Number Systems

Last Week

- Analog vs. Digital
- Number Systems
- Arithmetic Operations on Binary Numbers
- One's Complement and Two's Complement
- Signed Numbers

This Week

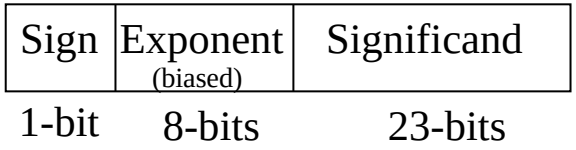
- Floating Point Numbers
- Arithmetic Operations with Signed Numbers
  - Addition and Subtraction
  - Multiplication and Division

Floating Point Numbers

We can represent very small numbers (e.g. 0.00000000000000000001) and very large numbers (e.g. 99999999999999999999) using floating point numbers in scientific format.

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

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



32-bit representation

Normalization

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

$1 = 0.1x2^1 = 0.01x2^2 = \ldots$

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 use the exponent directly. Instead, we use the biased exponent. Bias exponent is calculated by this formula:

Exponent 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.

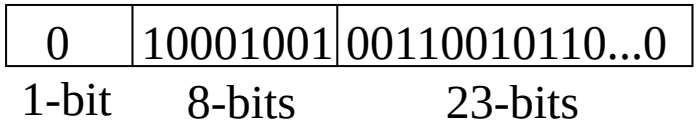
The original exponent is between -126 and 128. But we need to place a positive number. That's why we add bias to the original exponent. Biased exponent is always a positive number.

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 (bias)=137=10001001_2. And finally, the significand is 0011001011_2.



If the same number was negative, the result would be the same except that the sign bit would be 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

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

All the positive numbers that can be stored are;

$1.0\times2^{-1} = 0.5$ $1.1\times2^{-1} = 0.75$

$1.0\times2^{0} = 1$ $1.1\times2^{0} = 1.5$

$1.0\times2^{1} = 2$ $1.1\times2^{1} = 3$

$1.0\times2^{2} = 4$ $1.1\times2^{2} = 6$

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.

- What if exponent part was 1 bit and significand part was 2 bits? Since we have 4 bits in total, we could still store 16 different numbers. But The numbers would be different. (1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5).

This time, gaps between numbers would be smaller, but we would store smaller numbers.

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 positive.

Example: 00001001 (9)

\+ 00000100 (4)

00001101 (13)

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

Example: 00001001 (9)

\+ 11111010 (-6)

1**0**0000011 (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 negative. We ignore the carry bit.

Example: 11111101 (-3)

\+ 11111110 (-2)

11111011 (-5)

Arithmetic Operations on Signed Numbers

- In addition, we ignore the carry bit. But there are two exceptions. The first one is when we add two numbers together but the sum is negative, the second one is when we add two negative numbers together but the sum is positive. In both situations, there's an overflow which means that we need an extra bit to represent the result.

Example: **0**1101110 (110)
+ **0**0011001 (25)
10000111 (overflow)

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).

00001000 (8)
+ 11111101 (-3)
100000101 (5)

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. But if we need to do too many additions, then it would be too hard. So, we do partial multiplication.

- First, we check if both numbers have the same sign. If both numbers are negative or positive, the product is positive.
- Then, we convert negative numbers to positive.
- Then, starting from LSB of the first number and we calculate the partial products. We shift second partial product left by 1 bit, third partial product by 2 bits, and so on.
- Then, we add all the partial products together and find the absolute value of the 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×(-4)

11001	
× 100	-4 converted to 4
00000	1st partial product
+ 00000	2nd partial product (shifted left by 1 bit)
000000	subtotal
+ 11001	3rd partial product (shifted left by 2 bits)
1100100	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.

$25 = 00011001_2$ and $6 = 00000110_2$

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

```
  00011001
+ 11111010   (Two's complement form of 6)
-----------
 100010011   (carry bit is ignored) Since partial remainder is positive;
```

quotient=0+1=1

We subtract the divisor from the partial remainder.

```
  00010011
+ 11111010
-----------
 100001101  (carry bit is ignored) Since partial remainder is positive;
```

quotient=1+1=2

We subtract the divisor from the partial remainder again.

```
  00001101
+ 11111010
-----------
 100000111  (carry bit is ignored) Since partial remainder is positive;
```

quotient =2+1=3

Division Example (Continuing)

We subtract the divisor from the partial remainder again.

```
  00000111
+ 11111010
----------
 100000001
```

(carry bit is ignored) Since partial remainder is positive; quotient=3+1=4

We subtract the divisor from the partial remainder again.

```
  00000001
+ 11111010
----------
  11111011
```

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 in binary).

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