usually done in hardware in 64-bit machines. The data type `long double` uses at least 12 bytes and this format is

often implemented in software. This can be either same as the extended double precision format of IEEE or it could be slightly different. The different parameters of floating point representations are defined in the library file `float.h`. The following program displays some of these parameters.

```
#include<stdio.h>
#include<float.h>
int main()
{
   printf("a float variable occupies %d bytes \n",sizeof(float));
   printf("a double variable occupies %d bytes \n",sizeof(double));
   printf("maximum float value is %g \n",FLT_MAX);
   printf("minimum positive float is %g \n",FLT_MIN);
   printf("epsilon for float is %g\n",FLT_EPSILON);
   printf("maximum exponent for a float variable in base 2 is %d \n",FLT_MAX_EXP)

   return 0;
}
```

The output of this program is given below.

```
a float variable occupies 4 bytes
a double variable occupies 8 bytes
maximum float value is 3.40282e+38
minimum positive float is 1.17549e-38
epsilon for float is 1.19209e-07
maximum exponent for a float variable in base 2 is 128
```