

Introduction à l'architecture des ordinateurs

Présentation du cours

- Semaines 36 à 41
- 4 créneaux par semaine : 1 CM, 1 TD, 2TP
- Évaluation : devoir surveillé sur table en semaine 41 + note de td
- Équipe pédagogique :
 - CM, TD et TP groupe 1 : Sébastien Faucou
 - TD et TP groupe 2 : Jean-François Remm
 - TD et TP groupe 3 : Théo Serru
 - TD et TP groupe 4 : Antoine Bernabeu

Questions, remarques : **sebastien.faucouATuniv-nantes.fr**

Bureau E1/19.

Jours de présence à l'IUT variables → envoyer un mail pour prendre RDV.

Objectifs pédagogiques :

- Se faire une idée générale du fonctionnement d'un ordinateur
- Comprendre comment l'information est codée et manipulée
- Appréhender les limites de ce que peut faire un ordinateur

Supports de cours, sujets de TD et de TP sont en ligne :

<https://gitlab.univ-nantes.fr/faucou-s/but1info-r103>

- Le support de cours sera mis à jour au fur et à mesure des séances
- Le support de cours est nécessaire mais pas suffisant → il peut être utile de prendre des notes
- Les sujets de TD seront fournis en version papier → m'indiquer les besoins spécifiques (par exemple taille et/ou type de police)

N'hésitez pas à poser des questions, y compris en cours.

Faites attention cependant à ne pas interrompre un·e camarade.

Ordinateur?

Machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques. (Larousse en ligne).

[...] système de traitement de l'information programmable tel que défini par Alan Turing et qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques. (Wikipedia)

Est-ce un ordinateur?

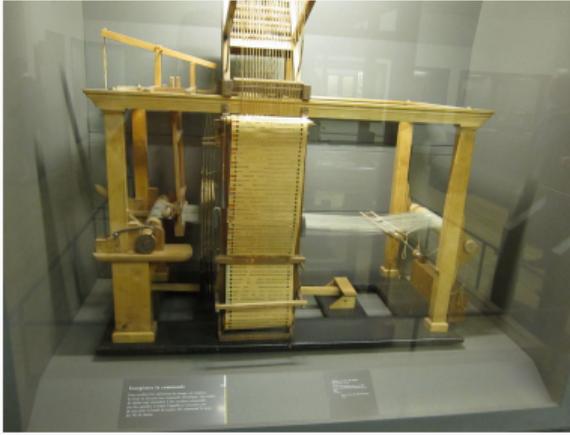


Machine à calculer, Blaise Pascal (1642)



© CNAM, Paris

Les métiers à tisser programmables (1725 et plus)



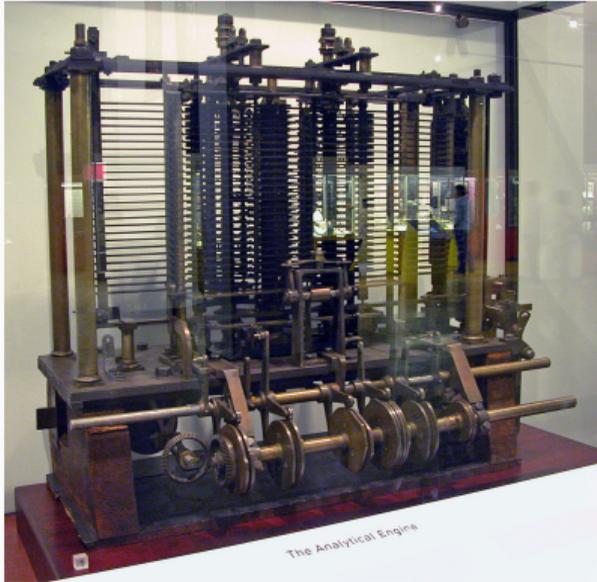
Basile Bouchon (1725)

Joseph Marie Jacquard (1801)



La première machine à calculer programmable (183X)

La **machine analytique** imaginée par Charles Babbage : mécanique, à vapeur, jamais achevée

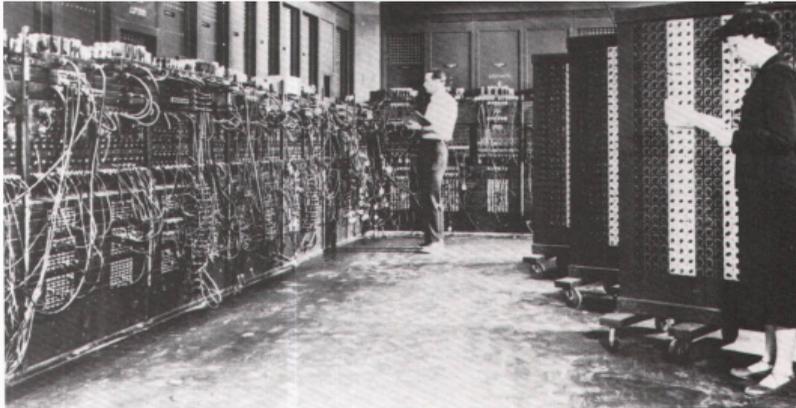


Prototype inachevé (1871)



Ada Lovelace, programmeuse

Le premier ordinateur : l'ENIAC (1946)



L'ENIAC c'est :

- 167 m²
- 100 000 additions/s
- Une mémoire pour 20 nombres décimaux à 10 chiffres
- L'information est manipulée via des « interrupteurs » pilotables en tension (tubes à vide)
- Unité centrale avec une architecture von Neumann

On distingue :

- L'architecture externe, ou **jeu d'instructions** (*instruction set architecture*), qui décrit les opérations que sait faire l'ordinateur. Elle est souvent décrite par la spécification du langage machine.
- L'architecture interne, ou **micro-architecture**, qui décrit l'organisation des composants internes de l'ordinateur (CPU, mémoires). Il s'agit d'un ensemble de **circuits électroniques**, qui :
 - implémentent les opérations du langage machine,
 - assurent le contrôle de l'exécution des programmes.

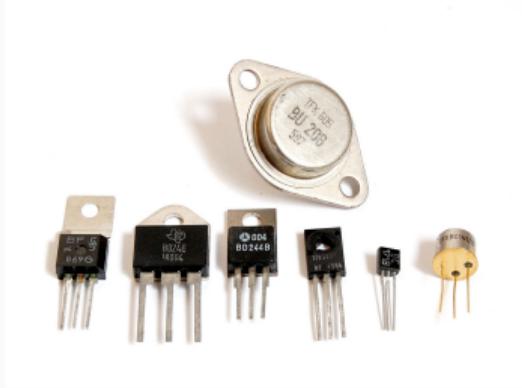
L'élément de base qui permet la réalisation des circuits est le **transistor**.

Du transistor aux portes logiques

Miniaturisation



Lampes (~ 1910)
ou tubes à vide,
tubes électroniques



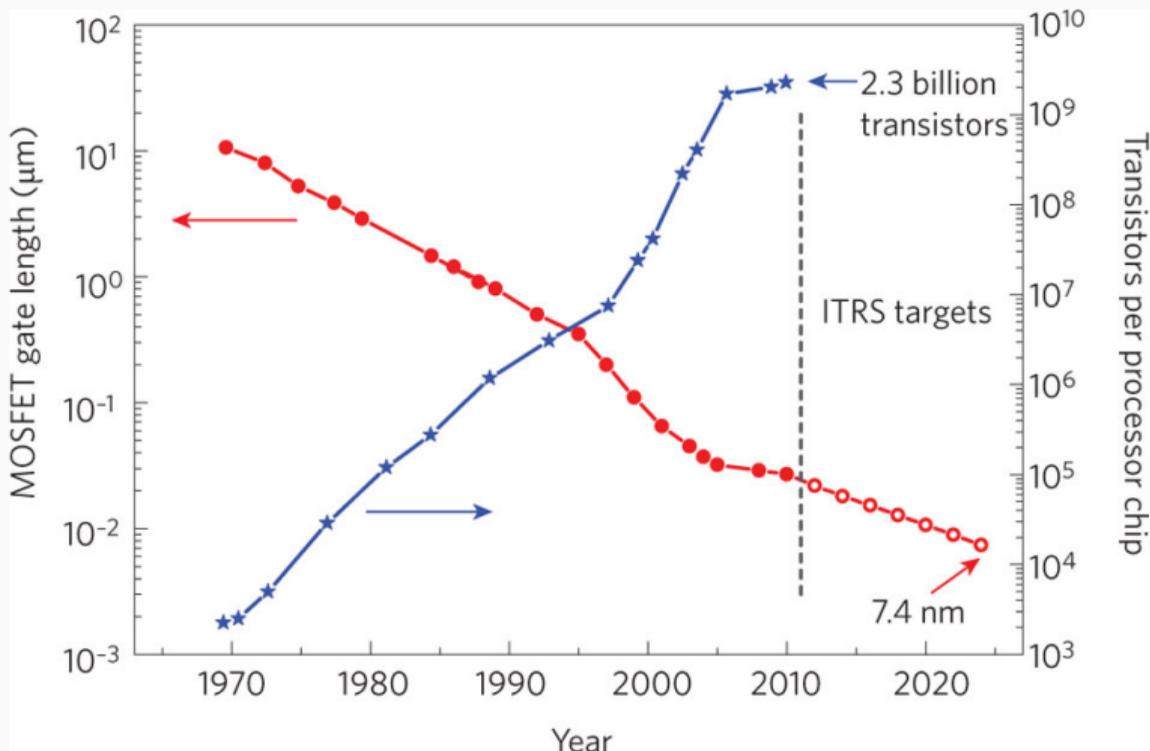
Transistors (1947)
J. Bardeen, W. Shockley, W.
Brattain (Nobel de physique
1956)



Circuits intégrés (1958)
J. Kilby (Nobel de physique
2000)

Progrès technologique

Source : Nature Nanotechnology



Notion de signal numérique

Un transistor peut être vu comme un interrupteur qu'on ouvre ou ferme en appliquant une commande sous forme d'une tension électrique.

Ainsi, grâce aux transistors, l'information est manipulée sous la forme d'un signal électrique :

- au-dessus d'une certaine tension, le signal « vaut » 1 (associé à la valeur logique vrai)
- en-dessous d'une certaine tension, le signal « vaut » 0 (associé à la valeur logique faux)
- une bande de garde sépare les deux zones : quand la tension est dans cette bande, la valeur du signal est indéterminée.

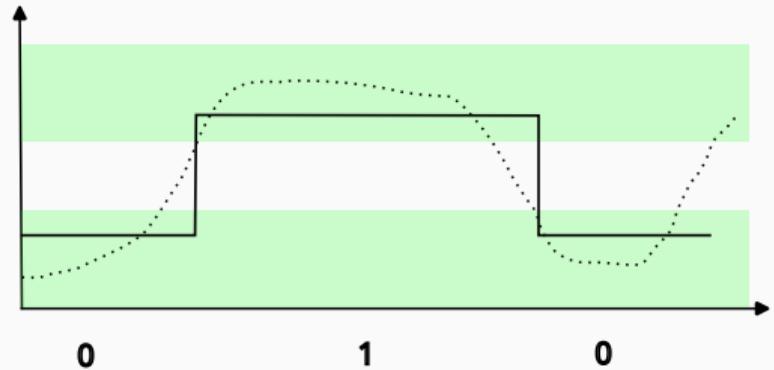
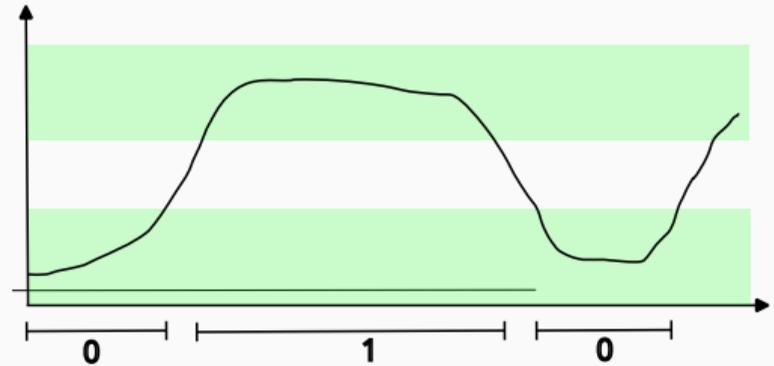
Signal physique vs. signal logique

Un signal physique subit :

- des délais de propagation.
- des délais de commutation.

Les circuits sont conçus pour cacher ces effets.

Au niveau programmation, on considère un **signal logique**.



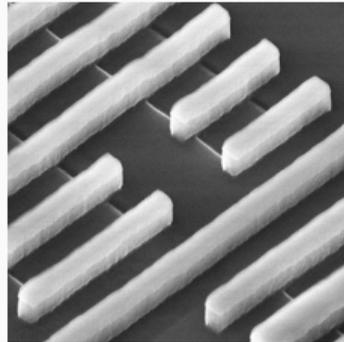
Technologie CMOS

CMOS (Complementary Metal-Oxide-Semiconductor)

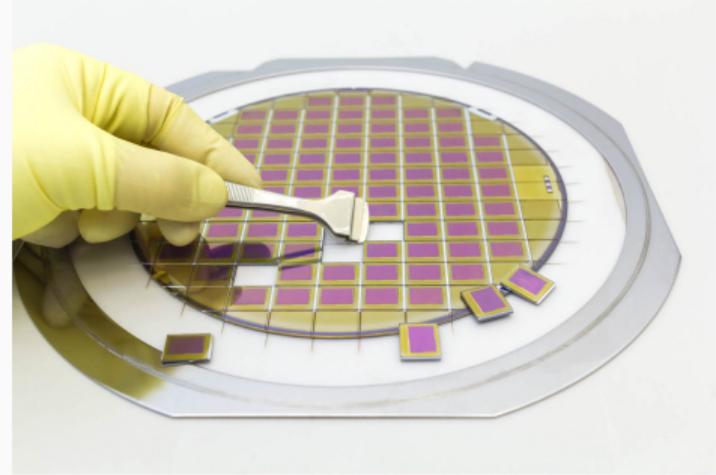
Procédé de fabrication des transistors à effet de champ (*MOSFET* pour Metal-Oxide-Semiconductor Field Effect Transistor) née à la fin des années 60.

C'est une technologie planaire (les transistors sont implantés à la surface du silicium).

C'est toujours la technologie la plus utilisée aujourd'hui, que ce soit pour concevoir des processeurs ou d'autres types de composants électroniques



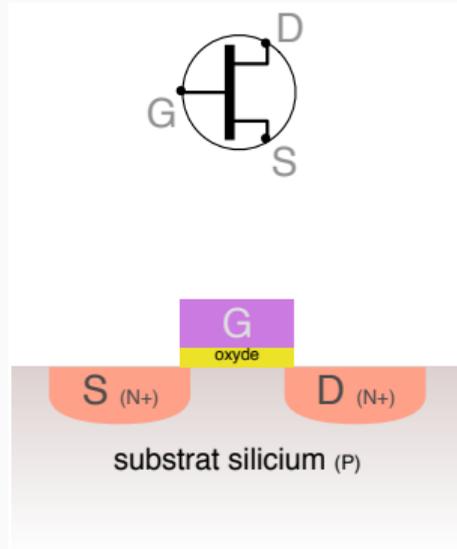
Du sable au circuit intégré



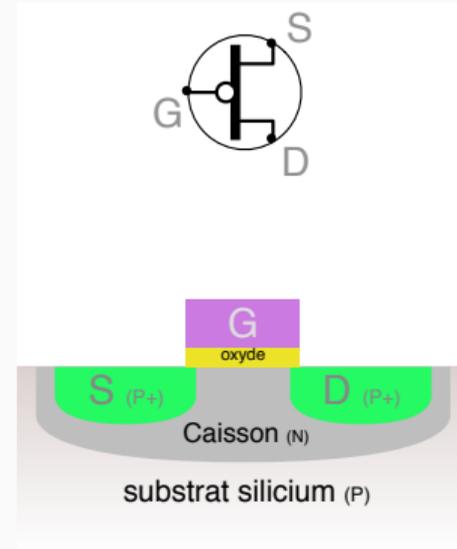
Ici une vidéo décrivant les principes du procédé de fabrication (attention, la fin de la vidéo est à objectif commercial)

Transistors MOS

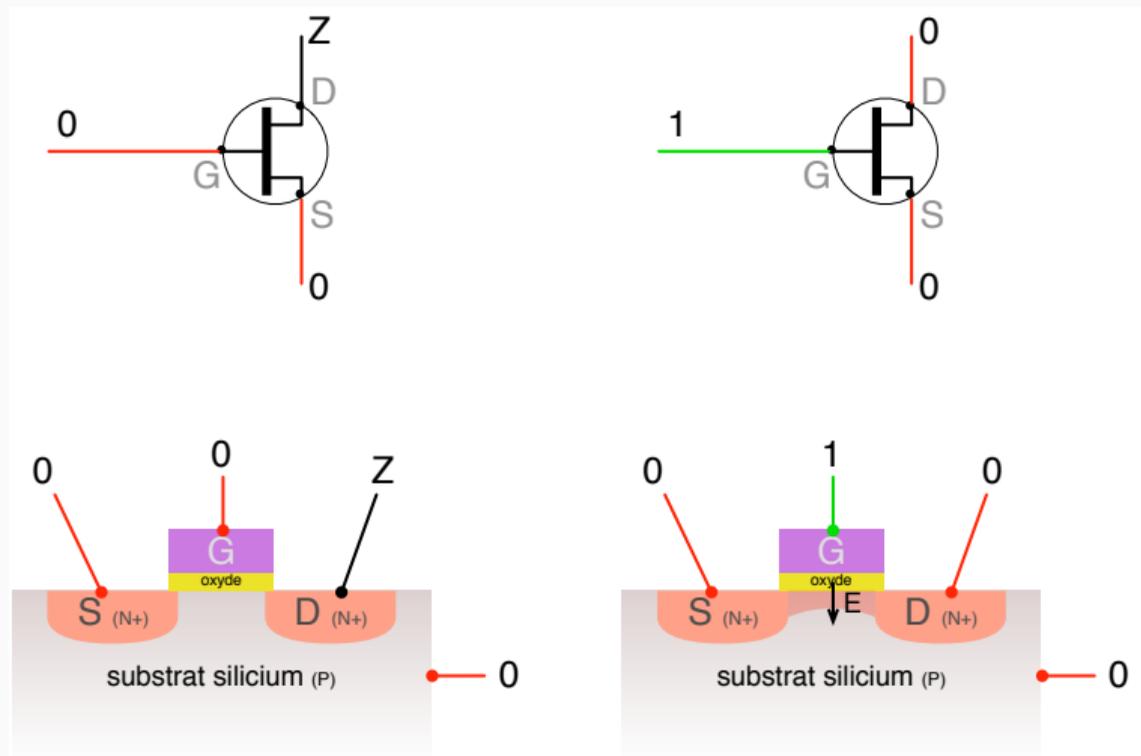
Transistor NMOS



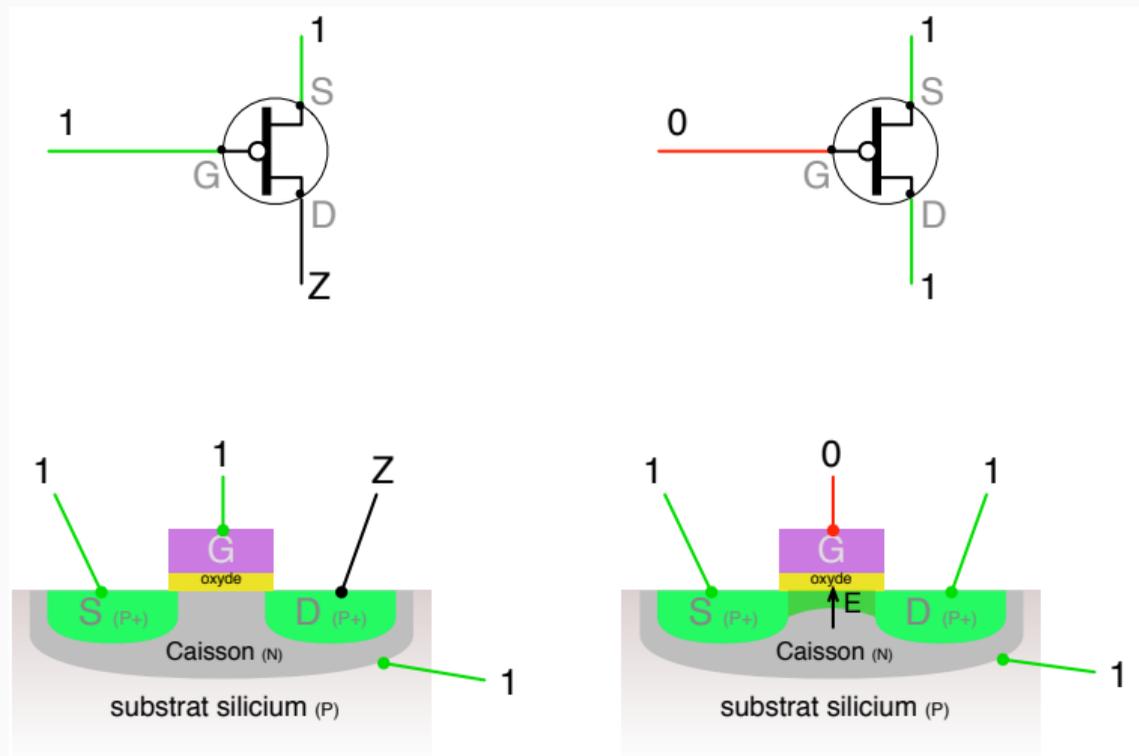
Transistor PMOS



Fonctionnement du transistors NMOS



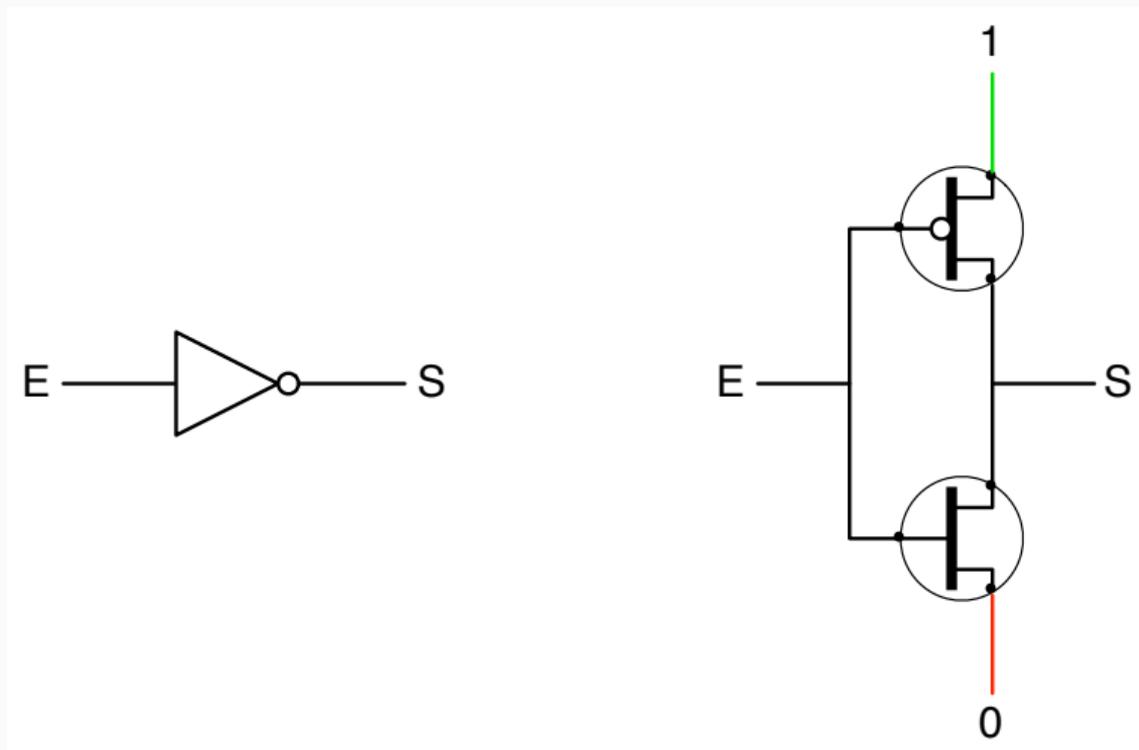
Fonctionnement du transistors PMOS



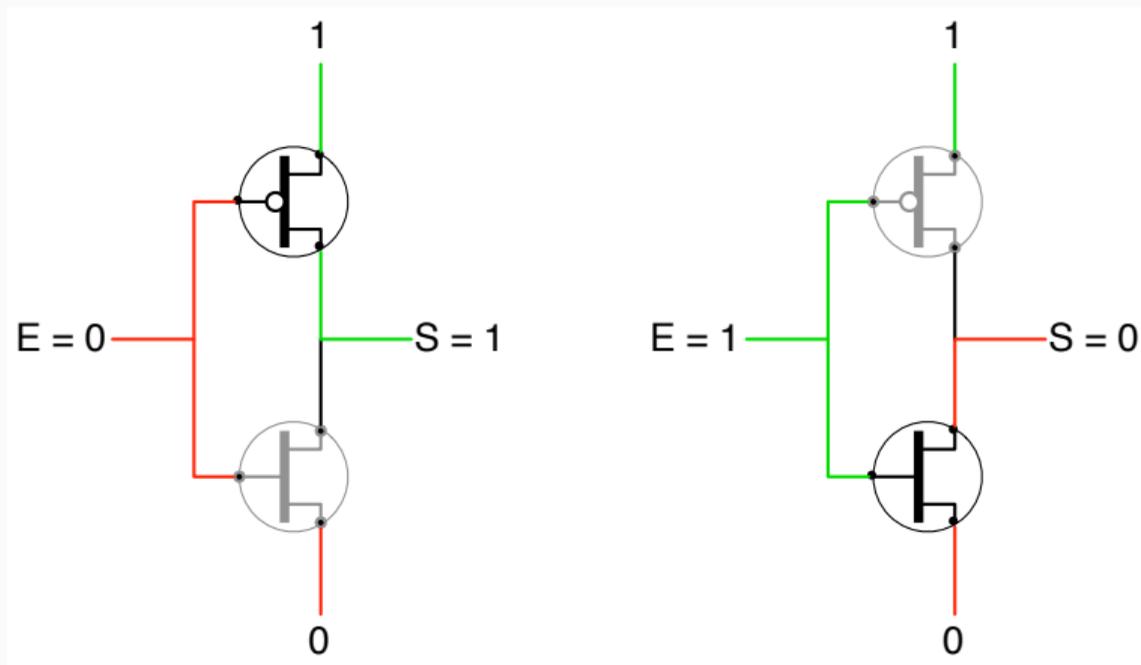
Si les transistors sont le composant de base des circuits, on préfère manipuler des composants de plus haut niveau pour construire les circuits : **les portes logiques**.

Une porte logique est constituée d'un ensemble de transistors connectés, qui permet de calculer un signal de sortie qui est une **fonction logique** des entrées : non, et, ou, non-et, non-ou, ou-exclusif, non-ou-exclusif

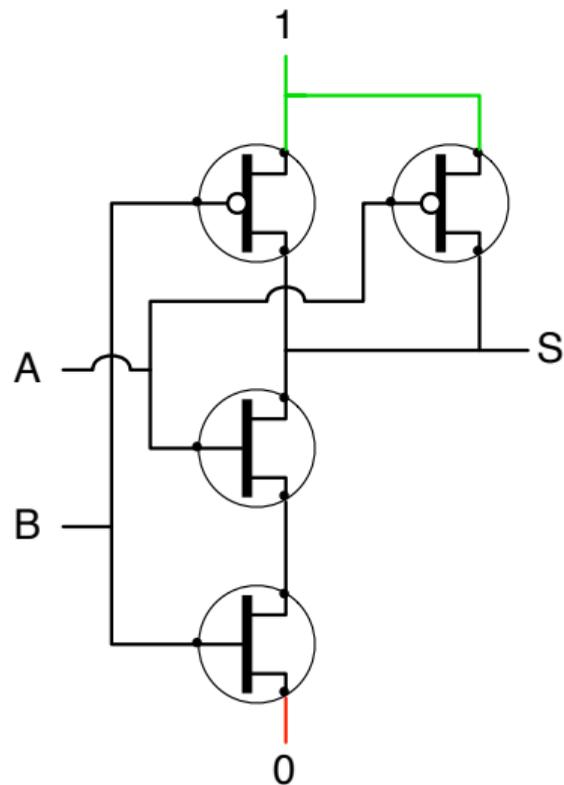
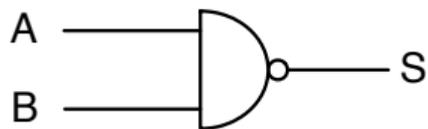
Inverseur CMOS (1)



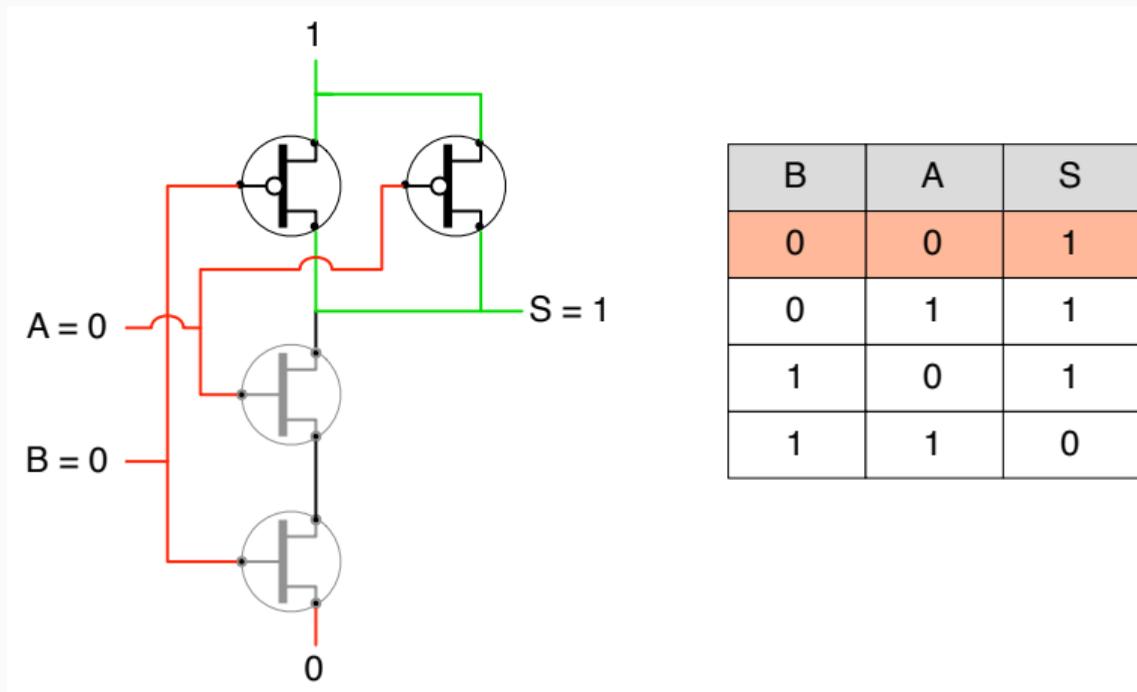
Inverseur CMOS (2)



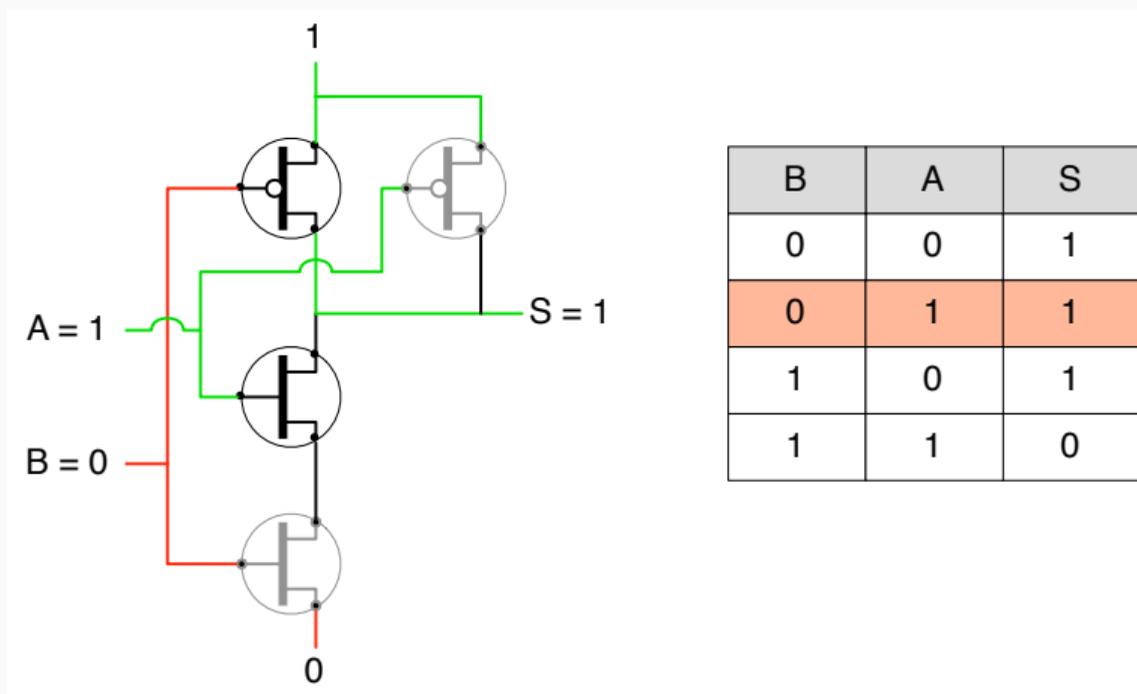
Porte NAND 2 entrées (1)



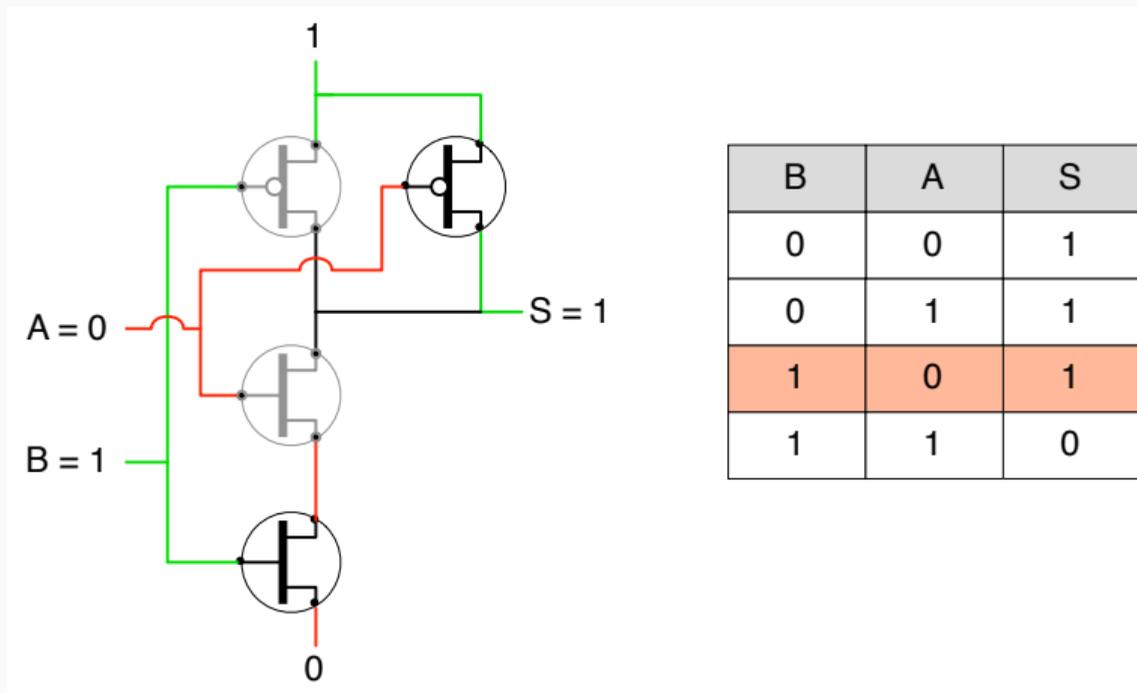
Porte NAND 2 entrées (2)



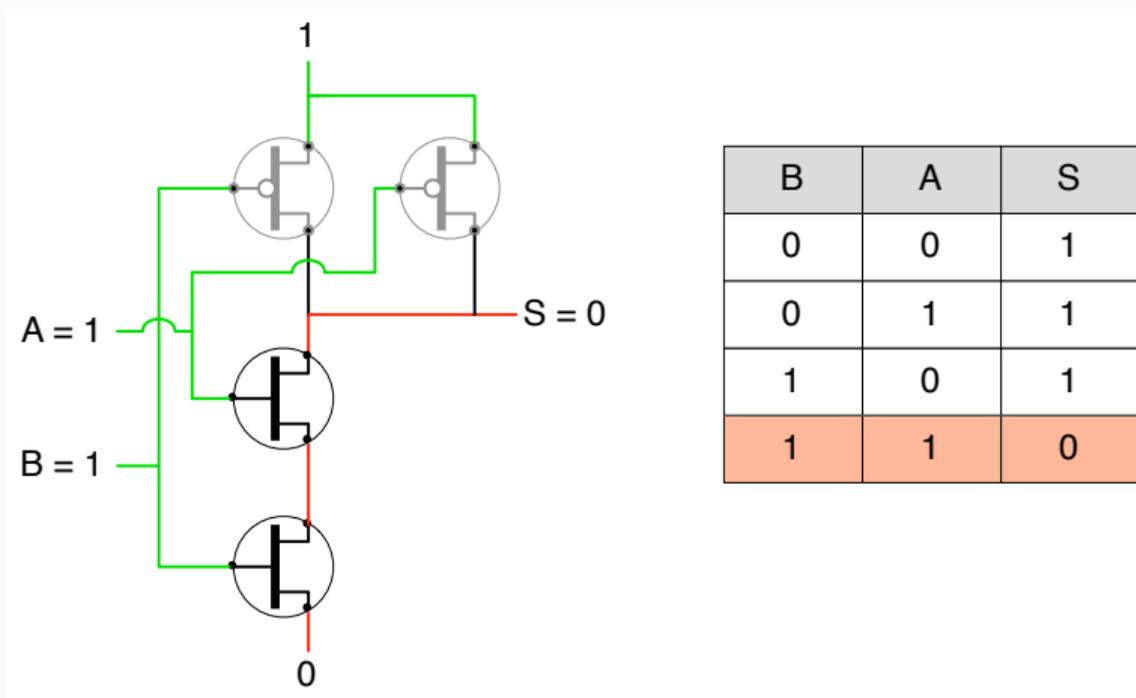
Porte NAND 2 entrées (2)



Porte NAND 2 entrées (2)



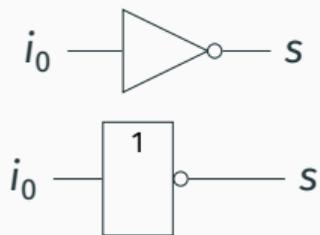
Porte NAND 2 entrées (2)



Les portes logiques

La porte non

Représentation graphique



Notation textuelle

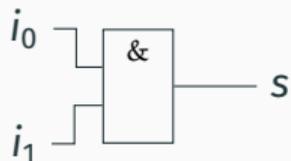
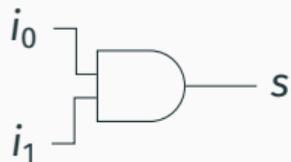
Algèbre de Boole
 $s = \bar{i}_0$

Mathématiques
 $s = \neg i_0$

Table de vérité

i_0	s
0	1
1	0

Représentation graphique



Notation textuelle

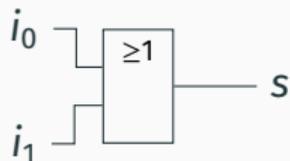
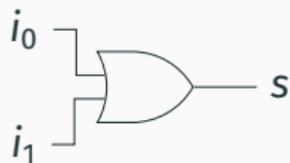
Algèbre de Boole
 $s = i_0 \cdot i_1$ ou $s = i_0 i_1$

Mathématiques
 $s = i_0 \wedge i_1$

Table de vérité

i_0	i_1	s
0	0	0
0	1	0
1	0	0
1	1	1

Représentation graphique



Notation textuelle

Algèbre de Boole

$$s = i_0 + i_1$$

Mathématiques

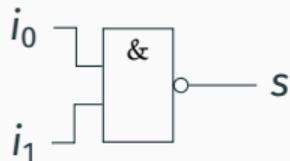
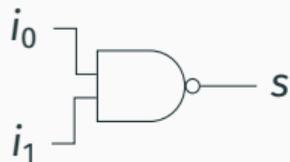
$$s = i_0 \vee i_1$$

Table de vérité

i_0	i_1	s
0	0	0
0	1	1
1	0	1
1	1	1

La porte non-et (nand)

Représentation graphique



Notation textuelle

Algèbre de Boole

$$s = \overline{i_0 i_1}$$

Mathématiques

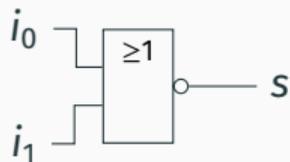
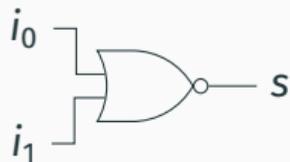
$$s = \neg(i_0 \wedge i_1)$$

Table de vérité

i_0	i_1	s
0	0	1
0	1	1
1	0	1
1	1	0

La porte non-ou (nor)

Représentation graphique



Notation textuelle

Algèbre de Boole

$$s = \overline{i_0 + i_1}$$

Mathématiques

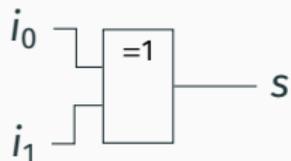
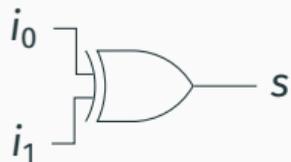
$$s = \neg(i_0 \vee i_1)$$

Table de vérité

i_0	i_1	s
0	0	1
0	1	0
1	0	0
1	1	0

La porte ou-exclusif (xor)

Représentation graphique



Notation textuelle

Algèbre de Boole

$$s = i_0 \bar{i}_1 + \bar{i}_0 i_1$$

Mathématiques

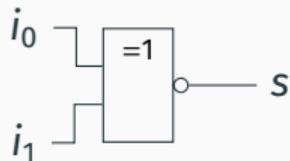
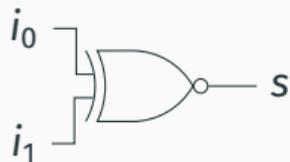
$$s = (i_0 \wedge \neg i_1) \vee (\neg i_0 \wedge i_1)$$

Table de vérité

i_0	i_1	s
0	0	0
0	1	1
1	0	1
1	1	0

La porte non-ou-exclusif (xnor)

Représentation graphique



Notation textuelle

Algèbre de Boole

$$s = i_0 i_1 + \bar{i}_0 \bar{i}_1$$

Mathématiques

$$s = (i_0 \wedge i_1) \vee (\neg i_0 \wedge \neg i_1)$$

ou

$$s = (i_0 \Leftrightarrow i_1)$$

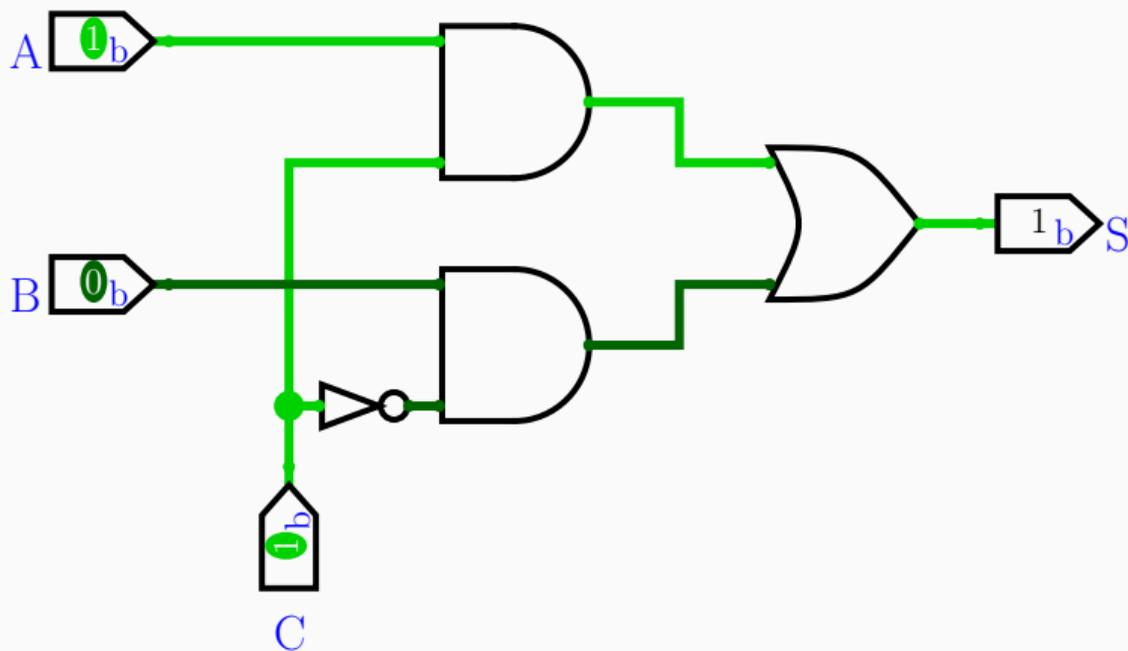
Table de vérité

i_0	i_1	s
0	0	1
0	1	0
1	0	0
1	1	1

Un premier circuit

Que fait ce circuit ?

Pouvez-vous donner sa table de vérité ? son équation ?



Mémoires et circuits séquentiels

En utilisant les portes logiques vues jusqu'à maintenant, on peut construire des **circuits combinatoires**, c'est-à-dire que les sorties dépendent uniquement des entrées.

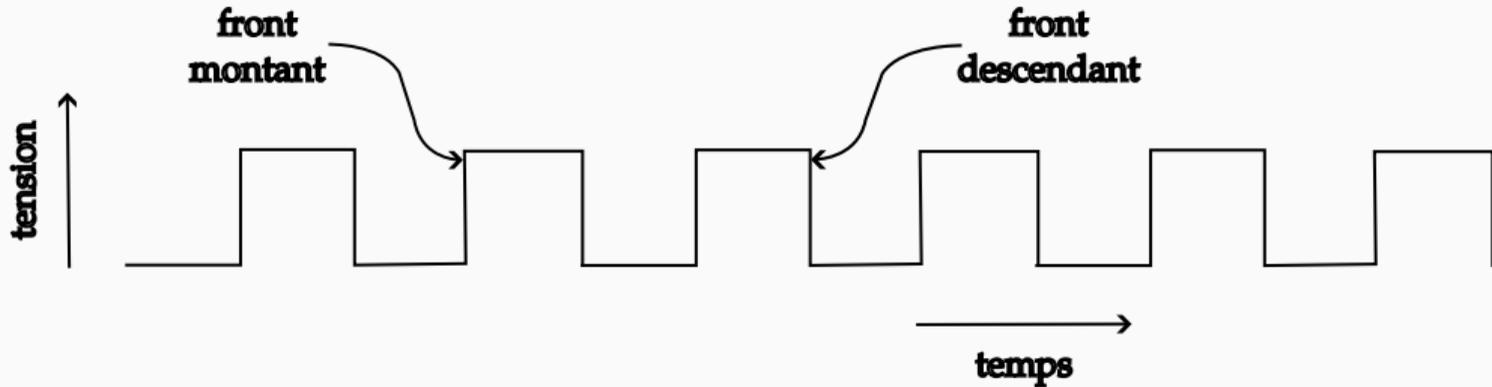
Pour construire un ordinateur, il nous faut également des **circuits séquentiels**, dont les sorties à l'instant t dépendent des entrées à l'instant t et des sorties à l'instant $t-1$.

Pour cela il nous faut :

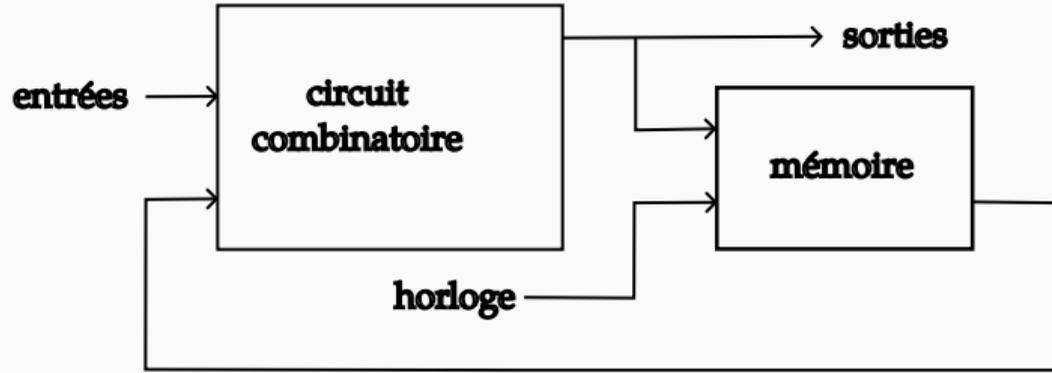
- une horloge qui permet de distinguer les instants t et $t-1$;
- un moyen de mémoriser la valeurs des sorties entre les instants $t-1$ et t .

Signal d'horloge

Les circuits de nos ordinateurs sont des circuits **synchrones**. Cela signifie que les changements de valeurs des sorties sont synchronisés avec un **signal d'horloge**



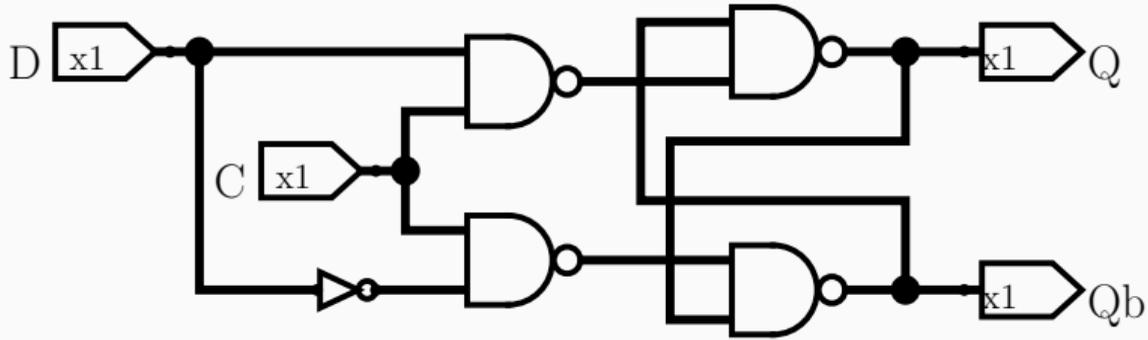
Circuit séquentiel



Dans un circuit synchrone, les valeurs des sorties changent à des dates séparées par un délai égal à la période de l'horloge.

En couplant mémoire et boucles de rétro-action, on peut utiliser la valeur d'une sortie à l'instant $t - 1$ comme entrée à l'instant t .

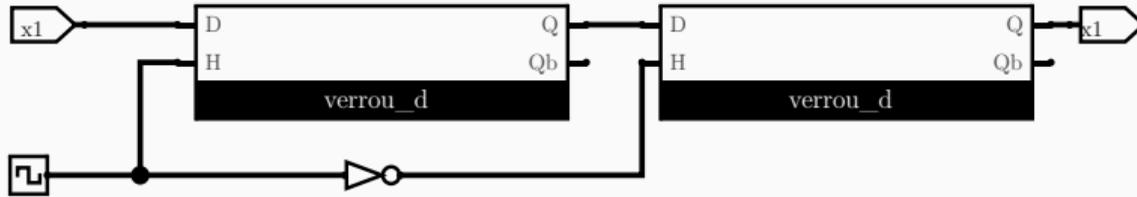
Le verrou D (D latch)



Tant que C est maintenu à 0, les sorties Q et Q_b maintiennent leur valeur, et ce que D changent ou pas de valeur.

Quand C devient égal à 1, la sortie Q prend la valeur de D et Q_b la valeur complémentaire.

La bascule D (D flip-flop)



Pour contrôler précisément la date de changement de valeur de la sortie, la bascule D utilise le principe du sas :

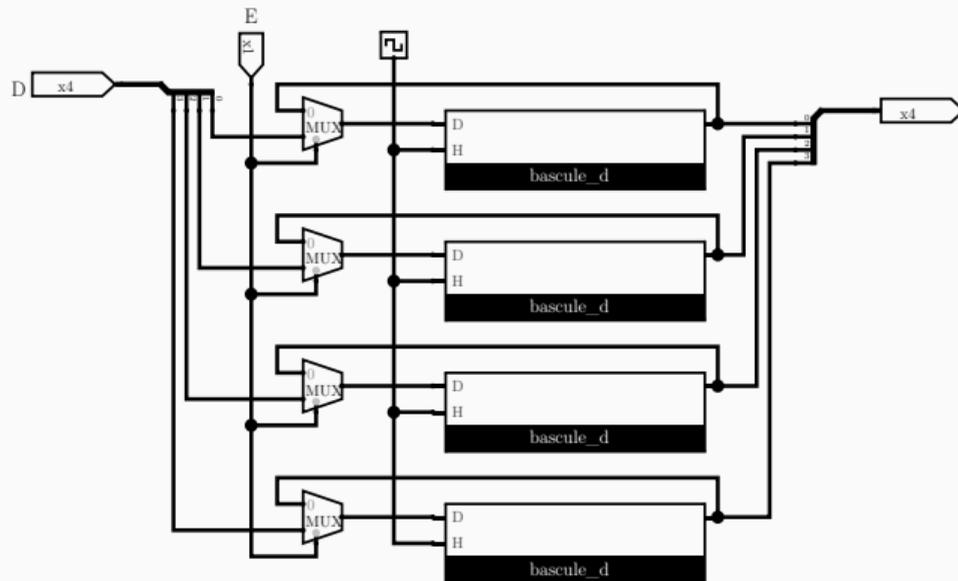
- Quand H vaut 1, le premier verrou est passant, le second est fermé
- Quand H devient égal à 0, le premier verrou se ferme et le second s'ouvre : la sortie change
- Quand H est égal à 0, le premier verrou est fermé et donc la sortie est maintenue

Les registres

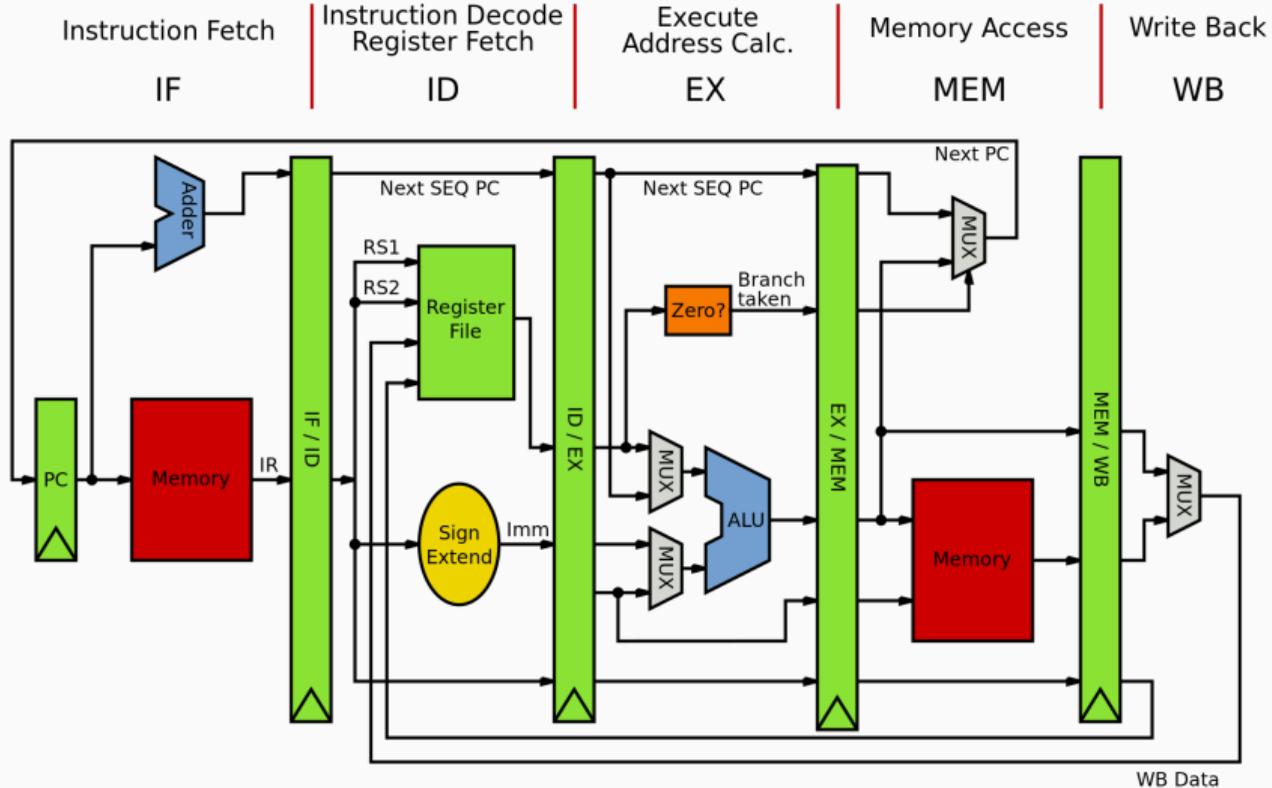
Un registre est construit à partir d'un ensemble de bascules pilotées par le même signal d'horloge.

Le signal d'entrée E contrôle l'écriture dans le registre

- si l'écriture est activée, l'entrée de donnée est prise en compte
- sinon, la valeur précédente est ré-injectée



Aperçu de l'architecture d'une unité centrale simple



Représentation de l'information et codage binaire

Bit

Le bit (pour *binary digit*) est l'unité fondamentale de mesure d'une quantité d'information. Il représente **une valeur logique pouvant prendre deux états : 0 ou 1**.

Conceptuellement, le codage de l'information par des bits est utilisé dès le XVIII^{ème} siècle par Basile Bouchon et Jean-Baptiste Falcon dans les cartes perforées utilisées pour configurer les métiers à tisser automatisés.

Le mot bit est utilisé pour la première fois par Claude E. Shannon dans *A Mathematical Theory of Communication*¹

¹Shannon, C.E. (1948), "**A Mathematical Theory of Communication**", Bell System Technical Journal, 27, pp. 379-423 & 623-656, July & October, 1948.

Bit, Octet, etc.

Le bit étant une unité très petite, il est nécessaire de dériver des unités permettant de représenter des quantités plus grandes.

Nom	Symbole	En octets	En bits
bit	b	-	1
octet	B	1	8
Kibiocet	KiB	2^{10}	2^{13}
Mébioctet	MiB	2^{20}	2^{23}
Gibioctet	GiB	2^{30}	2^{33}
Tébioctet	TiB	2^{40}	2^{43}
Kilooctet	KB	10^3	8×10^3
Mégaocet	MB	10^6	8×10^6
Gigaocet	GB	10^9	8×10^9
Téraocet	TB	10^{12}	8×10^{12}

Pour caractériser les débits des canaux de communication, on utilise parfois également les Kilobits (Kb), les Megabits (Mb), etc.

Le bit est le concept utilisé pour représenter l'information au sein d'un ordinateur.

Pour chaque type d'information que l'on souhaite manipuler, on doit donc définir un **codage binaire**, c'est-à-dire une fonction permettant de représenter une information sous la forme d'une séquence de bits.

Selon le type d'information, **le codage peut être exact ou avec perte** si l'exactitude n'est pas nécessaire (image, son) ou tout simplement impossible (nombres réels).

Codage à base de nombres entiers

Comme on va le voir dans la suite, il est simple de faire correspondre une séquence de bit avec un nombre entier. Cette correspondance est utilisée pour coder les autres type d'information :

- Les nombres fractionnaires sont codés sous forme d'une fraction entre deux entiers
- Les caractères sont codés à l'aide de tables de correspondance
- Les images sont codées sous forme de pixels, chaque pixel est codé par des entiers qui déterminent sa couleur
- Les sons sont codées par des échantillons, chacun codé par des entiers (fréquence, amplitude)
- Les programmes sont codés sous forme d'instructions composées de différents entiers (code de l'opération, code des opérandes)
- ...

En pratique, un ordinateur ne sait manipuler que des nombres entiers!

Codage des entiers naturels

Rappel : notation positionnelle en base 10

Aujourd'hui, nous utilisons majoritairement la notation positionnelle en base 10

- positionnelle = le poids d'un symbole (ou chiffre en base 10) dépend de sa position absolue dans l'écriture du nombre

Par exemple : $2\ 457 = 2 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Notation positionnelle : de la base 10 vers la base 2

Notation positionnelle en base 10

$$\begin{aligned} s_{n-1}s_{n-2} \dots s_0 &= s_{n-1} \times 10^{n-1} + s_{n-2} \times 10^{n-2} + \dots + s_0 \times 10^0 \\ &= \sum_{i=0}^{i=n-1} s_i \times 10^i \end{aligned}$$

avec $\forall i, s_i \in [0..10 - 1]$

Notation positionnelle : de la base 10 vers la base 2

Notation positionnelle en base 10

$$\begin{aligned} s_{n-1}s_{n-2} \dots s_0 &= s_{n-1} \times 10^{n-1} + s_{n-2} \times 10^{n-2} + \dots + s_0 \times 10^0 \\ &= \sum_{i=0}^{i=n-1} s_i \times 10^i \end{aligned}$$

avec $\forall i, s_i \in [0..10 - 1]$

Pour coder les entiers naturels en binaire, il est naturel d'utiliser une notation positionnelle en base 2.

Notation positionnelle : de la base 10 vers la base 2

Notation positionnelle en base 10

$$\begin{aligned} s_{n-1}s_{n-2} \dots s_0 &= s_{n-1} \times 10^{n-1} + s_{n-2} \times 10^{n-2} + \dots + s_0 \times 10^0 \\ &= \sum_{i=0}^{i=n-1} s_i \times 10^i \end{aligned}$$

avec $\forall i, s_i \in [0..10 - 1]$

Pour coder les entiers naturels en binaire, il est naturel d'utiliser une notation positionnelle en base 2.

Notation positionnelle en base 2

$$\begin{aligned} s_{n-1}s_{n-2} \dots s_0 &= s_{n-1} \times 2^{n-1} + s_{n-2} \times 2^{n-2} + \dots + s_0 \times 2^0 \\ &= \sum_{i=0}^{i=n-1} s_i \times 2^i \end{aligned}$$

avec $\forall i, s_i \in [0..2 - 1]$

Exemple

$$\begin{aligned}1000\ 0011_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 \\ &\quad + 1 \times 2^0 \\ &= 2^7 + 2^1 + 2^0 \\ &= 128 + 2 + 1 \\ &= 131\end{aligned}$$

Exemple

$$\begin{aligned}1000\ 0011_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 \\ &\quad + 1 \times 2^0 \\ &= 2^7 + 2^1 + 2^0 \\ &= 128 + 2 + 1 \\ &= 131\end{aligned}$$

On écrira $1000\ 0011_2 = 131_{10}$

Exemple

$$\begin{aligned}0110\ 0111_2 &= 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 64 + 32 + 4 + 2 + 1 \\ &= 103\end{aligned}$$

On écrit $0110\ 0111_2 = 103_{10}$

L'écriture d'un nombre en base 2 comporte vite beaucoup de bit, ce qui rend sa manipulation fastidieuse et source d'erreurs.

C'est pour cette raison qu'on a pris l'habitude d'utiliser la base 16 → on parle de **représentation hexadécimale**.

Les 16 symboles utilisés sont **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.

Notation positionnelle **en base 16**

$$\begin{aligned} s_{n-1}s_{n-2} \dots s_0 &= s_{n-1} \times 16^{n-1} + s_{n-2} \times 16^{n-2} + \dots + s_0 \times 16^0 \\ &= \sum_{i=0}^{i=n-1} s_i \times 16^i \end{aligned}$$

avec $\forall i, s_i \in [0..F]$

Exemples

$$\begin{aligned}1CAFE_{16} &= 1_{16} \times 16^4 + C_{16} \times 16^3 + A_{16} \times 16^2 + F_{16} \times 16^1 + E_{16} \times 16^0 \\ &= 1 \times 16^4 + 12 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 14 \times 16^0 \\ &= 1 \times 65\,536 + 12 \times 4\,096 + 10 \times 256 + 15 \times 16 + 14 \times 1 \\ &= 117\,502_{10}\end{aligned}$$

$$\begin{aligned}4\,242_{10} &= 4\,096 + 144 + 2 \\ &= 1 \times 16^3 + 9 \times 16^1 + 2 \times 16^0 \\ &= 1092_{16}\end{aligned}$$

Pourquoi la base 16 ?

Un digit en base 16 représente un nombre de 4 bits et donc un nombre à deux digits représente 1 octet.

Base 2	Base 16	Base 10	Base 2	Base 16	Base 10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Pourquoi la base 16 ?

Un digit en base 16 représente un nombre de 4 bits et donc un nombre à deux digits représente 1 octet.

Base 2	Base 16	Base 10	Base 2	Base 16	Base 10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

$$11011001_2 = D9_{16}$$

Pourquoi la base 16 ?

Un digit en base 16 représente un nombre de 4 bits et donc un nombre à deux digits représente 1 octet.

Base 2	Base 16	Base 10	Base 2	Base 16	Base 10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

$$1101\ 1001_2 = D9_{16}$$

Pourquoi la base 16 ?

Un digit en base 16 représente un nombre de 4 bits et donc un nombre à deux digits représente 1 octet.

Base 2	Base 16	Base 10	Base 2	Base 16	Base 10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

$$1101\ 1001_2 = D9_{16}$$

Constantes entières dans les langages de programmation

Dans la plupart des langages de programmation², une séquence de caractères entre '0' et '9' ne commençant pas par '0' est **en base 10**, par exemple 15 ou 75.

Une séquence de caractères commençant par '0', suivi de caractères entre '0' et '7' est **en base 8**, par exemple 017 ou 0213.

Une séquence de caractères commençant par '0', suivi de 'b' ou 'B', suivi de caractères entre '0' et '1' est **en base 2** par exemple 0b1111 ou 0B01001011.

Une séquence de caractères commençant par '0', suivi de 'x' ou 'X', suivi de caractères entre '0' et '9', ou entre 'a' et 'f', ou entre 'A' et 'F', est **en base 16** par exemple 0xF ou 0X4B.

²dont golang, python et java

Un mot sur l'arithmétique entière des ordinateurs

Au sein d'un processeur, l'**unité arithmétique et logique** (UAL) réalise les opérations arithmétiques usuelles (+, −, ×, ÷), les comparaisons (=, ≠, <, ≤, ...), et quelques autres opérations (p. e.x décalages, ou rotation)

Une UAL est un ensemble de circuits conçus pour travailler sur **des mots de taille fixe**. La plupart des machines modernes savent manipuler des mots de **8, 16, 32** et **64** bits.

Cela signifie que les opérations sont celles du **calcul modulaire** sur les restes des entiers dans la division par 2^k (où k est la taille du mot).

Exemples

Avec des mots de 8 bit, on travaille modulo $2^8 = 256$:

$$257 \equiv 1$$

car $257 = 256 \times 1 + 1$ et donc $257 \bmod 256 = 1$.

$$0xFE + 0x10 \equiv 0xE$$

car $FE_{16} + 10_{16} = 10E_{16} = 270_{10}$, $270 = 256 + 14$, soit $270 \bmod 256 = 14$ et $14_{10} = E_{16}$

Exemples

Avec des mots de 8 bit, on travaille modulo $2^8 = 256$:

$$257 \equiv 1$$

car $257 = 256 \times 1 + 1$ et donc $257 \bmod 256 = 1$.

$$0xFE + 0x10 \equiv 0xE$$

car $FE_{16} + 10_{16} = 10E_{16} = 270_{10}$, $270 = 256 + 14$, soit $270 \bmod 256 = 14$ et $14_{10} = E_{16}$

Certains langages (comme python) cachent ces effets en agrandissant automatiquement la taille de données.

D'autres (comme golang) demandent d'explicitement la taille des variables (p.ex. `uint8`, ou `uint64`).

Codage des entiers relatifs

Codage en complément à 2

Il est souvent utile de pouvoir manipuler des valeurs entières négatives, pas uniquement positives. Aujourd'hui, l'immense majorité des machines (toutes?) utilisent le **codage binaire en complément à 2** pour manipuler les entiers relatifs.

Sur un mot de taille donnée k , le codage en complément à 2 se définit simplement **en donnant une valeur négative au bit de poids fort** :

$$\begin{aligned} s_{k-1} \dots s_0 &= s_{k-1} \times -(2^{k-1}) + s_{k-2} \times 2^{k-2} + \dots + s_0 \times 2^0 \\ &= s_{k-1} \times -(2^{k-1}) + \sum_{i=0}^{i=k-2} s_i \times 2^i \end{aligned}$$

avec $\forall i, s_i \in [0..1]$

Exemples

$$1010_{c2} = 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6 \neq 1010_2$$

Exemples

$$1010_{c2} = 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6 \neq 1010_2$$

$$0110_{c2} = 0 \times -(2^3) + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 4 + 2 = 6 = 0110_2$$

Exemples

$$1010_{c2} = 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6 \neq 1010_2$$

$$0110_{c2} = 0 \times -(2^3) + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 4 + 2 = 6 = 0110_2$$

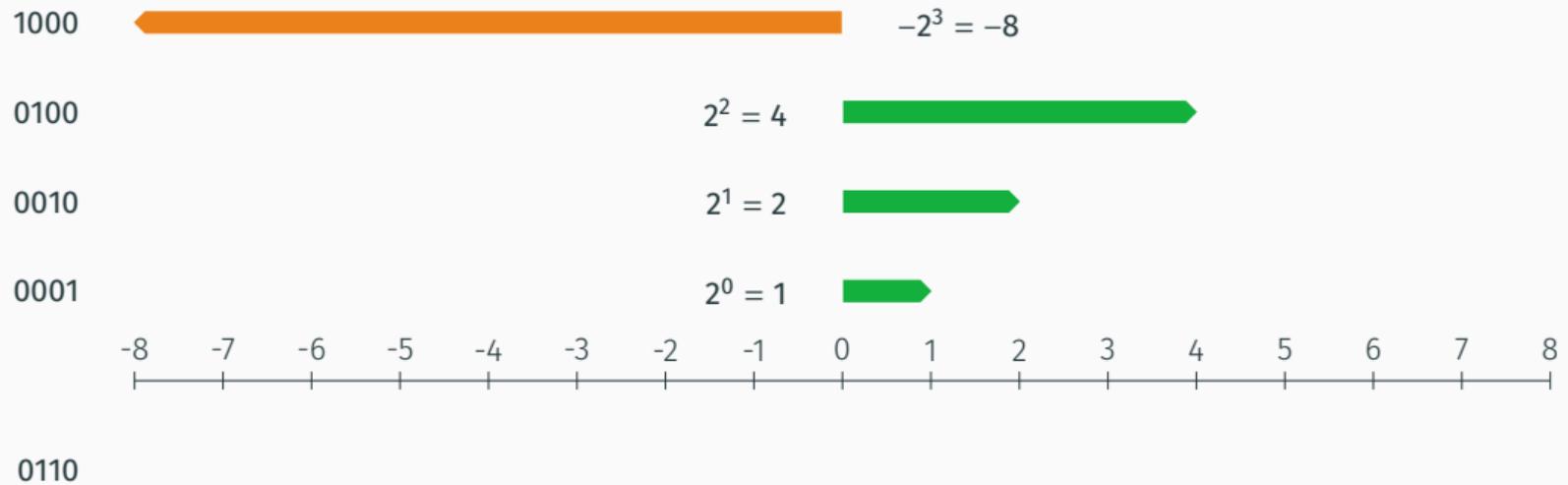
Si le bit de poids fort est 0 : nombre positif.

Si le bit de poids fort est 1 : nombre négatif.

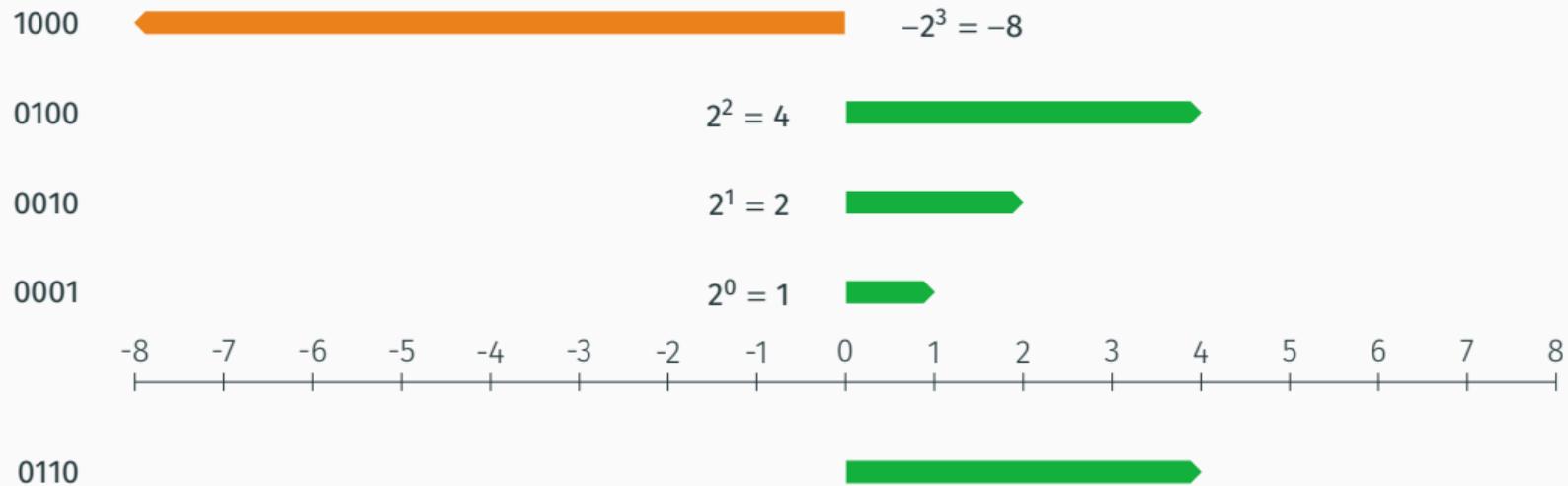
Visualisation du complément à 2 (mots de 4 bits)



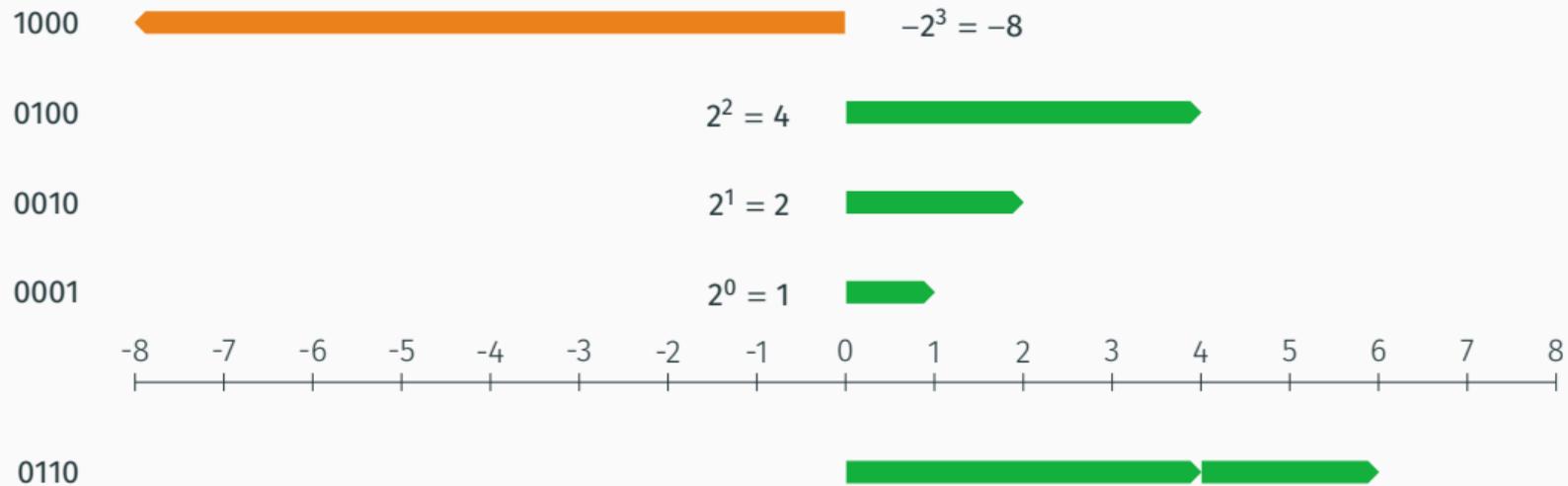
Visualisation du complément à 2 (mots de 4 bits)



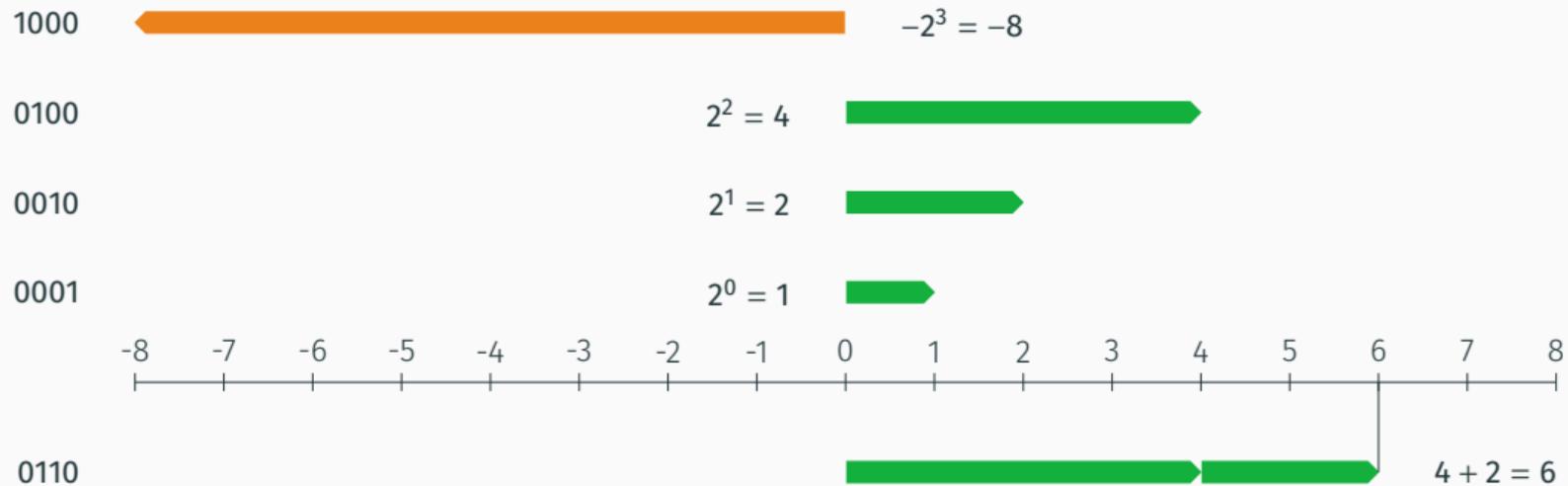
Visualisation du complément à 2 (mots de 4 bits)



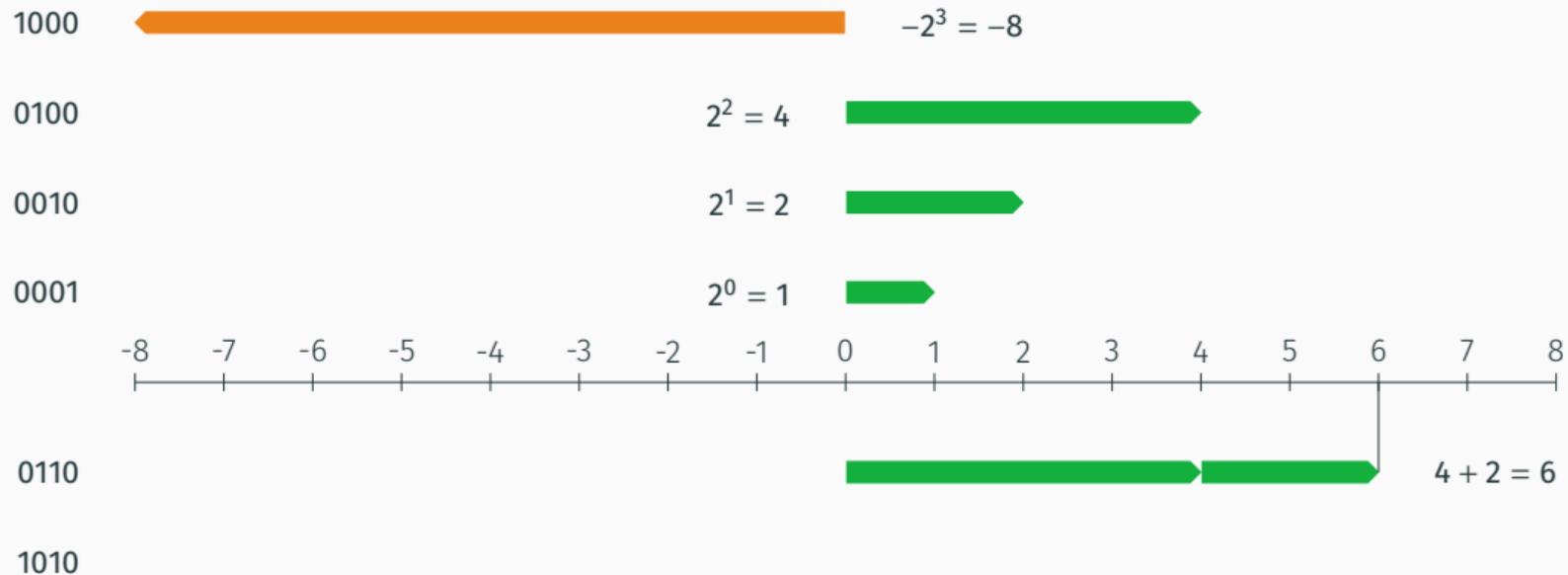
Visualisation du complément à 2 (mots de 4 bits)



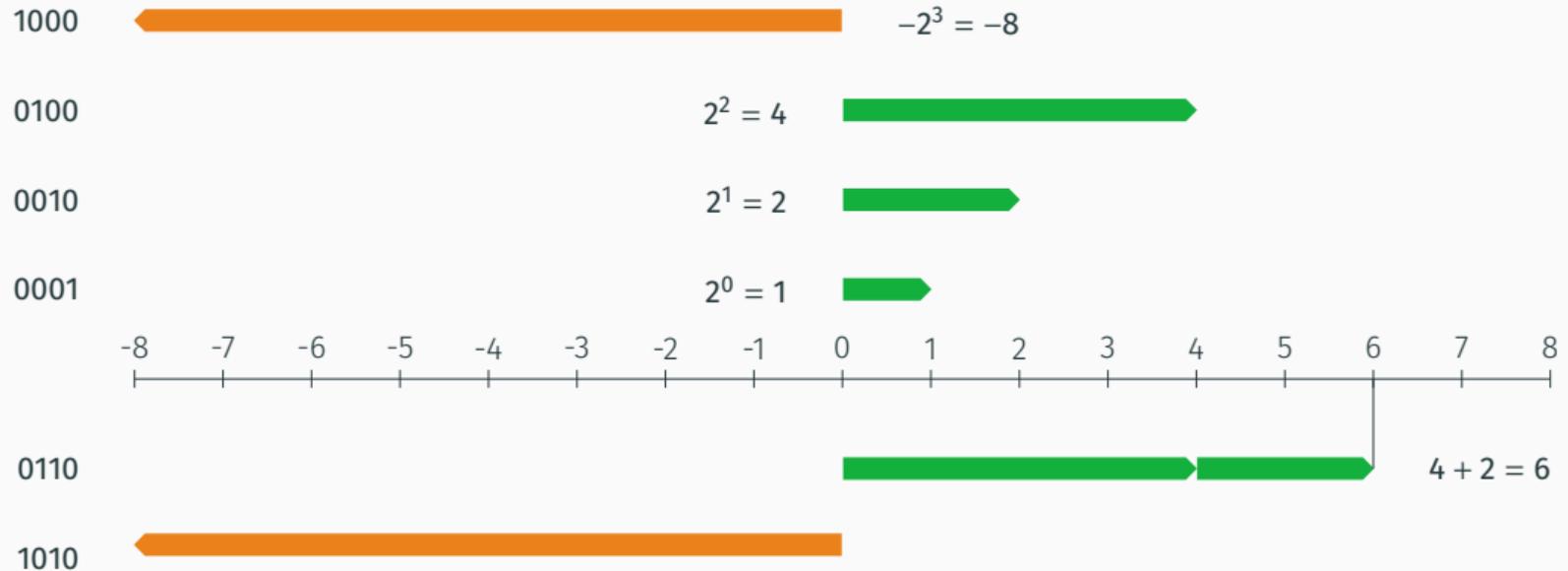
Visualisation du complément à 2 (mots de 4 bits)



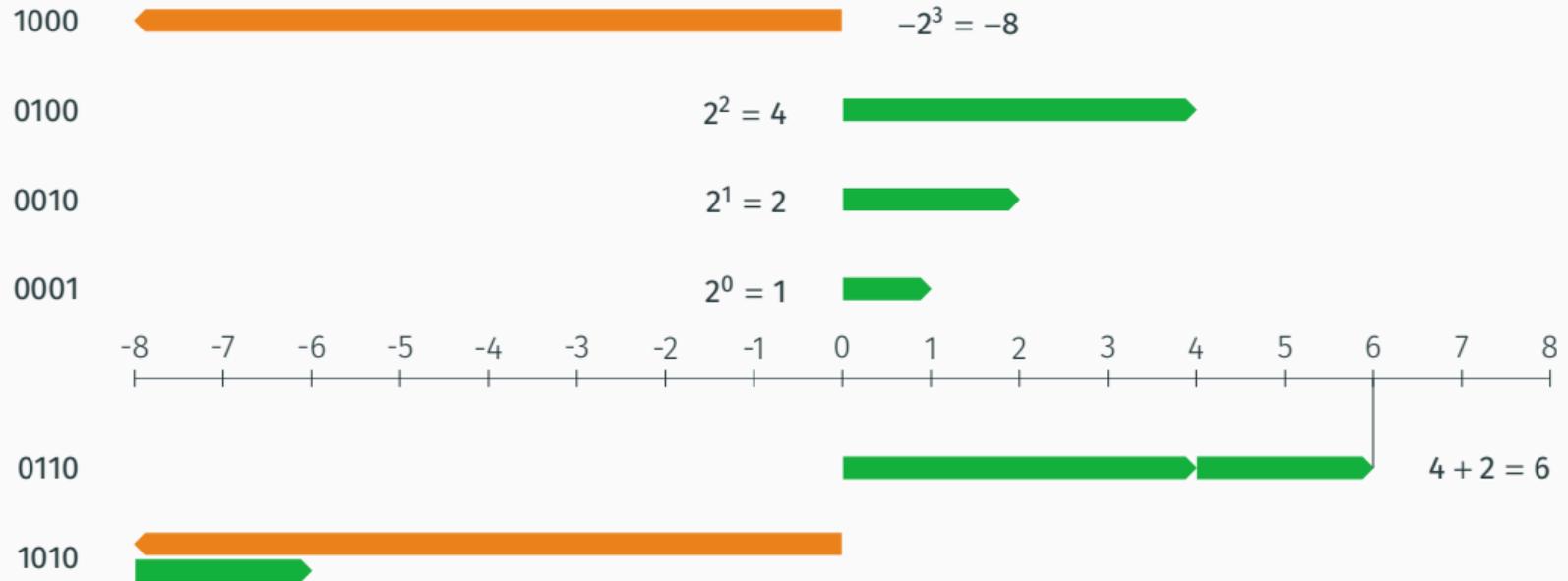
Visualisation du complément à 2 (mots de 4 bits)



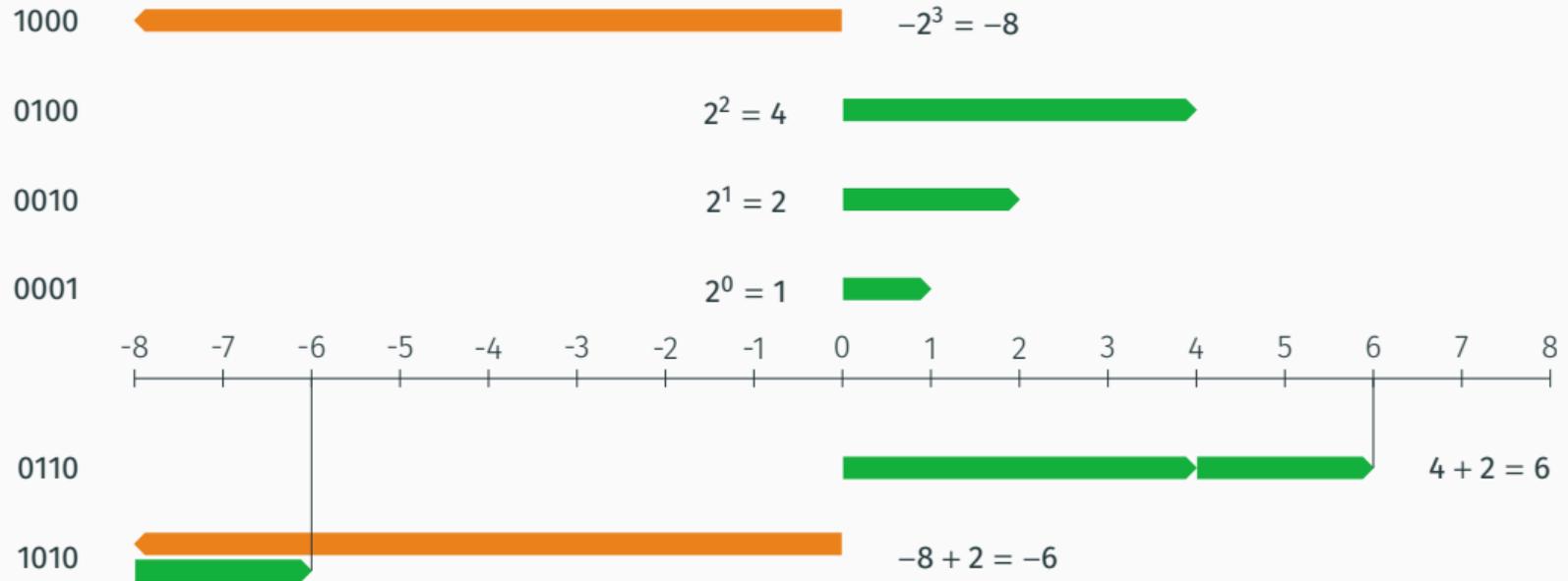
Visualisation du complément à 2 (mots de 4 bits)



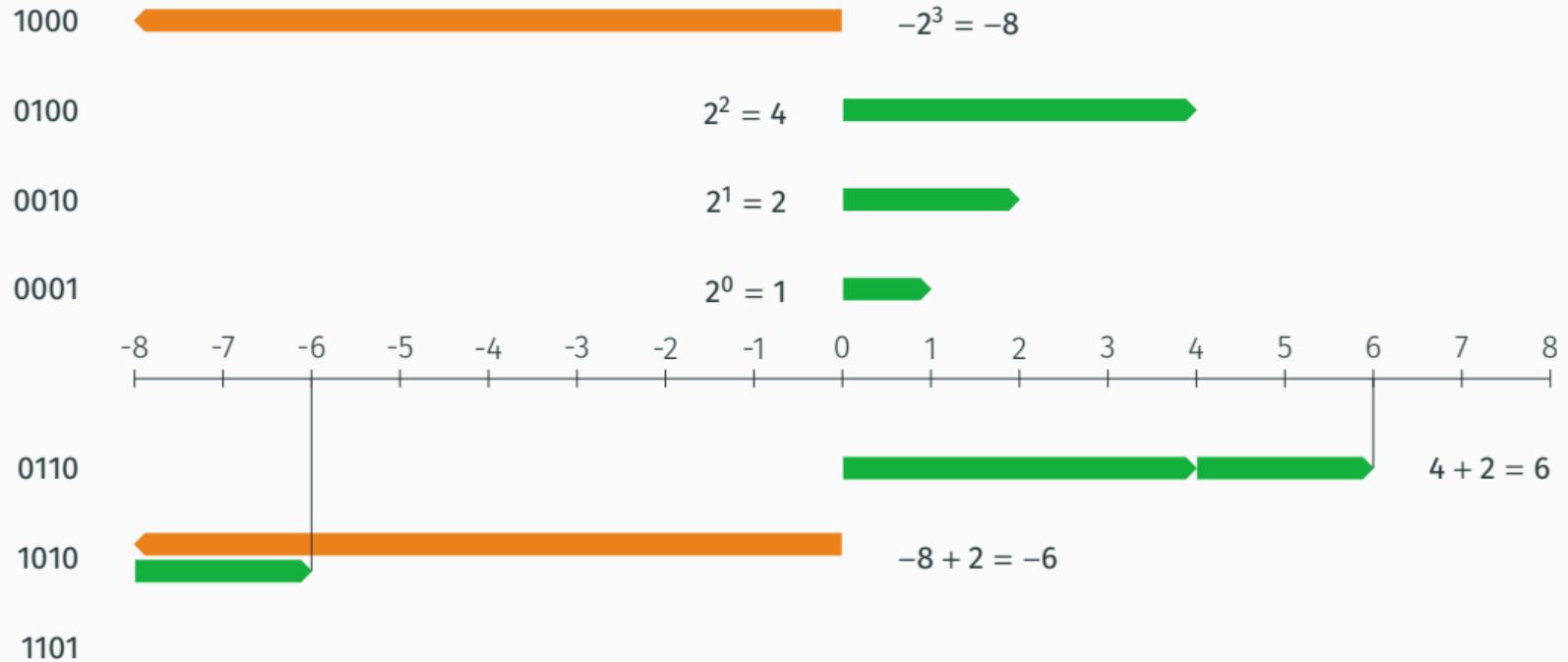
Visualisation du complément à 2 (mots de 4 bits)



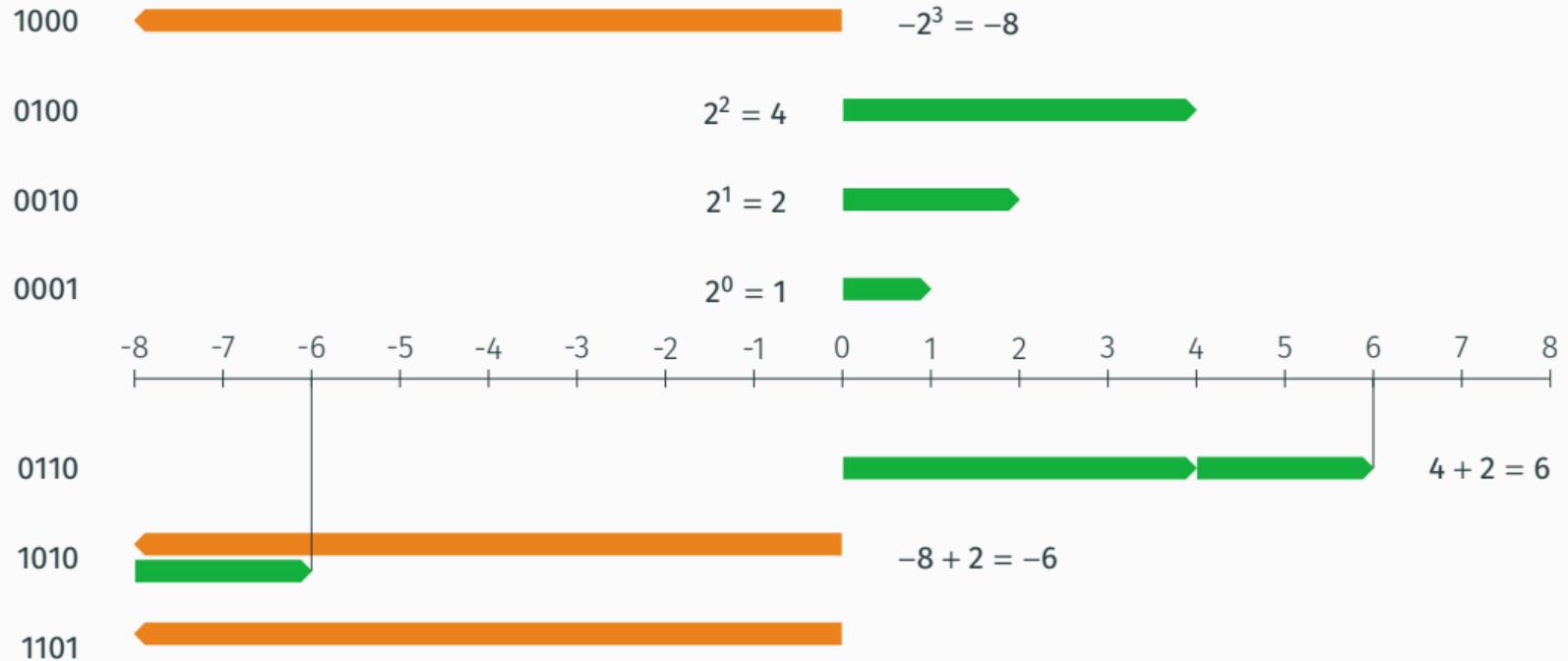
Visualisation du complément à 2 (mots de 4 bits)



Visualisation du complément à 2 (mots de 4 bits)



Visualisation du complément à 2 (mots de 4 bits)



Visualisation du complément à 2 (mots de 4 bits)



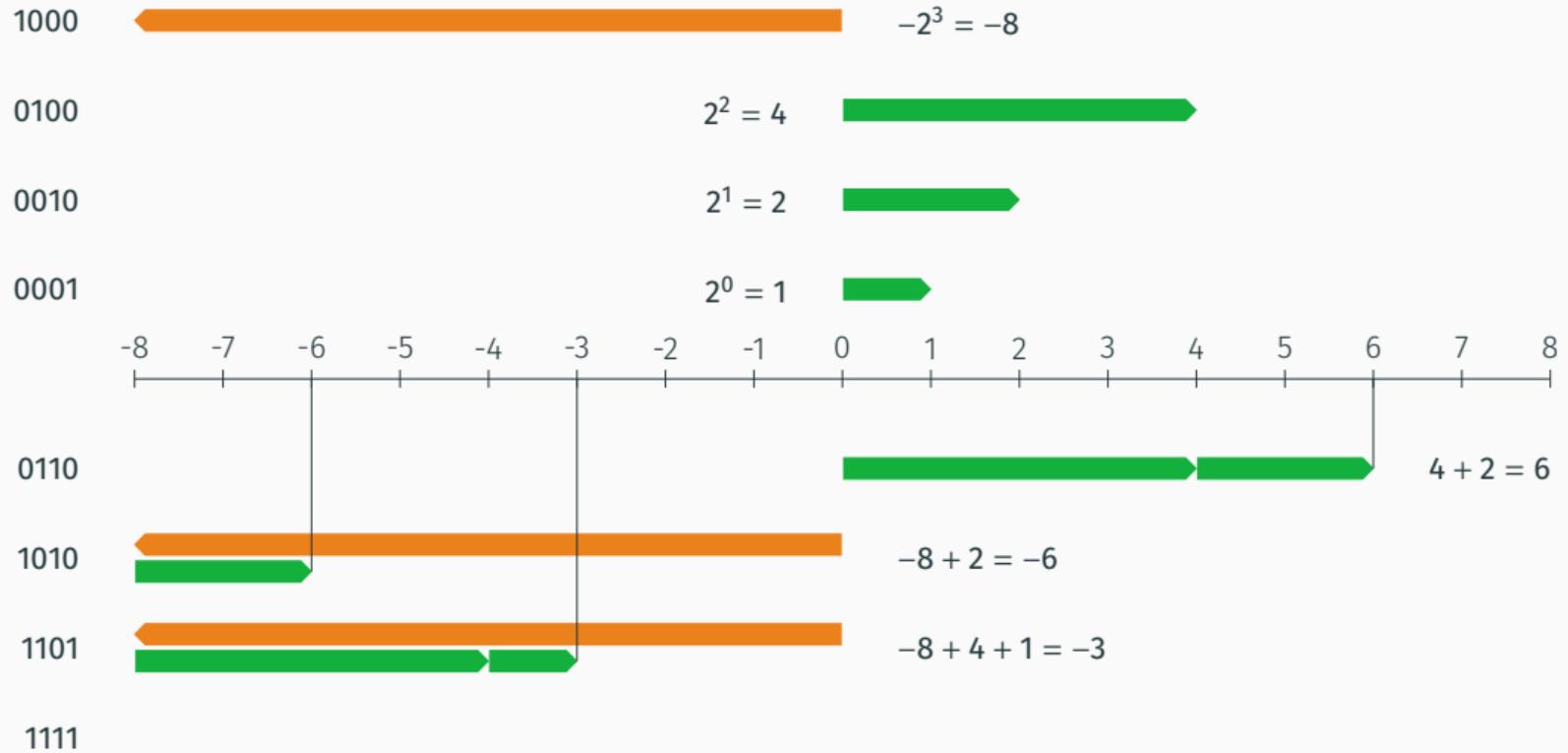
Visualisation du complément à 2 (mots de 4 bits)



Visualisation du complément à 2 (mots de 4 bits)



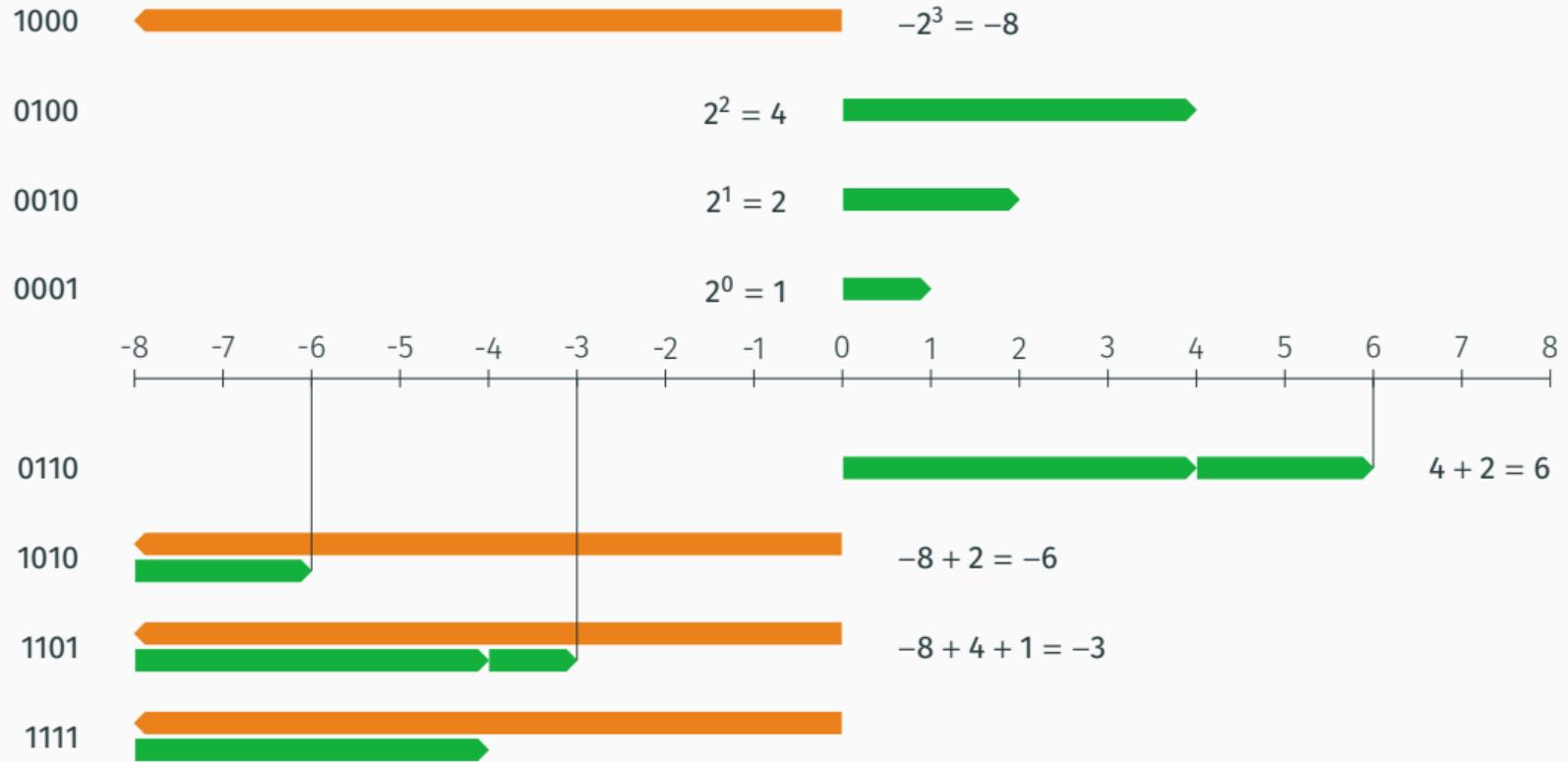
Visualisation du complément à 2 (mots de 4 bits)



Visualisation du complément à 2 (mots de 4 bits)



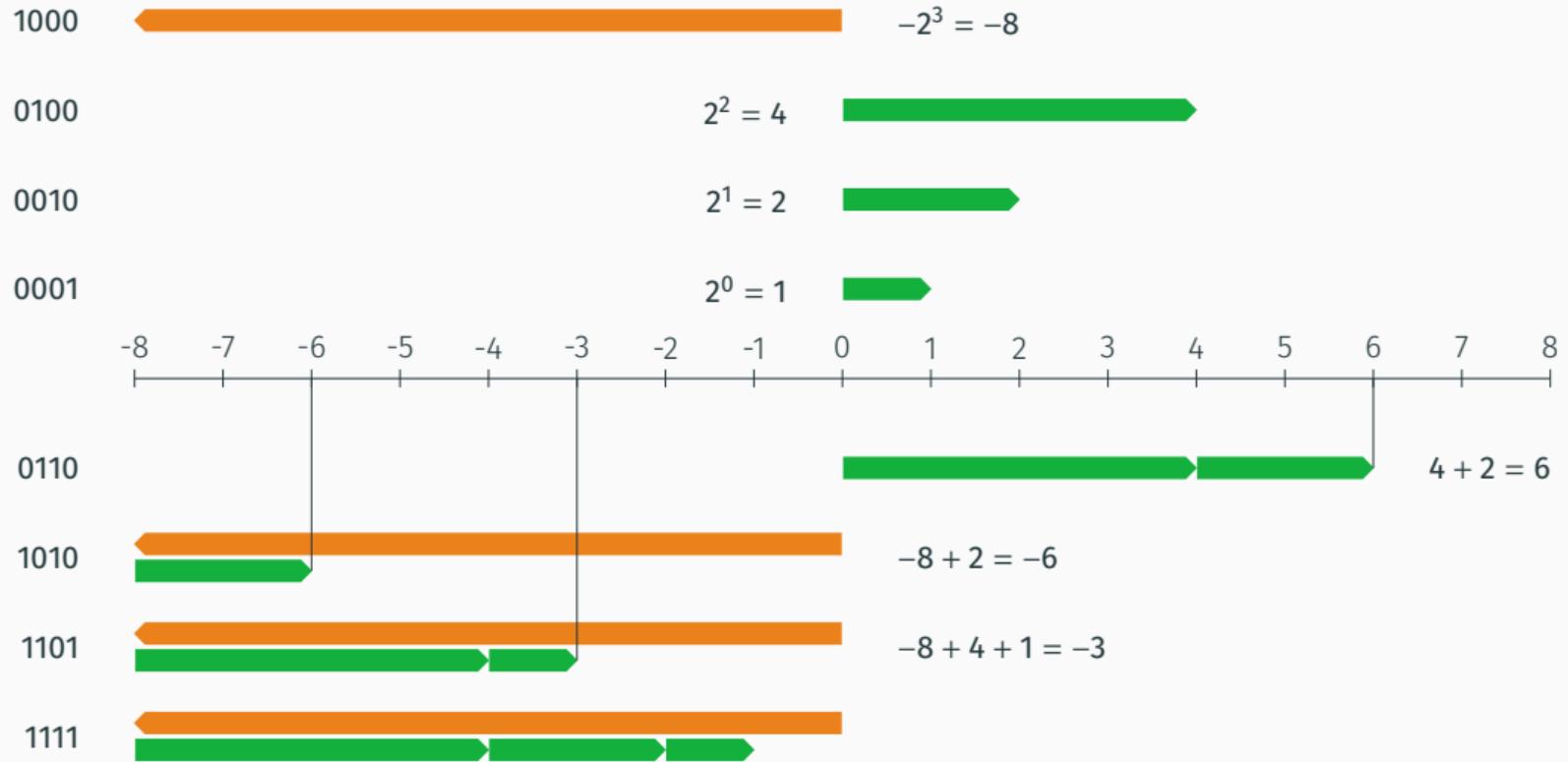
Visualisation du complément à 2 (mots de 4 bits)



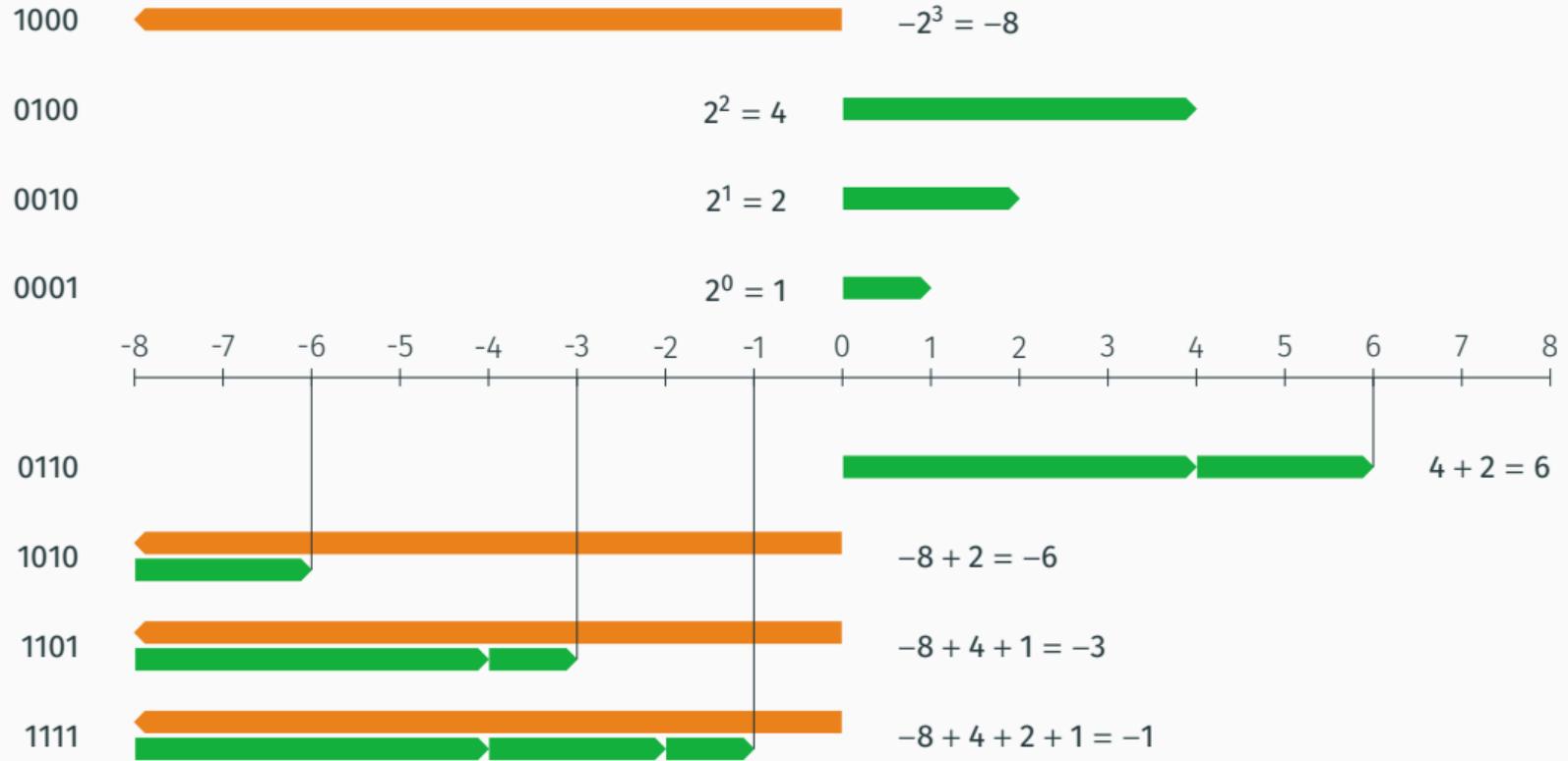
Visualisation du complément à 2 (mots de 4 bits)



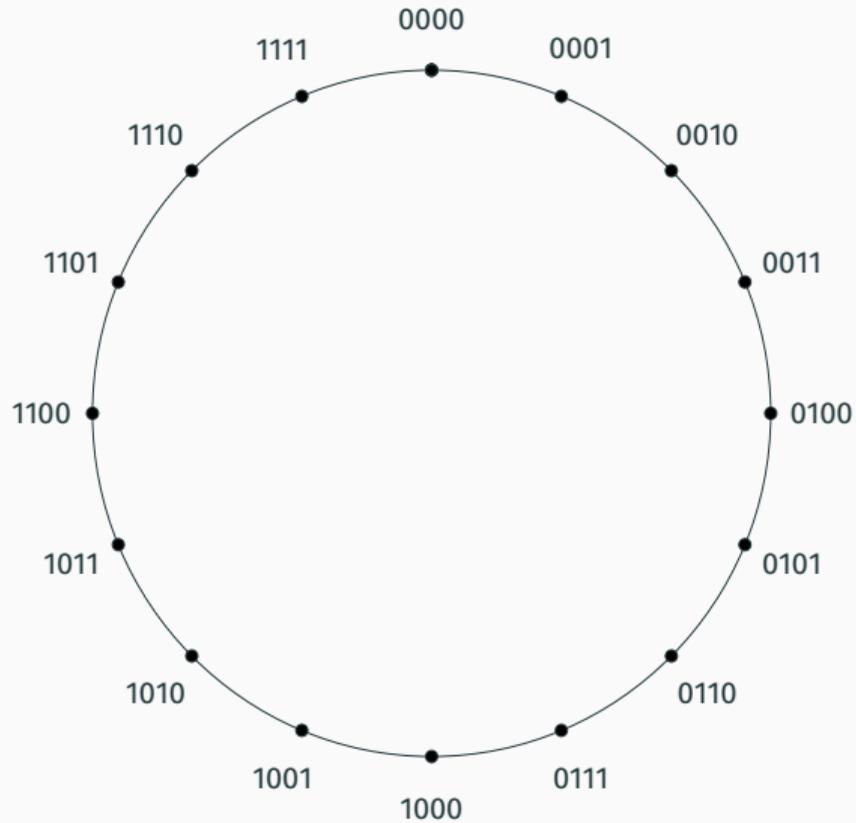
Visualisation du complément à 2 (mots de 4 bits)



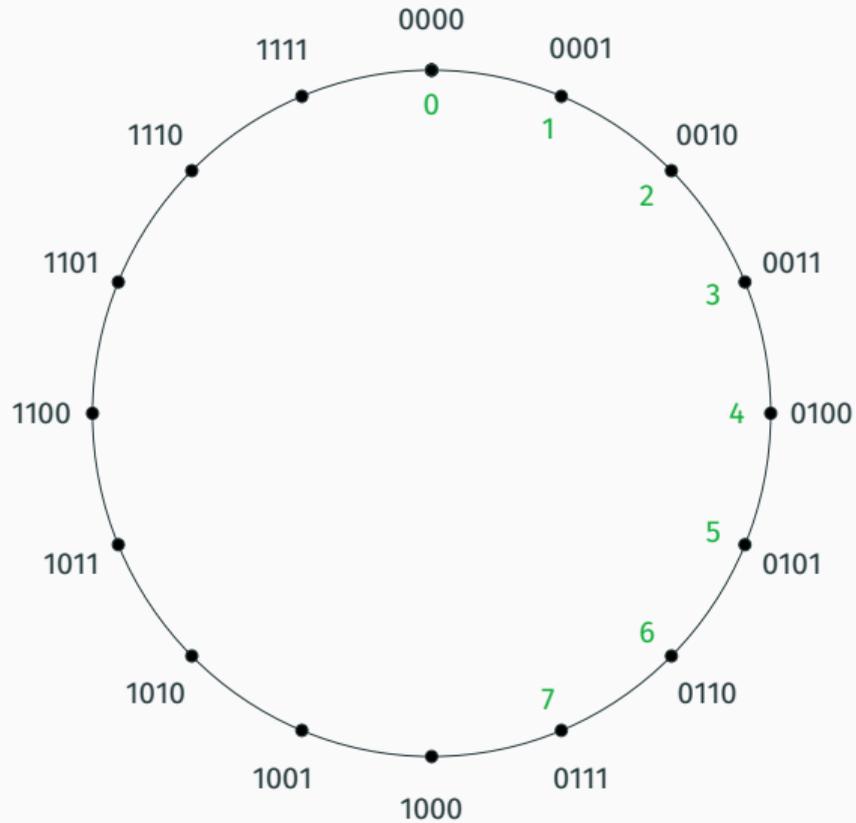
Visualisation du complément à 2 (mots de 4 bits)



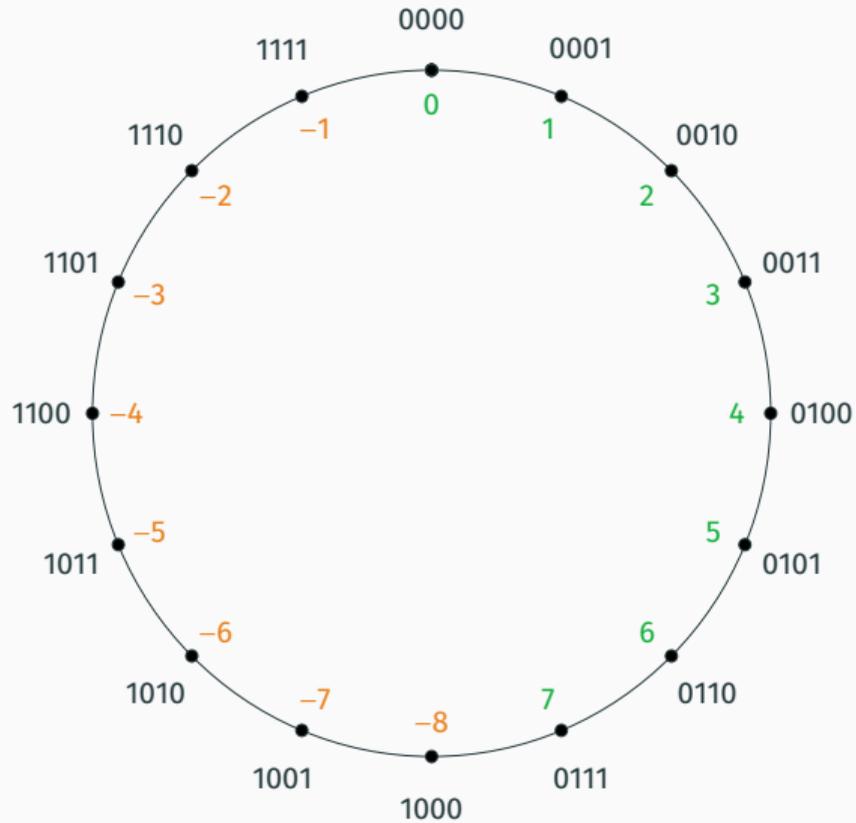
Autre visualiation du complément à 2 (mots de 4 bits)



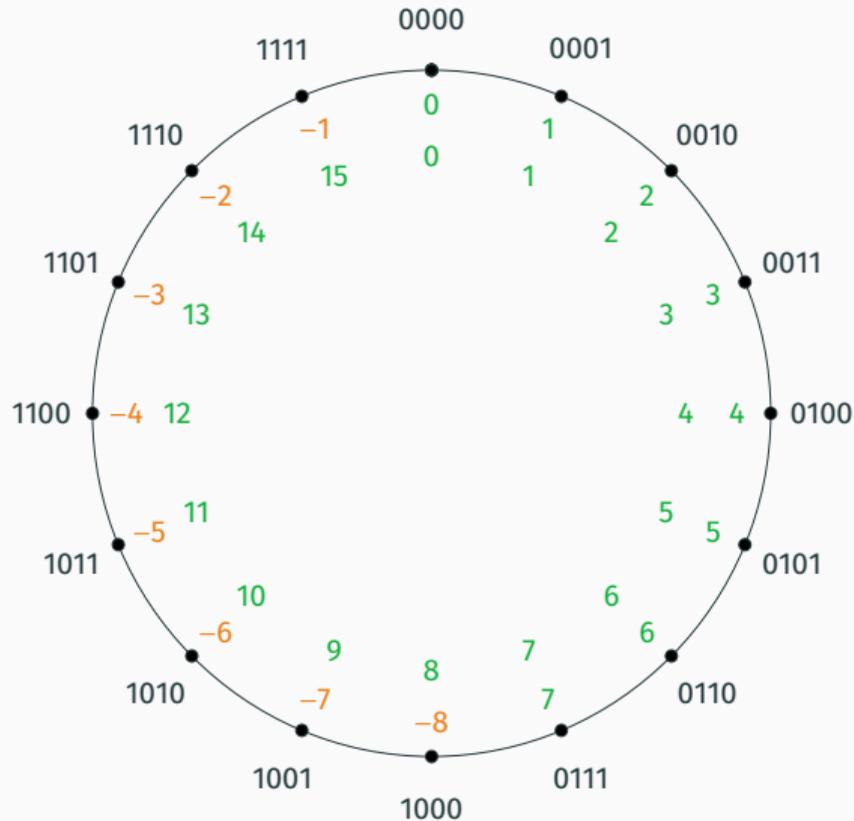
Autre visualiation du complément à 2 (mots de 4 bits)



Autre visualiation du complément à 2 (mots de 4 bits)



Autre visualiation du complément à 2 (mots de 4 bits)



Puisque les mots sont de taille fixe, il s'agit aussi d'une arithmétique modulaire.

On a vu dans la visualiation précédente qu'avec le complément à 2, les bits sont « dans le même ordre » que dans la notation positionnelle en base 2 :

- Utilisation d'un même circuit pour l'addition
- Soustraction = addition avec l'opposé → utilisation de l'additionneur

Si les machines utilisent le complément à 2, c'est parce que cela simplifie grandement la conception des UAL et cela améliore leurs performances!

Avec des mots de 4 bits :

$$6 + 3 \equiv -7$$

En effet, $0110 + 0011 = 1001$, et $1001_{c2} = -7$ (alors que $1001_2 = 9$).

On a bien $-7 \equiv 9 \pmod{2^4}$.

$$-5 + (-6) = 5$$

En effet $1011 + 1010 = 10101$, mais sur 4 bits le résultat est 0101 et $0101_{c2} = 5$.

On a bien $-11 \equiv 5 \pmod{2^4}$.

Status d'un calcul dans une UAL

Pour avertir de certaines situations, une UAL produit, en plus du résultat du calcul, un certain nombre de bits de status. Les plus classiques sont :

- retenue (*carry*) : la retenue sortante du calcul
- débordement (*overflow*) : en complément à 2, le résultat déborde de la capacité du mot
- zéro : le résultat est nul
- signe : en complément à 2, le résultat est négatif

Rappels sur les nombres décimaux

Si les machines manipulent assez facilement les entiers (naturels ou relatifs), il en est autrement des autres ensembles de nombre que l'on utilise couramment en mathématiques comme \mathbb{R} , \mathbb{Q} , ou encore \mathbb{D} .

Pour mémoire :

- \mathbb{R} est l'ensemble des réels.
- $\mathbb{Q} = \{q = \frac{n}{d} \mid n \in \mathbb{Z} \wedge d \in \mathbb{N}^+\}$ est l'ensemble des rationnels.
- $\mathbb{D} = \{d = \frac{n}{10^k} \mid n \in \mathbb{Z} \wedge k \in \mathbb{N}\}$ est l'ensemble des décimaux.
- $\mathbb{D} \subsetneq \mathbb{Q} \subsetneq \mathbb{R}$.

Notation décimale positionnelle

$$\mathbb{D} = \left\{ d = \frac{n}{10^k} \mid n \in \mathbb{Z} \wedge k \in \mathbb{N} \right\}$$

Pour écrire les nombres décimaux, on peut étendre la notation positionnelle pour inclure la partie fractionnaire : on parle alors de **notation décimale positionnelle**.

$$\begin{aligned} s_n \dots s_0, s_{-1} \dots s_{-m} &= s_n \times 10^n + \dots + s_0 \times 10^0 \\ &\quad + s_{-1} \times 10^{-1} + \dots + s_{-m} \times 10^{-m} \\ &= \sum_{i=-m}^{i=n} s_i \times 10^i \end{aligned}$$

avec $\forall i, s_i \in [0..10 - 1]$.

Exemples

$$\frac{123}{10^2} = 1,23$$

$$\frac{123}{10^4} = 0,0123$$

$$\frac{314159}{10^5} = 3,14159$$

Nombres « décimaux binaires »

Sur le modèle de \mathbb{D} , on peut construire l'ensemble suivant :

$$\left\{ b = \frac{n}{2^k} \mid n \in \mathbb{Z} \wedge k \in \mathbb{N} \right\}$$

Et pour représenter les nombres de cet ensemble, on peut étendre la notation positionnelle en base 2 :

$$\begin{aligned} b_n \dots b_0, b_{-1} \dots b_{-m} &= b_n \times 2^n + \dots + b_0 \times 2^0 \\ &\quad + b_{-1} \times 2^{-1} + \dots + b_{-m} \times 2^{-m} \\ &= \sum_{i=-m}^{i=n} b_i \times 2^i \end{aligned}$$

avec $\forall i, b_i \in [0..2 - 1]$.

Exemples

$$\frac{111}{10^{11}} = 0,111$$

soit en base 10 : $2^{-1} + 2^{-2} + 2^{-3} = 0,875$

$$\frac{1010101}{10^{11}} = 1010,101$$

soit en base 10 : $2^3 + 2^1 + 2^{-1} + 2^{-3} = 10,625$

$$\frac{11001001001}{10^{1001}} = 11,001001001$$

soit en base 10 : $2^1 + 2^0 + 2^{-3} + 2^{-6} + 2^{-9} = 3,142578125_{10}$

Représentations en machine des nombres fractionnaires

Comment représenter les nombres fractionnaires en machine ?

Pour les entiers, le choix est fait de se limiter aux nombres d'un intervalle dont la taille dépend de la « largeur des circuits » de l'UAL, typiquement 64 bits sur une machine moderne :

- Entiers naturels de $[0, 2^{64} - 1]$
- Entiers relatifs de $[-2^{63}, 2^{63} - 1]$

Pour les besoins plus spécifiques (p.ex. cryptographie), il existe des bibliothèques de « grands entiers » (*BigInt*) qui permettent d'atteindre une précision arbitraire, au prix de performances détériorées (pourquoi?)

Peut-on utiliser cette approche pour manipuler des nombres rationnels ?

Comment représenter un nombre décimal

Les rationnels sont un **ensemble « dense »**, et la représentation choisie ne comptera nécessairement qu'un nombre limité d'éléments $\{x_1, x_2, \dots, x_n\}$.

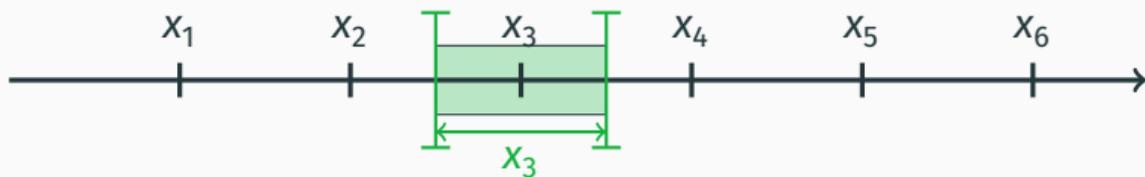
Comment représenter un nombre décimal

Les rationnels sont un **ensemble « dense »**, et la représentation choisie ne comptera nécessairement qu'un nombre limité d'éléments $\{x_1, x_2, \dots, x_n\}$.



Comment représenter un nombre décimal

Les rationnels sont un **ensemble « dense »**, et la représentation choisie ne comptera nécessairement qu'un nombre limité d'éléments $\{x_1, x_2, \dots, x_n\}$.



Chaque élément x_i doit donc être utilisé pour représenter **l'ensemble des nombres de l'intervalle** $[x_i - \delta_i, x_i + \delta_i]$.

Choisir une représentation pour les décimaux

Choisir une représentation, c'est choisir les x_i et les δ_i :

- Quelle **taille** (en bits) donner à la partie entière ? à la partie fractionnaire ?
- Comment concilier **précision** de la représentation et **amplitude** de la plage des nombres représentables ?
- La représentation garantit-elle l'**unicité** de la représentation des nombres ?
- La représentation permet-elle une réalisation matérielle **efficace** (temps, surface de silicium, énergie) des fonctions arithmétiques de base ?

Choisir une représentation pour les décimaux

Choisir une représentation, c'est choisir les x_i et les δ_i :

- Quelle **taille** (en bits) donner à la partie entière ? à la partie fractionnaire ?
- Comment concilier **précision** de la représentation et **amplitude** de la plage des nombres représentables ?
- La représentation garantit-elle l'**unicité** de la représentation des nombres ?
- La représentation permet-elle une réalisation matérielle **efficace** (temps, surface de silicium, énergie) des fonctions arithmétiques de base ?

Deux grandes approches sont possibles : la représentation **à virgule fixe**, et la représentation **à virgule flottante**.

Représentation à virgule fixe

Représentation à virgule fixe

Le nombre de bit de la partie entière et celui de la partie fractionnaire sont fixés une fois pour toute.

Avantages

- Utilisation des opérateurs d'arithmétique entière :
 - Pas d'unité de calcul dédiée.
 - Performances élevées.
- Assez simple d'estimer l'erreur d'un calcul complexe.
- Permet d'optimiser le format en fonction de la précision et de l'amplitude ciblée pour un cas particulier.

Inconvénients

- Solution spécialisée par cas d'usage → s'oppose à la logique « *general purpose computer* »

Exemple : le format $Q_{m.n}$

Le format $Q_{m.n}$

Codage qui utilise la représentation à virgule fixe : le nombre est codé sous la forme d'un entier en complément à 2 sur $m + n$ bits, qui est ensuite divisé par 2^n .

Exemple : le format $Q_{4.12}$

$$0x\text{CAFE}_{Q_{4.12}} = \frac{-13\,570}{2^{12}} = -3,312\,988\,281\,25$$

Le format $Q_{4.12}$ a les caractéristiques suivantes :

- Mot de 16 bits, dont 12 pour la partie fractionnaire
- Valeur minimale : $\frac{-2^{15}}{2^{12}} = -2^3 = -8$ (codage : $0x8000$)
- Valeur maximale : $\frac{2^{15}-1}{2^{12}} = 2^3 - \frac{1}{2^{12}} = 7,99975585938$ (codage : $0x7FFF$)
- Précision : $\frac{1}{2^{12}} = \frac{1}{4096} = 0,000244140625$

Représentation à virgule flottante

Représentation à virgule flottante

La position de la virgule n'est pas fixe : reprend le principe de la notation scientifique, avec un nombre fixe de chiffre significatifs et une plage limitée d'exposants.

Avantages

Allie précision importante pour les petits nombres, et capacité à représenter l'ordre de grandeur des grands nombres → s'inscrit dans la logique « *general purpose computer* ».

Inconvénients

Les calculs sont plus complexes qu'en virgule fixe :

- Nécessite l'utilisation d'unité de calcul dédiée → les FPU (pour *Floating Point Unit*)
- Performance moins bonne qu'en virgule fixe.
- Difficile d'évaluer les erreurs de calculs induites par l'approximation.

Rappel : notation scientifique

La notation scientifique du décimal x est définie par le triplet (s, m, e) tel que

$$x = (-1)^s \times m \times 10^e$$

avec $s \in \{0, 1\}$, $m \in [1, 10[$ et $e \in \mathbb{Z}$.

Exemple

$$-12.375 = (-1)^1 \times 1.2375 \times 10^1$$

$$123\,456\,789 = (-1)^0 \times 1.23456789 \times 10^8$$

Notation scientifique en base 2

La notation scientifique est indépendante de la base, et peut donc en particulier être instanciée **en base 2** :

$$x = (-1)^s \times m \times 2^e$$

avec $s \in \{0, 1\}$, $m \in [1, 2[$ et $e \in \mathbb{Z}$.

Exemple

$$-12.375_{10} = -1100,011_2 = (-1)^1 \times 1,100011_2 \times 2^3$$

$$123\,456\,789 = 111010110111100110100010101_2 = (-1)^0 \times 1,11010110111100110100010101_2 \times 2^{26}$$

La suite, ce n'est que de la syntaxe!

Les représentations à virgule flottante codent le triplet (s, m, e) de la notation scientifique en base 2.

Le format IEEE 754

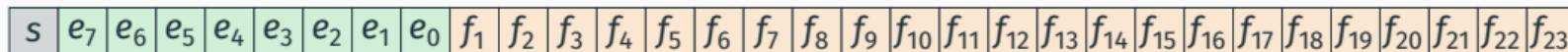
L'IEEE (*Institute of Electrical and Electronics Engineers*) a proposé un format standardisé pour la représentation binaire en virgule flottante, qu'on appelle communément IEEE 754.

Le format définit plusieurs précisions.

Les langages et processeurs modernes utilisent le format *simple precision* sur 32 bits et *double precision* sur 64 bits.

Ces formats correspondent respectivement aux types `float32` et `float64` de golang.

Le format IEEE 754 simple précision



$$x = (-1)^s \times \left(1 + \sum_{i=1}^{23} f_i \times 2^{-i}\right) \times 2^{\left(\sum_{i=0}^7 e_i \times 2^i\right) - 127}$$

- La partie entière est **toujours** égale à 1 et n'est pas codée.
- Les 22 premiers bits après la virgule sont codés et **le 23ème est un arrondi**.
- L'exposant est codé avec un biais de 127 (plage de valeur : [-126, 127] car les valeurs **0x00** et **0xFF** sont réservées)
- La précision (= poids le plus faible) varie avec l'exposant : de 2^{-149} à 2^{104} .

Exemple

On veut coder 1234,5

1. $1234,5 = 10011010010,1_2$

Exemple

On veut coder $1234,5$

1. $1234,5 = 10011010010,1_2$

2. $10011010010,1_2 = (-1)^0 + 1,00110100101_2 \times 2^{10}$

Exemple

On veut coder 1234,5

1. $1234,5 = 10011010010,1_2$

2. $10011010010,1_2 = (-1)^0 + 1,00110100101_2 \times 2^{10}$

3. $(-1)^0 + 1,00110100101_2 \times 2^{10} = (-1)^0 + 1 + 0,00110100101_2 \times 2^{137-127}$

Exemple

On veut coder 1234,5

1. $1234,5 = 10011010010,1_2$

2. $10011010010,1_2 = (-1)^0 + 1,00110100101_2 \times 2^{10}$

3. $(-1)^0 + 1,00110100101_2 \times 2^{10} = (-1)^0 + 1 + 0,00110100101_2 \times 2^{137-127}$

4. $s = 0, e = 10001001, f = 001101001010000000000000$

Exemple

On veut coder 1234,5

1. $1234,5 = 10011010010,1_2$
2. $10011010010,1_2 = (-1)^0 + 1,00110100101_2 \times 2^{10}$
3. $(-1)^0 + 1,00110100101_2 \times 2^{10} = (-1)^0 + 1 + 0,00110100101_2 \times 2^{137-127}$
4. $s = 0, e = 10001001, f = 001101001010000000000000$
5. $1234,5 = 01000100100110100101000000000000_{ieee754sp}$

Exemple

On veut coder 1234,5

1. $1234,5 = 10011010010,1_2$
2. $10011010010,1_2 = (-1)^0 + 1,00110100101_2 \times 2^{10}$
3. $(-1)^0 + 1,00110100101_2 \times 2^{10} = (-1)^0 + 1 + 0,00110100101_2 \times 2^{137-127}$
4. $s = 0, e = 10001001, f = 001101001010000000000000$
5. $1234,5 = 01000100100110100101000000000000_{ieee754sp}$
6. $1234,5 = 0x449A5000$

Les valeurs dénormalisées

Pour augmenter la densité de valeurs autour de 0, le codage change quand $e = 0x00$:

$$x = (-1)^s \times \left(0 + \sum_{i=1}^{23} f_i \times 2^{-i} \right) \times 2^{-126}$$

Ce codage permet de coder les valeurs $+0$ et -0 .

Par ailleurs, $e = 0xFF$ est réservé pour les valeurs spéciales :

- NaN pour *Not a Number*, quand $f \neq 0$
- $+\infty$, quand $s = 0$ et $f = 0$
- $-\infty$, quand $s = 1$ et $f = 0$

Le format double précision

Le format IEEE754 double précision utilise 64 bits :

- un bit de signe
- 11 bit d'exposant (biais : **-1023**)
- 52 bit de partie fractionnaire (dans l'écriture scientifique en base 2)

Il fonctionne exactement comme le format simple précision. En particuliers, les valeurs spéciales et les valeurs dénormalisées sont codées avec les mêmes conventions.

On veut effectuer une opération arithmétique binaire entre deux nombres IEEE754, x_1 et x_2 . On suppose que $e_2 < e_1$. Pour effectuer l'opération, il faut :

1. « ré-écrire » x_2 sous la forme $(-1)^{s_2} \times 2^{e_1-127} \times (f'_2) \rightarrow f'_2 = (1 + f_2) \div 2^{e_1-e_2}$
2. calculer l'opération : $f_s = op(f_1, f'_2)$
3. re-normaliser le nombre $(-1)^{s_1 \times s_2} \times 2^{e_1-127} \times f_s$

Lors de l'étape 1, tous les bits significatifs de x_2 peuvent disparaître : on dit que x_2 est absorbé par x_1 . De ce fait, l'arithmétique IEEE754 n'a pas toutes les propriétés usuelles !

Arithmétique IEEE754 : propriétés élémentaires

Les propriétés ci-dessous portent sur *FP* l'ensemble des valeurs codables avec le standard IEEE754.

Propriétés conservées :

- **0** élément neutre pour +, absorbant pour \times .
- **1** élément neutre pour \times .
- $\forall x, x - x = 0$
- \times et + sont commutatives : $a + b = b + a$ et $a \times b = b \times a$.

Propriétés non conservées

- Dans FP, + et \times **ne sont pas associatives** : $(a + b) + c \neq a + (b + c)$ et $(a \times b) \times c \neq a \times (b \times c)$
- Dans FP, \times **ne se distribue pas** sur + : $a \times (b + c) \neq (a \times b) + (a \times c)$
- $\exists x, x \times \frac{1}{x} \neq 1$
- $\exists x, \exists y, x \neq 0 \wedge y \neq 0 \wedge x + y = x$
- ...