Fundamentals of Digital Systems - Notes and Summary

Faustine Flicoteaux

Spring Semester 2025

Contents

Introduction

These are my notes for the Fundamentals of Digitals System (CS-173) course given during the spring semester of 2025 at EPFL. Please note that the content is not mine but belongs to Professor Mirjana Stojilovic, who taught it.

The first chapter of this document is also heavily based on the one of Ali El Azdi, that can be found at `https://github.com/elazdi-al/FDS/blob/main/FDS.pdf`. I have however changed some formulations and such.

This summary is not exempt of errors. If you find one, you can contact me at my EPFL e-mail address: `faustine.flicoteaux@epfl.ch` or through the GitHub page `https://github.com/FocusedFaust/LectureNotes`.

Note that the GitHub repository is also where I have the latest pdfs and TEXdocuments, for this course and others.

Chapter 1

Number Systems (W1.1)

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. (with leftward-increasing indexing)

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

X_0 is called the zero-origin.
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 non-redundant if each digit-vector represents a different integer, meaning that they have different weights. For example, the decimal system is non-redundant.

1.2 Weighted Number Systems

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

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

where

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

Each W_i is the weight of the the i^{th} digit, its importance in the representation of the number.

1.2.1 Radix Systems

When weights are in this format :

$$\begin{cases} W_0 = 1 \\ W_{i+1} = W_i R_i \text{ with } 1 \leq i \leq n-1 \end{cases} \tag{1.1}$$

Also written : $W_0 = 1$, $\prod_{j=0}^{i-1} R_j$, it is a **radix** system.

1.2.2 Fixed and Mixed-Radix Number Systems

In a **fixed-radix system**, all elements of the radix-vector (i.e. all R_i) 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

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 $W = (3600, 60, 1)$

1.2.3 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, …, 9}
- Binary: {0, 1}
- Hexadecimal: {0, 1, 2, …, 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.

1.3 Representation of Signed Integers

1.3.1 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* (a positive integer).

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

0 for positive

1 for negative

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

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

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)*:

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

The main disadvantage of SM is that it's complex to design digital circuits for arithmetic operations (addition, subtraction, etc.).

1.3.2 True-and-Complement (TC)

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:

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

Unambiguous Representation

To have an unambiguous representation, the two regions should not overlap, translating to the condition: $\forall x, \max |x| < \frac{C}{2}$

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.

1.3.3 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.

The range is asymmetrical:

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

but the representation of zero is unique.

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:

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

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

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

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

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

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

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.

1.4 Range Extension and Arithmetic Shifts

1.4.1 Range Extension

This is 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 represented as:

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

1.4.2 Range Extension Algorithm in Sign-and-Magnitude

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 $X = 11010101_2 = -85_{10}$, is equivalent to $Z = 10001010101 = -85_{10}$ in an 8-bit system.

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

1.4.3 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 $z = x/2$, with rounding towards zero when x is negative.

Left Arithmetic Shift in Sign-and-Magnitude System

Algorithm (assuming the overflow does not occur):

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

Right Arithmetic Shift in Sign-and-Magnitude System

Algorithm:

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

Left Arithmetic Shift in Two's Complement System

Algorithm (assuming that overflow does not occur):

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

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.

1.5 Hamming Weight and Distance

1.5.1 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.

1.5.2 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.

Chapter 2

Number systems (Part II) (W1.2)

2.1 Addition and Substraction of Unsigned Integers

2.1.1 Addition of binary numbers

Binary numbers act relatively close to the "exclusive or" in propositional logic, with the difference being the "carry". Just like regular addition by hand, we right-align the LS digits and start from the least significant column. Then, the rules are:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 10$ we carry the 1 to the next digit

The carry taking part in the sum are called "**carry-in**" and those produced as part of the partial result (which we write on top of the addition) are called "**carry-out**".

How many bits?

How many bits are needed to represent the sum of two n-bit unsigned binary numbers?

The minimum possible value is $s_{min} = 0 + 0 = 0$

The maximum possible value is $s_{max} = (2^n - 1) + (2^n - 1) = 2 \cdot 2^n - 2 = 2^{n+1} - 2$ This leads us to conclude that we need $n + 1$ bits.

However, in hardware, we do not always need the extra bit. When the magnitude of the result exceeds the largest representable value, we say an **overflow** occurs and the result in incorrect.

2.1.2 Substraction of binary numbers

The idea is the same as for decimal numbers. The direction is from the right to the left, starting with the least-significant digit. Instead of the carry, we have the borrow, which is substracted in the next column. The rules are:

- $0 - 0 = 0$
- $1 - 0 = 1$
- $0 - 1 = 1$ with a borrow on the next column
- $1 - 1 = 0$

Negative Result? If the result should be negative, we cannot represent it using an unsigned system. Therefore, an **underflow** occurs, and the result is incorrect.

2.2 Two's Complement Arithmetic

We represent integers graphically as being around a circle. When Performing an addition, we go clockwise. When performing a substraction, we go counter-clockwise.

There is a line dividing the circle in two, representing the sign change. This is because at some point when doing additions/substraction, the carry-out/borrow is stored in the sign bit, changing its value. This is the same as adding a positive value to a negative integer and getting a positive result and vice-versa.

This means that the result will always be correct as long as the range is not exceeded.

This is a good technique for hardware implementation, as the same hardware can perform the addition of unsigned numbers.

Overflow Adding two numbers of different signs never produces an overflow. If the signs of the two numbers are the same but different from the sign of the sum, then overflow occurred.

Otherwise, if the carry-in and the carry-out of the sign position are different, overflow occurred.

2.2.1 Two's complement Substraction

If we remember last chapter, to substract x, we can add the opposite value that we now know how to construct ($-x =\sim x + 1$).

2.2.2 Unsigned Integer Multiplication

The formal calculation is:

$$\begin{aligned} X \cdot Y &= X \cdot \sum_{0}^{n-1} Y_i \cdot 2^i \\ &= \sum_{0}^{n-1} X \cdot Y \cdot 2^i \\ &= Y_{n-1} \cdot \underbrace{X \cdot 2^{n-1}}_{\text{left-shifted by n-1}} + \ldots + Y_1 \cdot \underbrace{X \cdot 2^1}_{\text{left-shifted by 1}} + Y_0 \cdot \underbrace{X \cdot 2^0}_{\text{multiplicand}} \end{aligned}$$

This means that we multiply by each digit of Y, shifting each time by the radix of that digit and summing with the previous result.

How many bits? When multiplying a number with n bits by another with m bits. How many bits at most will be needed to represent the result ?

$n + m$, because we shift each time and each intermediate result needs one more bit than the one before.

2.2.3 Two's Complement Multiplication

The algorithm for Two's Complement Multiplication is inspired by the previous algorithm, except that the $n - 1^{th}$ bit is multiplied by minus one.

$$X \cdot Y = \underbrace{-}_{sign} Y_{n-1} \cdot \underbrace{X \cdot 2^{n-1}}_{\text{left-shifted by n-1}} + \ldots + Y_1 \cdot \underbrace{X \cdot 2^1}_{\text{left-shifted by 1}} + Y_0 \cdot \underbrace{X \cdot 2^0}_{\text{multiplicand}}$$

When doing hand multiplication, we have to be aware of the sign. To do that, we extend each multiplicand with another sign bit.

Chapter 3

Number systems (Part III) (W2.1)

3.1 Fixed-Point Numbers

Fixed-point numbers are either

1. Integers
2. Rational numbers of the form $x = a/2^f$

the fixed-point representation of a number x consists of integer x_{int} and fraction x_{fr} such that $x = x_{int} + x_{fr}$.

The digit point representation is

$$X = (\underbrace{X_{m-1}X_{m-2}\dots X_1X_0}_{\text{integer component}} \underbrace{.}_{radix-point} \underbrace{X_{-1}X_{-2}\dots X_{-f}}_{\text{fractional component}})$$

The position of the radix-point is assumed to be fixed (hence the name). If it is not shown, it is assumed that the number is an integer.

We see that each X_i with $i < 0$ has a weight that is also < 0. In decimal system, the weight would be 10^i so $0.0\dots01$.

3.2 Finite Precision Maths

3.2.1 Precision

The **precision** is the maximum number of non-zero bits (see earlier).

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

If $m = 5$ and $f = 5$, the precision is 10.

If $m = 10$ and $f = 6$, the precision is 16.

If $m = 8$ and $f = 0$ (integer), the precision is 8. In the general case,

Precision$(x) = m + f$

3.2.2 Resolution

The **resolution** is the smallest possible difference between two consecutive numbers.

Examples $X = (X_{m-1}X_{m-2}\dots X_1X_0.X_{-1}X_{-2}\dots X_{-f})$
If $m = 5$ and $f = 5$, the resolution is $1/2^5 = 1/32 = 0.03125$.
If $m = 10$ and $f = 6$, the resolution is $1/2^6$.
If $m = 8$ and $f = 0$ (integer), the resolution is $1/2^0 = 1/1 = 1$. In the general case, Resolution$(x) = 2^{-f}$ and it is **fixed**.

3.2.3 Range

The **range** is the difference between the most positive and the most negative number representable.

In the general case, for fixed-point and two's complement,

$$\text{Range}(x) = x_{max} - x_{min} = \sum_{i=-f}^{m-2} 2^i - (-2^{m-1})$$

3.2.4 Accuracy

The **accuracy** is the magnitude of the maximum difference between a **real** value and its representation.

The worst case (max difference) occurs for a real value exactly between two representable numbers.

In the general case, Accuracy(x) =Resolution$(x)/2$

3.2.5 Dynamic Range

The **dynamic range** is the ratio of the the **maximum absolute value** representable and the **minimum positive absolute** value representable.

Example Two's complement with $m = 5$ and $f = 3$

- The maximum absolute value representable is $|x|_{max} = |-2^4| = 16$
- The minimum positive absolute value representable is $|x_{positive,nonzero}|_{min} = |2^{-3}| = 1/8$
- The dynamic range is $|x|_{max}/|x_{positive,nonzero}|_{min} = 128$

In the general case, for fixed-point and two's complement, Dynamic Range$(x) = 2^{m-1}/2^{-f} = 2^{m-1+f}$

3.3 Floating-Point Number Representation

As with any other number representation in a digital system, Floating-Point (FP) representation is encoded in a finite number of bits. It represents only a finite subset of the infinite set of real numbers.

3.3.1 Significand, Exponent, Base

FP representation is similar to the "notation scientifique" that many of us know and which is in base 10.

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

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

E is the signed **exponent**

b is a constant called the **base**

3.3.2 Significand

Sign-and-Magnitude

This the the most used representation for significand because it simplifies multiplication in hardware. 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.

The representation is

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

with a $(n+1)$-bit significand in sign-and-magnitude and an m-bit exponent.

This representation however is redundant, as multiple magnitude and exponent combinations can give the same number. This is the case unless we **normalize the magnitude**: $1 \leq M < 2$ (this would be the same as moving the floating-point and multiplying by b^i with i depending on the movement).

Hidden Bit and Fraction

As the significand is normalized, the first digit of the magnitude is always binary 1 (because normalization makes $1 \leq M < 2$).

The first digit of the significand is omitted (hidden bit) because it is always the same.

The binary point is assumed to the right of the hidden bit and the represented part of the significand is called a fraction F ans

Remark: Here, hidden means "given". It is a given that there is a bit here.

Summary

The common significand representation is the following:

- Sign-and-Magnitude
- Normalized
- One hidden bit

The corresponding significand value becomes

$$(-1)^S \times (\underbrace{1}_{\substack{hidden \\ bit}} + \underbrace{\sum_{i=1}^{n} M_{n-i} 2^{-i}}_{fraction})$$

3.3.3 Exponent

The exponent needs to be signed, because when **positive**, it represents very large numbers (large absolute value) and when **negative**, it represents very small numbers (small absolute value).

Biased Representation

Exponents can take any representation but one representation is called **biased** and simplifies comparison in hardware.

The biased representation of a digit vector $X = (X_{n-1} \dots X_1 X_0)$ is

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

where B is the bias. Typically, $B = 2^{n-1} - 1$, which centers representable numbers around 0. They stay sorted just like unsigned integers but cover both the positive and negative numbers.

Summary

For the binary digit vector $X = (S E_{m-1} E_{m-2} \dots E_1 E_0 . M_{n-1} M_{n-2} \dots M_0)$ the biased exponent value becomes

$$e = \sum_{j=0}^{m-1} E_j 2^j - (2^{m-1} - 1)$$

3.3.4 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. To obtain a representation close to the exact result, we perform what is called **rounding**.

3.3.5 Floating-Point Format Summary

$$X = (S\ E_{m-1} E_{m-2} \dots E_1 E_0\ .\ M_{n-1} M_{n-2} \dots M_0\ )$$

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

Rounding Modes

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

There are various rounding modes, such as:

- Round to the **nearest** or to **even** when tie

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

- Round towards **zero** (truncate)

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

- Round towards plus or towards minus **infinity**

$$R_{pinf}(x_{real}) = F_2, R_{ninf}(x_{real}) = F_1$$

3.4 IEEE Standard 754

The FP format in IEEE 754 is what we described earlier, with
$X = (SE_{m-1}E_{m-2} \dots E_1E_0M_{n-1}M_{n-2} \dots M_0)$.
Basic formats include:

- 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

The default rounding mode is to round to nearest and to even when tie.

3.4.1 Special Values

The floating-point **zero** is represented with $E = 0, F = 0$. The sign S differentiates between positive and negative zero.

Positive and negative **infinity** are with biased exponents to all ones and $F = 0$.

NaN (not a number) is used to represent results of invalid operations. The sign is either 0 or 1, biased exponents are all ones, $F \neq 0$.

3.4.2 Exceptions Handling

The following five exceptions set a flag (i.e., "activate an alarm") and the computation continues:

- Overflow (value is too large, result is set to infinity)
- Underflow (value is too small)
- Division by zero
- Inexact result (not an exact floating-point number)
- Invalid result (NaN is produced)

Chapter 4

Number systems (Part IV) (W3.1)

4.1 Fixed-Point Arithmetic

4.1.1 + and -

Performing additions or substractions on two binary numbers $x(m, f)$ and $y(m, f)$ is done in **the same way** as if the operands were integers. However, overflow can still happen.

$$\begin{cases} x + y = x_{int} + x_{fr} + y_{int} + y_{fr} \\ x - y = x_{int} + x_{fr} - (y_{int} + y_{fr}) \end{cases}$$

How many bits?

The largest integer-part exponent is $max(m_x - 1, m_y - 1)$. Consequently, $m_{x \pm y} = max(m_x, m_y) + 1$.

The smallest fractional-part exponent is $min(-f_x, -f_y)$. Consequently, $f_{x \pm y} = max(f_x, f_y)$.

4.1.2 x

Multiplication on two binary numbers $x(m, f)$ and $y(m, f)$ uses the same algorithm as if the operands were integers. Binary point location changes and overflow can still happen.

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 signed significands are normalized.

4.2.1 + and -

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

There are four main steps to computing an addition or substraction.

1. Add/subtract significand (mantissa) and set the exponent. The mantissa of the number with the smaller exponent has to be multiplied by two to the power of the difference between the exponents (this operation is called alignment) and then added/subtracted to the mantissa of the other number

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

2. **Normalize** the result and then, if required, **adjust the exponent**
3. **Round** the result and then, if required, normalizeitand adjust the exponent
4. Set flags for **special values**, if required