Architecture des Ordinateurs

Chapitre 4 : Systèmes de nombres

Jacques Supcik, Daniel Gachet, Luca Haab

2023-11-30

Table des matières

4 Systèmes de nombres

Les ordinateurs utilisent des portes logiques binaires et représentent l'information uniquement avec des suites de zéros et de uns. Dans ce chapitre, nous étudions comment les ordinateurs combinent ces zéros et ces uns pour représenter des nombres.

4.1 Les nombres entiers

Les nombres les plus simples sont les nombres entiers, en particulier les nombres entiers positifs. En mathématique, on parle d'*ensemble de nombres* et les entiers positifs sont dans l'ensemble des *entiers naturels* $\mathbb{N}$.

Nous avons l'habitude de représenter les nombres avec le système *décimal*, c'est-à-dire que nous utilisons des suites de chiffres entre zéro (0) et neuf (9) pour représenter ces nombres. On dit que le système *décimal* est un système en base 10.

Quand nous écrivons la suite de chiffres 123, le 1 représente les centaines (10^2), le 2 représente les dizaines (10^1) et le 3 représente les unités (10^0).

De manière générale, si un nombre en base 10 est représenté par la suite de n chiffres :

$$a_{n-1}a_{n-2}...a_1a_0 \ (0 \leq a_i < 10)$$

alors la valeur du nombre N est :

$$N = \sum_{i=0}^{n-1} a_i 10^i$$

- a_0 est le chiffre le plus à droite et on l'appelle le chiffre de poids faible (*LSD : Least Significant Digit*). Ce chiffre représente aussi les *unités* du nombre.
- a_{n-1} est le chiffre le plus à gauche et on l'appelle le chiffre de poids fort (*MSD : Most Significant Digit*).
- n est le nombre de chiffres (*Digit*).

Le choix du système décimal (base 10) vient du fait que nous utilisons nos doigts pour compter et que nous en avons 10. Mais il est tout à fait possible de représenter des nombres avec une autre base. La civilisation Maya ainsi que les langues celtiques utilisaient la base 20 (Système vicésimal) et l'on en trouve aujourd'hui encore des traces dans le *quatre vingts* des Français.

On a vu plus haut que les ordinateurs n'ont que *deux états* (0 ou 1) pour représenter les nombres et c'est donc tout naturellement que les ordinateurs utilisent un système binaire (ou base 2).

un nombre en base 2 est représenté par la suite de n "bits" :

$$a_{n-1}a_{n-2}...a_1a_0 \ (0 \leq a_i < 2)$$

et sa valeur est :

$$N = \sum_{i=0}^{n-1} a_i 2^i$$

Pour éviter toute confusion, on indique la base d'un nombre à l'aide d'un indice. Par exemple 1101_2 signifie le nombre 1101 en base 2, soit le nombre 13 en base 10. Pour plus de clarté, on peut aussi préciser la base 10 avec un indice (13_{10}), mais si on n'indique rien, alors on utilise la base 10.

Écrire un nombre en binaire peut s'avérer long et fastidieux. Par exemple $1\,500\,000\,000$ devient $1011001011010000010111100000000_2$ et ce n'est pas très pratique à écrire. Pour remédier à ça, les informaticiens utilisent souvent le système *héxadécimal* ou base 16. Comme 16 est une puissance de 2 ($2^4 = 16$), un "chiffre" de la base 16 correspond à 4 bits de la base 2. On peut donc facilement passer de la base 2 à la base 16 en faisant des groupes de 4 bits. En reprenant l'exemple précédent, $1\,500\,000\,000$ est égal à $101\ 1001\ 0110\ 1000\ 0010\ 1111\ 0000\ 0000_2$

Chaque groupe de 4 bits correspond à un "chiffre" en base 16 selon la table suivante :

Groupe de 4 bits	"Chiffre" en base 16	Valeur en base 10
0000_2	0_{16}	0_{10}
0001_2	1_{16}	1_{10}
0010_2	2_{16}	2_{10}
0011_2	3_{16}	3_{10}
0100_2	4_{16}	4_{10}
0101_2	5_{16}	5_{10}
0110_2	6_{16}	6_{10}

Groupe de 4 bits	“Chiffre” en base 16	Valeur en base 10
0111_2	7_{16}	7_{10}
1000_2	8_{16}	8_{10}
1001_2	9_{16}	9_{10}
1010_2	A_{16}	10_{10}
1011_2	B_{16}	11_{10}
1100_2	C_{16}	12_{10}
1101_2	D_{16}	13_{10}
1110_2	E_{16}	14_{10}
1111_2	F_{16}	15_{10}

La représentation hexadécimale de $1\,500\,000\,000$ est donc $5968\,2\mathrm{F}00_{16}$.

Au lieu de regrouper les bits par groupe de 4, on peut les regrouper par groupe de 3 et on obtient ainsi la base 8 ou l'*octal*. L'intérêt de la base 8 est de pouvoir représenter les nombres avec les chiffres de 0 et 7 et de ne pas avoir besoin de lettres. Cette notation n'est cependant plus beaucoup utilisée aujourd'hui.

L'ensemble des nombres naturels peut être étendu avec les nombres négatifs. En mathématique on parle d'ensemble d'entiers relatifs et on le note $\mathbb{Z}$. Un nombre négatif se distingue avec le symbole “$-$” placé devant le nombre.

4.2 Représentations des nombres entiers

Comme nous l'avons vu plus haut, l'ordinateur encode les nombres en binaires, mais contrairement aux mathématiques qui s'accommodent de nombres infiniment grands, les ordinateurs ont un espace limité pour représenter les nombres. Ils ont également des contraintes dues à l'architecture de leur mémoire. En effet, la mémoire des ordinateurs est organisée en octets (8 bits) et on utilise un ou plusieurs octets pour représenter l'information.

Les ordinateurs actuels permettent de représenter et de traiter des nombres sur 64 bits, (8 octets). Cette taille nous permet de représenter 1.8×10^{19} nombres différents. On peut donc représenter les nombres entre 0 et 1.8×10^{19} si on se contente de nombres positifs ou alors entre -9.2×10^{18} et $+9.2 \times 10^{18}$ si on souhaite des nombres signés.

Si on n'a pas besoin de nombres aussi grands, on peut représenter les nombres sur 32 bits (4 octets) Cette taille nous permet de représenter les entiers positifs entre 0 et $4\,294\,967\,297$ ou les entiers signés entre $-2\,147\,483\,648$ et $+2\,147\,483\,647$. Notez que 32 bits c'est aussi la taille "standard" pour les plus petits ordinateurs de type "systèmes embarqués".

4.3 Addition binaire

L'addition de deux nombres binaires est semblable à l'addition en base 10. On commence par additionner les bits de poids faibles. Si les deux bits sont à 1, le résultat dépasse la capacité d'un bit et on a alors une retenue qu'on ajoutera au bit suivant. Cette retenue s'appelle *carry* en anglais. Le tableau ci-dessous représente l'addition de deux bits A et B et donne les valeurs de la somme (*Sum*) et de la retenue (*carry*)

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Cette opération est bien connue et le circuit logique qui l'implémente et le demi-additionneur ou le *half-adder* en anglais (figure 4.1).

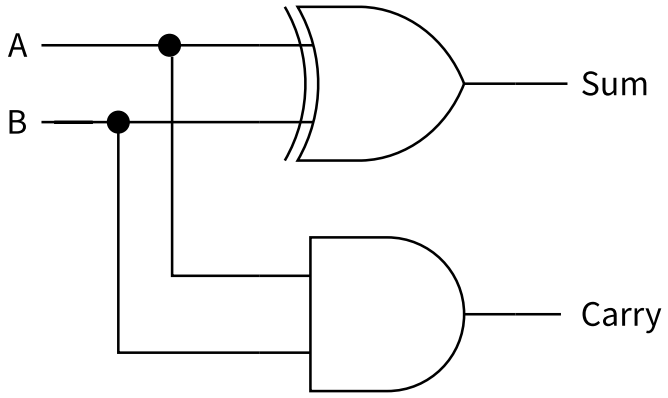

FIGURE 4.1 – Half-adder

A partir du deuxième bit, on doit tenir compte de la retenue. Nous distinguons la retenue entrante (résultant de l'addition du bit précédent) que nous appelons *carry in* et de la retenue sortante (résultant

de l'addition du bit courant) que nous appelons *carry out*.

Le tableau ci-dessous représente la table de vérité de cette opération :

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Le circuit qui implémente cette opération est l'additionneur complet ou *full-adder* (figure 4.2).

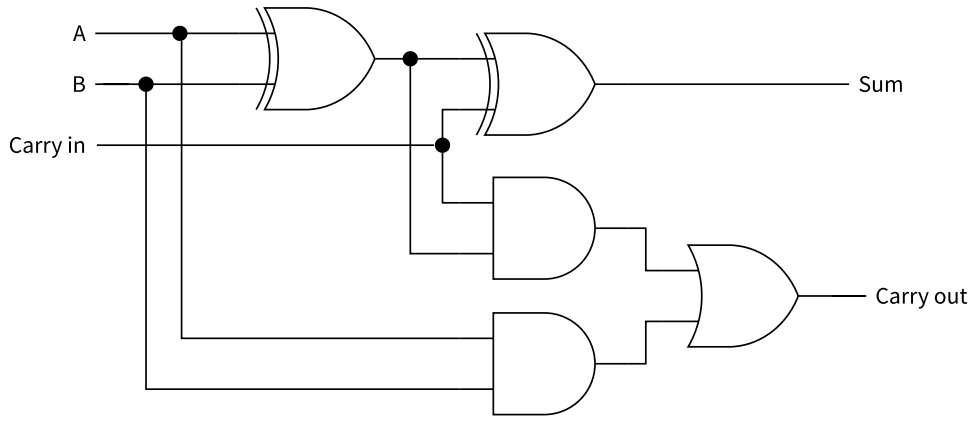

FIGURE 4.2 – Full-adder

Pour additionner deux nombres de 32 bits, il suffit de chaîner 32 additionneurs complets et de laisser entrer un 0 comme *carry in* dans l'additionneur de la première addition. Si le *carry out* de la dernière addition est à 1, ça signifie que le résultat dépasse la capacité du mot de 32 bits.

Les systèmes modernes utilisent une construction plus complexe qu'une suite d'additionneurs complets, car la propagation de la retenue d'un étage à l'autre (*ripple-carry*) ralentit l'addition. Ces systèmes plus performants s'appellent *carry-lookahead adder*, mais leur étude sort du cadre de ce cours.

On peut illustrer la retenue (*carry out*) avec la figure 4.3 qui représente tous les nombres non signés de 3 bits sur un cercle.

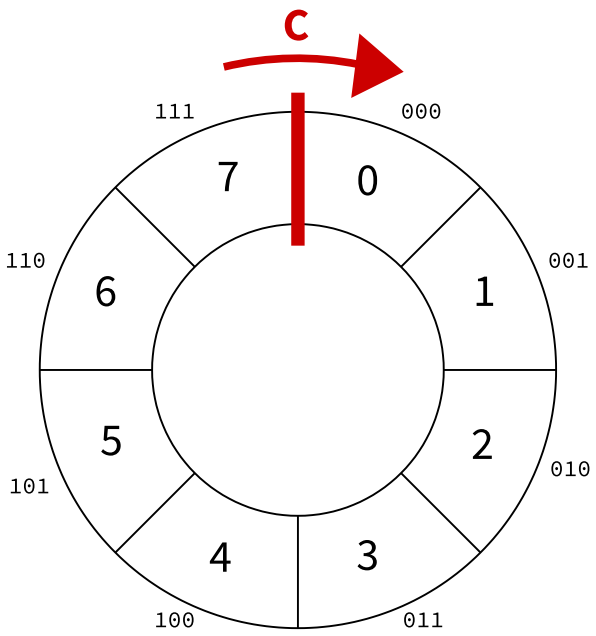

FIGURE 4.3 – Cercle des nombres non signées

Si l'on additionne deux de ces nombres et qu'on ne repasse pas par le zéro, alors la retenue est à 0 et le résultat est correct. Par exemple $3 + 2 = 5$, on ne repasse pas par zéro, la retenue est à 0 et le résultat est bien 5. Si par contre, on repasse par le zéro alors la retenue est à 1. Si on fait $4 + 5$, on repasse par le zéro, la retenue est à 1 et le résultat n'est pas 9 (car on ne peut pas le représenter avec 3 bits), mais il est tronqué à 1.

La retenue est sauvegardée dans un registre spécial du microprocesseur (le registre d'état), plus précisément dans le "flag" C (pour *Carry*) de ce registre.

4.4 Les nombres entiers signés

En mathématique, il existe un ensemble de nombres qui étend l'ensemble des nombres naturels avec des nombres négatifs. Cet ensemble est appelé l'ensemble des *entiers relatifs* $\mathbb{Z}$. Par rapport aux entiers non signés, on adjoint un signe à chaque entier signé. Le symbole $+$ signifie que le nombre est positif et le symbole $-$ signifie que le nombre est négatif. Le symbole $+$ peut être omis et dans ce cas le nombre est positif.

4.5 Nombres binaires signés

Les nombres binaires ne sont représentés que par des 0 et des 1 et nous n'avons pas la possibilité de représenter un *signe* $+$ ou $-$ comme nous le faisons en mathématiques. Une des premières idées qui nous vient pour représenter ce signe serait de détourner le bit de poids fort pour qu'il représente le signe. Par convention, on pourrait dire que si le bit de poids fort est à 0, le nombre est positif et que s'il est à 1, le nombre est négatif.

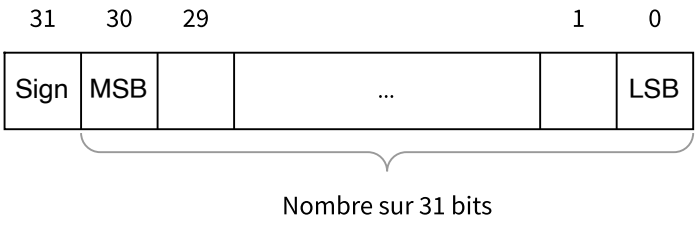

FIGURE 4.4 – Bit de signe

Nous verrons plus tard, dans un contexte bien précis, que la représentation du signe par un bit est en effet utilisée, mais dans le contexte des nombres entiers qui nous concerne actuellement, cette solution est plutôt une mauvaise idée :

- La première faiblesse de ce système c'est que nous avons deux représentations pour le nombre 0, une fois avec le bit de poids fort à zéro $(+0)$ et une fois avec le bit de poids fort à un (-0).
- La deuxième faiblesse c'est que nous ne pouvons pas réutiliser notre système de chaîne de *full adders* pour additionner deux nombres signés.

Il nous faut donc trouver une autre solution pour représenter les nombres binaires signés. Si on reprend le cercle des nombres et que nous remplissons avec les premiers nombres positifs, nous obtenons la figure 4.5.

Nous observons que lorsque nous tournons dans le sens des aiguilles d'une montre, nous incrémentons la valeur du nombre et lorsque nous tournons dans le sens contraire, nous décrémentons la valeur du nombre représenté. Pour compléter ce cercle avec les nombres négatifs, nous commençons

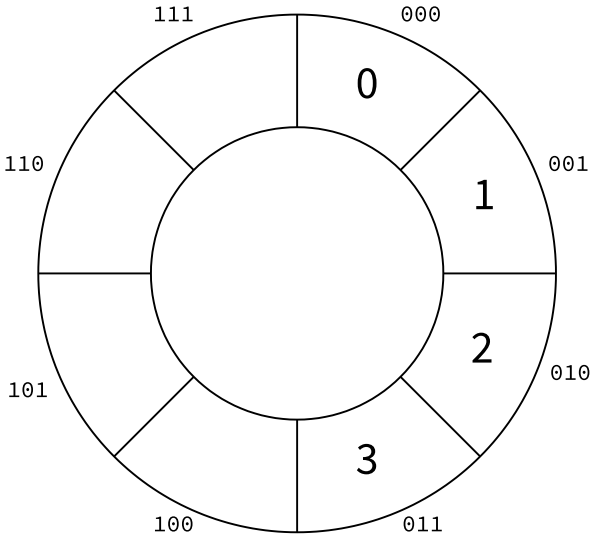

FIGURE 4.5 – Demi cercle des nombres signées

par le plus petit de ces nombres (-1) et nous le plaçons à gauche du 0. Nous complétons avec les nombres négatifs suivants et nous obtenons la figure 4.6.

Cette manière de faire offre de nombreux avantages :

- Le nombre zéro n'a qu'une seule représentation (tous les bits à 0).
- Les nombres positifs signés ont la même représentation que les nombres non signés.
- Le bit de poids fort nous permet toujours de savoir si un nombre est positif (MSB $= 0$) ou négatif (MSB $= 1$).
- On a bien partagé l'espace en deux parties égales pour les nombres positifs et les nombres négatifs. Comme on considère le 0 comme un nombre positif, le côté négatif, en valeur absolue, va plus loin que le côté positif, mais ça n'a pas vraiment d'importance.
- L'addition de deux nombres signés peut réutiliser le système de chaîne de *full adders*. Nous pouvons illustrer avec $-1 + 2$. En binaire, ça fait $111_2 + 010_2 = 001_2$ (avec une retenue à 1) et $001_2 = 1_{10}$ ce qui est bien le résultat de $-1 + 2$. On voit que la retenue (*carry*) telle que nous la calculons n'a plus vraiment de sens dans l'addition des nombres signés et nous y reviendrons plus tard.

Cette manière de coder les nombres entiers signés s'appelle le **complément à 2** [1] et c'est la méthode la plus souvent utilisée par les ordinateurs pour représenter les nombres binaires signés.

Si nous considérons des nombres sur N bits :

- les nombres entre 0 et $2^{N-1} - 1$ sont représentés comme les nombres non signés ;

1. [https ://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux]

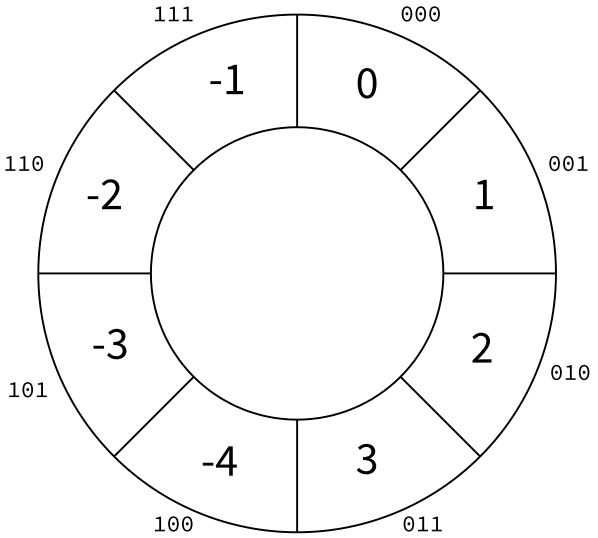

FIGURE 4.6 – Cercle des nombres signées

— pour représenter les nombres x entre -2^{N-1} et -1, on ajoute on biais de 2^N. On calcule $x' = x + 2^N$ et on représente x' comme un nombre non signé. Par exemple, avec un système sur 8 bits, pour représenter $x = -42$, nous calculons $x' = -42 + 2^8 = -42 + 256 = 214$ et $214_{10} = 11010110_2$ ce qui correspond à la représentation binaire de -42. Notez que le bit de poids fort et bien 1, ce qui est correct pour un nombre négatif.

Dans la technique ci-dessus, nous avons ajouté un biais de 2^N pour représenter les nombres négatifs. Que se passe-t-il si nous ajoutons également ce biais aux nombres positifs ?
Ajouter un biais de 2^N à un nombre positif sur N bits ne change pas la valeur de ce nombre. En effet, 2^N c'est 1 suivi de N zéros, et ce 1 est donc en dehors des N bits. On peut comparer ce 1 à la retenue (*carry*) et on peut dire que d'ajouter un biais de 2^N à un nombre positif met la retenue à 1, mais ça ne change pas la valeur du nombre.
On peut donc représenter **tous les nombres signées** en ajoutant un biais de 2^N. Si le nombre est négatif, la retenue est à 0 et si le nombre est positif, la retenue est à 1.

Nous pouvons reprendre cette idée de biais pour calculer l'*opposé* d'un nombre. Pour représenter $-x$, nous pouvons ajouter le biais de 2^N et calculer $2^N - x$. Avec cette technique -42 est représenté sur 8 bits $2^8 - 42 = 256 - 42 = 214$ et ça correspond bien à ce que nous avons écrit plus haut. Pour calculer $2^N - x$, le microprocesseur utilise une technique pour remplacer la soustraction par

une addition. Nous pouvons réécrire $2^N - x$ comme $(2^N - 1) - x + 1$, et $(2^N - 1) - x$ revient à inverser tous les bits de x (les 0 deviennent de 1 et les 1 deviennent des 0). Nous pouvons donc dire que $-x = 2^N - x = (2^N - 1) - x + 1 = \overline{x} + 1$. Autement dit, pour calculer l'opposé de x, il suffit d'inverser tous les bits de x et d'ajouter 1.

Illustrons cette technique avec $42_{10} = 101010_2$. Pour calculer -42, on inverse les bits de 42 et on ajoute 1 :

$$-42_{10} = \overline{00101010_2} + 1 = 11010101_2 + 1_2 = 11010110_2$$

Notez que l'opposé est une opération symétrique et l'opposé de -42 doit redonner 42 :

$$-(-42)_{10} = \overline{11010110_2} + 1_2 = 00101001_2 + 1_2 = 101010_2 = 42$$

Nous venons de voir deux techniques simples pour calculer la représentation d'un nombre négatif en complément à 2. L'opération inverse est simple elle aussi. Supposons que vous souhaitez trouver le nombre qui correspond à la représentation binaire 10101010_2 codée sur 8 bits. Le 1 en bit de poids fort indique que le nombre est négatif et la première technique consiste à soustraire 2^N : $x = 10101010_2 - 2^8_{10} = 170_{10} - 256_{10} = -86_{10}$. La réponse est donc -86_{10}. La deuxième technique est de calculer l'opposé du nombre en inversant les bits et en ajoutant 1 : $-x = \overline{10101010_2} + 1_2 = 1010101_2 + 1_2 = 1010110_2 = 86_{10}$, donc $x = -86_{10}$.

4.6 Addition binaire de nombres signés

Dans le chapitre précédent, nous avons vu que nous pouvions réutiliser le système de chaîne de *full adders* pour l'addition de nombres signés, mais nous devons encore gérer le problème de la retenue.

Reprenons notre cercle des nombres signés. Nous observons que la "rupture" n'est plus en haut comme avec les nombres non signés, mais en bas, lorsqu'on passe du grand nombre positif au plus petit nombre négatif et inversement.

Le résultat d'une addition provoque un dépassement de capacité (et donc un résultat faux) dans deux cas :

- si le résultat de l'addition de deux nombres positifs est plus grand que le plus grand nombre positif
- si le résultat de l'addition de deux nombres négatifs est plus petit que le plus petit nombre négatif

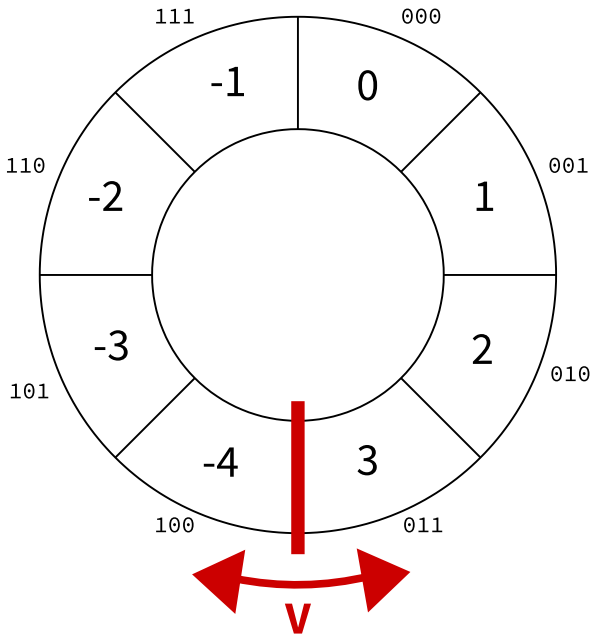

FIGURE 4.7 – Cercle des nombres signées avec overflow

Dans ces deux cas, le système notifie le dépassement de capacité en mettant le flag V (pour *oVerflow*) du registre d'état à 1.

Notez que le microprocesseur n'a qu'une seule opération pour faire une addition ; et il ne sait pas s'il additionne des nombres signés ou pas. On a vu que le résultat était le même et le microprocesseur calcule toujours les deux flags (C et V). C'est au programmeur de déterminer lequel de ces flags est relevant pour le résultat (C pour des entiers non signés et V pour des entiers signés).

4.7 Soustraction binaire

Pour effectuer une soustraction, le microprocesseur possède une opération spécifique, mais en interne, il utilise le même circuit que pour l'addition en inversant simplement le deuxième nombre (avec le complément à 2). Autrement dit, pour calculer $a - b$, le processeur calcule $a + (\overline{b} + 1)$. Par exemple $3 - 1$ est calculé comme $011_2 + \overline{001_2} + 1 = 011_2 + 110_2 + 001_2 = (1)010_2 = 2_{10}$. Notez que dans cet exemple le flag de retenue (C) est à 1, mais si on considère les nombres non signés, le résultat est bien correct. Si l'on reste avec des nombres non signés et qu'on calcule $2 - 3$, le microprocesseur calcule

$010_2 + \overline{011_2} + 1 = 010_2 + 100_2 + 001_2 = (0)111_2 = 7_{10}$. La retenue (C) est à 0 mais le résultat (7) est faux.

Le flag C doit donc être interprété différemment pour une soustraction de nombres non signés et on devrait plutôt parler d'*emprunt* plutôt que de *retenue*.

- Si le flag C est à 1, il n'y pas eu d'emprunt et le résultat est correct.
- Si le flag C est à 0, il y a eu un emprunt, ça correspond à un dépassement de capacité et le résultat est faux

Pour les nombres signés, c'est le flag V qui indique un dépassement de capacité et la logique est la même que pour l'addition :

- Si le flag V est à 0, le résultat est correct
- Si le flag V est à 1, il y a eu un dépassement de capacité et le résultat est faux

Notez que le flag V sera toujours 0 si les nombres sont de même signe.

4.8 Complément sur les flags

Pour mieux comprendre les flags (N, Z, C, V), nous pouvons les visualiser dans un graph.

4.8.1 Nombres non signés

La figure 4.8 représente le flag Z lors de l'addition de deux nombres non signés. Dans la figure l'origine $(0, 0)$ est en bas à gauche.

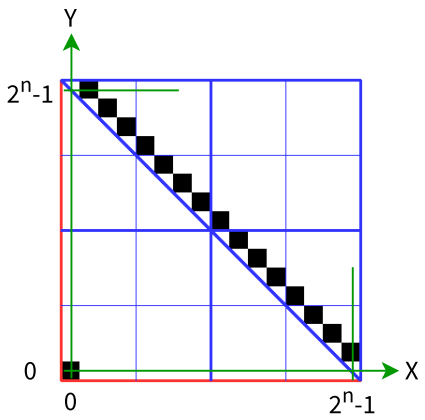

FIGURE 4.8 – Flag Z de l'addition X+Y

Pour rappel, le flag Z est à 1 si le résultat de l'addition est zéro et on constate que $0 + 0$ est bien égal à zéro (flag Z = 1). Les autres combinaisons de X et Y qui donnent zéro sont sur la diagonale et représentent les additions qui font un dépassement de capacité et qui reviennent à zéro. Par exemple, sur un système 8 bits, $1 + 255 = 256$ ou $2 + 254 = 256$. Le nombre 256 est trop grand pour être représenté sur 8 bits, le 9^{e}bit est perdu et le résultat est 0.

La figure 4.9 représente les 4 flags lors de l'addition et la soustraction de deux nombres non signés.

Pour le flag Z, on retrouve l'addition que nous venons de décrire. On voit aussi que la soustraction de deux nombres non signés donne zéro si les deux nombres sont égaux ($0 - 0 = 0$ ou $1 - 1 = 0$).

La deuxième ligne représente le flag C. Sur la figure 4.9, on voit bien que ce flag est à 1 si on additionne deux nombres trop grands pour être représentés sur les bits à disposition. Sur un système 8 bits, on peut encore additionner $127 + 127 = 254$ mais $128 + 128 = 256$ et 256 est trop grand pour être représenté sur 8 bits.

Notez que pour l'addition, si le flag Z est à un, alors le flag C est aussi à un, sauf pour $0 + 0$ où le flag Z est à un et le flag C est à zéro.

Pour la soustraction, le flag C est à un si l'opération ne fait pas de dépassement de capacité. On constate que C est à 1 si Y est plus petit ou égal à X. En effet, is $Y \leq X$, alors $X - Y \geq 0$ et le résultat de la soustraction est correct.

Pour être complète, la figure 4.9 représente aussi les flags N et V, mais ces flags n'ont pas beaucoup d'intérêt pour les nombres non signés.

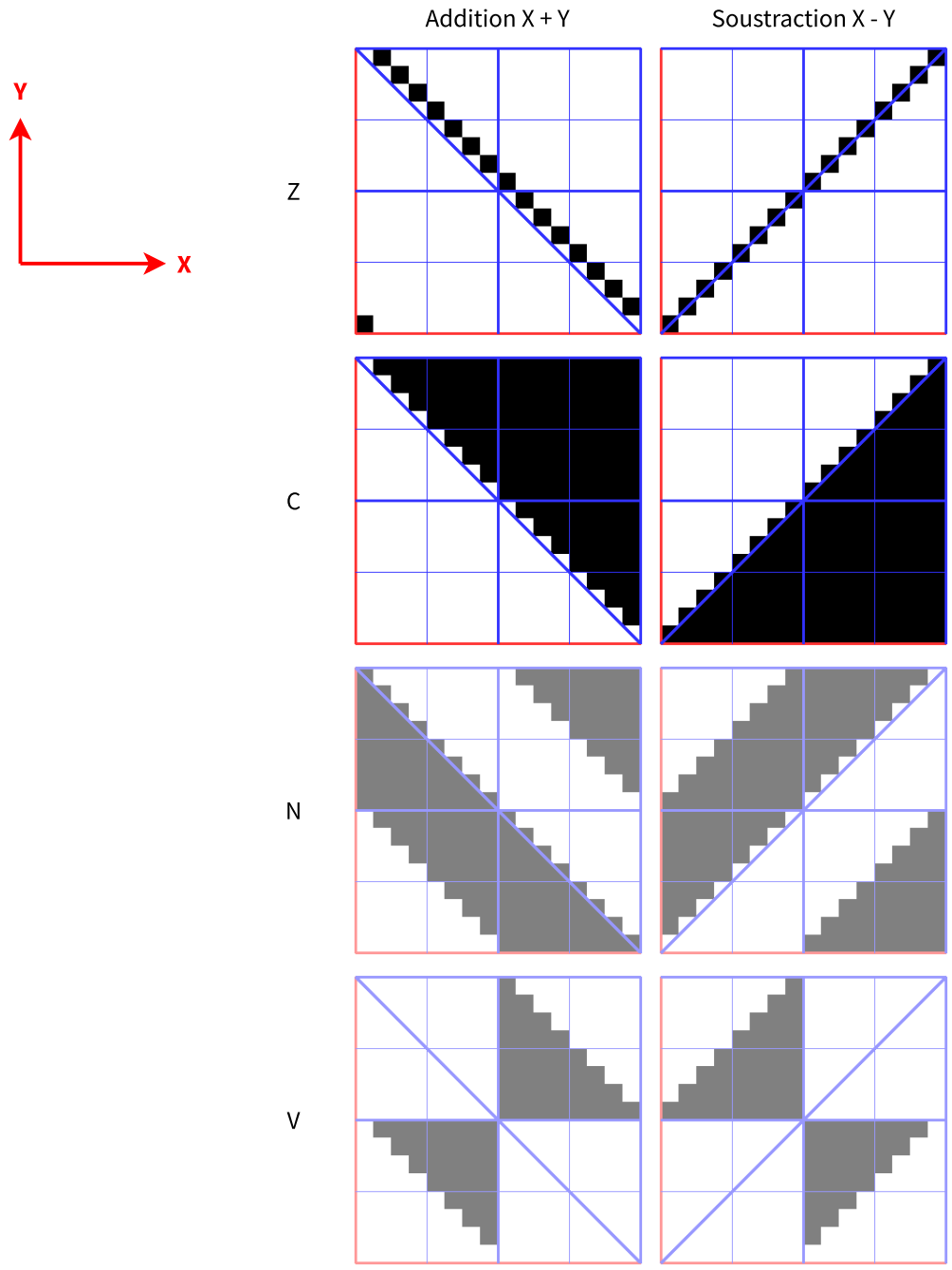

FIGURE 4.9 – Flags de l'addition et de la soustraction pour les nombres non signés

Pour terminer avec les nombres non signés, voici toutes les combinaisons possibles pour les flags N, Z, C et V lors de l'addition et de la soustraction de deux nombres sur 8 bits non signés.

N	Z	C	V	Exemple d'addition	soustraction
0	0	0	0	$32 + 32$	$32 - 224$
0	0	0	1	impossible	impossible
0	0	1	0	$96 + 224$	$224 - 160$
0	0	1	1	$160 + 160$	$160 - 96$
0	1	0	0	$0 + 0$	impossible
0	1	0	1	impossible	impossible
0	1	1	0	$32 + 224$	$64 - 64$
0	1	1	1	$128 + 128$	impossible
1	0	0	0	$32 + 160$	$160 - 224$
1	0	0	1	$96 + 96$	$96 - 160$
1	0	1	0	$224 + 224$	$224 - 32$
1	0	1	1	impossible	impossible
1	1	X	X	impossible	impossible

4.8.2 Nombres signés

Pour les nombres signés, nous déplaçons le référentiel pour que $(0, 0)$ soit au centre du graph. La figure 4.10 représente le flag Z lors de l'addition de deux nombres signés.

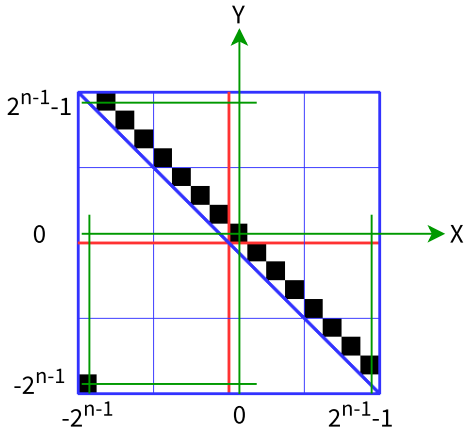

FIGURE 4.10 – Flag Z de l'addition X+Y pour les nombres signés

Notez au passage que malgré le changement de référentiel, l'image est strictement identique à celle que nous avons pour l'addition de nombres non signés.

La figure 4.11 représente les 4 flags lors de l'addition et la soustraction de deux nombres signés.

L'explication pour le flag Z est la même que pour les nombres non signés.

Pour rappel, le flag N est à un quand le nombre est négatif. Pour l'addition, le résultat est négatif quand au moins un des deux nombres est négatif et que le résultat est lui aussi négatif. C'est le cas par exemple pour $1 + (-2) = -1$, $(-2) + 1 = -1$ ou $(-1) + (-1) = -2$. Mathématiquement parlant, le résultat est négatif dans toute la zone en dessous de la diagonale, mais à cause du nombre de bits limité, on a un dépassement de capacité dans le coin en haut à droite et dans le coin en bas à gauche. Dans ces deux coins, on a un dépassement de capacité et le signe du résultat est faux. Ce dépassement de capacité est indiqué par le flag V qui est à un dans ces deux coins.

Pour la soustraction, le résultat est négatif si X est plus petit que Y. Mathématiquement parlant, le résultat est négatif dans toute la zone en dessus de la diagonale, mais comme pour l'addition, on a un dépassement de capacité dans le coin en haut à gauche et en bas à droite. Le flag V est à un dans ces deux coins pour indiquer le dépassement de capacité.

La figure 4.11 montre que le flag V est toujours à zéro lorsqu'on additionne deux nombres de signes opposés ou si on soustrait deux nombres de même signe.

Pour être complète, la figure 4.11 représente aussi le flag C, mais ce flag n'a pas beaucoup d'intérêt

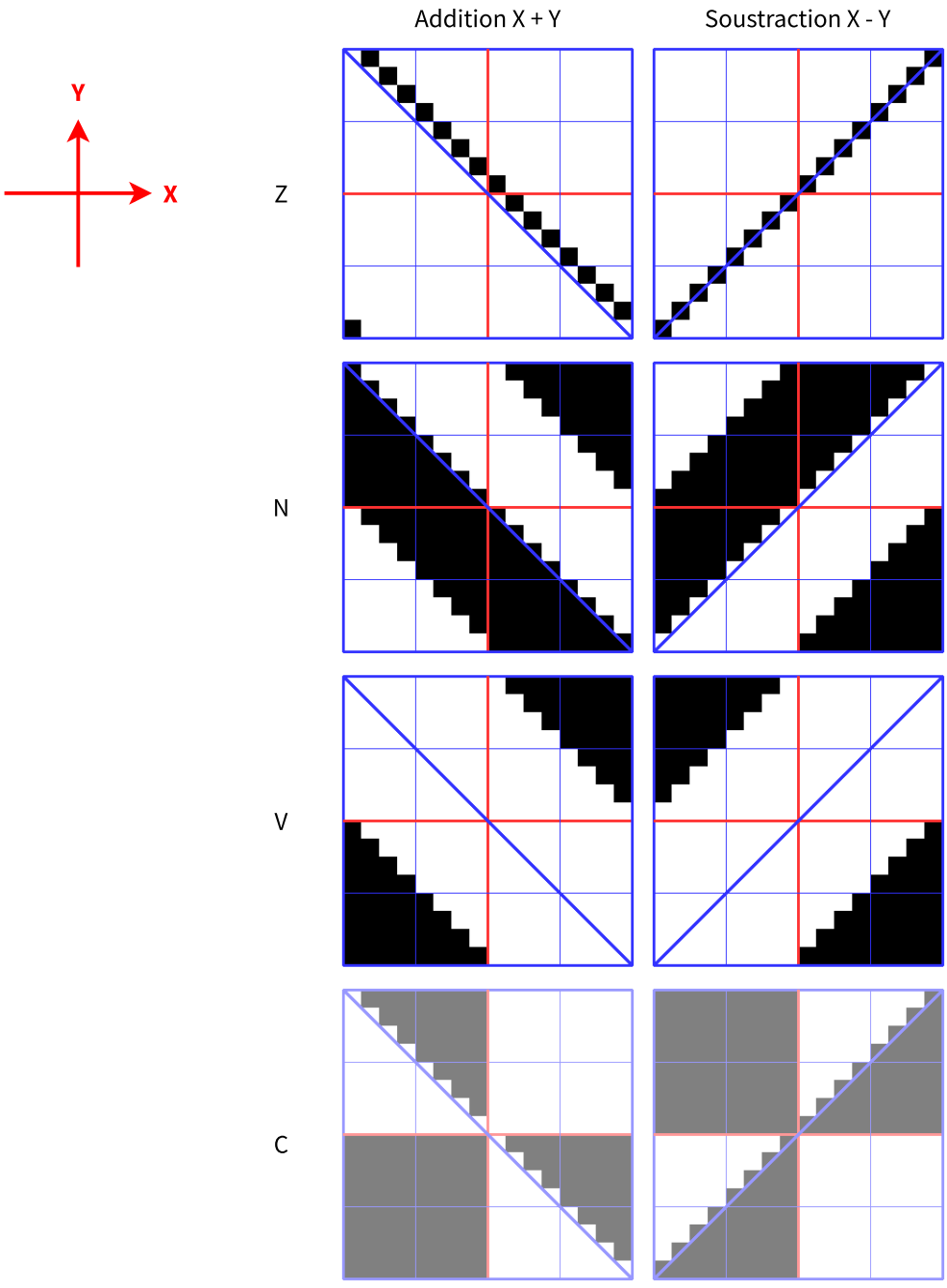

FIGURE 4.11 – Flags de l'addition et de la soustraction pour les nombres signés

pour les nombres signés.

Pour terminer avec les nombres signés, voici toutes les combinaisons possibles pour les flags N, Z, C et V lors de l'addition et de la soustraction de deux nombres sur 8 bits non signés.

N	Z	C	V	Exemple d'addition	soustraction
0	0	0	0	$32 + 32$	$32 - (-32)$
0	0	0	1	impossible	impossible
0	0	1	0	$96 + (-32)$	$(-32) - (-96)$
0	0	1	1	$(-96) + (-96)$	$(-96) - 96$
0	1	0	0	$0 + 0$	impossible
0	1	0	1	impossible	impossible
0	1	1	0	$32 + (-32)$	$64 - 64$
0	1	1	1	$(-128) + (-128)$	impossible
1	0	0	0	$32 + (-96)$	$(-96) - (-32)$
1	0	0	1	$96 + 96$	$96 - (-96)$
1	0	1	0	$(-32) + (-32)$	$(-32) - 32$
1	0	1	1	impossible	impossible
1	1	X	X	impossible	impossible

4.9 Comparaison

Le mécanisme des flags que nous avons vu nous permettent de comparer deux nombres et d'implémenter les opérations suivantes :

- Egal : ==
- Différent : ! =
- Plus petit que : <
- Plus petit ou égal : <=
- Plus grand que : >
- Plus grand ou égal : >=

Pour comparer deux nombres, le microprocesseur effectue simplement une soustraction. Il ignore le résultat et ne considère que les flags (N, Z, C, V).

$$a \,\text{op}\, b \Rightarrow a - b \,\text{op}\, 0$$

Pour l'égalité "$a = b$", on calcule $a - b$ et on regarde le flag Z (zéro). Si le flag Z est à 1, alors $a - b = 0$, donc $a = b$. Si le flag Z est à 0, alors $a \neq b$.

Pour tester l'égalité, nous n'avons pas besoin de faire de différence entre les nombres signés et non signés.

Pour savoir si $a < b$, on calcule $a - b$. Si le résultat est négatif, ça veut dire que a est bien plus petit que b. Si par contre le résultat est positif (ou zéro), c'est que a est plus grand ou égal à b.

Nous devons considérer les nombres signées et les nombres non signées de manière différentes. Pour des nombres non signés, nous ne pouvons pas vraiment dire que le résultat est "négatif" ou "positif", mais nous pouvons savoir si il y a un un dépassement de capacité avec le flag C. Si C est à 0, c'est qu'il y a eu un *underflow* et ça veut donc dire que $a < b$.

Pour les nombres signés, nous considérons les flags N et V. Si le résultat de la soustraction est négatif ($N = 1$), et qu'il n'y a pas eu de dépassement de capacité ($V = 0$), alors nous pouvons dire que $a < b$. De même, si le résultat est positif ($N = 0$) mais qu'il y a eu un dépassement de capacité ($V = 1$), ca veut dire que le résultat devrait plutôt être négatif et donc aussi que $a < b$.

Le tableau ci-dessous résume comment sont évaluées les différentes comparaisons avec des nombres non signés et des nombres signés :

Comparaison	Pour A et B **non signés**	Pour A et B **signés**
$A = B$	Z	Z
$A \neq B$	$\overline{Z}$	$\overline{Z}$
$A < B$	$\overline{C}$	$N \oplus V$
$A \leq B$	$\overline{C} \vee Z$	$(N \oplus V) \vee Z$
$A > B$	$C \wedge \overline{Z}$	$(N = V) \wedge \overline{Z}$
$A \geq B$	C	$N = V$

4.10 Nombres en virgules flottantes

En plus des nombres naturels ($\mathbb{N}$) et des nombres relatifs ($\mathbb{Z}$) il y a les nombres rationnels ($\mathbb{Q}$) et les nombres réels ($\mathbb{R}$). Ces deux derniers ensembles contiennent des nombres qui ne sont forcément entiers et on les représente souvent avec une virgule (ou un point décimal).

Les nombres rationnels ($\mathbb{Q}$) peuvent tous être représentés par une fraction de deux nombres relatifs et il existe des fonctions informatiques qui utilise cette représentation, mais nous n'aborderons pas ce point dans ce cours.

Dans l'arithmétique traditionnelle en base 10, le nombre 123.45 peut se lire comme :

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

C'est juste une extension des nombres entiers avec des puissances de 10 négatives dès qu'on passe la virgule. Pour la représentation binaire, nous pouvons transposer cette idée en utilisant des puissances de 2 à la place des puissances de 10. Le nombre 1010.11_2 peut se lire comme :

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 1^{-2}$$

ce qui donne :

$$8 + 2 + \frac{1}{2} + \frac{1}{4} = 10.75_{10}$$

Comme pour les entiers, nous devons maintenant faire tenir ces nombres dans des cases mémoires des ordinateurs. Supposons que nous souhaitions faire tenir ces nombres sur 3 bits. Une première idée consiste à *fixer* la position de la virgule, par exemple sur le dernier bit du nombre. Il nous reste donc 2 bits pour coder le nombre et on peut représenter les nombres $0.0, 0.5, 1.0, 1.5 \ldots 3.5$:

Si on souhaite également représenter les nombres négatifs, on peut reprendre le concept du complément à deux comme illustré sur la figure 4.13.

La représentation en virgule fixe est certes simple, mais elle limite beaucoup l'éventail des valeurs que nous pouvons représenter et nous ne pouvons ni représenter de très petits nombres, ni de très grands. Les mathématiciens, les scientifiques et les ingénieurs étaient aussi confrontés à cette problématique et ils ont inventé la notation **scientifique**. Dans cette notation, un nombre est représenté par :

$$\pm m \cdot 10^e$$

Le nombre m est appelé la *mantisse* et sa valeur est entre 1 et 10 non compris ($1 \leq m < 10$). Le nombre e est appelé l'*exposant*.

Grâce à cette notation, nous pouvons représenter de très petits nombres : Le rayon d'un atome d'hydrogène mesure $0.000000000053\,\mathrm{m}$ mais c'est plus simple d'écrire $5.3 \cdot 10^{-11}\,\mathrm{m}$. Idem pour de très

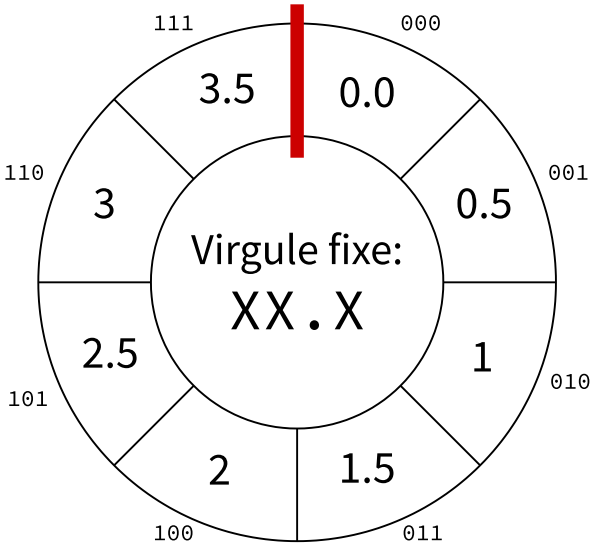

FIGURE 4.12 – Cercle des nombres en virgule fixe

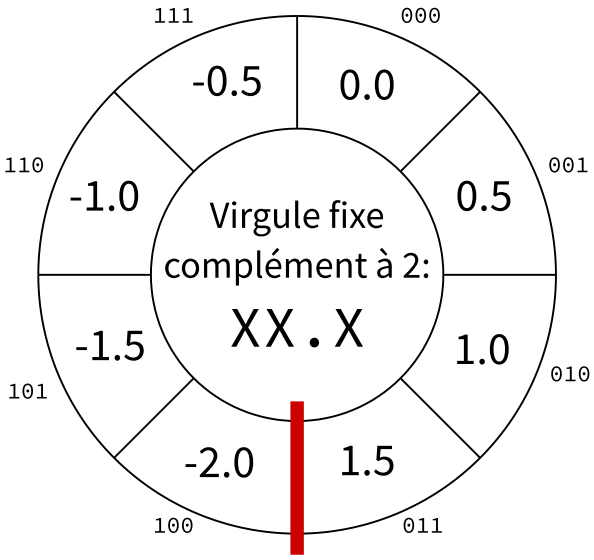

FIGURE 4.13 – Cercle des nombres en virgule fixe (complément à deux)

grands nombres : La distance entre la terre et le soleil est d'environ $150000000000\,\mathrm{m}$, mais c'est plus simple d'écrire $1.5 \cdot 10^{11}\,\mathrm{m}$.

Dans les ordinateurs, nous nous sommes inspiré de cette notation pour représenter les nombres en *virgules flottantes* (ou *floating point*). Nous utilisons naturellement des puissances de 2 à la place des puissances de 10, mais l'idée et la même. Par exemple, on peut représenter le nombre 1011000000000000_2 par $1.011_2 \cdot 2^{15}$ (en base 10, ça fait $45056 = 1.375 \cdot 2^{15}$).

Tous les nombres à virgules peuvent donc être représentés par :

$$\pm m \cdot 2^e \text{ avec } 1 \leq m < 2$$

Nous devons connaître :

— le signe du nombre ;
— la mantisse entre 1 et 2 non compris avec la précision souhaitée ;
— l'exposant qui peut être positif ou négatif.

Nous pouvons illustrer avec un exemple :

$42.5 = +1.328125 \cdot 2^5$. Le signe est positif, la mantisse est 1.328125 (ou 1.010101_2 en base 2) et l'exposant est 5.

Pour certains nombres, c'est plus difficile. Par exemple $0.1 \approx 1.600000023841858 \ldots 2^{-4}$. Le signe et l'exposant sont définis, mais la mantisse contient un nombre infini de chiffres après la virgule. En binaire, on note que mantisse est $1.1\overline{0011}_2$ (avec les 4 derniers bits qui se répètent à l'infini). On voit donc que l'ordinateur ne peut pas représenter tous les nombres à virgule et il doit se contenter d'approximer la plupart des nombres.

Pour utiliser les nombres à virgules dans les ordinateurs, il nous reste à définir le format sous lequel on enregistre les nombres en mémoire. Pour cela, il existe le format standard IEEE 754. Cette norme décrit les formats suivants :

— simple précision (32 bits)
— double précision (64 bits)
— quadruple précision (128 bits)

Avec le format *simple précision*, les 32 bits sont répartis comme suit :

— le bit de poids fort (bit 31) représente le signe du nombre. Ce bit est à zéro pour un nombre positif et à un pour un nombre négatif.
— les 8 bits suivants (bits 23 à 30) représentent l'exposant. L'exposant peut être positif ou négatif et on aurait pu le représenter en complément à 2, mais en l'occurrence, l'exposant est codé avec un biais (offset) de 127. Ca signifie qu'un exposant de 0 est code par 127 ou $7\mathrm{F}_{16}$ (en binaire). La

valeur 0 étant réservée pour le nombre 0.0 et FF_{16} étant réservée pour les nombres spéciaux, le plus petit exposant possible est donc de -126 (codé par $127 - 126 = 1$) et le plus grand exposant est 127 (codé par $127 + 120 = 254 = \mathrm{FE}_{16}$).
— les 23 derniers bits représentent la mantisse. Comme cette dernière est comprise entre 1 et 2 non compris, il y a toujours un seul 1 à gauche du point décimal et ce 1 ne fait partie des 23 derniers bits.

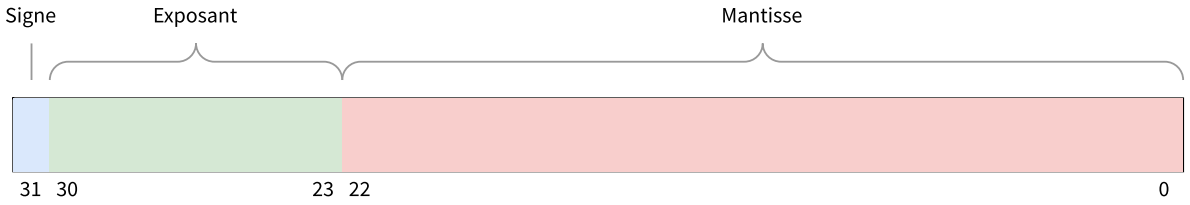

FIGURE 4.14 – number :IEEE 754 - simple précision

Nous pouvons illustrer l'encodage de nombres à virgules avec l'exemple de 42.5 vu précédemment :

— 42.5 est positif, donc le bit de signe est 0
— $42.5 = +1.328125 \cdot 2^5$, donc l'exposant est 5. Avec le biais de 127, cet exposant est codé par 132 ou 10000100_2.
— la mantisse est 1.010101_2 ; on ignore le 1 à gauche de la virgule et on complète avec les 0 et les 23 derniers bits sont donc $01010100000000000000000_2$.

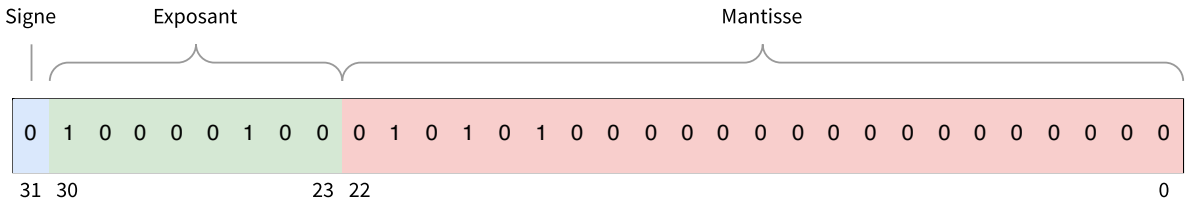

FIGURE 4.15 – number :42.5 en IEEE 754 simple précision

En hexadécimal, ça donne $422\mathrm{A}\ 0000_{16}$.

La norme IEEE 754 définit également quelques nombres spéciaux :

— Le zéro (0.0) est codé avec tous les bits à zéro.
— L'*infini* est représenté par un exposant avec tous les bits à 1 et la mantisse à 0. Le signe permet de représenter $+\infty$ et $-\infty$.
— Si l'exposant a tous les bits à 1 mais que la mantisse est différente de zéro, alors ça représente le nombre NaN (*Not a Number*).

Format IEEE	Nombre de bits pour la mantisse	Nombre de bits pour l'exposant	biais (offset)
Single precision	23	8	127
Double precision	52	11	1023
Quadruple precision	112	15	16383

Attention, en quadruple précision, contrairement aux autres formats, le 1 à gauche du point décimal n'est pas omis et est toujours présent dans la mantisse.

4.11 Exercice 1 : Les nombres entiers non-signés

Convertir en binaire les nombres suivants :

- 125_{10} (Base 10)
- 0377_{8} (Base 8)
- $0x\mathrm{ADE1}_{16}$ (Base 16)

Convertir en décimal les nombres suivants :

- $0b10011000_{2}$ (Base 2)
- 0177_{8} (Base 8)
- $0x25\mathrm{E1}_{16}$ (Base 16)

Additionner les nombres binaires suivants et donner le résultat ainsi que l'état des flags C et Z. On considère des nombres sur 8 bits :

- $0b10011000_{2} + 0b10011000_{2}$
- $0b11111101_{2} + 0b00000011_{2}$
- $0b00011000_{2} + 0b10011100_{2}$

Soustraire les nombres binaires suivants et donner le résultat ainsi que l'état des flags C et Z :

- $0b10011000_{2} - 0b10011000_{2}$
- $0b11111101_{2} - 0b00000011_{2}$
- $0b00011000_{2} - 0b10011100_{2}$

Donner le résultat ainsi que l'état des flags C et Z pour les soustractions suivantes :

- $125 - 128$
- $77 - 26$
- $254 - 254$
- $255 - 0$

4.12 Exercice 2 : Les nombres entiers signés

Convertir en binaire les nombres suivants et indiquer l'état du flag N :

- -125_{10} (Base 10)
- 0271_{8} (Base 8)
- $0x50\mathrm{F}1_{16}$ (Base 16)

Convertir en décimal les nombres suivants :

- $0b10011000_{2}$ (Base 2)
- 0177_{8} (Base 8)
- $0x85\mathrm{E}1_{|6}$ (Base 16)

Additionner les nombres binaires suivants et donner le résultat ainsi que l'état des flags V, N et Z :

- $0b10011000_{2} + 0b10011000_{2}$
- $0b11111101_{2} + 0b00000011_{2}$
- $0b00011000_{2} + 0b10011100_{2}$

Soustraire les nombres binaires suivants et donner le résultat ainsi que l'état des flags V, N et Z :

- $0b10011000_{2} - 0b10011000_{2}$
- $0b11111101_{2} - 0b00000011_{2}$
- $0b00011000_{2} - 0b10011100_{2}$

Donner le résultat ainsi que l'état des flags V et N et Z pour les soustractions suivantes :

- $127 - (-125)$
- $77 - (-26)$
- $-30 - (-34)$
- $55 - 66$

4.13 Exercice 3 : Evaluation des flags

Pour les opérations suivantes, calculez :

- l'état des flags Z, C, N et V

- le résultat de l'opération si on interprète ce résultat comme un nombre signé
- le résultat de l'opération si on interprète ce résultat comme un nombre non signé

On considère des nombres sur 8 bits.

- $128 - (-128)$
- $64 + (-128)$
- $228 - 128$
- $240 - (-16)$
- $0 - 0$
- $-7 + 249$
- $248 + (-128)$
- $128 + 0$
- $62 - 200$
- $-8 - (-96)$

4.14 Exercice 4 : Nombres réels

Représenter en hexadécimal sur 32 bits (simple précision) les valeurs réelles suivantes :

- $1\,048\,576$
- 2048
- 55.75
- $5 \div 4096$
- $-25 \div 2$