

Number Systems

Last Week

- Analog and Digital Signals
- Number Systems
- Arithmetic Operations in Binary System
- One's Complement and Two's Complement
- Signed Numbers

This Week

- Floating Point Numbers
- Arithmetic Operations with Signed Numbers

Addition and Subtraction

Multiplication and Division

Floating Point Numbers

We can represent very small numbers (e.g. 0.000000000000000000000001) and very large numbers (e.g. 999999999999999999999999) using floating-point numbers in scientific format.

For example, in IEEE-754 standard, floating-point numbers are represented in 32-bits. 1 bit for the sign, 8 bits for exponent, and 23 bits for significand.

There are 64-bit and 80-bit representations as well.

Sign	Exponent (biased)	Significand
1-bit	8-bits	23-bits

32-bit representation

Normalization

A binary number can be in different forms in scientific notation.

$$1 = 0.1 \times 2^1 = 0.01 \times 2^2 = \dots$$

As a standard, we make sure that the whole part of the number is 1. This process is called normalization.

For example, we normalize the number 10011001011_2 by converting it to $1.0011001011 \times 2^{10}$.

Biased Exponent

In floating point representation, we do not store the exponent directly. Instead, we use the biased exponent. Bias exponent is calculated by this formula:

$$\text{Bias} = 2^{[\text{exponent bit count}]-1} - 1$$

In 32-bit floating point representation, the exponent bias is 127 ($2^{8-1} - 1$). So, we add 127 to the exponent.

The original exponent can be between -126 and 128. But we need to store a positive number. That's why we add bias to the original exponent. Biased exponent is always a positive number.

In floating point representation, when all the bits are 0, then the number is considered to be 0. When exponent part is all 1's and significand part is all 0's, then the number is considered to be infinite.

Floating Point Numbers

Example: Let's convert $1.0011001011 \times 2^{10}$ to floating number.

Since the number is positive, the sign bit is 0. The exponent is 10. So, biased exponent is $10 + 127 \text{ (bias)} = 137 = 10001001_2$. And finally, the significand is 0011001011_2 .

0	10001001	00110010110...0
1-bit	8-bits	23-bits

If the number above were negative, the result would remain the same, but the sign bit would be set to 1.

Floating point representation lets us store larger numbers using less bits. For instance, let's take the largest number that can be stored 32-bit floating point representation. If we wanted to store the same number in binary form, we would need 129 bits.

Floating Point Numbers

In a 4-bit floating point representation, let's assume that the sign part is 1 bit, biased exponent part is 2 bits, and the significand is 1 bit.

Let's analyze the numbers that can be stored using this representation.

Sign	B. Exponent	Significand
1 bit	2 bits	1 bit

$$\text{Bias} = 2^{2-1} - 1 = 1$$

All the positive numbers that can be stored are;

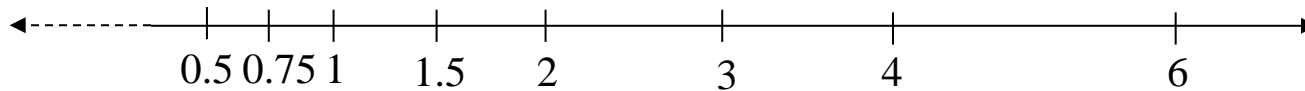
$$1.0 \times 2^{-1} = 0.5 \quad 1.1 \times 2^{-1} = 0.75$$

$$1.0 \times 2^0 = 1 \quad 1.1 \times 2^0 = 1.5$$

$$1.0 \times 2^1 = 2 \quad 1.1 \times 2^1 = 3$$

$$1.0 \times 2^2 = 4 \quad 1.1 \times 2^2 = 6$$

Floating Point Numbers



- Although they are not shown above, we also have the same numbers with negative sign on the number axis. Since we have 4 bits, we can store $2^4=16$ different numbers. As you can see above, gaps between numbers are larger in greater numbers.
- You can see that; we can store 3 numbers between 0.5 and 1. The same rule applies to 1 and 2 and also 2 and 4.
- If the exponent part were 1 bit and the significand part 2 bits, we would still have 4 bits in total, allowing us to store 16 different bit combinations. However, the numbers represented would be different: 1, 1.25, 1.5, 1.75, 2, 2.5, 3, and 3.5.

In this scenario, the gaps between the numbers would be smaller, allowing for finer precision, but the range of representable numbers would be limited to relatively smaller values.

Arithmetic Operations on Signed Numbers

In this section, we will use two's complement form.

Addition and subtraction:

- The sum of two positive numbers is a positive number.

Example: 00001001 (9)
+ 00000100 (4)

00001101 (13)

- The sum of a positive number and a smaller negative number is a positive number. We ignore the carry bit.

Example: 00001001 (9)
+ 11111010 (-6)

100000011 (3)

Arithmetic Operations on Signed Numbers

- The sum of a positive number and a greater negative number is a negative number in two's complement form.

Example: 00001000 (8)

+ 11110100 (-12)

11111100 (-4 in two's complement form)

- The sum of two negative numbers is a negative number. We ignore the carry bit.

Example: 11111101 (-3)

+ 11111110 (-2)

111111011 (-5)

Arithmetic Operations on Signed Numbers

- When signed numbers are added, any carry bit that occurs is ignored. There are two exceptional cases: the first is when both numbers are positive, but the sign bit is 1, and the second is when both numbers are negative, but the sign bit is 0. In both cases, an overflow has occurred, meaning it can be said that one more bit is required to represent the sum.

Example:
$$\begin{array}{r} 01101110 \quad (110) \\ + 00011001 \quad (25) \\ \hline 10000111 \quad (\text{overflow}) \end{array}$$

Subtraction on signed numbers is just a different form of addition. We convert the number to subtract to two's complement form and add two numbers.

Example: 8-3 operation is performed as 8+(-3).

$$\begin{array}{r} 00001000 \quad (8) \\ + 11111101 \quad (-3) \\ \hline \underline{100000101} \quad (5) \end{array}$$

Multiplication

Multiplication can be performed by adding the number to itself as many times as necessary. For example, 4×2 is considered as $4 + 4$. When the multiplier value is large, it becomes tedious, so the partial product method is used.

- First, it is checked whether the multiplicand and the multiplier have the same sign, and the sign of the result is determined.
- Then, we convert negative numbers to positive.
- Then, starting from LSB of the first number, partial products are obtained. If the multiplier is 1, the multiplicand itself is used; if it is 0, a partial product of 0 is obtained. Each partial product is shifted left by one.
- All partial products are summed to obtain the final product.
- Then, from the first step, if the product must be negative, we convert the number to two's complement form. If the product must be positive, we leave the result as it is. Then we add a sign bit to the left.

Multiplication

Example: Let's calculate $25 \times (-4)$

11001	
$\times 100$	-4 converted to 4
<u>00000</u>	1 st partial product
$+ 00000$	2 nd partial product (shifted left by 1 bit)
<u>000000</u>	subtotal
$+ 11001$	3 rd partial product (shifted left by 2 bits)
<u>1100100</u>	absolute value of the product

Since the product must be negative, we convert the result to two's complement form.

Two's complement of 1100100_2 is 0011100_2 .

Eventually, we add a sign bit and get the final product: 10011100_2 .

Division

Division can be performed with subtraction. As we know, subtraction is performed by addition. So, division can also be performed by addition. The quotient can be calculated by subtracting the divisor from the dividend multiple times. The number of subtraction operations gives us the quotient. We do the division operation as follows:

- First, we check both numbers have the same sign and detect the sign of the result.
- Then, we convert negative numbers to positive.
- Then, we assign 0 to quotient.
- Then, we subtract the divisor from the dividend. We get the partial remainder and increase quotient by 1. If the partial remainder is positive, then the division is considered as complete. If the partial remainder is negative, we proceed to the next step.
- We subtract the divisor from the partial remainder and increase quotient by 1. If the result is positive, then we repeat the same operation with the next partial remainder. If the result is 0 or negative, then the operation is considered as complete.

Division

Example: Let's do the operation $25/6$.

Since both numbers are positive, the result should be positive.

$$25 = 00011001_2 \text{ and } 6 = 00000110_2$$

We assign 0 to quotient and subtract the divisor from the dividend.

$$\begin{array}{r} 00011001 \\ + 11111010 \text{ (Two's complement form of 6)} \\ \hline 100010011 \end{array} \text{ (carry bit is ignored) Since partial remainder is positive;}$$

$\text{quotient} = 0 + 1 = 1$

We subtract the divisor from the partial remainder.

$$\begin{array}{r} 00010011 \\ + 11111010 \\ \hline 100001101 \end{array} \text{ (carry bit is ignored) Since partial remainder is positive;}$$

$\text{quotient} = 1 + 1 = 2$

We subtract the divisor from the partial remainder again.

$$\begin{array}{r} 00001101 \\ + 11111010 \\ \hline 100000111 \end{array} \text{ (carry bit is ignored) Since partial remainder is positive;}$$

$\text{quotient} = 2 + 1 = 3$

Division Example (Continuing)

We subtract the divisor from the partial remainder again.

$$\begin{array}{r} 00000111 \\ + 11111010 \\ \hline 100000001 \end{array} \text{ (carry bit is ignored) Since partial remainder is positive; } \text{quotient}=3+1=4$$

We subtract the divisor from the partial remainder again.

$$\begin{array}{r} 00000001 \\ + 11111010 \\ \hline 11111011 \end{array} \text{ There's no carry bit. The result is negative. The division is completed. The result of the division operation is quotient's current value, which is 4 (00000100}_2 \text{ in binary).}$$

The remainder is the result of previous subtraction (attention, not the last one), which is 00000001_2