3.1.3 Real numebrs and normalized floating-point representation

In decimal notation, the number “23.456” can be written as “$0.23456x10^{2}$”. This means that in decimal notation, we only need to store the numbers “0.23456” and “2”. The number “0.23456” is called the **mantissa** and the number “2” is called the **exponent**. This is what happens in binary.

For example, consider the binary number “10111”. This could be represented as “$0.10111x2^{5}$” or “$0.10111x2^{101}$”. Here “0.10111” is the **mantissa** and “101” is the **exponent**.

Similarly, in decimal, 0.0000246 can be written as “$0.246x10^{-4}$”. Now the **mantissa** is “0.246” and the **exponent** is “-4”.

Thus, in binary, “0.00010101” can be written as “$0.10101x2^{-11}$”. Now the **mantissa** is “0.10101” and the **exponent** is “-11”.

By now it should be clear that we need to store 2 numbers, the **mantissa** and the **exponent**. This form of representation is called **floating point form**. Numbers that involve a fractional part like “2.467_{10}” and “101.0101_{2}” are called **real numbers**.

Converting binary floating-point real numbers into denary and vice versa

- **Convert "2.625" into 8-bit floating point format**

Converting the integral part is simple; you simply keep adding bits of increasing power until you get the number that you want, so in this case:

2 (the integral part) =

128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
0	0	0	0	0	0	1	0

Or simply "10"

As for the fractional part, you must do repeated multiplication by 2 until your remainder is zero, so in this case:

0.625 (the fractional part)

- 0.625 x 2 = 1.25 [1] (Generate "1",continue with the rest)
- 0.25 x 2 = 0.5 [0] (Because the whole part is "0" we continue multiplying by 2)
- 0.5 x 2 = 1.0 [1] (Generate "1" and nothing remains in the fractional part

So $\mathbf{2_{10} = 10_2}$ and $\mathbf{0.625_{10} = 0.101_2}$

So $2.625_{10} = 10.101_2$ (binary floating-point real number)

Convert 0.40625 to 8-bit floating point format

- 0.40625 x 2 = 0.8125 | 0
- 0.8125 x 2 = 1.625 | 1
- 0.625 x 2 = 1.25 | 1
- 0.25 x 2 = 0.5 | 0
- 0.5 x 2 = 1.0 | 1

So 0.40625_{10} = 0.01101_2 (binary floating-point real number)

Convert 14.7 into 8-bit floating point format

14 (the integral part) =

128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
0	0	0	0	1	1	1	0

Or simply “1110”

0.7 (the fractional part)

- 0.7 x 2 = 1.4 | 1
- 0.4 x 2 = 0.8 | 0
- 0.8 x 2 = 1.6 | 1
- 0.6 x 2 = 1.2 | 1
- 0.2 x 2 = 0.4 | 0
- 0.4 x 2 = 0.8 | 0
- 0.8 x 2 = 1.6 | 1
- 0.6 x 2 = 1.2 | 1
- …

This process seems to go on endlessly. The number “7/10”, which is a perfectly normal decimal fraction, is a repeating fraction in binary, just as the fraction “1/3” is a repeating fraction in decimal. (It repeats in binary as well.) We can’t represent this number precisely as a floating point number. The closest we can get with four bits is “**.1011**”. Since we already have a leading “14”, the best eight

bit number we can make is "**1110.1011**" **So 14.7_{10} = 1110.1011_2** (binary floating-point real number)

- **Convert 1101.1100 into denary format**

Integral part: 1101

Fractional part: 1100

Converting the integral part is simply converting from binary to decimal

128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
0	0	0	0	1	1	0	1

1101_2 = 8+4+1 = 13_{10}

Converting the fractional part is similar as integral part

½ (2^-1)	¼ (2^-2)	1/8 (2^-3)	1/16 (2^-4)	1/32 (2^-5)	1/64 (2^-6)	1/128 (2^-7)	1/256 (2^-8)
1	1	0	0	0	0	0	0

0.1100_2 = (1/2) + (1/4) = 0.75_{10}

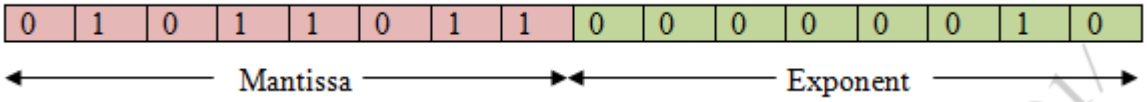
So 1101.1100_2 = 13.75_{10}

*red cells indicate that those bits are not used

Normalizing a Real Number

In the above examples, the decimal point in the mantissa was always placed immediately before the first non-zero digit. This is always done like this with positive numbers because it allows us to use the maximum number of digits.

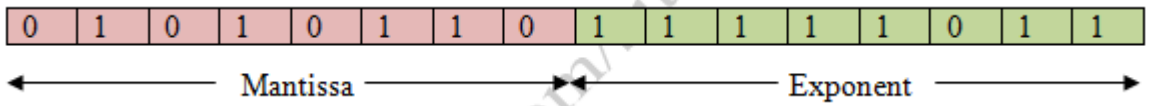
Suppose we use 8 bits to hold the mantissa and 8 bits to hold the exponent. The binary number



"**10.11011**" becomes "**0.1011011 x 2^{10}**" and can be held as:

Notice that the first digit of the mantissa is 0 and the second digit is 1. The mantissa is said to be "**normalized**" if the first 2 digits are different. Thus, for a positive number, the first digit is always 0 and the second digit is always 1. The exponent is always an integer and is help in 2's complement form.

Now consider the binary number "**0.00000101011**" which is "**0.1010110 x 2^{-101}**". Thus the mantissa is "**0.101011**" and the exponent is "**-101**". Again, using 8 bits for the mantissa and 8 bits for the exponent, we have:

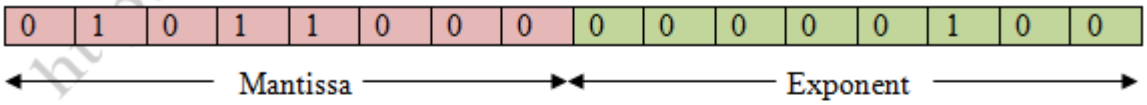


Because the 2's complement of "**-101**" using 8 bits is "**11111011**"

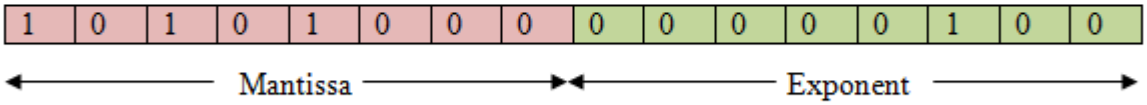
The main reason that we normalize floating-point numbers is in order to have as high degree of accuracy as possible.

Care must be taken when normalizing negative numbers. The easiest way to normalize negative numbers is to first normalize the positive version of the number. Consider the binary number "**-1011**". The positive version is "**1011**" = "**0.1011 x 2^{100}**" and can be represented by:

3.1.3 Real numebrs and normalized floating-point representation



Now find the two's complement of the mantissa and the result is



Notice that the first two digits are different.

As another example, change the fraction “-11/32” into a normalized floating point binary number.

Ignore the negative sign and solve for just “11/32”

11/32 = 0.34375

- 0.3475 x 2 = 0.6875 | 0
- 0.6875 x 2 = 1.375 | 1
- 0.375 x 2 = 0.75 | 0
- 0.75 x 2 = 1.50 | 1
- 0.50 x 2 = 1.00 | 1

= 0.01011_2

Now we have 8 bits mantissa and 8 bits exponent

½	¼	1/8	1/16	1/32	1/64	1/128	1/256	1	2	4	8	16	32	64	128
0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0

Mantissa — Exponent

But it is not normalized, so we do that by removing a “0” from bit “1/2” in the mantissa and subtracting that 1 location from exponent 0 revels -1.

That is:

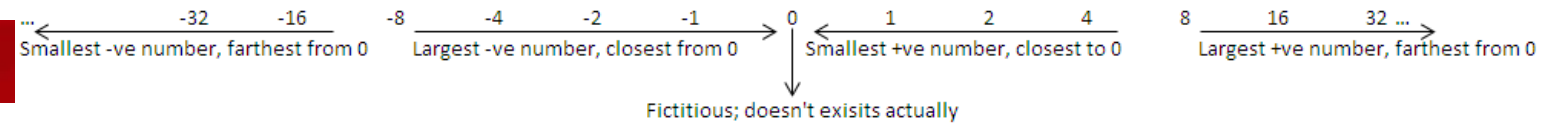
½	¼	1/8	1/16	1/32	1/64	1/128	1/256	1	2	4	8	16	32	64	128
0	1	0	1	1	0	0	0	1	1	1	1	1	1	1	1

Mantissa — Exponent

For “-11/32”, keep the exponent the same and in the mantissa, convert it into 2’s complement.

½	¼	1/8	1/16	1/32	1/64	1/128	1/256	1	2	4	8	16	32	64	128
1	0	1	0	1	0	0	0	1	1	1	1	1	1	1	1

Mantissa — Exponent



NEGATIVE SIDE POSITIVE SIDE

Number Ranges in Floating Point

Number	Mantissa (1 Byte)	Exponent (1 Byte)	Range Calculation	Comments
Largest +ve	Largest +ve 01111111	Largest +ve 01111111	01111111 01111111 => 1 x 2^{127} => 2^{127}	Farthest from 0
Smallest +ve	Smallest +ve 01000000 Roughly Normalised	Smallest -ve 10000000	01000000 10000000 => 0.1 x 2^{-128} => 2^{-129}	Closest to 0 Used as 0 in FP representations.
Largest -ve	Largest -ve 10111111 Roughly Normalised	Smallest -ve 10000000	10111111 10000000 +ve conversion -0.1000001 x 2^{-128} => -2^{-129}	Closest to 0
Smallest -ve	Smallest -ve 10000000	Largest +ve 01111111	10000000 01111111 +ve conversion => -1 x 2^{127} => -2^{127}	Farthest from 0

There are always a finite number of bits that can be used to represent numbers in a computer. This means that if we allocate more bits for the mantissa, we will have to allocate fewer bits for the exponent.

Let us start off by using 8 bits for the mantissa and 8 bits for the exponent. The largest positive value we can have for the mantissa is **0.1111111** and the largest positive number we can have for the exponent is **01111111**. This means that we can have

$$\mathbf{0.1111111 \times 2^{1111111} = 0.1111111 \times 2^{127}}$$

This means that the largest positive number is almost 1×2^{127}

The smallest positive mantissa is **0.1000000** and the smallest exponent is **10000000**. This represents:

$$\mathbf{0.1000000 \times 2^{10000000} = 0.1000000 \times 2^{-128}}$$

Which is very close to zero; in fact it is 2^{-129}.

The largest negative number (i.e. the negative number closest to zero) is

$$\mathbf{1.0111111 \times 2^{10000000} = -0.10000001 \times 2^{-128}}$$

Have you ever noticed that zero cannot be represented in normalized form? This is because "0.0000000" is not normalized because the first two 2 digits are the same. **Usually, the computer uses the smallest positive number to represent zero**. Keep in mind that when we are talking about the "size" of the number, as you move towards the right, the size of the number increases. So "-1" is **greater than** "-2" whereas, if we talk about the largest "magnitude" negative number, then "-2" is **greater than** -1 because the integer value is greater.

Hence, for a positive number n:

$$\mathbf{2^{-129} \leq n < 2^{127}}$$

And for a negative number n:

$$\mathbf{-2^{127} \leq n < -2^{-129}}$$

Now suppose we use 12 of the bits for the mantissa and the other 4 bits for the exponent (12+4 =16). This will **increase** the number of bits available for the mantissa that can be used to represent the number. That is, we shall have more binary places in the mantissa and hence greater accuracy. However, the range of values in the exponent is from -8 (1000) to +7 (0111) and this produces a small range of numbers. Hence, we have increased the accuracy of the number but at the expense of the range of numbers that can be represented.

Similarly, allowing fewer bytes to the mantissa will reduce the accuracy, but allows a much greater range of values because the size of the exponent has increased.

Integer Overflow

How are integers represented on a computer?

Most computers use the 2's complement representation.

Assume we have 4 bits to store and operate on an integer; we may end up with the following situation:

```
 0111 (7)
+0001 (1)
-----
 1000 (-8)
```

The numbers in parentheses are the decimal value represented by the 2's complement binary integers. The result is obviously wrong because adding positive numbers should never result in a negative number. The reason for this error is that the result exceeds the range of value the 4-bit can store with the 2's complement notation.

The following example gives the wrong answer for the same reason. The leading 1 cannot be stored.

```
 1000 (-8)
+1111 (-1)
-----
10111 (7)
```

Integer Underflow

Since we are only concerned with magnitude, why even talk about underflow? Talking about underflow makes sense for floating-point numbers.

For example, using a 4-bit 2's complement representation for an exponent of a floating-point number would enable us to represent numbers of the order 10^{-16} to 10^{15}. If the number is less than 10^{-16}, there would be no way to represent that number, and hence would lean to an "underflow". Underflow in floating-point arithmetic may be thought of as "overflow of the exponent".

In math, numbers have infinite precision, but numerals (representation of a number) have finite precision. One third (1/3) cannot be represented precisely in decimal otherwise it would need an infinite number of digits (which is impossible). Similarly, "0.2" cannot be represented precisely in binary, which means its binary representation is not precise (approximation).

A floating-point underflow happens when the result of a calculation is too small to be stored. An underflow could also be caused by the (negative) overflow of the exponent.

Rounding errors

Suppose you wish to represent the number 1/7 in decimal. This number goes on infinitely repeating the sequence "**0.142857**".

How would you represent this in 4 bit floating point? (3 bit mantissa and 1 bit exponent) 1.43×10^{-1} is the closest you can get.

Even with 10, 20, or 100 digits, you would need to do some rounding to represent an infinite number in a finite space. If you have a lot of digits, your rounding error might seem insignificant. But consider what happens if you add up these rounded numbers repeatedly for a long period of time. If you round 1/7 to 1.42×10^{-1} (0.142) and add up this representation 700 times, you would expect to get 100. (1/7 x 700 = 100) but instead you get 99.4 (0.142 x 700).

Relatively small rounding errors like the example above can have huge impacts. Knowing how these rounding errors can occur and being conscious of them will help you become a better and more precise programmer.

Consequences of binary representation of a real number

While some rounding errors seem to be very insignificant, small errors can quickly add up. One of the most tragic events caused by rounding errors was the Patriot Missile Crisis in 1991.

Patriot Missiles, which stand for Phased Array Tracking intercept of Target, were originally designed to be mobile defenses against enemy aircraft. These missiles were supposed to explode right before encountering an incoming object. However, on the 25th of February 1991, a Patriot Missile failed to intercept an incoming Iraqi Scud missile which then struck an army barrack in Dhahran killing 28 American soldiers and injuring over 100 other people!

So what went wrong?

The system's internal clock recorded passage of time in tenths of seconds. However, as explained earlier, 1/10 has a non-terminating binary representation, which could lead to problems. Let's look into why this happens.

This is an approximation of what 1/10 looks like in binary.

0.00011001100110011001100110011001100...

The internal clock used by the computer system only saved 24 bits. So this is what was saved every tenth of a second:

0.00011001100110011001100

Chopping off any digits beyond the first 24 bits introduced an error of about:

0.00000000000000000000000011001100

This is about 0.000000095 seconds for each tenth of a second

The missile's battery had been running for about 100 hours before the incident. Imagine this error accumulating for 100 hours (10 times each second!)

The small error for each tenth of a second was not believed to be a problem. However, the missile's battery had been running for over 100 hours, introducing an error of about 0.34 seconds!

Given the missile was supposed to intercept a Scud traveling 1,624 meters per second, 0.34 second delay turned out to be a huge problem!