Fundamental of digital systems

Arthur Herbette
Prof. Mirjana Stojilovic

Lundi 10 mars 2025

Table des matières

Liste des cours

Lundi 17 février 2025 — **Cours 1 : Number Systems**

Chapitre 1

Numbser Systems

1.1 Digital representations

Introduction

- In mathematics, a **tuple** is a finite ordered sequence of elements.
  - An **n-tuple** is a tuple of n elements, where n is a nonnegative integer
- In a **digital representation**, a number is represented by an **ordered n-tuple**
  - Each element of the n-tuple is called a **digit**
  - The n-tuple is called a **digit vector** (or string of digits)
  - The number of digits n is called the **precision** of the representation

1.1.1 Representation of nonnegative integers

Integer Digit-Vector

- **Digit-vector (string)** representing the integer x is denoted by :

$$X = (X_{n-1}, X_{n-2}, \ldots, X_1, \overbrace{X_0}^{\text{zero-origin}})$$

We see here that it is a leftward-increasing indexing

- **Least-significant** digit (also called low order digit) : X_0
- **Most-significant** digit (also called high-order digit) : X_{n-1}

Elements of a number System

$$X = (X_{n-1}, X_{n-2}, \ldots, X_1, X_0)$$

- The number system to represent the integer x consists of
  - the number of **digits** n
  - A set of numerical **values** for the digits
    - if a **set of values for a digit** X_i is D_i, the cardinality of D_i is $|D_i|$
  - A rule of interpretation
    - Mapping between the set of digit-vector values and the set of integers
  - **Set size**
    - The set of integers is a finite set of at most K elements

$$K = \Pi_{i=0}^{n-1} |D_i|$$

Example : Decimal number system

$$X = (X_{n-1}, X_{n-2}, \dots, X_1, X_0)$$

- Number of digits n
  - Can be any, but let us consider $n = 6 (e.g., 17, 9899, 676799, \dots)$
  - Leading zeros are irrelevant
- Digit set in decimal number system
  - $D_i = \{0, 1, 2, \dots, 9\}$ of cardinality 10
- The correponding set size of K is one million values, from 0 to $K - 1$
  - $K = \prod_{i=0}^{n-1} 10 = 10^6$

(Non)Redundant Number systems

- A number system is **nonredundant** if
  - ... each digit-vector represents a **different** integer
  - E.g., the decimal system is nonredundant as every number is unique
- Alternatively, a number system is **redundant** if ...
  - ... there are integers represented by **more than one** digit-vector

Weighted (Positional number systems

- Most frequently used number systems are **weighted systems**
- The rule of representation :

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

Where $W = (W_{n-1}, W_{n-2}, \dots, W_1, W_0)$ is the **weight-vector** of size n

- Equivalent formulation :

$$x = X_{n-1}W_{n-1} + X_{n-2}W_{n-2} + \cdots + X_1 W_1 + X_0 W_0$$

Example Decimal Number system

- Weights are a power of 10. Example :
  - Digit Vector $X = (8, 5, 4, 6, 0, 3)$
  - Weight vector $W = (10^5, 10^4, 10^3, 10^2.10^1, 10^0)$

$x = 8 \cdot 10^5 + 5 \cdot 10^4 + 4 \cdot 10^3 + 7 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$
$x = 854703_{10}$

- When weights are of the format
  - $W_0 = 1$ and
  - $W_i = W_{i-1} R_{i-1}, \;\; i \leq i \leq n-1$

We have a **radix number system**

Radix number system

Définition 1 *Radix number systems are weighted number system in which the weight vector is related to the* ***radix vector*** $R = (R_{n-1}, R_{n-2}, \dots, R_1, R_0)$ *as follows :*

$$W_0 = 1; \; W_i = W_{i-1} R_{i-1}, \; 1 \leq i \leq n-1$$

- Equivalent to

$$W_0 = 1; \; W_i = \prod_{j=0}^{i-1} R_j$$

- E.g., in the decimal number system $W_0 = 1; W_i = \prod_{j=0}^{i-1} 10$

Fixed and Mixed-Radix number systems

- In a **fixed-radix** system, all elements of the radic-vector have the same value **r (the radix)**
- The weight vector in a fixed-radix system :

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

and the integer x becomes

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

Example

- Characteristics of the decimal number system :
  — Radix $r = 10$
  — **Fixed-radix** system

Binary/Octal/-Hexadecimal to/from Decimal I won't really go into the details here but the main thing to know is to convert from a system to one another (with the most famous ones)

Representation of signed Integers

Sign-and-Magnitude (SM)

- A signed integer x is represented by a pair

$$(x_s, x_m)$$

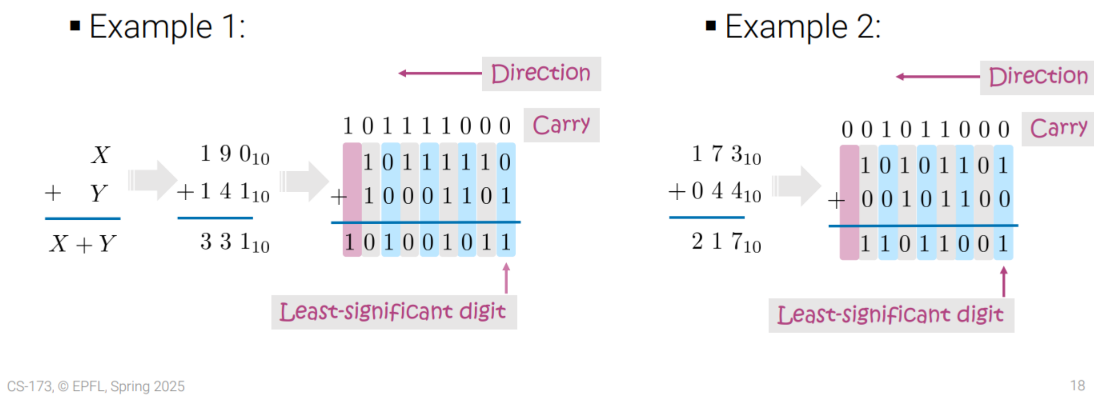
where x_s is the **sign** and x_m is the **magnitude** (positive integer)
- Sign (positive, negative) is represented by a binary bariables
  — $0 \implies$ positive ; $1 \implies$ negative
- Magnitude can be represented as any positive integer
  — In a conventional radix-r system, the range of n-digit magnitude is :

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

Jeudi 20 février 2025 — **Cours 2 : Arithmetic Operations with integers**

1.2 Addition of unsigned Integers

By hand We use here the same principle as a classical addition by hand of decimal numbers :



How many Bits are needed To represent the **sum of two n-bit unsigned numbers** we use $\boldsymbol{n+1}$
For exemple the minimum space is when there are $0 + 0$ which leads to :

$$s_{min} = 0 + 0 = 0$$

and for the maximum :

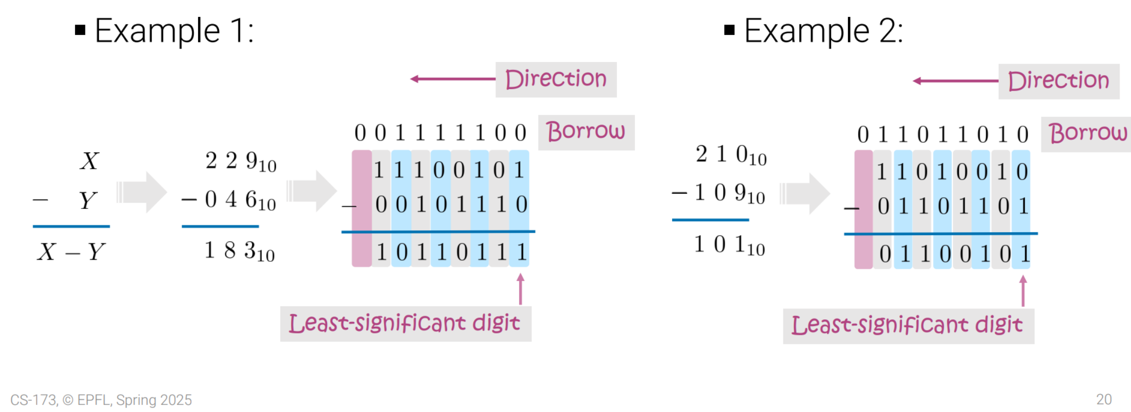
$$s_{max} = (2^n - 1) + (2^n - 1) = 2 \cdot 2^n - 2 = 2^{n+1} - 2$$

which makes it $n + 1$ bits for the sum.

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

1.2.1 Substraction of Unsigned Integers

- We use here the same idea as for decimal numbers :



Negative result

- Negative results cannot be represented using an unsigned system
- When trying to represent a value smaller than the minimu representable by the given number of bits n, an integer **underflow** occurs, and the result is incorrect.

1.2.2 Two's Complement Addition/substraction

Addition We use here the same algorithm as for the unsigned numbers, and if the result exceeds the range, **overflow** occurs.

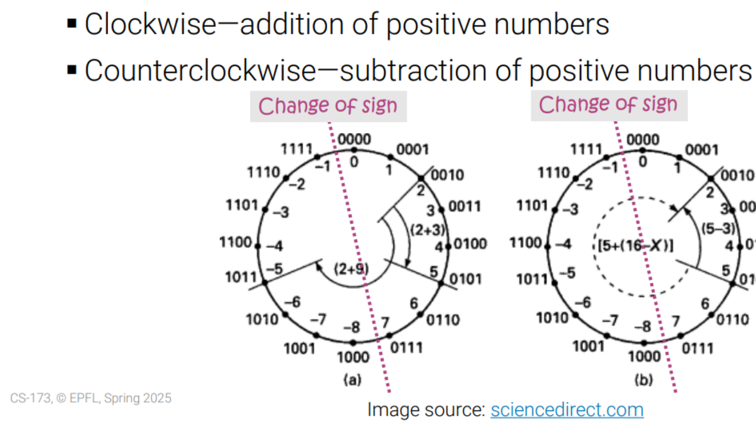
To refresh how signes numbers works, for example $1000_2 = -8_{10}$ which is the "*most negative*" number with 4 bits. Then we add the right side of the number as positive integers like this :

$$\underbrace{1}_{-8_{10}}\overbrace{010_2}^{2} = -6_{10}$$

If we want to sum up -5 and 7 for exemple :

$$\begin{aligned} 1011 + 0111 \\ \overbrace{1+1}^{1}0 \\ \overbrace{1\ }^{1}\ 10 \\ \overbrace{1\ }^{1}\ 010 \\ 0010 = 2_{10} \end{aligned}$$

We use it as a clock :



The preferred representation in digital Systems

- If we start with the smallest (most negative) number $1000_2 = -8_{10}$ and count up all succesive numbers up to $0111_2 = 7_{10}$ can be obtained by adding 1 to the previous one :
  - The result will always be correct as long as the range is not exceeded
  - Simple operation
  - Not as simple for sign and magnitude
  - Good for hardware implementation
    - **Win-win** : the same hardware can perform the addition of unsigned numbers

Overflow Detection rules

- Same algorithm as for the unsigned numbers
- If the result exceeds the range, **overflow** occurs
- **Overflow detection rules**
  - If the signes of the two numbers are the same but different from the sign of the sum, the overflow occured
  - Alternative formulation : if c_{in} into c_{out} out of the sign position are different, the overflow occured
  - Adding two numbers of different signs never produces an overflow

1.2.3 Binary multiplication

How

We use the same "*algorithm*" that the one we use by hand. For a binary representation :

$$\begin{aligned} X \cdot Y &= X \cdot \sum_{i=0}^{n-1} Y_i \cdot 2^i \\ &= \sum_{i=0}^{n-1} X \cdot Y_i \cdot 2^i \\ &= Y_{n-1} \cdot \underbrace{X \cdot 2^{n-1}}_{\text{Mulit Left-shifted by } n-1} + \cdots + Y_2 \overbrace{X \cdot 2^2}^{\text{Mult Left shifted by } 2} + Y_1 \cdot X \cdot 2^1 + Y_0 \cdot X \cdot 2^0 \end{aligned}$$

How many bits

Théorème 1 *Given a n-bits intger and a m-bits intgers, there product can at most require $n + m$ bits.*

We can see the multiplication as a sequence of m additions with an n-bit number.

Two's Complement multiplication

Recall of a value in two's complement (signed byte) :

$$x = -X_{n-1}2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i$$

- Inspired by the previous algorithm :

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

Let us not forget the sign-extend the partial result

For this only $n + m$ bits are kept ; any higher-order bits are discarded. (that the "*reason*" how $-5 \cdot -3 = 15$
I juste want to underline the difference between those two representation.

Sign-Magnitude and Two's Complement

Sign Magnitude

In the Sign magnitude representation with n bits, we use the **most significant bit** (MSB) to use it as a sign :

- 0 for positive numbers
- 1 for negative numbers

The remaining bits represent the absolute magnitude of the number :
To write 5 in a 4-bits number, we use $0101_2 = +5_{10}$
To write -5 in a 4-bits number, we use $1101_2 = -5_{10}$
We see here that it is very intuitive and mirrors human notation with a sign.

Two's Complement Representation

Here, there is two point of view the one introduce in the course is to see it as a clock, in a clockwise (le sens des aiguilles d'une montre) it is positive, and unclockwise (dans le sens contraire à celui d'une montre) it is negative and begin. The negative also start at -1 but the bit to represent -1 is 1111_2 which is just on the left.

The other way is to see it as the most significant bit (MSB) as negative, $1000_2 = -8$ and the rest of the bits being positive. To write -5 you have to write it as $-8 + 3 = -5$ which goes to $1011_2 = -5_{10}$
The pros for this notation is that there is only one representation for 0 where there is two for the other ($1000 = 0000 = 0$), The arithmetic operation are easier because we don't have to carry a sign everywhere and it is mor efficient in hardware implementation.

Figure 1.1 – Comparison table

Feature	Sign-Magnitude	Two's complement
-5_{10}	1101	1011
Zero representation	0000(+0) and 1000(−0)	0000
Range (4-bit)	$[-7, 7]$	$[-8, +7]$

Lundi 24 février 2025 — **Cours 3 : Fractional (Nointeger) Number**

1.3 Fractional number

1.3.1 Fixed-Point Representation

General Format

Définition 2 *Fixed-Point Numbers are :*

- *Integers*

$$I = -N, \dots, N$$

- *Rational numbers ("binary" rationals) of the form :*

$$x = \frac{a}{2^f}$$

where $a \in I$ and f positive integer

The fixed-point representation of a number x consists of integer x_{int} and fraction x_{fr} components represented by m and f digits, respectively :

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

Définition 3 *Digit-vector representation :*

$$X = (X_{m-1}X_{m-2}\dots X_1X_0 \underbrace{.}_{\textit{Radix point}} X_{-1}X_{-2}\dots X_{-f})$$

- *For **unsigned** numbers :*

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

- *For **signed** number in two's complement :*

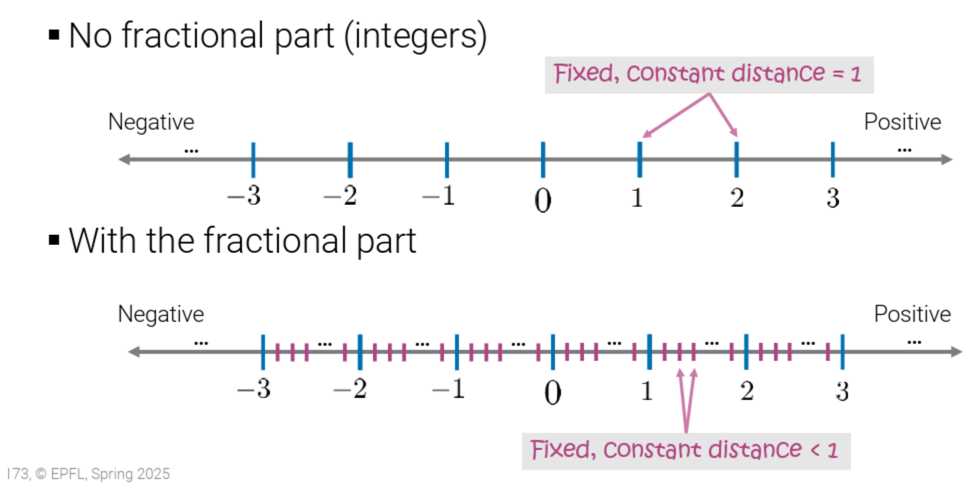
$$x = -X_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} X_i 2^i$$

1.3.2 Radix point

Separator between the integer and fractional parts

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

- The position of the radix-point is assumed to be fixed
  — Hence the name fixed-point
- If the radix point is not shown, it is assumed to be to the right of the least significant digit (i.e, no fractional part)
  — In that case, the number is an integer
- Also known as decimal point, binary point, etc...



Example

- Decimal number system and $m = 5, f = 5$
- Example decimal digit vector :
  — $X = (10077.01690)$
  — $x = 1 \cdot 10^4 + 7 \cdot 10^1 \cdots + 0,009$

- Most negative (min) :

$$x_{min} = -99999.99999 = -99999\frac{99999}{10^5}$$

- Largest number (max, positive) :

$$x_{max} = +99999.99999 = +99999\frac{99999}{10^5}$$

Fixed-point Representation

- Given an unsigned fixed-point binary format $m = 3, f = 4$
  — and an example binary digit vector :

$$X = (101.0111)$$

- Q : Find the equivalent decimal number :

$$X = (101.0111); x = 2^2 + 2^0 + 2^{-2} + 2^{-3} + 2^{-4} = 5.4375$$

Example With sign-and-magnitude and $m = 5, f = 3$, Example of a binary digit vector :

$$X = (10101.110);$$
$$x = -(4 + 1 + 0.5 + 0.25) = -5.75$$

Therefore, The most negative number can be

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

On the other hand, the largest number :

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

Two's complement With two's complement the work is the same as usual (the first digit is negative) :

$$X = (1010.1101);$$
$$x = -8 + 2 + 0.5 + 0.0625 = -5.1875$$

Here the most negative number is $x_{min} = 1000.0000_2 = -8$ and the largest one is $x_{max} = 0111.1111_2 = 7\frac{15}{16}$

1.4 Concepts of finite precision math

Precision

Définition 4 *The precision is the maximum number of non-zero bits*

For example if we have $X = (X_{m-1}X_{m-2}\dots X_1X_1.X_{-1}X_{-2}\dots X_{-f}$ then, the precisions is the sum of f and m :

$$\text{Precisions}(x) = m + f$$

Resolution

Définition 5 *The resolution is the smallest possible difference between two consecutive numbers*

For example if a number as $f = 5$ (5 digits for is fractional part) then we know that the smallest possible difference between the number is $\frac{1}{2^5} = \frac{1}{32}$, for integer ($f = 0$) the resolution is $\frac{1}{2^0} = 1$
However, in the general case :

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

Rang

Définition 6 *The range is the difference between the most positive and the most negative number representable.*

For example with **two's complement**
If we take, $m = 5, f = 3$, we compute $x_{max} = \sum_{i=-f}^{m-2} 2^i = 15\frac{7}{8}$, $x_{min} = -2^{m-1} = -16$. Then, the range is equal to $x_{max} - x_{min} = 31\frac{7}{8}$
In the general case, for fixed point and two's complement :

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

1.4.1 Accuracy

definition

Définition 7 *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 in the middle between two subsequent representable numbers (the real value lays between two equaly distant representation).
In the general case =

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

Dynamic Range

Définition 8 *The dynamic range is the* ***ratio*** *of, the maximum* ***absolute*** *value representable and the minimum positive value absolute (i.e nonzero) value representable.*

If we take the two's complement, with $m = 5, f = 3$ then the maximum absolute value is $- - 2^4 = 16$. For the minimum positive value we have

$2^{-3} = \frac{1}{8}$.
The dynamic range is said $= \frac{x_{max}}{x_{min}} = 128$ In the general case, for fixed-point and two's complement :

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

Personal remark You can see as the *size* of all the representable value divided by 2, 128. We have here 8 bits which means that we have 2^8 possible value which goes exactly to 256.

1.4.2 Floating-Point Number representation

Floating-Point (FP) Representation As with any other number representation in a digital system, FP representation is encoded in a finite number of bits. It represents only a **finite subset** of the **infinite set** of real numbers.
A real number that is **exactly** represented is called a **floating-point (FP) number**. All other real number either fall out of range (overflow or underflow) or are represented by FP numbers that approximate their value. The process of approximation is called **roundoff** and produces a **roundoff error**.

Significand, Exponent, Base FP representation consists of two components :

- the signed **significand** (also called **mantissa**) M^*
- the signed **exponent** E

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

where b is a constant called the **base**

Reminds us of the usual scientific notation, base 10 :

$$+35200 = \underbrace{3.52}_{\text{Coefficient}} \cdot 10^{+4} \quad -0.099 = -9.9 \cdot \overbrace{10^{-2}}^{\text{Exponent}}$$

Benefits of Floating-Point Consider 32 bit two's complement signed integers :

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

New, let's consider alors a 32 bit but floating-point number, with 24-significand in sign and magnitude and 8-bits exponent in two's complement.

$$\text{Dynamic Range}_2(x) = \frac{x_{max}}{x_{min}} = \frac{(2^{23}-1) \cdot 2^{2^{(8-1)}-1}}{2^0 \cdot 2^{-2^{8-1}}} = (2^{23}-1) \cdot 2^{255} \approx 5 \cdot 10^{83}$$

We can see here that the dynamic range increase a lot by a factor of $\approx 10^{74}$

Benefit We can also see the benefits the resolution which also reduces of for example when taking a 32-bits with 8 fractional bits (fixed-point) and on the other side, 24 bits significand in sign and magnitude and 8 bit exponent in two's complement. If we compute each resolutions :

$$\text{Resolution}_1(x) = 2^{-8} = 0.00390625$$
$$\text{Resolution}_2(x) = 2^0 \cdot 2^{-2^{(8-1)}} = 2^{-2^7} = 2^{-128}$$

If we compute the ratio :

$$\frac{\text{Resolution}_2(x)}{\text{Resolution}_1(x)} = \frac{2^{-128}}{2^{-8}} = 2^{-120} \approx 7.523 \cdot 10^{-37}$$

1.4.3 Significand : Sign-and-Magnitude

Floating-Point Representation

Today, the most used representation for significand is sign and magnitude 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

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

Digit vector

Many digit vectors are conceivable, but we focus on the following :

$$X = (\underbrace{SE_{m-1}}_{\text{Sign}E_{m-2}...E_1E_0M_{n-1}M_{n-2}...M_0}$$

Where E_i is the exponent and M_i is the magnitude.
There is $(n+1)$ bit significand in **sign and magnitude** and m bit exponent.

Redundant

In the most general case, the representation :

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

is redundant. Sign and magnitude is redundant, Multiple magnitude and exponent combinations can give the same number.

Example

If we take for example :

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

Floating-point representation is **redundant unless it is normalized**!
If we take a magnitude that is **normalized** :

$$1 \leq M < 2$$

Then :

$$1010.1000_2 = 1.0101_2 \times 2^3 = 10.5$$
$$-(0.00000011)_2 = -1.1_2 \times 2^{-7} = -0.01171875$$

Juste to be clearer, the normalized one here, is $1.0101_2 \times 2^3$ and $-1.1_2 \times 2^{-7}$.
For example let put 20_{10} normalized.
First, $20_{10} = 10100_2$, however $1 \leq M < 2$, which leads us to : $1.0100_2 \times 2^4$. The M being between 1 and 2 doesn't mean that the decimal number has a 1, ...

Hidden Bit and Fraction As the significand is normalized, the first digit of the magnitude is **always** binary 1. If something is always the same, it can be omitted (saving precious bits)
The first digit of the significand is omitted and called **hidden bit**.
The binary point is assumed to the right of the hidden bit. The represented part of the significand is called **fraction F**.

Example

$$\overbrace{101.001_2}^{\text{unormalized significand}} \times 2^{-4} = \underbrace{1.01001_2}_{\text{Normalized significand}} \times 2^{-2} = \overbrace{.01001_2}^{\text{hidden is not represented}} \times 2^{-}$$

Summary

- Common significand representation is the following :
  - Sign-and-magnitude
  - Normalized
  - One hidden bit
- Corresponding significand value becomes :

$$(-1)^S \times (1 + \sum_{i=1}^{n} M_{n-1} 2^{-1})$$

1.5 Exponent

Exponent Exponent needs to be signed

- **Positive** for representing very large numbers (**large absolute** value)
- **Negative** for representing very small numbers (**small absolute** value)

Biased representation Exponent can take any signed representation we know but there is one particular representation, called **biased**, which simplifies comparing two FP numbers in hardware.
Biased representation of a digit vector $X = (X_{n-1} \ldots X_1 X_0)$

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

Biased representation, Cntd.

Typically, the bias equals $B = 2^{n-1} - 1$

Where's the catch ?

- Resulting number are sorted just like unsigned integers but cover both the positive and negative numbers
- efficient hardware (superior to two's complement)
- Min exponent is represented as all zeros
  — FP zero can be represented as all zeros (significand and exponent)

Summary

Exponent

- Common representation of an -m bit exponent is biased with base $B = 2^{m-1} - 1$
- For the binary digit vector :

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

this biased exponent **value** becomes :

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

Floating point format

There could be many floating point formats, but we will often assume :

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

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

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

1.6 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**

Rounding modes

Various rounding modes exist

- Round to **nearest**, to **even** when **tie**
- Round towards **zero** (truncate)
- Round towards plus or towards minus **infinity**

Consider the real number x_{real} and the consecutive floating-point number F_1 and F_2 such that $F_1 \leq x_{real} \leq F_2$, we round it like always (normal definition)

1.6.1 IEE Standard 754

***FP* format in IEEE 754**

Exactly what we described

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

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

There is two types of formats : Basic and extended format :

Basic formats

- Sign S 1 bit
- Exponent E : 8 bits
- Fraction F : 23 bits

The default rounding mode is to the nearest, to even when there is a tie.

Double precision (64 bits)

- Sign S : 1 bit
- Exponent E : 11 bits
- Fraction F : 52 bits

How to convert fixed-point to IEEE 754

In order to do this, we need 5 steps :

1. Normalize the fixed point representation by shifting the binary point to the right of leftmost
2. The mantissa is the entire binary string after the binary point (the mantissa stores the fractional part after the leading 1 which is **implicit**.
3. Calculate the exponent bits by adding the bias of 127 to the power of 2
4. Convert exponent to unsigned binary representation
5. If the number is positive, the sign bit is zero, otherwise it is 1.

Let us see for example the number such as : $13_{10}.625$.
Firstly, we convert it to binary

$$13.625_{10} = 1101.101_2$$

Now we have to normalized the binary number :

$$1.101101 \cdot 2^3$$

So :

- Mantissa : 1.101101 (the leading 1 is **implicit** in $IEEE$ 754).
- Exponent : 3 (since we moved the decimal point 3 point left)

Then we compute the biased Exponent. In $IEEE$ 754, we use a bias of 127 :

$$E = 3 + 127 = 130$$
$$130_{10} = 10000010$$

We then drop the leading 1 from the 1.101101, fille the rest of the bit on the right :

$$1011010000000000000000000000$$

Which is the mantassima. this leads finally to :

$$1\,\overbrace{10000010}^{\text{Exponent}}\,\underbrace{10110100000000000000000000000}_{\text{Mantissa}}$$

Which in hexadecimal :

$$C16C0000$$

A cool video to understand how to convert any number to $IEEE$ 754 format :

`https://www.youtube.com/watch?v=RuKkePyo9zk`

Special Values

- Floating-point **zero** : $E = 0, F = 0$
  — The sign S differentiates between positive and negative zero, Value 1.0×2^{-B} is not represented.
- Positive and negative **infinity**
  — Biased exponents all ones, $F = 0$
- **NaN** (not a number)
  — To represent results of invalid operations(for example, the square root of a negative number)
  — Sign = 0 or 1 biased exponents all ones, $F \neq 0$

Exceptions : Handling of special situations

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

- **Overflow**, when the rounded value is **to large** to be represented
  — Result is set to infinity
- **Underflow**, when the rounded value is to small to be represented s to small to be represented
- Division by zero
- Inexact result, when the result is not an exact floating-point number
- invalid result, When NaN is produce by zero
- Inexact result, when the result is not an exact floating-point number
- invalid result, When NaN is produced

Vendredi 28 février 2025 — **Cours 4 : Arithmetic operation**

Difference between Fixed and Floating point representation In a fixed point representation, the " *distance*" between each point is fixed, on the other and When using Floating point representation, this distance isn't fixed it is **floating**. The reason for this is the definition of the floating point representation :

$$X = (SE_{m-1}E_{m-2}\dots E_1E_0M_{n-1}M_{n-2}\dots M_0)$$
$$x = (-1)^S \times M \times b^E$$

As you can see, the number can be much more precise as it goes near zero.

1.6.2 Arithmetic operations

Fixed-Point arithmetic Performing $+$ or $-$ on two binary numbers $x(m,f)$ and $y(m,f)$ is done **the same way** as if the operands were integers.

- Overflow can happen

Example (Slide 11) If I forgot to put the screenshot here is the example :

$$X = 000101.110_2 = 5.75_{10}$$
$$Y = 001100.011_2 = 12.375_{10}$$

And we want to sum up these two number,

$$\begin{array}{r} 00101.110 \\ +001100.011 \end{array}$$

We begin at the right, $0+1=1$, $X+Y$ =??????.??1 then $1+1=10$ there for we put a 0 and carry it over, $X+X=$??????.?01, then carry$+1+0=10$ same method at the next index so $X+Y$ =?????0.001 then we get the carry alone, … and we end up with $010010.001 = 18.125$

Personal remark It is the same way because we are always adding power of the 2 event when we are in the " *fractional world*" it is still power of two. We also do the same with decimal number in base 10.

Two's Complement For the two's complement the formula is :

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

The largest integer-part exponent : $\max(m_x-1, m_y-1)$ Consequently $m_{x\pm y} = \max(m_x, m_y) + 1$
The smallest fractional part exponent : $\min(-f_x, -f_y)$ Consequently $f_{x\pm y} =$

$\max(f_x, f_y)$
$m_{x\pm y}$ is the number of bits for the integer component that is needed (usual addition), same thing for the $f_{x\pm y}$

Fixed Point arithmetic Multiplication

Introduction For the multiplication on two binary numbers $x(m, f)$ and $y(m, f)$, we use the same algorithm as if the operands were integers but, the **binary point location changes**.
In two's complement :

$$x \cdot y = \left(-X_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} X_i 2^i\right) \cdot \left(-Y_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} Y_i 2^i\right)$$

The largest integer-part exponent $(m-1)+(m-1)$ Consequently $m_{xy} = 2m$
The smallest fractional-part exponent : $(-f)+(-f)$ Consequently $f_{xy} = 2f$

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

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

In two's complement :

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

- $m_{xy} = m_x + m_y$
- $f_{xy} = f_x + f_y$

Example : Analogy with Decimal numbers let us take for example

- 9.99 $m_x = 1, f_x = 2$
- 999.9999, $m_y = 3,\ f_x = 4$

If we take the multiplication :

$$9989.999001 \quad m_{xy} = 1+3 = 4; f_{xy} = 2+4 = 6$$

Example For example if we take two number with the format,

- $m_x = m_y = 3$
- $f_x = f_y = 2$

and $X = 010.11$,$Y = 011.01$. (screenshot slide 17)
To explain it in spoken English we do it as a loop of addition without the format (like it is integer) and then with the result, we convert it to fixed-point.

> We have to be careful here to not forget to change the format ($m_{xy} = m_x + m_y \dots$).

Pros and cons of fixed Point representation

Pros

- Arithmetic operations on integers can be applied to fixed-point numbers without modifications
  - Portable : we can reuse the same inetger processing hardware
  - Like with intgers, arithmetic operations are performed efficiently (fast)
  - Used in image and signal processing and communication

Cons

- Complex data and algorithm analysis
  - Where to put the binary point to maximize accuracy
- There are other number formats, namely floating-point, that provide more extensive dynamic range and better precision

Floating-Point Arithmetic

Addition/Subtraction

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 :

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

Steps

Four main steps to compute and produce the result $+/-$

- Add/substract significand and set exponent
  The significand 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 the added/subtracted to the other significand

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

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

- Normalize the result and update the exponent, if required
- Round the result, normalize, and adjust exponent, if required
- Set flags for special values, if required

Recap

- Recal Step 1 : Add/substract significand and set exponent
- Algorithm
  - Substract 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

1.6.3 Floating Point +/-

Normalization Various situations may occur

- Scenario 2 : When the effective operation is an **addition**, the significand might **overflow**. Steps to perform normalization :
  — Shift right the significand one position
  — Increment the exponent by one
- Example :

$$\begin{aligned} &1.1001111 \\ +\ &0.0110110 \\ =\ &10.0000101 \end{aligned}$$

Normalization

Shift right >> 1

Increment the exponent $E = E + 1$

Rounding The intermediate result may not be representable with the given format, in this case we perform a rounding.

- Towards zero : truncate the lsb
- Twords $\pm\infty$: requires addition
- To nearest : require addition

Tie to even The FP result is as close as possible to the exact value :

- Minimized rounfoff error (default rounding mode in $IEEE$ 754)
- Tie to even is preferred because it leads to smaller error when the result is divided by two -a frequent operation

Assuming as significand of infinite precision and radix r, round to the nearest can be obtained by **adding** ($\frac{r^{-f}}{2}$) to the infinite precision significandd and keeping the resulting f fractional digits

- In case of overflow : normalization and the exponent update are needed

Max round-off Error Rounding to nearest. f fractional digits. What is the maximumm difference between the exact value and its FP representation ?

$$d_{max} = \frac{2^{-f}}{2} \times 2^{E_{max}}$$

Lundi 3 mars 2025 — **Cours 5 : Low precision Compute**

1.6.4 Not in the course

Exponential Growh Ai is taking on an increasingly important role. Deep neural Networks are the most widespread.

- E.g, Large models (LLM) generate human-like content

The challenge with those LLM is their size, GPT3 has 175 **Billion** parameters. Larg models mean a lot of data, any computations and a fast result.

Challenges and limitations What is the pros and cons of formats :
32-bits or 64 bit floating-point formats :

- (-) Arithmetic unit are large (many bits $\implies$ high area, high energy
- (-) We can put fewer units per chip (e.g, less compute power in GPU)
  — Poor arithmetic density (in number of ops/1mm^2)
  — Fewer units, fewer computations
- (+) The model predictions are accurate, but it takes a long time to compute them

Fixed Point or integer format :

- (+) Arithmetic units are smaller and faster ($\sim 10\times$ area savings)
- (+) Better arithmetic density and lower delays
- (-) The error due to limited dynamic range are too significant for most ML models ; The accuracy of their predictions suffers

New number formats are needed : The best of both world

Low Precision Compute

> *Idea* The idea is to replace the 32 bits or 64 bit FP number traditionaly used for machine learning with reduced precision formats derived from the floating point representation

Build new, specialized hardware to accelerate ML training (**application domain specific** hardware)

Properties of low precision compute Advantages :

- Fit more numbers in memory (larger datasets, larger models)
- Move (read/write) more number per second
- Compute faster by using more arithmetic circuit in parallel
- Energy effiency

Disadvantages :

- Low precision
  — limits even more the set of number can represent
  — Accumulation of rounding error
- Less accurate neural network model predictions, but acceptable

Block Floating Point

Imagine a block (vector) of binary numbers in FP. Every vector element (every number) will have its own $S/M/E$. If the exponents in the block are not too different, we could use a single **shared exponent** per block :

- **Block-floating point**

To find which shared exponent to use in a BFP format, we need to find the largest exponent in the block of FP numbers.

> We use the largest because of the addition/substraction which works fine for every one of them if we take the largest

This exponent will be the shared exponent E_{block}.
Then we find the difference $d_i = E_{block} - E_i$ between the shared and each of the other exponents E_i in the block.
We then adjust the mantissa by shifting to the right the signed mantissa of each number by d_i.

> Because of these adjustements, mantissa in BFP cannot be normalized, therefore, there is no hidden bit either.

Block floating point is only the beginning

BFP strikes a balance between arithmetic density (fewer bits used, less silicon/chip area) and range
There are many other ideas to try,

- FP/BFP numbers with different exponent/mantissa sizes
- Fixed point numbers with nonstandars widths
- Industry and academia are coming up with new AI-targeted version of number formats.

Modern application (AI) demande innovation in computing

1.6.5 Point arithmetic

Fixed Point arithmetic Addition(substraction in two's complement

The largest integer-part exponent $\max(m_x - 1, m_y - 1)$ consequently : $m_{x\pm y} = max(m_x, m_y) + 1$
The smallest fractional part exponent : $\min(-f_x, -f_y)$ consequently, $f_{x\pm y} = \max(f_x, f_y)$

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

Multiplication (Fixed Point)

$$x \cdot y = \left(-X_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} X_i 2^i\right) \cdot \left(-Y_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} Y_i 2^i\right)$$

Jeudi 6 mars 2025 — **Cours 6 : Introduction to logic circuits**

1.7 Digital circuit

Introduction Logic circuits is the foundations of digital systems. In smartphones, computers, control systems, digital communication devices, ...
The smallest unit of digital information is one bit, represented as a binary value **0 and 1**.
In a binary logic circuit, the electrical signals are constrained to two discrete values.
The key to binary circuits dominance is **simplicity**. In practice, the two discrete values are implemented as voltage levels (the supply voltage or the ground).

Two states of a switch If controlled by an **input variable** x, the switch is open if $x = 0$ and closed if $x = 1$.

Symbol The symbol for a switch controlled by an input variable :

Analyses of a logic Network Example logic network
The sequence of input value in the truth table visualized in the network. Any sequence can be visualized in a **timing diagram**.

Cost of logic circuit The total cost of a logic circuit is typically defined as the total **number if gates** **plus** the total **number of gates input**

- Each logic gate (AND, OR, NOT, etc) contributes to the cost
- More inputs to gates often mean larger, more costly gates
- in simplified cost models, weights may be assigned to different types of gates, depending on their complexity or physical implementation.

Functionally Equivalent Networks A logic function can be implemented with a variety of different logic networks of different cost :

$$f(x) = \overline{x_1} + x_1x_2 = \overline{x_1} + x_2 = g(x)$$

The above two networks are functionally **equivalent**

How to check for Equivalence

$$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n), \forall x_1, x_n$$

Two logic networks are equivalent if :

- Their **truth tables** are the same
- There exists a sequence of algebraic manipulation to transform one logic expression to the other (these algebraic manipulations are defined as **Boolean algebra**
- Their **Venn diagrams** are the same

How to find the best equivalent network Logic function can be implement with a variety of different networks. How does one find the best (simplest, least costly)

The process of finding the best equivalent logical expression describing a logic network is called **minimization**

- **Approach 1** : Apply a sequence of algebraic transformation
  — Now always abvious when to apply which transformation, tedious, impractical
- **Approach 2** : Use **Karnaugh maps** (an alternative to the truth table)
  — Simpler, but quickly becomes unmanageable by hand (up to 4 inputs acceptable)
- **Approach 3** (the winner) Automated techniques in synthesis software tools

1.7.1 Boolean algebra

A bit of history In 1849 George Boole published a scheme for the algebraic description processes involved in logical thought and reasoning. This scheme and its refinements became know as Boolean algebra
It the late 1930s, Claude Shannon showed hat Boolean algebra provides an effective means of describing circuits built with switches, therefore, Boolean algebra can be used to describe logic circuits. Boolean algebra is a powerful technique for designing and analyzing logic circuits ; it is the foundation for our modern digital technology,

Axioms Like any algebra, Boolean algebra is based on a set of rules derived from a small number of basic assumptions (i.e., **axioms**). Let us assume that boolean algebra involves the following axioms are true :

1. $0 \cdot 0 = 0$
   $1 + 1 = 1$
2. $1 \cdot 1 = 1$
   $0 + 0 = 0$
3. $0 \cdot 1 = 0 \cdot 1 = 0$
   $1 + 0 = 0 + 1 = 1$
4. $\text{if } x = 0, \text{ then } \overline{x} = 1$
   $\text{if } x = 1, \text{ then } \overline{x} = 0$

From the axioms, we can define some rule (i.e., **theorems**) for dealing with single boolean variables

Single variable theorems If x is a variable, then the following theorems hold :

1. $x \cdot 0 = 0$
   $x + 1 = 1$
2. $x \cdot 1 = x$
   $x + 0 = x$
3. $x \cdot x = x$
   $x + x = x$
4. $x \cdot \overline{x} = 0$
   $x + \overline{x} = 1$
5. $\overline{\overline{x}} = x$

Theorems grouped in pairs, emphasizing the **principle of duality**.
Dual Form is obtained by replacing all + operators with · operators, and vice versa ; and by replacing all 0s with 1s, and vice versa.
To prove the theorems, apply **perfect induction** (i.e., substitute the variable with 1 or 0) and use the axioms.

Two and three variable propreties

Given three Boolean variables, the following properties hold :

- Commutative $x \cdot y = y \cdot x$
  $x + y = y + x$
- Associative : $x \cdot (y \cdot z) = (xc.y) \cdot z$
  $x + (y + z) = (x + y) + z$
- Distributive : $x \cdot (y + z) = x \cdot y + x \cdot z$
  $x + y \cdot z = (x + y) \cdot (x + z)$

Example

Let us prove the validity of the following logic equation :

$$(x_1 + x_3)(\overline{x_1} + \overline{x_3}) = x_1\overline{x_3} + \overline{x_1}x_3$$

Let us manipulate the left hand side :

$$\begin{aligned}(x_1 + x_3)(\overline{x_1} + \overline{x_3}) &= (x_1 + x_3)\overline{x_1} + (x_1 + x_3)\overline{x_3} \\ &= x_1\overline{x_1} + x_3\overline{x_1} + x_1\overline{x_3} + x_3\overline{x_3} \\ &= 0 + x_3\overline{x_1} + x_1\overline{x_3} + 0 \\ &= x_1\overline{x_3} + \overline{x_1}x_3\end{aligned}$$

Purpose

The purpose of the axioms, theorems, and properties in Boolean Algebra is to perform algebraic transformation to do :

- **Check for equivalence**, Find if two logical expressions (i.e., logical circuits made of gates) are equivalent (i.e., perform the same functionality) without evaluating all input possibilities
- **Design efficient circuits** Simplify the logical expression to find a potentially more efficient equivalent variant (i.e., design a circuit of the same desires functionality but with fewer gates)

Two and three variable properties :

Given three boolean variable, the following properties hold :

- Absorption : $x + x \cdot y = x$
  $x \cdot (x + y) = x$
- Combining $x \cdot y + x \cdot \overline{y} = x$
  $(x + y) \cdot (x + \overline{y}) = x$
- DeMorgan's theorem : $\overline{x \cdot y} = \overline{x} + \overline{y}$
  $\overline{x + y} = \overline{x} \cdot \overline{y}$
- Redundancy : $x + \overline{x} \cdot y = x + y$
  $x \cdot (\overline{x} + y) = x \cdot y$
- Consensus : $x \cdot y + yc.z + \overline{x} \cdot z = x \cdot y = \overline{x} \cdot z$
  $(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$

Proof For example let us prrof the validity of the following logic equation :

$$x_1\overline{x_3} + \overline{x_2 x_3} + x_1x_3 + \overline{x_2}x_3 = \overline{x_1 x_2} + x_1x_2 + x_1\overline{x_2}$$

Use the left hand side for the manipulation :

$$\begin{aligned} x_1\overline{x_3} + \overline{x_2}\overline{x_3} + x_1x_3 + x_1x_2 + \overline{x_2}x_3 &= x_1\overline{x_3} + x_1x_3 + \overline{x_2}\overline{x_3} + \overline{x_2}x_3 \\ &= x_1(\overline{x_3} + x_3) + \overline{x_2}(\overline{x_3} + x_3) \\ &= x_1 \cdot 1 + \overline{x_2} \cdot 1 \\ &= x_1 + \overline{x_2} \end{aligned}$$

1.7.2 The Venn Diagram

Introduction Venn Diagram provides a graphical illustration of various operations and relations in the algebra of sets. Popularized by John Venn (1834-1923) in the 1880s.

Shades and Contours In the diagram, the elements of a set are represented by the area enclosed by a **contour of a circle**.

- Shaded area where the **logical function** value = binary 1
- The area within the contour : **variable** value = binary1
- The area outside the contour **variable** value = binary 0

Simple intersection Reminder : The union of the shaded areas corresponds to the logical expression (shaded when the expression is binary 1)

Lundi 10 mars 2025 — **Cours 7 : Logic Synthesis**

1.7.3 Logic synthesis

Minterms For a function $f = (x_1, x_2, \ldots, x_n)$ of n variables, a **product term** in which **each** of the n variables appears **once** is called a **minterm**.
Minterms are typically labeled as m_i, where $i \geq 0$ is an integer. An $n-$variable minterm m_i can be represented by an n-bit integer.

- Variable appears **complemented** if the corresponding bit in the binary representation of m_i is 0
- Otherwise, it appears **uncomplemented** (original)

Example Let us take $n = 3, i = 5$: three variables. $5 = 101_2$ and therefore, $m_5 = x_1\overline{x_2}x_3$
If we take now $n = 5, i = 3$: five variables. $3 = (00011)_2$ and therefore, $m_3 = \overline{x_1}\overline{x_2}\overline{x_3}x_4x_5$

Maxterms For a function $f = (x_1, x_2, \ldots, x_n)$ of n variables, a **sum term** in which **each** of the n variables appears **once** is called a **maxterm**. Maxterm are typically labeled as M_i, where $i \geq 0$ is an integer. An $n-$variable maxterm M_i can be represented by an $n-$bit intgerer.

- Variable appears **complemented** if the corresponding **bit** in the binary representation of M_i is 1.
- Othewise, it appears **uncomplemented** (original)

Example if we take the same as above, $n = 3, i = 5$ with $5 = 101_2$ we get :

$$M_5 = \overline{x_1} + x_2 + \overline{x_3}$$

And as we take the second way, $n = 5, i = 3$ we get :

$$x_1 + x_2 + x_3 + \overline{x_4} + \overline{x_5}$$

What we are doing here is the same thing as seen in AICC I, we use it the same way as CNF and DNF where one is with negation on the 1 and the *OR* between each variable and the other with *AND* everywhere but the opposite.
This is equivalent because of the Morgan's law.

Logic synthesis with Minterm/-Maxterms For a function f specified in the form of a truth table, a logic expression realizing the function can be obtained by considering :

- Only the rows in the table for which $f = 1$ or
- Only the rows in the table for which $f = 0$

If we are considering the rows where $\boldsymbol{f = 1}$, f is represented by the **sum of the minterms** corresponding to the rows where $f = 1$.
If we are considering the rows where $\boldsymbol{f = 0}$, f is decribed by **the product of the maxterms** corresponding to the rows where $f = 0$

Sum of products (SoP) form When we are considering the rows where $f = 1$, f is represented by the sum of the corresponding minterms. The resulting logical expression is correct but **not** necessarily the lowest cost (optimal) implementation of f.
Any logical expression consisting of product (AND) terms that are summed (OR) is said to be in the **sum-of-products (SoP)** form.

Définition 9 *We called the* ***canonical sum of products*** *where all the product are a minterm*

Example SoP Consider a function f of $n = 3$ variables and the truth table below :

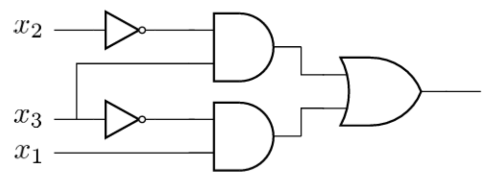
x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Then, the canonical SoP form :

$$\begin{aligned} f(x_1, x_2, x_3) &= \sum(m_1, m_4, m_5, m_6) \\ &= \sum m(1, 4, 5, 6) \end{aligned}$$

$$
\begin{aligned}
f(x_1,x_2,x_3) &= \overline{x_1}\,\overline{x_2}x_3 + x_1\overline{x_2}\,\overline{x_3} + x_1\overline{x_2}x_3 + x_1x_2\overline{x_3} \\
&= \overline{x_2}x_3 + x_1\overline{x_3}
\end{aligned}
$$

Which get us to :



A good indication of the **cost** of a logic circuit is the total number of **gates** and the **input** to the gates in the circuit.

- For the design above, the cost $= 5+1+1+2+2+2 = 13$ where 5 is the total gates, the one's are the NOT, 2's are the AND and OR.

Example PoS Now we consider with product instead of a sum. Consider a function f of $n = 3$ variables and the truth table below :

$$
\begin{aligned}
f(x_1,x_2,x_3) &= \prod(M_0, M_2, M_3, M_7) \\
&= \prod M(0,2,3,7)
\end{aligned}
$$

$$
\begin{aligned}
f(x_1,x_2,x_3) &= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\
&= (x_1+x_2+x_3)(x_1+\overline{x_2}+x_3)(x_1+\overline{x_2}+\overline{x_3})(\overline{x_1}+\overline{x_2}+\overline{x_3})
\end{aligned}
$$

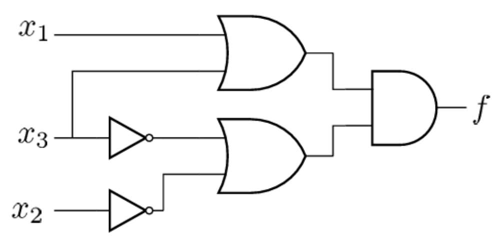
And now using the Morgan's theorem :

$$f = \overbrace{f}^{=} = \overline{m_0 + m_2 + m_3 + m - 7}$$

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Which as you can see is the same as the one before, but now we only take the line with 0 as a result.
After some trick, we finally get the result :

$$f(x_1,x_2,x_3) = (x_1+x_3)(\overline{x_2}+\overline{x_3})$$

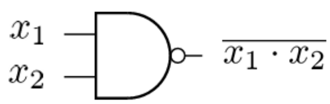


Where the cost = $5 + 1 + 1 + 2 + 2 + 2 = 13$

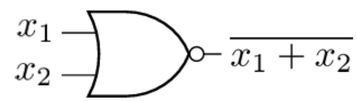
1.7.4 NANS and NOR logic Networks

NAND and NOR gates

NAND and NOR gates can be used to build logic circuits :

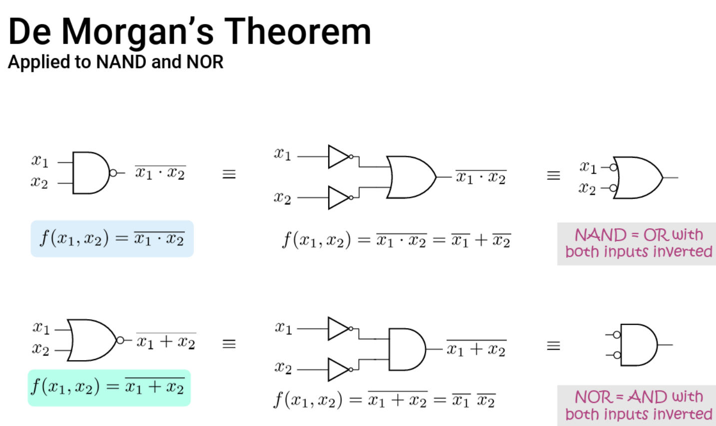


$$f(x_1, x_2) = \overline{x_1 \cdot x_2}$$



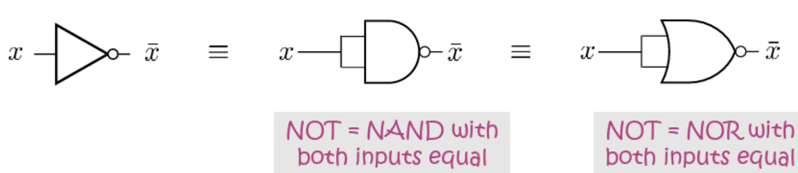
$$f(x_1, x_2) = \overline{x_1 + x_2}$$

NAND/NOR physical implementation is simpler (requires fewer transistor) and more efficient than AND/OR. In fact the AND and OR logic gates are implemented as NAND/NOR + not. How to de we that ?:



NOT gate using NAND or NOR

According to Boolean theorems, $\overline{x} = \overline{x \cdot x}$ and $\overline{x} = \overline{x + x}$ which are the $NAND$ and the NOR :



How to implement a function

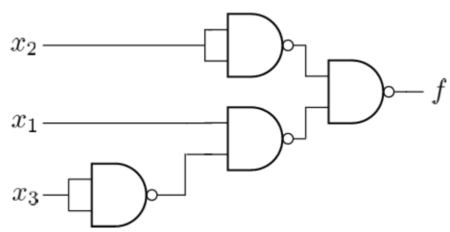
Now we try to implement the function f in the SoP form with $NAND$

$$f = x_2 + x_1\overline{x_3}$$

Algorithm : we start by applying double inversion and, then, the Morgan's theorem to simplify the expression.
We have :

$$\begin{aligned} f &= x_2 + x_1\overline{x_3} \\ &= \overline{\overline{x_2 + x_1\overline{x_3}}} \\ &= \overline{\overline{x_2} \cdot \overline{x_1\overline{x_3}}} \end{aligned}$$

Which gives us :



1.7.5 Incomplety definded Functions

Incompletely defined function are Boolean functions where some input combinations are not specified because they dont matter (e.g., they never occur), so the function does not need to define outputs for them

- Those input combinations are called **don't care conditions**

In logic optimization, don't care conditions can be assigned function value (output) either 0 or 1, to simplify the logic circuit

Example Imagine a lion's cage with an automated door control system including two sensors and a manual override switch. **Input** :

- **Sensor L** detects if the lion is inside (1 = inside, 0 = outside=
- **Sensor T** detects if the trainer is inside (1 = inside ; 0 =outside)
- **Override switch (S)** : The trainer can manually force the door open or closed irrespective of presence (1 = override enabled ; 0 = normal mode)

Outputs

- **Door control** 1 = open, 0 = closed.

In this case with si that when $S = 1$ then L and T doesn't matter, because the door will be open in any case.
What we are doing here is this :

$$D = \overline{L}T\overline{S} + LT\overline{S} + \overline{LT}S = T\overline{S} + \overline{LT}S$$

Which have a cost of $3 + 7 = 10$
However the result is the same by switching to

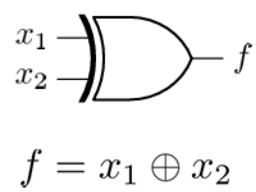
$$D = T + S$$

In spoken english this is saying, the door can only be open if the switch is on or the trainer is inside and if neither of those two are true, then the door is closed. Here the cost is $1 + 2 = 3$

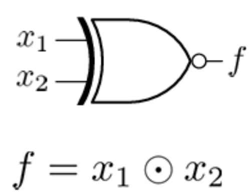
Even and Odd detectors

Given the function $f = \overline{x_1}x_2 + x_1\overline{x_2}$ which we called **Exclusive OR** or XOR writted as $\oplus$:

$$f = x_1 \oplus x_2$$



On the other side for the $XNOR$ which is defined as $f = \overline{x_1}\overline{x_2} + x_1x_2$



Number display

I skipped the previous slides (number display) because I am late but the goal was to write on a digital clock (with the 8 which as 7 lines that can be on or off) a value $(s_1, s_0)_2$ as a decimal number. To do so you have to do first a truth table depending of which line is on depending of the values, and the create a logic function from this truth table.

Data selector

It is often helpful to choose **precisely one** from several inputs. A circuit performing data selection (a **multiplixer**) has one or more **select** inputs dedicated to determining which of the remaining inputs to pass to the output. For example, a three input multiplixer (also called 2 to 2 MUX) :
Inputs

- One **selection** signal s
- Two data **input** x_1 and x_2

When the selection signal is $s = 0$ the output becomes $f = x_1$ otherwise, the output becomes $f = x_2$
To write this as a logical function :

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x_2} + \overline{s}x_1x_2 + s\overline{x_1}x_2 + sx_1x_2$$
$$\overline{s}x_1(\overline{x_2} + x_2) + s(\overline{x_1} + x_1)x_2$$
$$= \overline{s}x_1 + sx_2$$

Remark If there are n data inputs to select from, how many select signals MUX requires ?:

$$\lceil \log_2 n \rceil$$

Because if we have two data, this give only one combination, 4 data two select signal, 2^2, with eight data inputs, three select signals (2^3 combinations) and because we cannot take lower bound for data input that are not power of 2, we have to take the ceiling.