Fundamentals of digital systems

Ali EL AZDI

IC BA2 - Mirjana Stojilović

February 19, 2024

Introduction

This document is designed to offer a LaTeX-styled overview of the Fundamentals of Digital Systems course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please do not hesitate to reach out to me at ali.elazdi@epfl.ch for corrections. Hoping to provide an experience similar to the one provided by past generations of students in other subjects.

Contents

Chapter 1

Number Systems

1.1 Digital Representations

In a digital representation, a number is represented by an ordered n-tuple:

The n-tuple is called a **digit vector**, each element is a **digit**

The number of digits n is called the precision of the representation. (careful! leftward indexing)

$$X = (X_{n-1}, X_{n-1}, \ldots, X_0)$$

Each digit is given **a set of values** D_i (eg. For base 10 representation of numbers, $D_i = \{0, 1, 2, \ldots, 9\}$)

The **set size**, the maximum number of representable digit vectors is: $K = \prod_{i=0}^{n-1} |D_i|$

1.1.1 (Non)Redundant Number Systems

A number system is nonredundant if each digit-vector represents a different integer

1.1.2 Weighted Number Systems

The rule of representation if a Weighted (Positional) Number Systems is as follows :

$$\sum_{i=0}^{n-1} X_i W_i$$

where

$$W = (W_{n-1}, W_{n-2}, \ldots, W_0)$$

1.1.3 Radix Systems

When weights are in this format :

$$\begin{cases} W_0 = 1 \\ W_{i+1} = W_i R_i \text{ with } 1 \leq i \leq n-1 \end{cases}$$

Also written : $W_0 = 1$, $\prod_{j=0}^{i-1} R_j$

1.1.4 Fixed and Mixed-Radix Number Systems

In a **fixed-radix system**, all elements of the radix-vector have the same value r (*the radix*)

The weight vector in a fixed-radix system is given by:

$$W = \left(r^{n-1}, r^{n-2}, \ldots, r^2, r, 1\right)$$

and the integer x becomes:

$$x = \sum_{i=0}^{n-1} X_i \times r^i$$

In a **mixed-radix system**, the elements of the radix-vector differ

Example: Decimal Number System

The decimal number system has the following characteristics:

- Radix $r = 10$, it's a fixed-radix system.

The weight vector W is defined as:

$$W = \left(10^{n-1}, 10^{n-2}, \ldots, 10^2, 10, 1\right)$$

An integer x in this system is represented by:

$$x = \sum_{i=0}^{n-1} X_i \times 10^i$$

For example:

$$854703 = 8 \times 10^5 + 5 \times 10^4 + 4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 3 \times 10^0$$

Examples of Fixed and Mixed radix systems

Fixed: The base of number systems.

- Decimal – radix 10
- Binary – radix 2
- Octal – radix 8
- Hexadecimal – radix 16

Mixed: An example of a mixed radix representation, such as time:

- Radix-vector $R = (24, 60, 60)$
- Weight-vector $W = (3600, 60, 1)$

1.1.5 Canonical Number Systems

In a **canonical number system**, the set of values for a digit D_i is with $|D_i| = R_i$, the corresponding element of the radix vector

$$D_i = \{0, 1, \ldots, R_i - 1\}$$

Canonical digit sets with fixed radix:

- Decimal: $\{0, 1, \ldots, 9\}$
- Binary: $\{0, 1\}$
- Hexadecimal: $\{0, 1, 2, \ldots, 15\}$

Range of values of x represented with n fixed-radix-r digits:

$$0 \leq x \leq r^n - 1$$

A system with fixed positive radix r and a canonical set of digit values is called a radix-r conventional number system.

1.2 Binary/Octal/Hexadecimal to/from Decimal Conversion Table

The hexadecimal system supplements 0-9 digits with the letters A-F.

Remark. Programming languages often use the prefix 0x to denote a hexadecimal number.

Table 1.1: Conversion table up to 15.

Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

1.2.1 Convertion examples

Binary to Decimal:

To convert a binary number to decimal, multiply each bit by two raised to the power of its position number, starting from zero on the right.

Binary: $\underline{1011}$

Decimal: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11$

Decimal to Binary:

Let's convert the decimal number 25_{10} to binary.

$$\begin{array}{rcll} 25 \div 2 & = & 12 & \text{remainder 1 (LSB)} \\ 12 \div 2 & = & 6 & \text{remainder 0} \\ 6 \div 2 & = & 3 & \text{remainder 0} \\ 3 \div 2 & = & 1 & \text{remainder 1} \\ 1 \div 2 & = & 0 & \text{remainder 1 (MSB)} \end{array}$$

Thus, the binary representation of 25_{10} is 11001_2 (reading the remainders in reverse).

Personal Remark The trick is always to try to answer the question, what's the biggest power of 2 I need to form the number?. For 157, the biggest power would be $2^7 = 128$, then $128+64$ is greater than 157, $128+32$ is still greater than 157, $128+16 = 144$, and so on to obtain : $128+16+8+4+1 = 157$ which can be written as $2^7+2^4+2^3+2^2+2^0 = 157$. Written in binary as 10011101_2

Octal to Decimal:

Each octal digit is converted to decimal by multiplying it by eight raised to the power of its position number, starting from zero on the right.

Octal:	$\underline{257}$
Decimal:	$2 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 = 128 + 40 + 7 = 175$

Decimal to Octal:

To convert the decimal number 93_{10} to octal.

$$\begin{array}{rcll} 93 \div 8 & = & 11 & \text{remainder 5} \\ 11 \div 8 & = & 1 & \text{remainder 3} \\ 1 \div 8 & = & 0 & \text{remainder 1} \end{array}$$

Thus, the octal representation of 93_{10} is 135_8 (reading the remainders in reverse).

Hexadecimal to Decimal:

To convert the hexadecimal number $1A3_{16}$ to decimal.

Hexadecimal:	$\underline{1A3}$
Decimal:	$1 \times 16^2 + A \times 16^1 + 3 \times 16^0$
	$1 \times 256 + 10 \times 16 + 3 \times 1$
	$256 + 160 + 3$
	419

Here, A in hexadecimal corresponds to 10 in decimal.

Decimal to Hexadecimal:

To convert the decimal number 291_{10} to hexadecimal.

$$\begin{aligned} 291 \div 16 &= 18 \quad \text{remainder } 3 \\ 18 \div 16 &= 1 \quad \text{remainder } 2 \\ 1 \div 16 &= 0 \quad \text{remainder } 1 \end{aligned}$$

Thus, the hexadecimal representation of 291_{10} is 123_{16} (reading the remainders in reverse).

1.3 Octal/Hexadecimal to/from Binary

Bit-Vector Representation Summary

- Digit-vectors for binary, octal, and hexadecimal systems are represented using bit-vectors. In binary, 0 and 1 are directly represented as 0 and 1.
- In systems like octal or hexadecimal, a digit is a bit-vector of length k, where k is the number of bits needed to represent the base.

  $$k = \log_2(r)$$

  with r the radix of the system (eg. 8 for octal convertion).
- For example, the hexadecimal digit B is represented as the bit-vector 1101 in binary. *We obtain a length 4 bit-vector because the base is 16 and* $\log_2(16) = 4$

Binary to Octal:

To convert a binary number to octal, group every three binary digits into a single octal digit, because $k = \log_2 8 = 3$.

Binary:	$\underline{010}\,\underline{000}\,\underline{100}\,\underline{110}$
Octal:	$2_8\,0_8\,4_8\,6_8$

Binary to Hexadecimal:

To convert a binary number to hexadecimal, group every four binary digits into a single hexadecimal digit, because $k = \log_2 16 = 4$.

Binary:	$\underline{1011}\,\underline{1110}\,\underline{1010}\,\underline{1101}$
Hexadecimal:	$B_{16}\,E_{16}\,A_{16}\,D_{16}$

Octal to Hexadecimal:

Convert the octal number to binary, then group the binary digits in sets of four and convert each group to its hexadecimal equivalent.

Octal:	257
Binary:	010 101 111 (Octal to binary)
Binary grouped:	0101 0111
Hexadecimal:	5 7 (Binary to hexadecimal)

1.4 Representation of Signed Integers

1.4.1 Sign-Magnitude Representation (SM)

A signed integer x is represented by a pair (x_s, x_m), where x_s is the *sign* and x_m is the *magnitude* (positive integer).

The sign (positive, negative) is represented by a the most significant bit (MSB) of the digit vector:

$0 \rightarrow$ positive

$1 \rightarrow$ negative

The magnitude can be represented as any positive integer. In a conventional radix-r system, the range of n-digit magnitude is:

$$0 \leq x_m \leq r^n - 1$$

- Examples:

$$01010101_2 = +85_{10}$$
$$01111111_2 = +127_{10}$$
$$00000000_2 = +0_{10}$$
$$11010101_2 = -85_{10}$$
$$11111111_2 = -127_{10}$$
$$10000000_2 = -0_{10}$$

Note: The Sign-and-Magnitude representation is considered a redundant system because both 00000000_2 and 10000000_2 represent zero.

SM consists of an equal number of positive and negative integers.

An n-bit integer in sign-and-magnitude lies within the range *(because of 0's double representation and that MSB is used for the sign)*:

$$[-\left(2^{n-1}-1\right), +\left(2^{n-1}-1\right)]$$

Main disadvantage of SM: complex digital circuits for arithmetic operations (addition, subtraction, etc.).

1.5 True-and-Complement (TC)

1.5.1 Mapping

A signed integer x is represented by a positive integer x_R, C is a positive integer called the *complementation constant*.

$$x_R \equiv x \mod C$$

For $|x| < C$, by the definition of the modulo function, we have:

$$x_R = \begin{cases} x & \text{if } x \geq 0 \quad \text{(True form)} \\ C - |x| = C + x & \text{if } x < 0 \quad \text{(Complement form)} \end{cases}$$

1.5.2 Unambiguous Representation

To have an unambiguous representation, the two regions should not overlap, translating to the condition:

$$\max |x| < \frac{C}{2}$$

1.5.3 Converse Mapping

Converse mapping:

$$x = \begin{cases} x_R & \text{if } x_R < \frac{C}{2} \quad \text{(Positive values)} \\ x_R - C & \text{if } x_R > \frac{C}{2} \quad \text{(Negative values)} \end{cases}$$

When $x_R = \frac{C}{2}$, it is usually assigned to $x = -\frac{C}{2}$.

Asymmetrical representation simplifies sign detection.

1.6 Two's Complement System

This is the True-and-Compelement system with $C = 2^n$, where n is the number of bits used to represent the integer.

Range is asymmetrical:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

The representation of zero is unique.

1.6.1 Sign Detection in Two's Complement System

Since $|x| < C/2$ and assuming the sign is 0 for positive and 1 for negative numbers:

$$\text{sign}(x) = \begin{cases} 0 & \text{if } x_R < C/2 \\ 1 & \text{if } x_R \geq C/2 \end{cases}$$

Therefore, the sign is determined from the most-significant bit:

$$\text{sign}(x) = \begin{cases} 0 & \text{if } x_{n-1} = 0 \\ 1 & \text{if } x_{n-1} = 1 \end{cases} \quad \text{equivalent to} \quad \text{sign}(x) = x_{n-1}$$

1.6.2 Mapping from Bit-Vectors to Values

The value of an integer represented by a bit-vector $b_{n-1}b_{n-2}\ldots b_1b_0$ can be universally expressed as:

$$\text{Value} = (-2^{n-1} \cdot b_{n-1}) + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

where b_{n-1} is the MSB (sign bit) and is 0 for non-negative numbers and 1 for negative numbers.

Examples

$X = 011011_2 = 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 2 + 1 = 27_{10}$

$X = 11011_2 = -1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -16 + 8 + 2 + 1 = -5_{10}$

$X = 10000000_2 = -1 \cdot 2^7 = -128_{10}$

$X = 10000011_2 = -1 \cdot 2^7 + 1 \cdot 2^1 + 1 \cdot 2^0 = -128 + 2 + 1 = -125_{10}$

1.6.3 Change of Sign in Two's Complement System

The two's complement system represents negative numbers by inverting the bits of their positive counterparts and adding one. This process is equivalent to subtracting the number from 2^n.
For an n-bit number x:

$$z = -x = (\sim x) + 1 = C - x_R$$

where $(\sim x)$ is the bitwise NOT of x and x_R is the decimal representation of x.

Examples

Converting +17 to −17:

$$\begin{aligned} +17_{10} &= 00010001_2 \\ -17_{10} &= \overline{00010001_2} + 1 = 11101111_2 \\ 2^8 - 17 = 256 - 17 &= 239 = 11101111_2 \end{aligned}$$

Converting −99 to +99:

$$\begin{aligned} -99_{10} &= 10011101_2 \\ +99_{10} &= \overline{10011101_2} + 1 = 01100011_2 \qquad (TC) \\ 2^8 - 99 = 256 - 99 &= 157 = 01100011_2 \ (Substracting\ 99\ from\ 256) \end{aligned}$$

1.7 Range Extension and Arithmetic Shifts

1.7.1 Range Extension

Performed when a value x represented by a digit-vector of n bits needs to be represented by a digit-vector of m bits, where $m > n$.

x is equal to z with

$$\begin{aligned} X &= (X_{n-1}, X_{n-2}, \ldots, X_1, X_0) \\ Z &= (Z_{m-1}, Z_{m-2}, \ldots, Z_1, Z_0) \end{aligned}$$

1.8 Range Extension Algorithm in SM

In sign-and-magnitude system, the range-extension algorithm is defined as:

$$\begin{aligned} z_s &= x_s \text{ (sign bit)} \\ Z_i &= 0 \quad \text{for } i = m-1, m-2, \ldots, n \\ Z_i &= X_i \quad \text{for } i = n-1, \ldots, 0 \end{aligned}$$

Example: Consider $X = 11010101_2 = -45_{10}$ and $X = 100101101_2 = -45_{10}$.

The algorithm extends the range of X by adding zeros to the left of the most significant bit, preserving the sign bit.

1.8.1 Arithmetic Shifts

Two elementary transformations often used in arithmetic operations are scaling (multiplying and dividing) by the radix.

In the conventional radix-2 number system for integers:

Left arithmetic shift: multiplication by 2, expressed as $z = 2x$.

Right arithmetic shift: division by 2, expressed as $z = 2^{-1}x - \varepsilon$, where $|\varepsilon| < 1$ and the value of ε is such that it makes z an integer. *The value of ε is the remainder of the division.*

1.8.2 Left Arithmetic Shift in Sign-and-Magnitude System

Algorithm (assuming the overflow does not occur):

$$\begin{aligned} z_s &= x_s \text{ (sign bit retained)} \\ Z_{i+1} &= X_i, \quad \text{for } i = 0, \ldots, n-2 \\ Z_0 &= 0 \text{ (insert zero at the least significant bit)} \end{aligned}$$

Example:

Given $X = 100101101_2 = -45_{10}$,

The left arithmetic shift $SL(X)$ would be $101011010_2 = -90_{10}$.

1.8.3 Right Arithmetic Shift in Sign-and-Magnitude System

Algorithm:

$$\begin{aligned} z_s &= x_s \text{ (sign bit retained)} \\ Z_{i-1} &= X_i, \quad \text{for } i = 1, \ldots, n-1 \\ Z_{n-1} &= 0 \text{ (insert zero at the most significant bit)} \end{aligned}$$

Example:

Given $X = 100101101_2 = -45_{10}$,

The right arithmetic shift $SR(X)$ would be $100010110_2 = -22_{10}$.

1.8.4 Left Arithmetic Shift in Two's Complement System

Algorithm (assuming that overflow does not occur):

$$Z_{i+1} = X_i, \quad \text{for } i = 0, \ldots, n-2$$
$$Z_0 = 0 \text{ (insert zero at the least significant bit)}$$

Examples:

Given $X = 00110101_2 = 13_{10}$,

The left arithmetic shift $SL(X)$ is $01101010_2 = 26_{10}$.

Given $Y = 11010101_2 = -11_{10}$,

The left arithmetic shift $SL(Y)$ is $10101010_2 = -22_{10}$.

1.8.5 Right Arithmetic Shift in Two's Complement System

Algorithm (assuming that overflow does not occur):

$$Z_{n-1} = X_{n-1}$$
$$Z_{i-1} = X_i, \quad \text{for } i = 1, \ldots, n-1$$

The most significant bit (MSB) is duplicated to keep the sign of the number the same.

Examples:

For $X = 001101_2 = 13_{10}$, the right arithmetic shift is $SR(X) = 000110_2 = 6_{10}$.

For $Y = 110101_2 = -11_{10}$ (in two's complement), the right arithmetic shift is $SR(Y) = 111010_2 = -6_{10}$.

1.9 Hamming Weight and Distance

1.9.1 Hamming Weight (HW)

The Hamming weight of a binary sequence is the number of symbols that are equal to one (1s).

For example, the Hamming weight of 11010101 is 5, as there are five 1s in the bit sequence.

1.9.2 Hamming Distance (HD)

The Hamming distance between two binary sequences of equal length is the number of positions at which the corresponding symbols are different.

For example, the Hamming distance between 11010101 and 01000111 is 3, as they differ in three positions.

1.10 Binary Coded Decimal (BCD)

Binary Coded Decimal (BCD) represents decimal numbers where each decimal digit is encoded as a four-bit binary number. This method allows decimal numbers to be represented in a format that is easy for digital systems to process.

1.10.1 BCD Encoding

- In BCD, each of the decimal digits 0 through 9 is represented by a four-bit binary number, ranging from 0000 to 1001.
- Binary values from $1010_2(10_{10})$ to $1111_2(15_{10})$ are not used in standard BCD encoding.
- For example, 25 is represented as $\underline{0010\ 0101}_2$.

1.10.2 Conversion Algorithms

From BCD to Decimal

To convert a BCD-encoded number to its decimal representation:

```
1. Initialize i to the highest index of BCD digits (n-1), D to 0.
2. While i is greater than or equal to 0:
   a. Multiply D by 10.
   b. Add the decimal value of the BCD digit at index i to D.
   c. Decrement i.
```

From Decimal to BCD

To convert a decimal number to its BCD representation:

```
1. Initialize i to 0, D to the decimal number.
2. While D is not equal to 0:
   a. Calculate D mod 10 and store it as the current BCD digit.
   b. Divide D by 10.
   c. Increment i.
```

1.11 Gray Code Conversion Algorithm

Rule for Conversion:

For bit i in the Gray code, look at bits i and $i+1$ in the binary code (bit n in binary is zero if $i+1=n$).

If bits i and $i+1$ in the binary are the same, bit i in the Gray code is 0.

If they are different, bit i in the Gray code is 1.

1.11.1 Example

To convert the binary number 1011 to Gray code.

let: $\underline{1011}_2 = b_3b_2b_1b_0$.

Apply the conversion rule:

$g_3 = b_3$ since there is no b_4 (assume $b_4 = 0$).

$g_2 = b_3 \oplus b_2$.

$g_1 = b_2 \oplus b_1$.

$g_0 = b_1 \oplus b_0$.

Calculate the Gray code bits:

$g_3 = 1 \oplus 0 = 1$ as b_4 doesn't exist and is thus a 0.

$g_2 = 1 \oplus 0 = 1$.

$g_1 = 0 \oplus 1 = 1$.

$g_0 = 1 \oplus 1 = 0$.

The Gray code is: $g_3g_2g_1g_0 = \underline{1110}_{\text{gray code}}$.

Chapter 2

Number Systems (Part II)

2.1 Addition and Subtraction of Unsigned Integers

Personal Remark. In case this is not clear, this video explains it pretty good:
https://www.youtube.com/watch?v=sJXTo3EZoxM

2.1.1 Addition of Binary Numbers

To add binary numbers, follow these rules:

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10$ (0 and carry 1 to the next higher bit)

$1 + 1 + 1 = 11$ (1 and carry 1 to the next higher bit)

Example:

Adding 1101_2 and 1011_2:

	1	1	1	0	(Carry)
	1	1	0	1	
+	1	0	1	1	
1	1	0	0	0	

2.1.2 Subtracting Two Binary Numbers

Works exactly like subtracting decimal numbers, but with a borrow of 2 instead of 10. The rules for binary subtraction include:

$0 - 0 = 0$

$1 - 0 = 1$

$1 - 1 = 0$

$0 - 1 = 1$ (with a borrow from the next higher bit)

Example:

Subtracting 1000_2 from 1101_2:

$$\begin{array}{rcccc} & 1 & 1 & 0 & 1 \\ - & 1 & 0 & 0 & 0 \\ \hline & 0 & 1 & 0 & 1 \end{array}$$

Therefore, the difference between 1101_2 and 1011_2 is 0010_2.

2.2 Overflow and Underflow in Unsigned Binary Arithmetic

2.2.1 Overflow

Overflow in unsigned binary arithmetic occurs when the sum of two numbers exceeds the maximum value that can be represented by the given number of bits. For an n-bit unsigned number, the maximum value that can be represented is $2^n - 1$. If the result of an addition is greater than this maximum value, the system experiences overflow, leading to an incorrect result.
Example: Consider adding two 4-bit unsigned numbers 1111_2 and 0001_2:

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array}$$

The result 10000_2 is a 5-bit number, but only the 4 least significant bits 0000_2 are kept in a 4-bit system, leading to overflow.

2.2.2 Underflow

Underflow in unsigned binary arithmetic occurs when the result of a subtraction is less than 0, which is not representable in unsigned arithmetic. Since unsigned numbers cannot represent negative values, any operation that would result in a negative value causes underflow.
Example: Consider subtracting a larger 4-bit unsigned number 1010_2 from a smaller one 0100_2:

$$\begin{array}{cccc} & 0 & 1 & 0 & 0 \\ - & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

underflow

Since the result would be negative, which cannot be represented in unsigned arithmetic, this situation is considered underflow.

2.3 Two's Complement Addition and Subtraction

Graphical Representation

In two's complement arithmetic, a circular graphical representation can be used to illustrate the addition and subtraction of numbers:

- Moving **clockwise** from 0 represents the *addition* of positive numbers.
- Moving **counterclockwise** represents the *subtraction* of positive numbers.
- Crossing the line where the sign changes indicates a *change of sign* from positive to negative or vice versa.

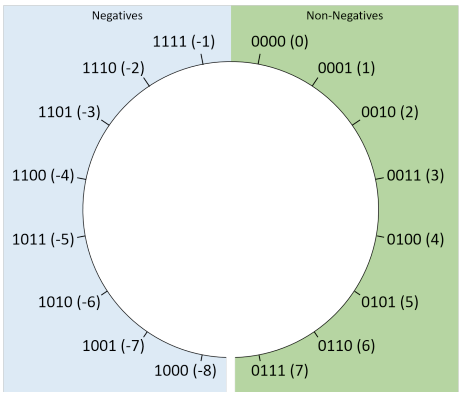

Figure 2.1: Circular representation of two's complement

Examples:

The addition of two positive numbers, such as $2 + 3$, is shown by moving clockwise from 0 to 2 and then moving 3 more steps clockwise, resulting in 5.

The subtraction of a smaller number from a larger number, such as $5 - 3$, is shown by moving clockwise from 0 to 5 and then moving 3 steps counterclockwise, resulting in 2.

2.3.1 Addition and Subtraction

2.3.2 Addition

Given two n-bit numbers A and B, their sum in two's complement arithmetic is obtained by directly adding them together as binary numbers:

$$\text{Sum} = A + B \tag{2.1}$$

If there is an overflow, i.e., a carry out of the most significant bit (MSB), it is discarded. The result is also represented in n bits.

2.3.3 Subtraction

To subtract one n-bit number B from another A using two's complement arithmetic, convert B to its two's complement and then add it to A:

1. Find the two's complement of B, denoted as $\bar{B}$, by inverting all the bits of B and adding 1.
2. Add A to the two's complement of B:

$$\text{Difference} = A + \bar{B} \tag{2.2}$$

As with addition, discard any overflow from the MSB.

2.4 Binary Multiplication

Proof. Let X and Y be two numbers, then their product can be represented as:

$$\begin{aligned} X \cdot Y &= X \cdot \left(-Y_{n-1} \cdot 2^{n-1} + X \sum_{i=0}^{n-2} Y_i \cdot 2^i \right) \\ &= -X \cdot Y_{n-1} \cdot 2^{n-1} + X \sum_{i=0}^{n-2} Y_i \cdot 2^i \\ &= -Y_{n-1} \cdot X \cdot 2^{n-1} + Y_{n-2} \cdot X \cdot 2^{n-2} + \ldots + Y_2 \cdot X \cdot 2^2 + Y_1 \cdot X \cdot 2^1 + Y_0 \cdot X \cdot 2^0 \end{aligned}$$

□

Binary multiplication operates similarly to decimal multiplication but is performed bit by bit. Here is a clearer example illustrating the multiplication of two binary numbers:

Multiplicand	1101	(This is the number to be multiplied)
Multiplier	$\times$ 0011	(This number multiplies the multiplicand)
	1101	(Multiply by 1, the least significant bit of the multiplier)
+	11010	(Multiply by 1, add one zero to the right, (left shift, $<< 1$))
+	000000	(Multiply by 0, add two zeros to the right, (left shift, $<< 2$))
+	0000000	(Multiply by 0, add three zeros to the right, (left shift, $<< 3$))
	1000111	(Sum of the partial products)

Chapter 3

Number Systems (Part III)

3.1 Fixed-Point Number Representation

Let x be an integer : $x = x_{int} + xfr$, with x_{int} the integer part and x_{fr} the fractional part.
Let X be a digit-vector:

$$X = (X_{m-1}X_{m-2}\dots X_1X_0 \,.\, X_{-1}X_{-2}\dots X_{-f})$$

where

X_{m-1} to X_0 represent the integer component

X_{-1} to X_{-f} represent the fractional component

The dot (.) represents the radix point (assumed to be fixed)

For unsigned numbers:

$$x = \sum_{i=-f}^{m-1} X_i \cdot 2^i$$

For signed numbers in two's complement:

$$x = -X_{m-1} \cdot 2^{m-1} + \sum_{i=-f}^{m-2} X_i \cdot 2^i$$

3.1.1 Examples of Fixed-Point Numbers

Decimal Numbers

Decimal number system with $m = 5, f = 5$

Example decimal digit vector

$$x = (10077.01690)$$

$$\begin{aligned} x &= 1 \cdot 10^4 + 7 \cdot 10^1 + 7 \cdot 10^0 + 1 \cdot 10^{-2} + 6 \cdot 10^{-3} + 9 \cdot 10^{-4} \\ &= 10000 + 70 + 7 + 0.01 + 0.006 + 0.0009 \\ &= 10077.0169 \end{aligned}$$

Most negative (min):

$$x_{\min} = -99999.99999 = \frac{-9999999999}{10^5}$$

Largest number (max, positive):

$$x_{\max} = +99999.99999 = \frac{+9999999999}{10^5}$$

Unsigned Binary Numbers

Unsigned with $m = 3, f = 4$

Example binary digit vector

$$X = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 5.4375$$

Smallest number (min):

$$x_{\min} = 000.0000_2 = 0$$

Largest number (max):

$$x_{\max} = 111.1111_2 = 7 + \frac{15}{16} = 7,9375$$

Sign-and-Magnitude Binary Numbers

Sign-and-magnitude with $m = 5, f = 3$

Example binary digit vector

$$X = (-1)^1 \cdot (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3}) = -5.75$$

Most negative number (min):

$$x_{\min} = 11111.111_2 = -15 + \frac{7}{8} = -15.875$$

Largest number (max, positive):

$$x_{\max} = 01111.111_2 = 15 + \frac{7}{8} = 15.875$$

3.2 Concepts of Finite Precision Math

3.2.1 Precision

Let X be a digit-vector:

$$X = (X_{m-1}X_{m-2}\dots X_1X_0 \,.\, X_{-1}X_{-2}\dots X_{-f})$$

Precision is the maximum number of non zero bits.

$$Precision(x) = m + f$$

with m the number of bits for the integer part and f the number of bits for the fractional part.

3.2.2 Resolution

Resolution is the smallest non-zero number that can be represented.

$$Resolution(x) = 2^{-f}$$

3.2.3 Range

The range of a fixed-point number is the difference between the largest and smallest numbers that can be represented.

$$Range(x) = x_{\max} - x_{\min}$$

In two's complement, the range is given by:

$$Range(x) = \sum_{i=-f}^{m-2} 2^i - \left(-2^{m-1}\right)$$

3.2.4 Accuracy

Accuracy is the maximum difference between a real value and the represented value.

$$Accuracy(x) = \frac{Resolution(x)}{2}$$

3.2.5 Dynamic Range

The dynamic range is the ratio of the largest and the smallest positive number that can be represented.

$$Dynamic\ Range(x) = \frac{x_{\max}}{x_{\min}}$$

In two's complement :

$$Dynamic\ Range(x) = \frac{2^{m-1}}{2^{-f}} = 2^{m-1+f}$$

3.3 Floating-Point Number Representation

Personal Remark. Please take the time to really understand this part. It is crucial for the understanding of the rest of the course. Take some time to understand the vocabulary and its meaning.

A real number that is exactly representable in a computer is called a **floating-point number**.

3.3.1 Significand, Base, Exponent

A floating-point number consits of a **significand** (or mantissa), a **base** (or radix), and an **exponent**.

the signed *significand* (also called *mantissa*) M^*

the signed *exponent* E

where b is a constant called the *base*

$$x = M^* \times b^E$$

This reminds us of the usual scientific notation, base 10:

$+35200 = +3.52 \times 10^4$ (Coefficient)

$-0.099 = -9.9 \times 10^{-2}$ (Exponent)

3.3.2 Benefits

1. *Consider 32-bit two's complement signed integers*

   The dynamic range for a 32-bit two's complement signed integer can be expressed as:

   $$\text{Dynamic Range}_1(x) = \frac{|x|_{\max}}{|x_{\text{positive, nonzero}}|_{\min}} = \frac{|-2^{32-1}|}{2^0} = 2^{31} \approx 2 \cdot 10^9$$

2. *Consider 32-bit floating-point number, with 24-bits significand and 8-bit exponent in two's complement*

   $\text{Dynamic Range}_2(x) = \frac{|x|_{\max}}{|x_{\text{positive, nonzero}}|_{\min}}$

   $$= \frac{(2^{23}-1) \cdot 2^{(2^{8-1})-1}}{2^{-128}} = \frac{(2^{23}-1) \cdot 2^{127}}{2^{-128}} = (2^{23}-1) \cdot 2^{255} \approx 5 \cdot 10^{83}$$

3. *Dynamic range increase*

   Comparing the two dynamic ranges:

   $$\frac{\text{Dynamic Range}_2(x)}{\text{Dynamic Range}_1(x)} \approx 10^{74}$$

1. *Consider a fixed-point number with 8 fractional bits*

$$\text{Resolution}_1(x) = 2^{-8}$$

2. *Consider 32-bit floating-point number, with 24-bits significand in sign-and-magnitude and 8-bit exponent in two's complement*

$$\text{Resolution}_2(x) = 2^0 \cdot 2^{-2(8-1)} = 2^{-2^7} = 2^{-128}$$

3. *Improved resolution*

$$\frac{\text{Resolution}_2(x)}{\text{Resolution}_1(x)} = \frac{2^{-128}}{2^{-8}} = 2^{-120}$$

3.3.3 Representation

We will be focusing on the following digit-vector:

$$X = (\underbrace{S}_{\text{Sign}} E_{m-1}E_{m-2}\dots E_1E_0 \underbrace{M_{n-1}M_{n-2}\dots M_0}_{\text{Magnitude}})$$

The floating-point representation becomes

$$x = (-1)^s \times M \times b^E$$

where $s \in \{0, 1\}$ is the **sign**, and M is the **magnitude** of the signed significant

In the rest of the lecture, we assume significand is represented in sign-and-magnitude.

Normalization

The goal of normalization is to represent the number such that the magnitude (M) is within the range $1 \leq M < 2$. This means that the leading digit before the binary point is always 1. Here are examples to demonstrate this process:

Positive Example:

Given: $+1010.1000_2$

Normalize: $+1.1010_2 \times 2^3$

Decimal Conversion: $1.3125 \times 8 = 10.5$

Explanation: The binary number 1010.1000_2 is normalized by shifting the binary point such that the first digit is 1, and adjusting the exponent (2^3) accordingly. The equivalent decimal number is 10.5.

Negative Example:

Given: $-(0.00000011)_2$

Normalize: $-1.1_2 \times 2^{-7}$

Decimal Conversion: $-(1.5)_{10} \times 2^{-7} = -0.01171875$

Explanation: The binary number 0.00000011_2 is normalized by shifting the binary point to get a leading 1, and adjusting the exponent (2^{-7}) to reflect the shift. The equivalent decimal number is -0.01171875.

Why Normalize?

Normalization removes redundancy from floating-point representation, making it unique. Consider these examples to understand the redundancy in non-normalized representations:

$$+ (1010)_2 \times 2^{-2} = 10 \times 2^{-2} = 2.5;$$
$$+ (1.01)_2 \times 2^{1} = 1.25 \times 2^{1} = 2.5;$$
$$+ (0101)_2 \times 2^{-1} = 5 \times 2^{-1} = 2.5;$$

In these cases, different representations lead to the same decimal value, illustrating redundancy. Normalization ensures that each number has a unique floating-point representation.

Conclusion: Floating-point representation is redundant unless it is normalized. By normalizing, we ensure a unique and efficient representation for computational purposes.

In normalized floating-point representation, the leading digit is always 1 and is omitted as a hidden bit to save space, with the remaining digits representing the fraction part of the significand :

$$+(101.001)_2 \times 2^{-4} \Rightarrow +(1.01001)_2 \times 2^{-2} \Rightarrow +(.01001)_2 \times 2^{-2}$$

3.3.4 Biased Representation

Given a binary number with n bits, the value of the biased representation can be calculated as follows:

$$x = \sum_{i=0}^{n-1} X_i \cdot 2^i - B$$

Where X_i represents the i^{th} bit of the binary number (starting from 0 for the least significant bit), and B is the bias, which is calculated by:

$$B = 2^{(n-1)} - 1$$

The exponent thus becomes :

$$e = \sum_{i=0}^{n-1} E_i \cdot 2^i - (2^{(n-1)} - 1)$$

For instance, with a 3-bit binary number, the bias B is $2^{(3-1)} - 1 = 3$. Therefore, the biased representation maps binary numbers to the integer range from $-B$ to $2^n - 1 - B$.

Example

For a 3-bit binary number, the biased representations would be:

Decimal	Binary	Biased Decimal
7	111	4
6	110	3
5	101	2
4	100	1
3	011	0
2	010	-1
1	001	-2
0	000	-3

Note that the minimum exponent in the biased representation is zero so that the representation of FP zero value is all zeros (zero sign, exponent, and mantissa).

Summary of the Floating-Point Representation

Let the binary vector :

$$X = (SE_{m-1}E_{m-2}\dots E_1E_0 \,.\, M_{n-1}M_{n-2}\dots M_0)$$

(m)-bit exponent

- Biased, $B = 2^{m-1} - 1$

$(n+1)$-bit significand

- Sign-and-magnitude
- Normalized, one hidden bit

$$x = (-1)^S \times \left(1 + \sum_{i=1}^{n} M_{n-i}2^{-i}\right) \times 2^{\left(\sum_{j=0}^{m-1} E_j 2^j\right) - (2^{m-1}-1)}$$

3.3.5 Rounding

The result of a floating-point operation is a real number that, to be represented exactly, might require a significand with an infinite number of digits.

For a representation close to the exact result, we perform **rounding**.

Consider the real number x_{real} and the consecutive floating-point numbers F_1 and F_2, such that $F_1 \leq x_{\text{real}} \leq F_2$.

We can perform several types of rounding:

Round to nearest (tie to even)
Round towards zero (truncation)
Round towards plus or towards minus infinity

- Round to nearest (tie to even)

$$R_{\text{near}}(x_{\text{real}}) = \begin{cases} F_1, & \text{if } |x_{\text{real}} - F_1| < |x_{\text{real}} - F_2| \\ F_2, & \text{if } |x_{\text{real}} - F_1| > |x_{\text{real}} - F_2| \\ \text{even}(F_1, F_2), & \text{if } |x_{\text{real}} - F_1| = |x_{\text{real}} - F_2| \end{cases}$$

- Round towards zero (truncation)

$$R_{\text{zero}}(x_{\text{real}}) = \begin{cases} F_1, & \text{if } x_{\text{real}} \geq 0 \\ F_2, & \text{if } x_{\text{real}} < 0 \end{cases}$$

- Round towards plus or minus (negative) infinity

$$R_{\text{pinf}}(x_{\text{real}}) = F_2$$

$$R_{\text{ninf}}(x_{\text{real}}) = F_1$$

Examples of Rounding Methods

Let's consider $x_{\text{real}} = 2.5$, $F_1 = 2$, and $F_2 = 3$ for our examples.

Round to nearest (tie to even)

$$R_{\text{near}}(2.5) = \text{even}(2, 3) = 2$$

Since both F_1 and F_2 are equidistant from x_{real}, we choose the even number which is 2.

Round towards zero (truncation)

$$R_{\text{zero}}(2.5) = 2$$

Since x_{real} is positive, we round towards zero, resulting in F_1.

Round towards plus infinity

$$R_{\text{pinf}}(2.5) = 3$$

When rounding towards plus infinity, we choose F_2.

Round towards minus infinity

$$R_{\text{ninf}}(2.5) = 2$$

When rounding towards minus infinity, we choose F_1.
Now let's consider a negative value $x_{\text{real}} = -2.5$, $F_1 = -3$, and $F_2 = -2$.

Round towards zero (truncation) for negative value

$$R_{\text{zero}}(-2.5) = -2$$

Since x_{real} is negative, we round towards zero, resulting in F_2.

Round towards plus infinity for negative value

$$R_{\text{pinf}}(-2.5) = -2$$

When rounding towards plus infinity for a negative number, we choose the larger number, which is F_2.

Round towards minus infinity for negative value

$$R_{\text{ninf}}(-2.5) = -3$$

When rounding towards minus infinity for a negative number, we choose the smaller number, which is F_1.

3.4 IEEE 754 Standard

FP Format in IEEE 754

Exactly what we described:

- $(n+1)$-bit significand
  - Sign-and-magnitude
  - Normalized, one hidden bit
- m-bit exponent
  - Biased, $B = 2^{m-1} - 1$

Let X a digit-vector represented as:

$$X = (SE_{m-1}E_{m-2}\dots E_1E_0.M_{n-1}M_{n-2}\dots M_1M_0)$$

Basic and extended formats:

+ Basic formats:
  - Single precision (32 bits)
    * Sign S: 1 bit
    * Exponent E: 8 bits
    * Fraction F: 23 bits
  - Double precision (64 bits)
    * Sign S: 1 bit
    * Exponent E: 11 bits
    * Fraction F: 52 bits
+ Default round to nearest (ties to even)

3.4.1 Special Values

The IEEE 754 standard defines special values with unique bit patterns:

Floating-point zero: is represented by all zeros in both the exponent and the significand fields.

Positive zero:	0	00000000	00000000000000000000000
Negative zero:	1	00000000	00000000000000000000000

The most significant bit (the sign bit) differentiates between positive and negative zero.

Positive and negative infinity: are represented by all ones in the exponent field and all zeros in the significand field.

Positive infinity:	0	11111111	00000000000000000000000
Negative infinity:	1	11111111	00000000000000000000000

NaN (Not a Number): is represented by all ones in the exponent field and a non-zero significand field.

NaN (example):	–	11111111	10000000000000000000000

NaN values represent indeterminate or undefined results, such as the square root of a negative number. The sign bit can be either 0 or 1, but the significand must not be all zeros.

3.4.2 Overflow, underflow, and others

Overflow: Occurs when the rounded value is too large to be represented by the floating-point format.

- The result is set to positive or negative infinity, depending on the sign.

Underflow: Happens when the rounded value is too small to be represented.

- Typically, the result is set to a denormalized number or zero.

Division by zero: Occurs when a finite non-zero number is divided by zero.

- The result is set to positive or negative infinity, based on the sign of the numerator.

Inexact result: Occurs when the result of an operation is not an exact floating-point number.

- The result is rounded to the nearest representable value.

Invalid: This flag is set when the result of an operation is not a real number (NaN).

- Examples include the square root of a negative number or the indeterminate form 0/0.

IEEE 754

Example: Converting single-precision FP to decimal

Find the decimal equivalent of

$$X = (1\ 01111100\ 01000000000000000000000)$$

Solution:

- Sign $S = 1$, hence negative.
- Exponent $E = 01111100$ represents the biased exponent. To find the actual exponent:
$$E_{\text{actual}} = 124 - (2^7 - 1) = 124 - 127 = -3$$
- Mantissa (including the hidden bit) $M = 1.01$ in binary represents:
$$M = 1 + 2^{-2} = 1.25$$
- Result:
$$x = -1.25 \times 2^{-3} = -0.15625$$

Example: Converting decimal to single-precision FP

Find the single-precision FP equivalent of $x = -0.8125$
Solution:

- Sign = 1, negative.
- Fraction bits can be obtained using multiplication by 2.
- Converting 0.8125 to binary by successive multiplication:
$$0.8125 \times 2 = 1.625 \to 1$$
$$0.625 \times 2 = 1.25 \to 1$$
$$0.25 \times 2 = 0.5 \to 0$$
$$0.5 \times 2 = 1.0 \to 1$$
Stop when the fractional part becomes zero.
- Mantissa M in binary is 0.1101, normalized is 1.101×2^{-1}.
- Exponent adjustment:
$$E_{\text{actual}} = -1$$
$$E = E_{\text{actual}} + B = -1 + (2^7 - 1) = 126$$
- Result:
$$X = (1\ 01111110\ 10100000000000000000000)$$

Chapter 4

Number Systems (Part IV)

4.1 Fixed Point Arithmetic

4.1.1 Addition and Subtraction

Let x and y be two fixed-point numbers with the same number of integer and fractional bits. The sum and difference of x and y can be calculated as follows:

$$x + y = x_{int} + y_{int} + x_{fr} + y_{fr}$$

$$x - y = x_{int} - y_{int} + x_{fr} - y_{fr}$$

Examples

Addition

011011	$(Carry)$
000101.110	$= 5.75$
001100.011	$= 12.375$
010010.001	$= 18.125$

Substraction

1110000 110	$(Carry)$
000101.110	$= 5.75$
$-$ 001100.011	$= 12.375$
111001.011	$= -6.625$

In two's complement

Given two fixed-point numbers x and y, the addition or subtraction in two's complement is given by:

$$x \pm y = \left(-X_{(m_x-1)} 2^{(m_x-1)} + \sum_{i=-f_x}^{m_x-2} X_i 2^i \right) \pm \left(-Y_{(m_y-1)} 2^{(m_y-1)} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i \right) \tag{4.1}$$

Where:

m_x and m_y are the total number of bits for the integer components of x and y respectively.

f_x and f_y are the number of bits for the fractional components of x and y respectively.

$X_{(m_x-1)}$ and $Y_{(m_y-1)}$ are the sign bits of x and y.

The largest integer-part exponent is $\max(m_x-1, m_y-1)$. Consequently, the number of bits for the resulting integer component is $\max(m_x, m_y)+1$.

The smallest fractional-part exponent is $\min(-f_x, -f_y)$. Consequently, the number of bits for the resulting fractional component is $\max(f_x, f_y)$.

4.1.2 Multiplication

	010.11	Multiplicand	
×	011.01	Multiplier	
	000000	First partial product (always zero), sign-extended	
+	001011	1 x multiplicand, sign-extended	
	001011	Intermediate result, sign-extended	
+	00000	0 x multiplicand, left-shifted by 1 place and sign-extended	
	00001011	Intermediate result, sign-extended	convert to fixed-point
+	001011	1 x multiplicand, left-shifted by 2 places and sign-extended	
	00010111	Intermediate result, sign-extended	
+	001011	1 x multiplicand, left-shifted by 3 places and sign-extended	
	001001111	Intermediate result, sign-extended	
+	000000	0 x multiplicand, left-shifted by 4 places and sign-extended	
	0010001111	Result, integer	

Multiplication in Two's Complement

Multiplication on two binary numbers $x(m_x, f_x)$ and $y(m_y, f_y)$

$$x \cdot y = (x_{\text{int}} + x_{\text{fr}}) \cdot (y_{\text{int}} + y_{\text{fr}})$$

In two's complement:

$$x \cdot y = \left(-X_{m_x-1} 2^{m_x-1} + \sum_{i=-f_x}^{m_x-2} X_i 2^i \right) \cdot \left(-Y_{m_y-1} 2^{m_y-1} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i \right)$$

The largest integer-part exponent: $(m_x - 1) + (m_y - 1)$. Consequently: $m_{xy} = m_x + m_y$

The smallest fractional-part exponent: $(-f_x) + (-f_y)$. Consequently: $f_{xy} = f_x + f_y$

4.2 Floating-Point Arithmetic

Let x and y be represented as (S_x, M_x, E_x) and (S_y, M_y, E_y)

The significands $M^* = (-1)^S M$ are normalized

Addition/subtraction result is z, also represented as (S_z, M_z, E_z):

$$z = x \pm y = M_x^* \times 2^{E_x} \pm M_y^* \times 2^{E_y}$$

The significand of the result is also normalized

$$z = M_z^* \times 2^{E_z}$$

Four main steps to compute the result of floating-point addition/subtraction:

1. **Add/Subtract significand and set exponent:**
   - Align the significands by shifting the one with the *smaller* exponent.
   - Perform addition/subtraction on the aligned significands.

$$M_z^* = \begin{cases} (M_x^* + (M_y^* \times 2^{(E_y - E_x)})) \times 2^{E_x} & \text{if } E_x \geq E_y \\ ((M_x^* \times 2^{(E_x - E_y)}) + M_y^*) \times 2^{E_y} & \text{if } E_x < E_y \end{cases}$$

$$E_z = \max(E_x, E_y)$$

2. **Normalize the result and update the exponent, if required:**
   - Check if the result's significand is within the normalized range.
   - If not, shift the significand to the right or left until it is normalized, adjusting the exponent accordingly to maintain the value.
3. **Round the result, normalize, and adjust exponent, if required:**
   - Apply rounding rules to the significand to fit within the precision limits.
   - After rounding, if the significand overflows (e.g., carries out during addition), normalize the result again and adjust the exponent.
4. **Set flags for special values, if required:**
   - Check for overflow or underflow conditions and set flags accordingly.
   - Identify and mark results that are special values (e.g., infinity, NaN) based on the operation and input values.

Step 1: Floating-Point Addition/Subtraction Detailed Algorithm

Algorithm steps:

- Subtract exponents $d = E_x - E_y$.
- Align significands:
  - Compare the exponents of the two operands.
  - Shift right d positions the significand of the operand with the smallest exponent.
  - Select as the exponent of the result the largest exponent.
- Add/subtract signed significands and produce the sign of the result.

Table 4.1: Floating-point operations based on the signs of the operands.

FP operation	Signs of the operands	Effective operation
+	=	add
+	≠	subtract
−	=	subtract
−	≠	add

Step 2: Normalize the Result

After the initial addition or subtraction, the result may not always be in the normalized form required by floating-point representation standards. Normalization ensures that the significand (mantissa) is within a specific range, usually just below 1 (for binary floating-point numbers, this means the leading bit is just to the right of the decimal point).

Steps for normalization:

- If the result of the operation causes the significand to exceed its predefined size (overflow), the significand is shifted to the right, and the exponent is increased accordingly.
- Conversely, if the operation results in a significand that's too small (underflow), the significand is shifted to the left, and the exponent is decreased.
- This process ensures that the floating-point number is as close to its true value as possible within the limits of the representation.

Step 3: Round the Result

After normalization, the next step is to round the result to fit within the target floating-point format's precision. Rounding is crucial because it affects the accuracy and representation of the result.

Steps for rounding:

- Evaluate the significand's precision and compare it with the format's limit.
- If the significand exceeds the precision limit, round it according to a rounding rule (e.g., round to nearest, round towards zero, round towards positive/negative infinity).
- Common rounding strategies include:
  - **Round to Nearest:** Round to the nearest value, with ties going to the nearest even number.
  - **Round Down (Towards Zero):** Always round towards zero, truncating any fractional part.
  - **Round Up (Away from Zero):** Always round away from zero, increasing the magnitude of the result.

- After rounding, if there's an overflow in the significand (e.g., a carry into a new digit), normalize the result again. This may involve shifting the significand and adjusting the exponent.

Step 4: Set Flags for Special Values

The final step in floating-point addition or subtraction involves handling special cases and setting flags accordingly. Special values include infinity, not-a-number (NaN), and potential overflow or underflow conditions.

Handling special values:

- **Infinity:** If the result of the operation is too large to be represented in the given floating-point format, set the result to infinity. The sign of infinity depends on the operation and operands.
- **NaN (Not-a-Number):** If the operation involves invalid operations (e.g., $0/0$, $\infty - \infty$), set the result to NaN. NaN propagates through most floating-point operations.
- **Overflow:** If the result exceeds the maximum representable value, set an overflow flag. The result is typically set to infinity with the appropriate sign.
- **Underflow:** If the result is too small to be represented (closer to zero than the smallest representable value), set an underflow flag. The result may be set to zero or the smallest denormalized number, depending on the format and flags.

4.2.1 An Example in Binary

Let's add two binary floating-point numbers: 1.01×2^3 and 1.1×2^2. Here's how we do it step by step:

1. **Line Up the Dots:** First, we need to align the exponents. We'll adjust the second number to have the same exponent as the first, by increasing its exponent and shifting its significand to the right:

   $$1.1 \times 2^2 = 0.11 \times 2^3$$

   Now, both numbers are 1.01×2^3 and 0.11×2^3.

2. **Add Them Up:** With the exponents aligned, we can now add the significands:

   $$1.01 + 0.11 = 10.00$$

   The result in binary is 10.00. Since we're working in binary, 10.00 is actually 2 in decimal.

3. **Make It Look Right:** The result 10.00×2^3 is already in the correct format, but let's note that if our result was something like 1.000×2^4, we would need to adjust it to keep it in normalized form.

4. **Round It Off:** Our result doesn't need rounding in this case, but if we had more digits than we could store, we'd round off to the nearest value we could represent.

5. **Check for Special Cases:** There are no special cases here, as our result is a regular binary floating-point number.

So, our final result is 10.00×2^3 in binary, which is 8 in decimal.