Introduction à l'architecture des ordinateurs

Cours 2
Codage des entiers naturels et relatifs

Version du 11 septembre 2023

Plan du cours

Représentation de l'information et codage binaire

Codage des entiers

Unités de mesure de l'information

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]

[1] Shannon, 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
Kibioctet	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
Mégaoctet	MB	10^6	8×10^6
Gigaoctet	GB	10^9	8×10^9
Téraoctet	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.*

Codage binaire

Le bit est le concept utiliser 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 (cf. cours suivant)
- 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 !

Plan du cours

Plan du cours

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 écrira $0110\,0111_2 = 103_{10}$

Base 2 et base 16

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

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

$$
\begin{aligned}
4\,242_{10} &= 4\,096 + 144 + 2 \\
&= 1 \times 16^3 + 9 \times 16^1 + 2 \times 16^0 \\
&= 1\,092_{16}
\end{aligned}
$$

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

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

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

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

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 programmation[2], 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`.

[2] dont golang, python et java

Un mot sur l'arithmétique entière des ordinateurs

Au sein d'un processeur (cf. cours suivant), l'unité arithmétique et logique (UAL) réalise les opérations arithmétiques usuelles ($+$, $-$, $\times$, $\div$), les comparaisons ($=$, $\neq$, $<$, $\leq$, ...), 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 == 1
```

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

```
0xFE+0x10 == 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 == 1
```

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

```
0xFE+0x10 == 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'expliciter la taille des variables (p.ex. `uint8`, ou `uint64`).

Plan du cours

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

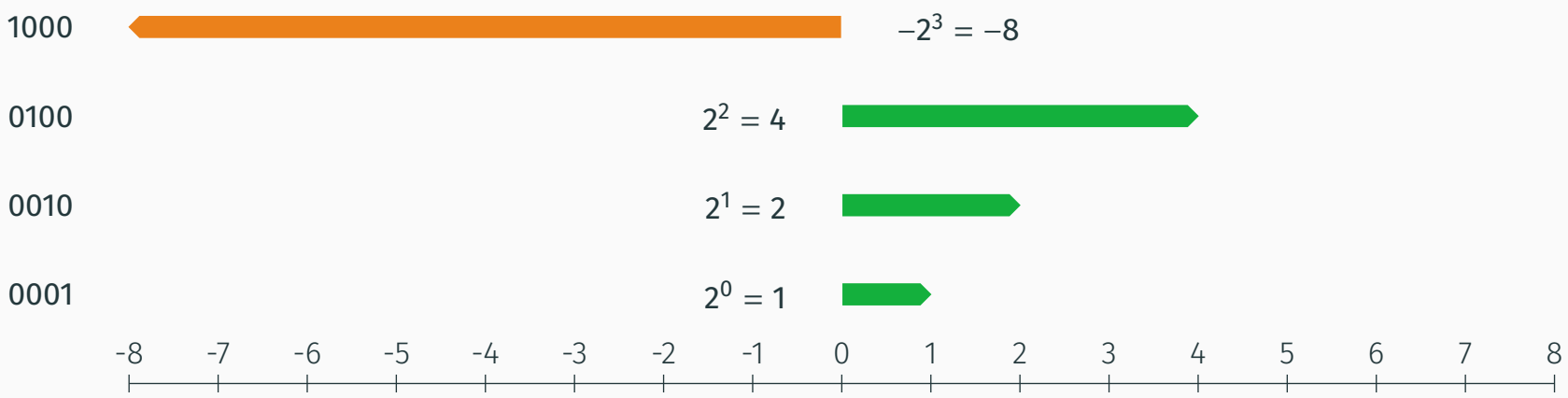
$$1010_{c2} = 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6 \neq 1010_2$$

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

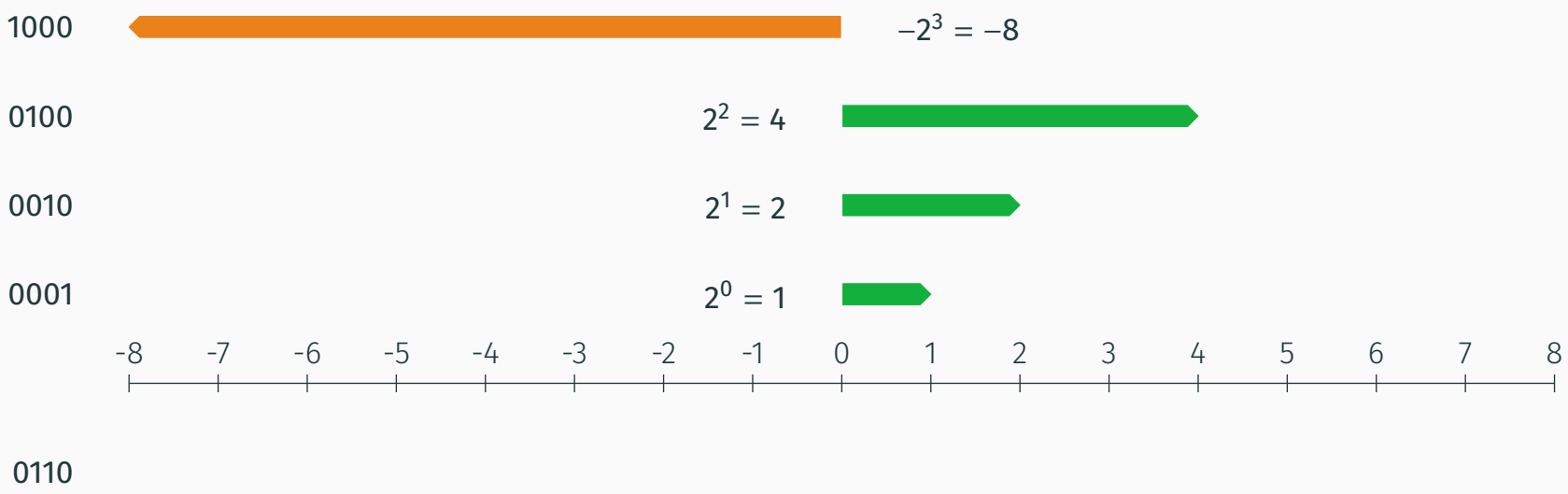
Si le bit de poids fort est **0** : nombre positif.

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

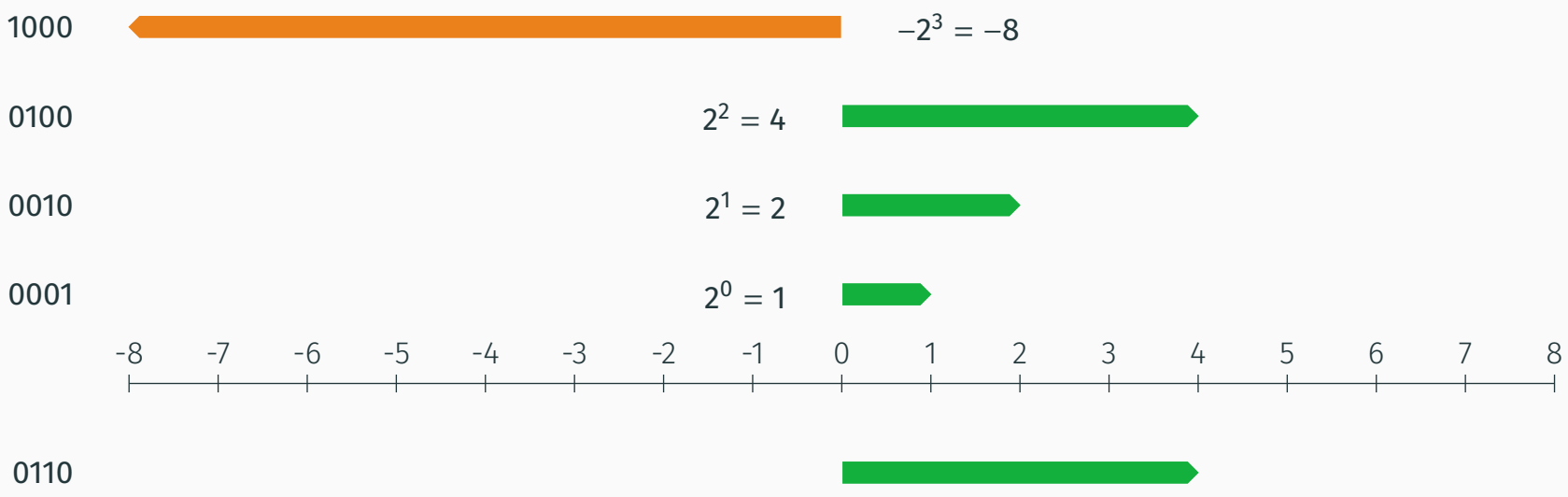
Visualiation du complément à 2 (mots de 4 bits)



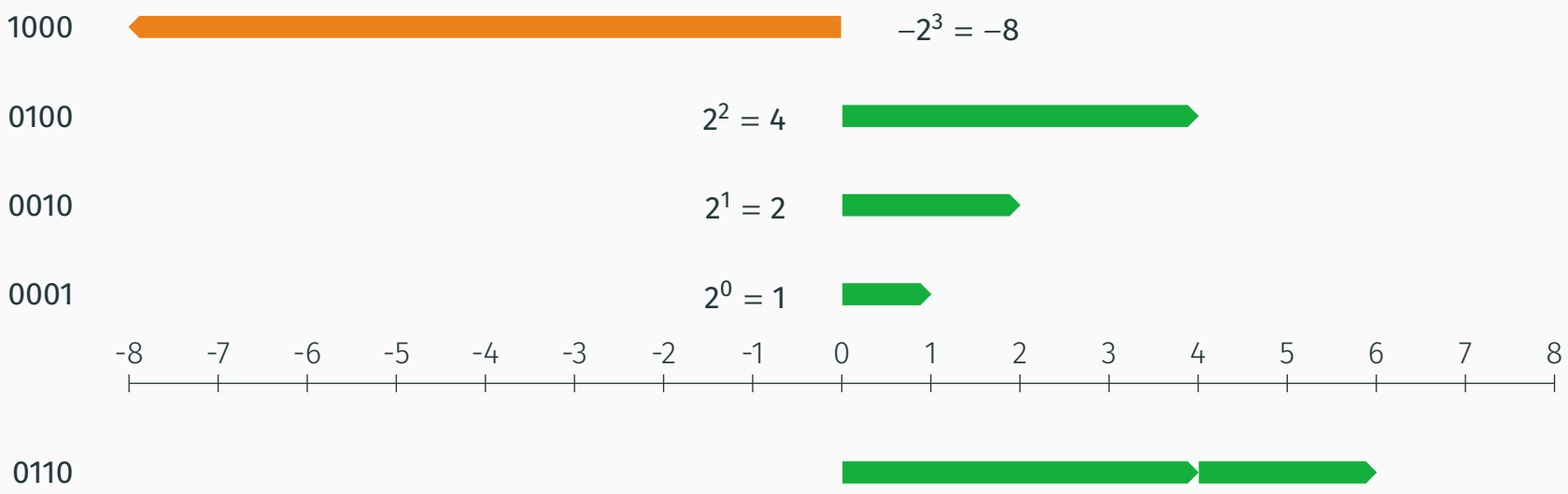
Visualiation du complément à 2 (mots de 4 bits)



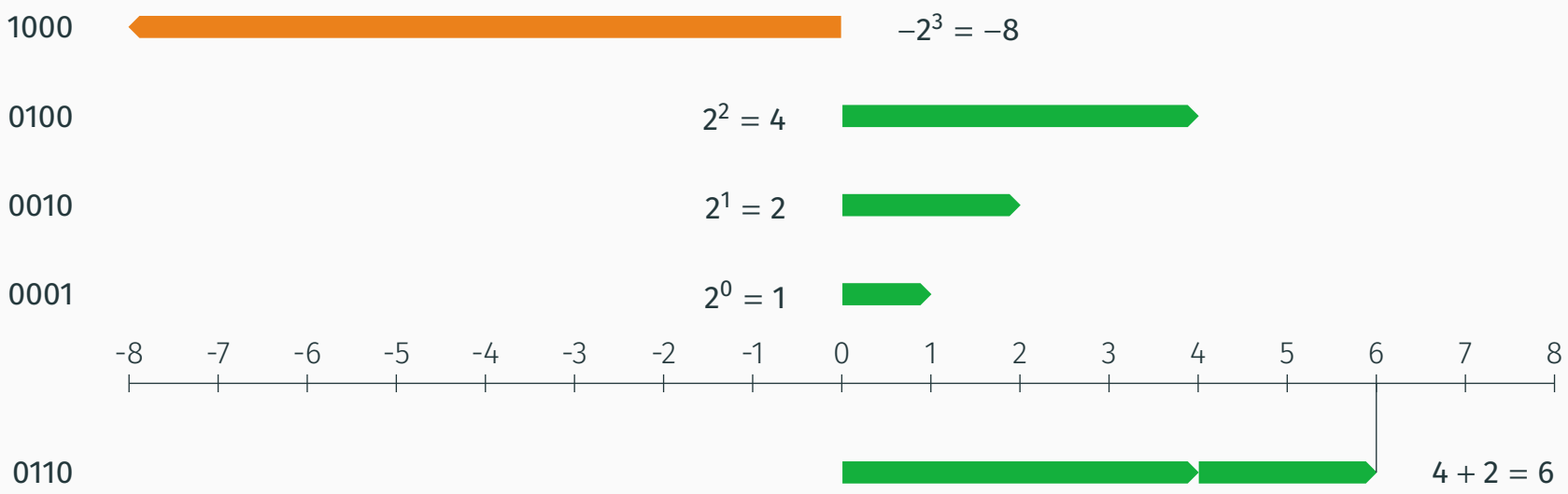
Visualiation du complément à 2 (mots de 4 bits)



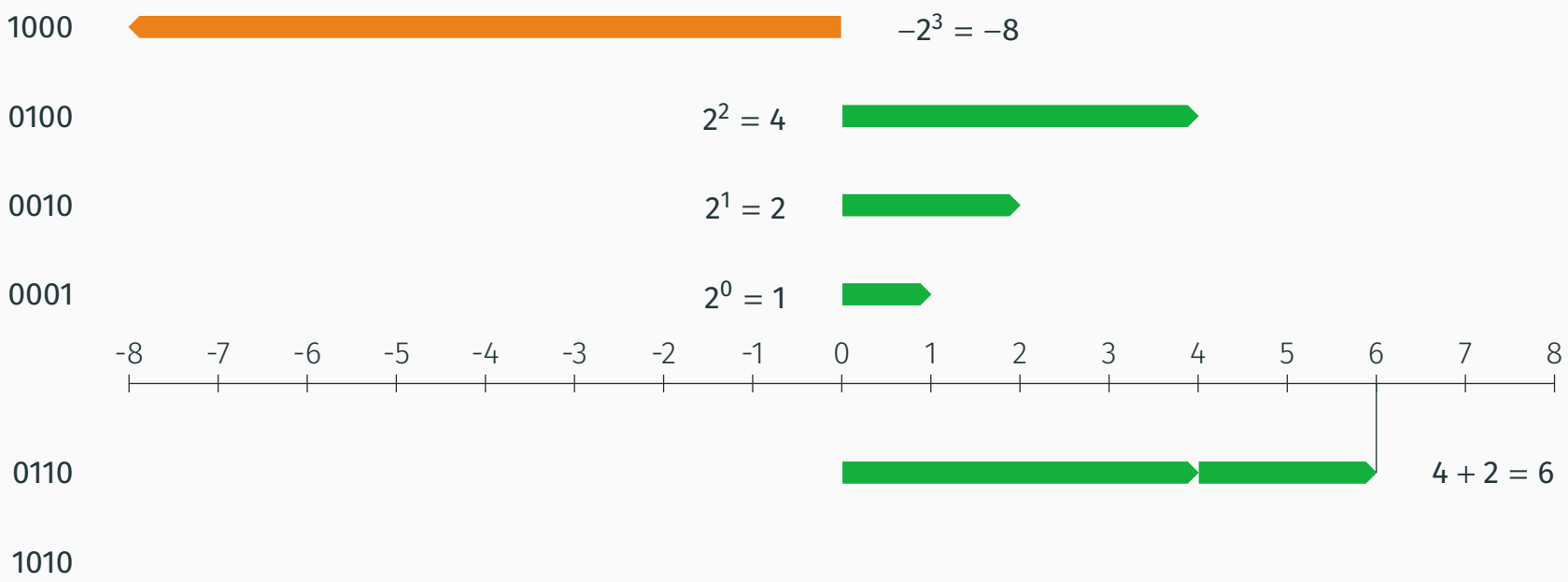
Visualiation du complément à 2 (mots de 4 bits)



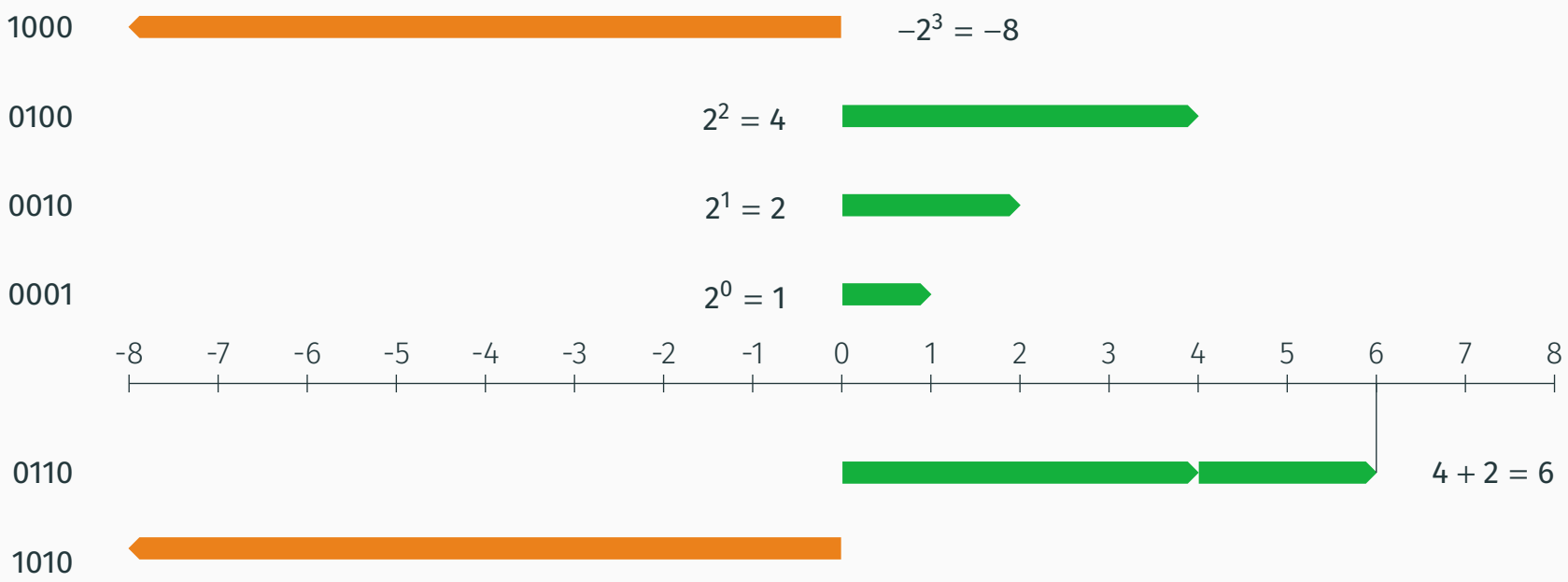
Visualiation du complément à 2 (mots de 4 bits)



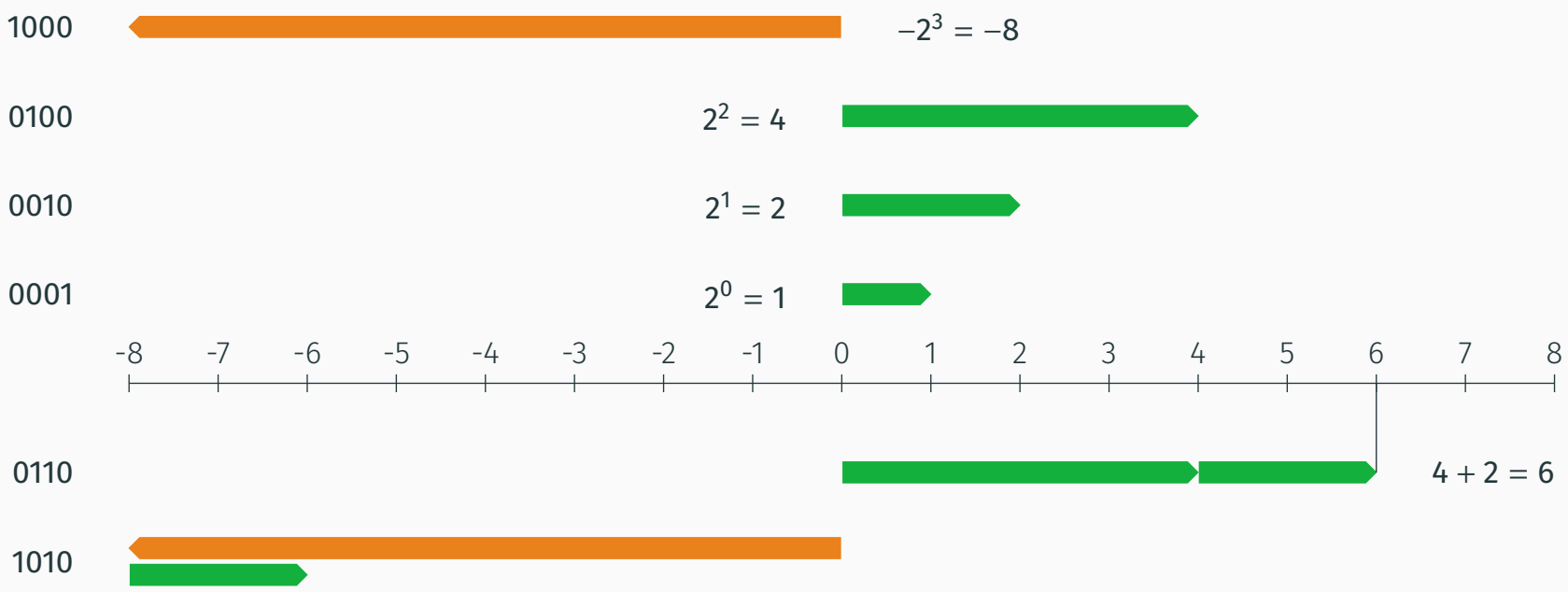
Visualiation du complément à 2 (mots de 4 bits)



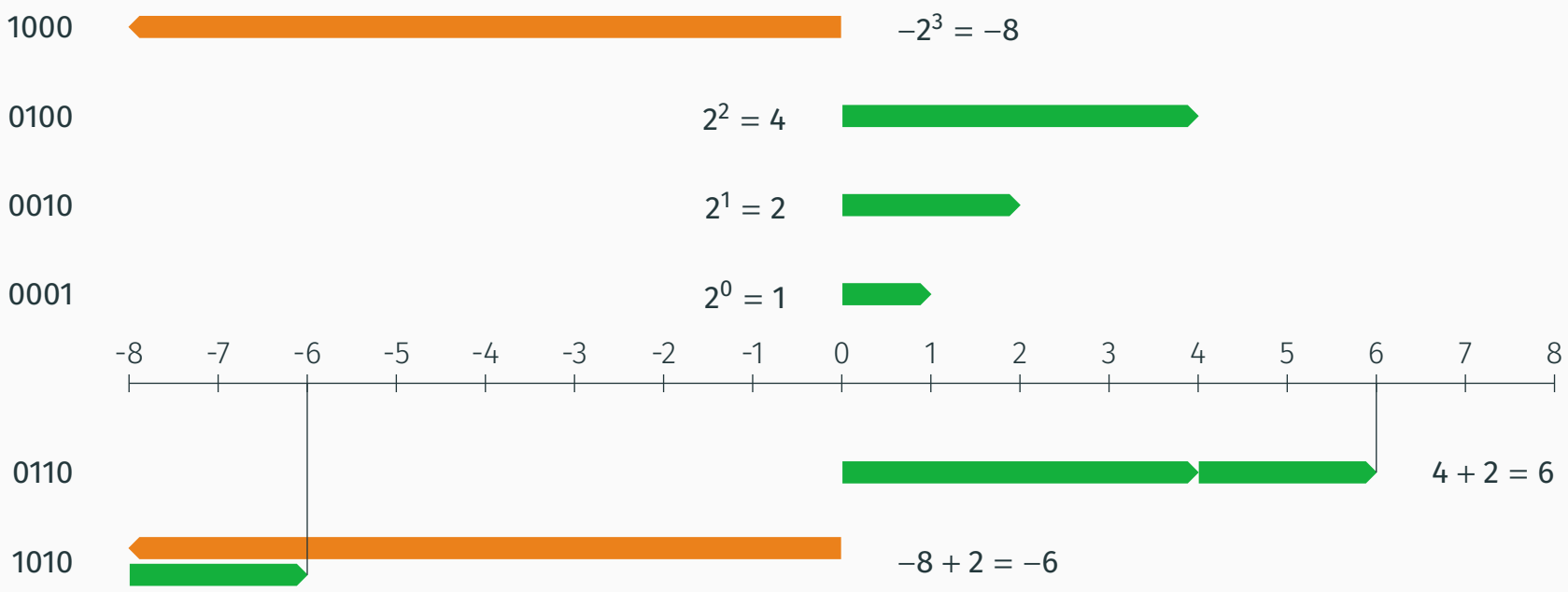
Visualiation du complément à 2 (mots de 4 bits)



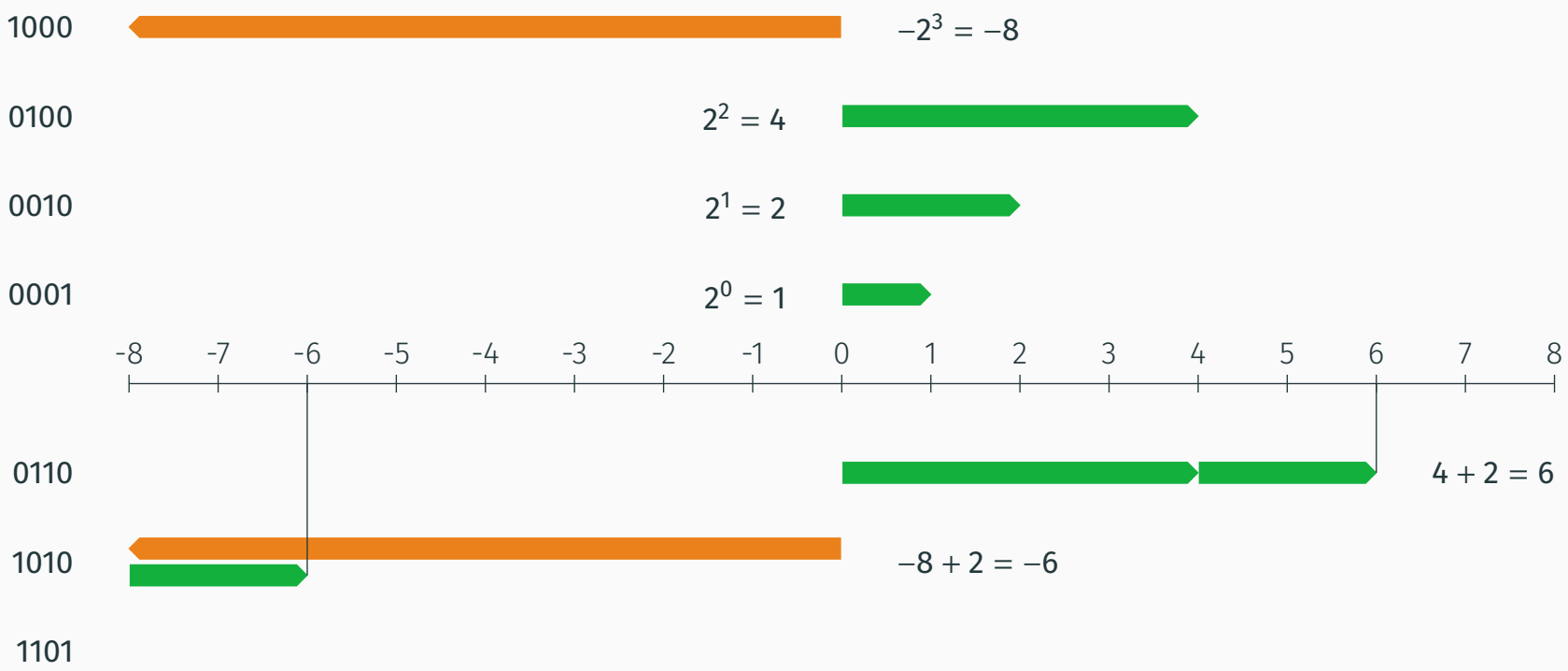
Visualiation du complément à 2 (mots de 4 bits)



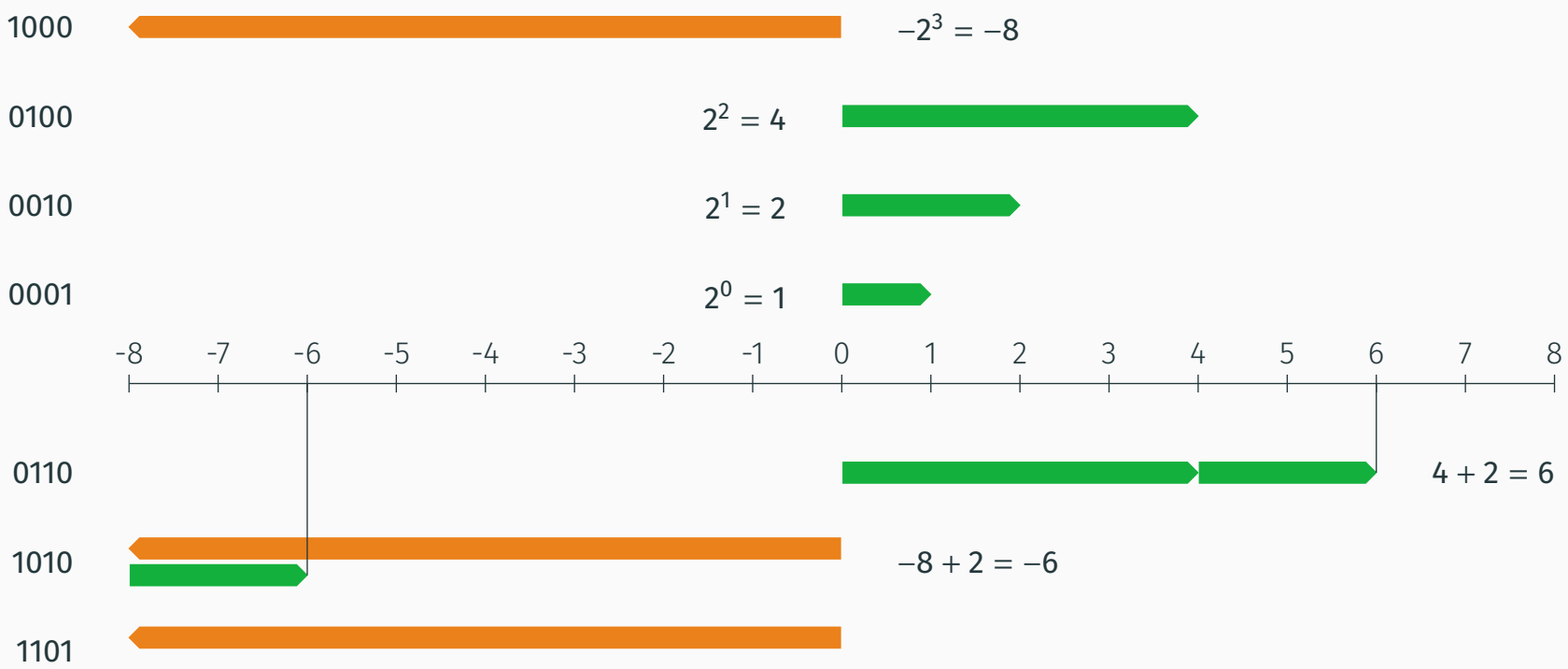
Visualiation du complément à 2 (mots de 4 bits)



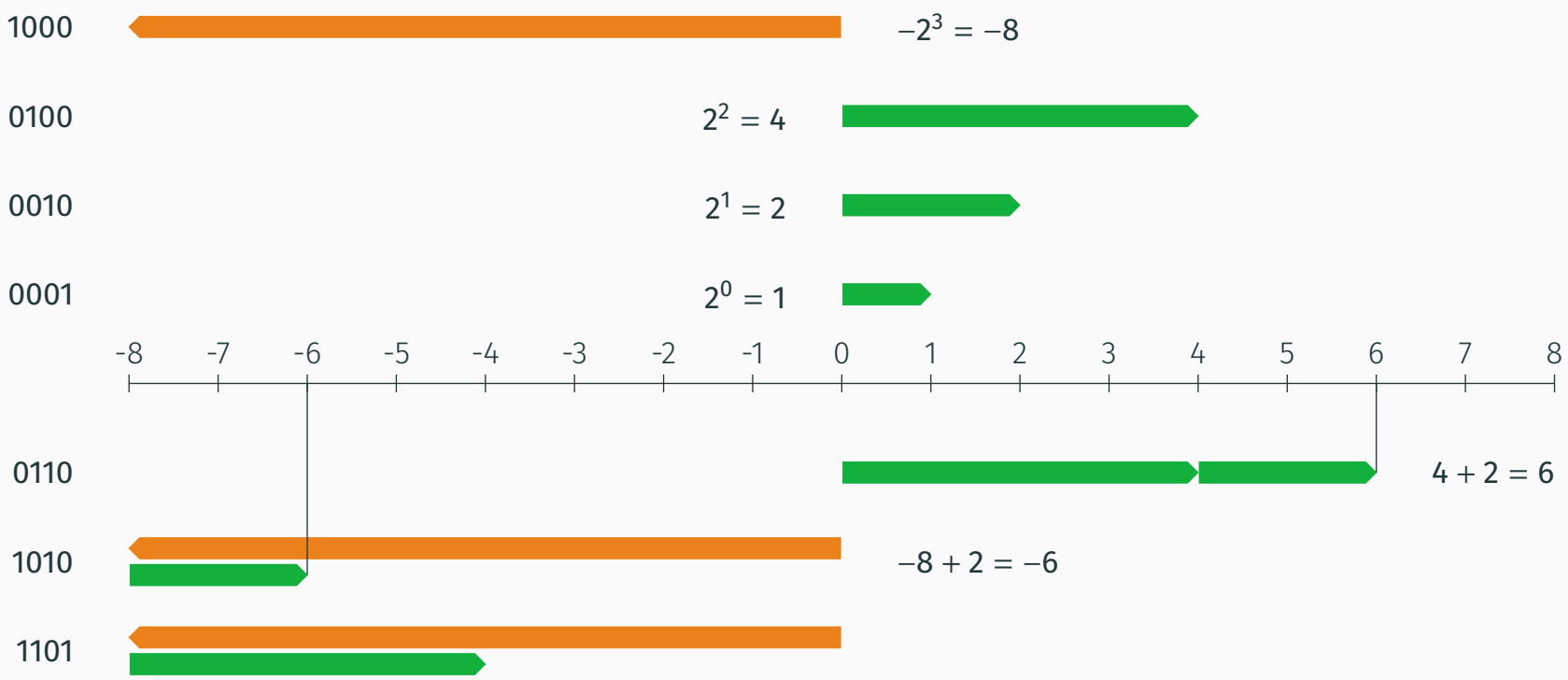
Visualiation du complément à 2 (mots de 4 bits)



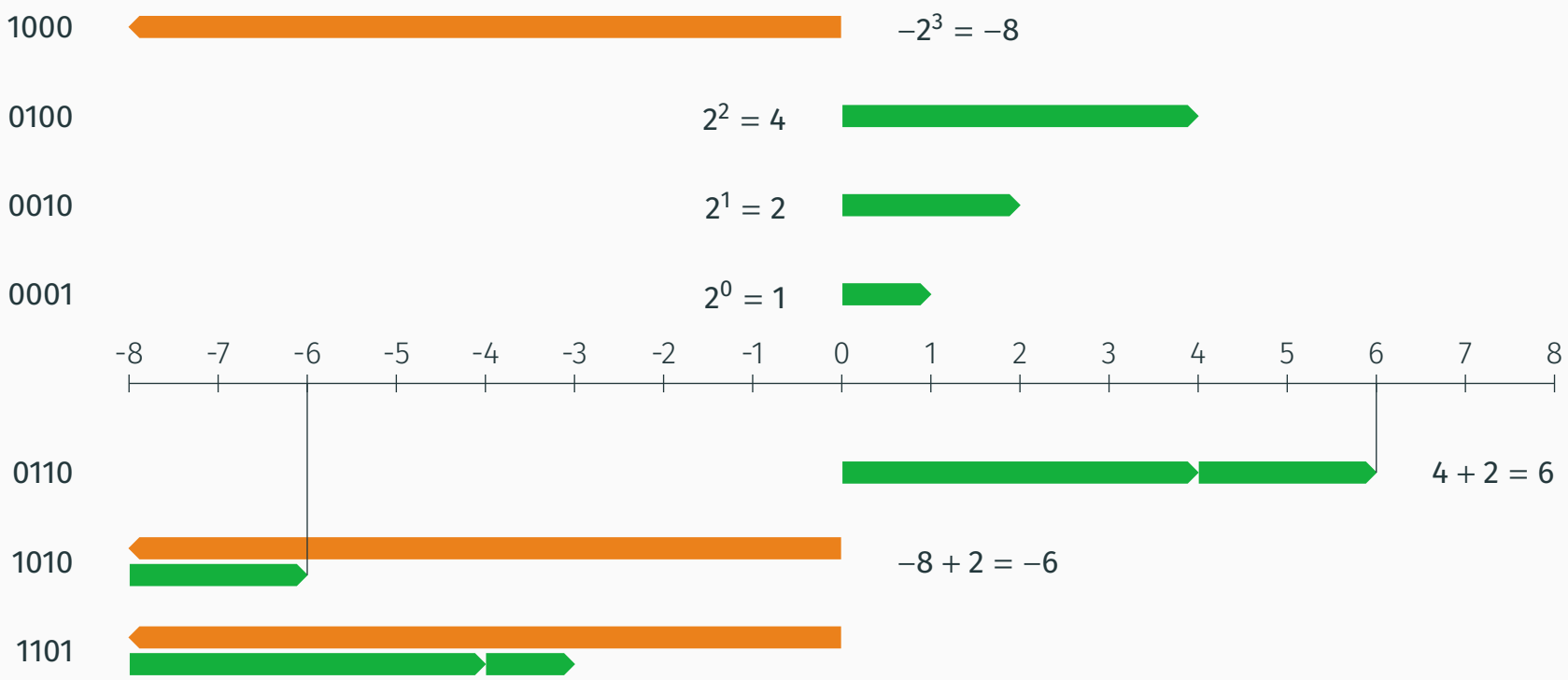
Visualiation du complément à 2 (mots de 4 bits)



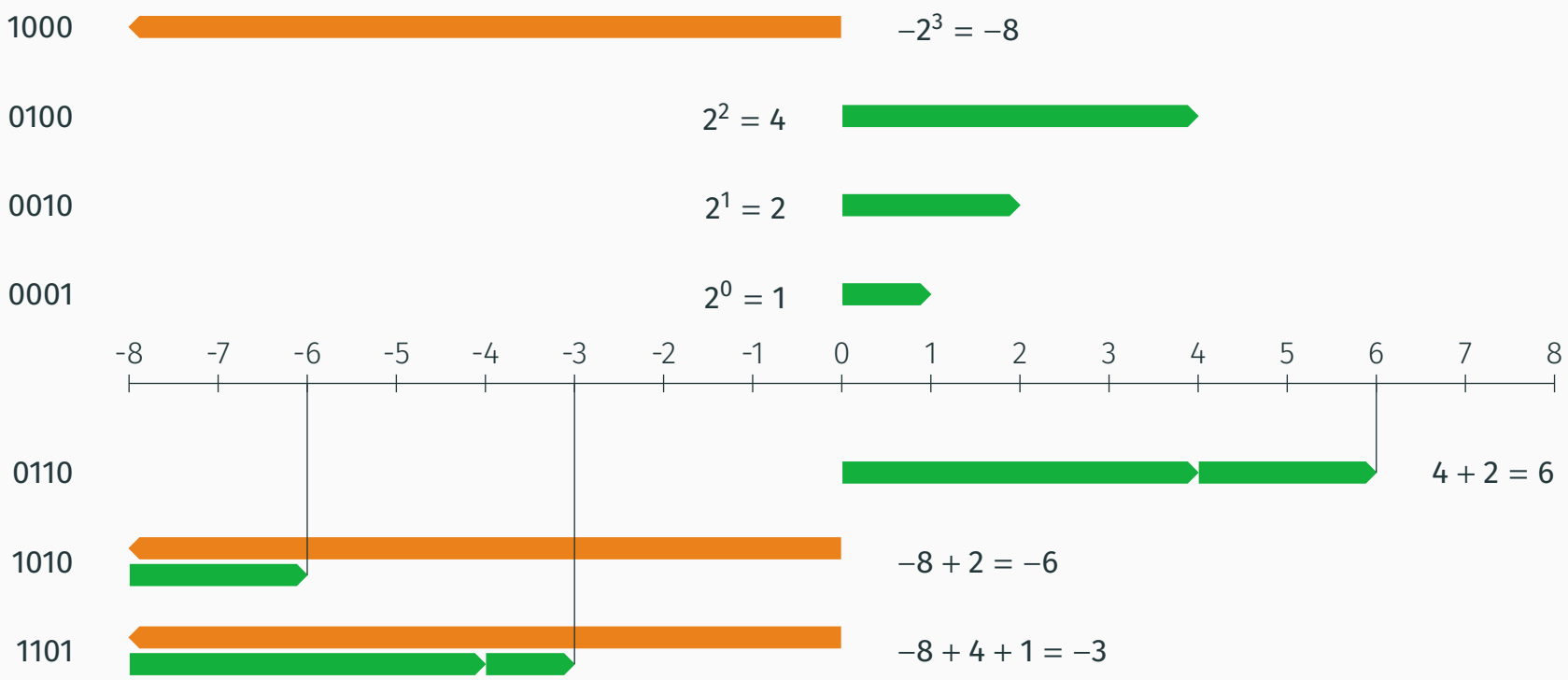
Visualiation du complément à 2 (mots de 4 bits)



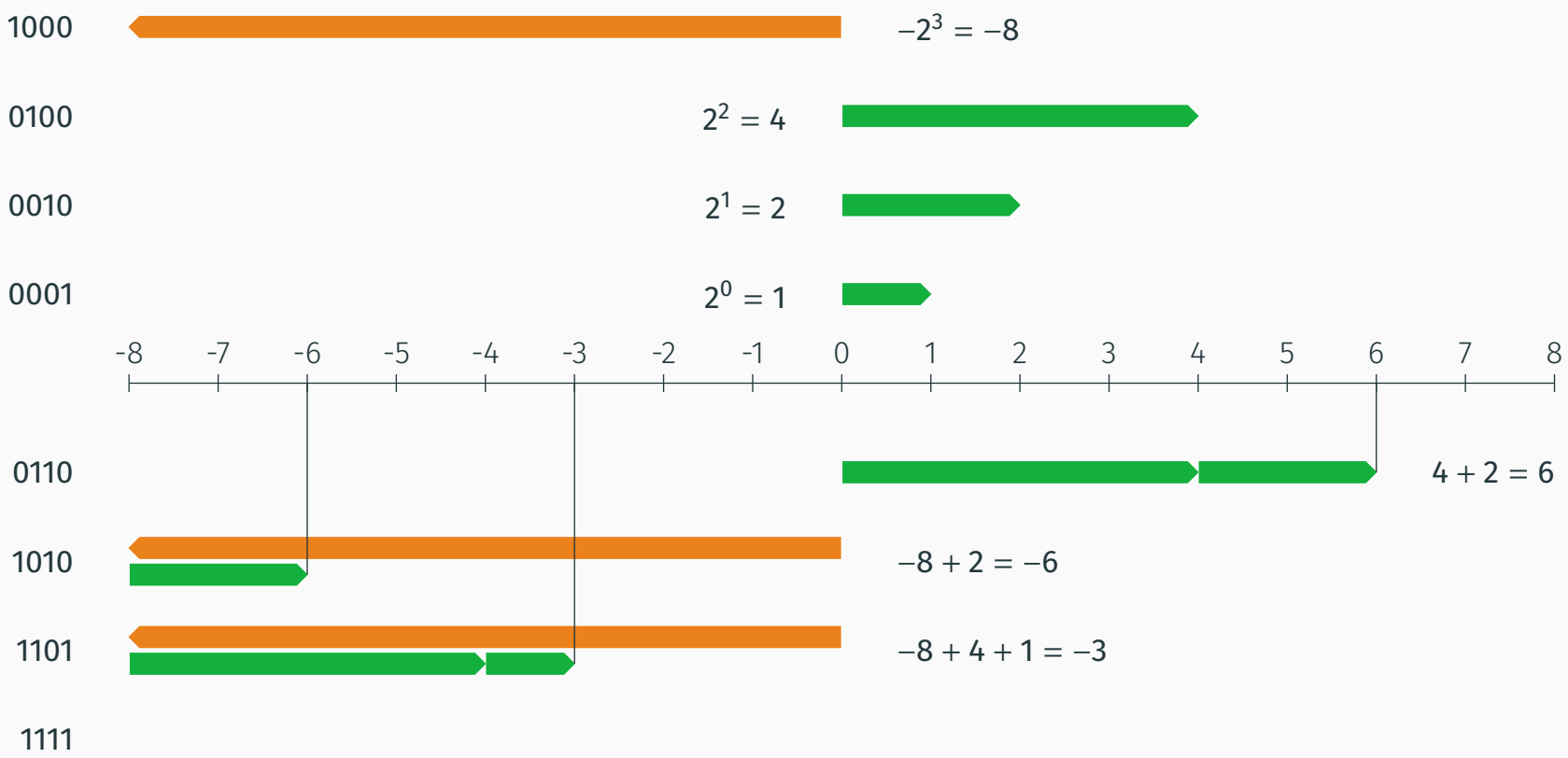
Visualiation du complément à 2 (mots de 4 bits)



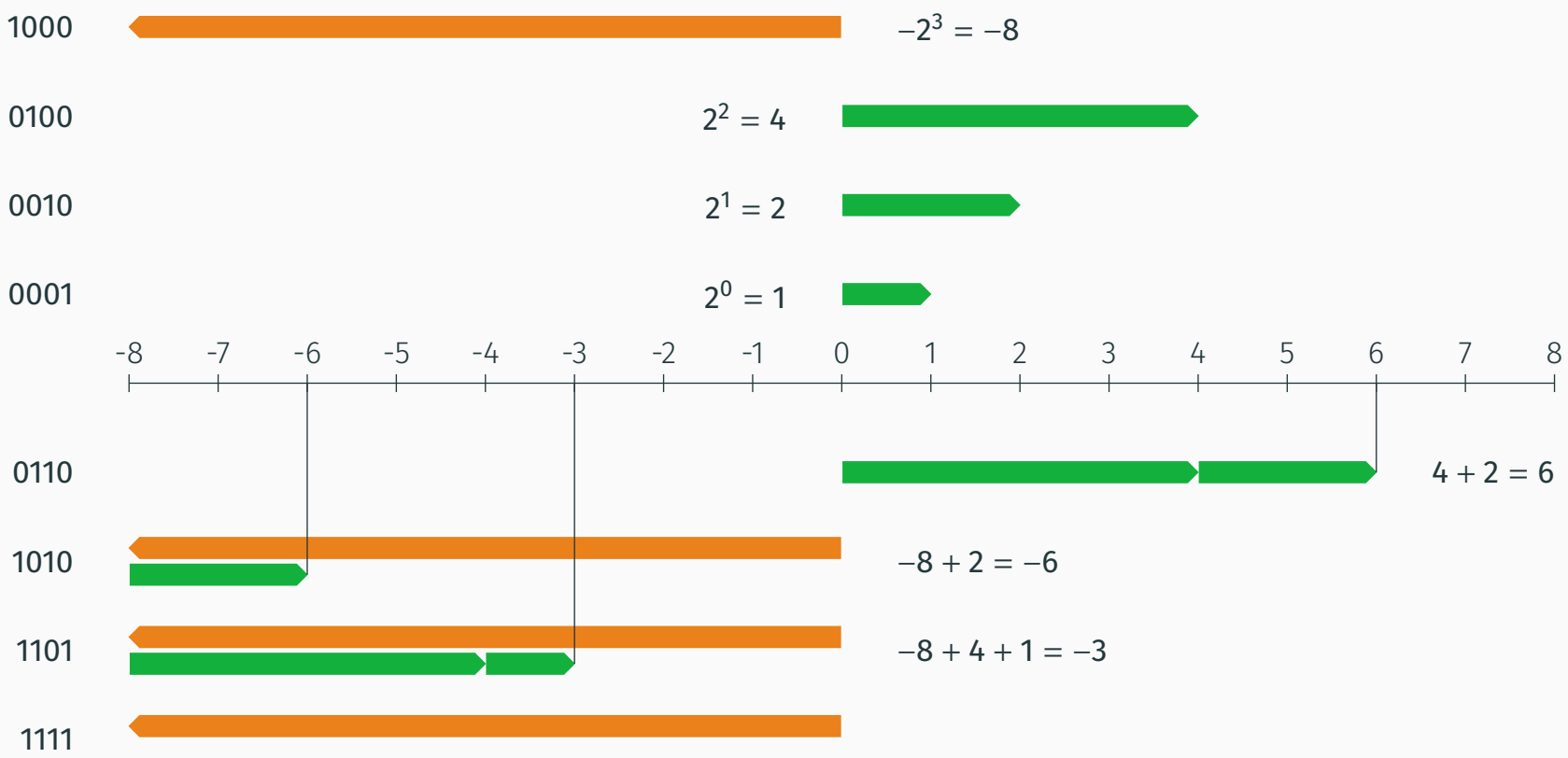
Visualiation du complément à 2 (mots de 4 bits)



Visualiation du complément à 2 (mots de 4 bits)



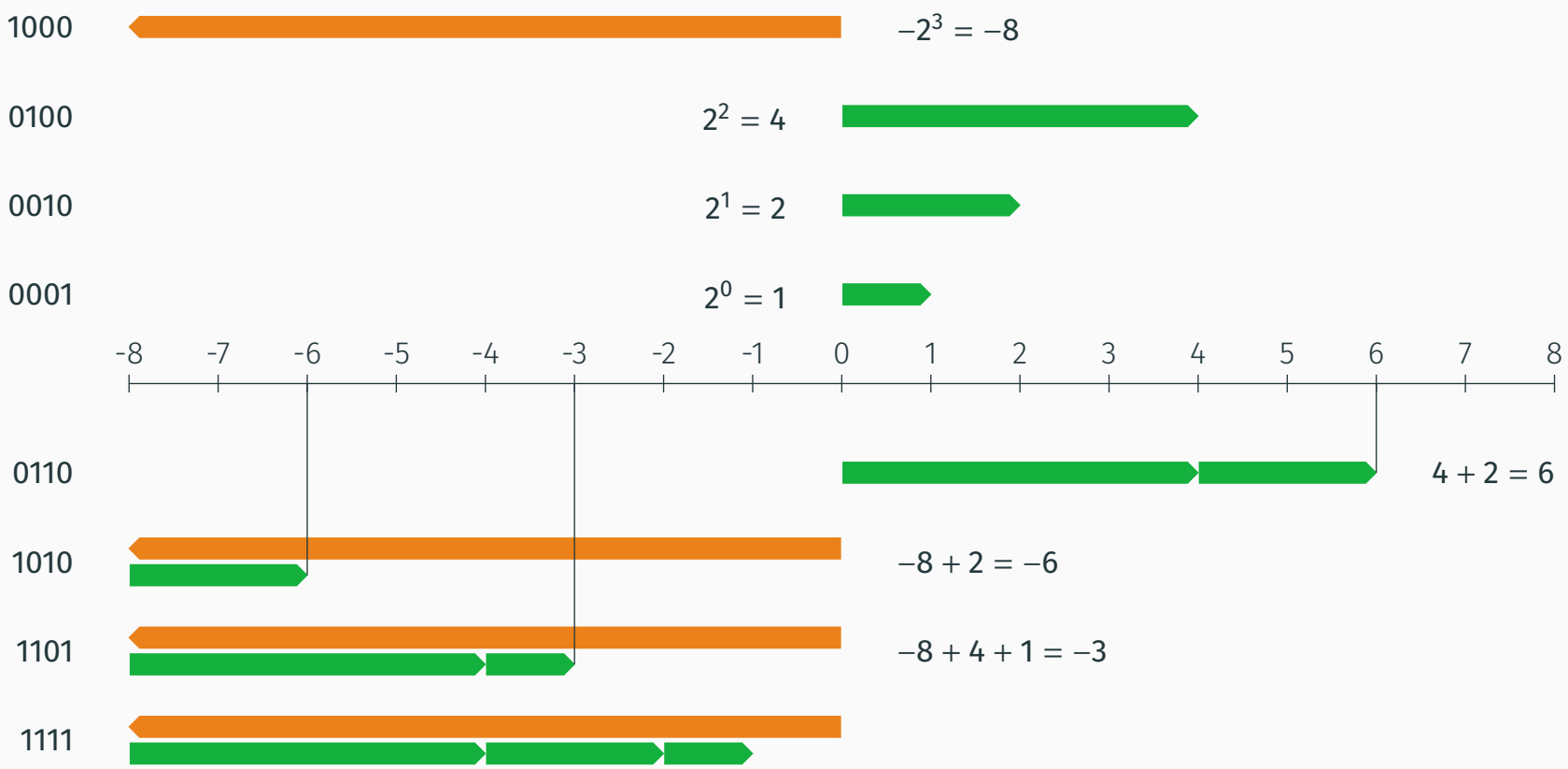
Visualiation du complément à 2 (mots de 4 bits)



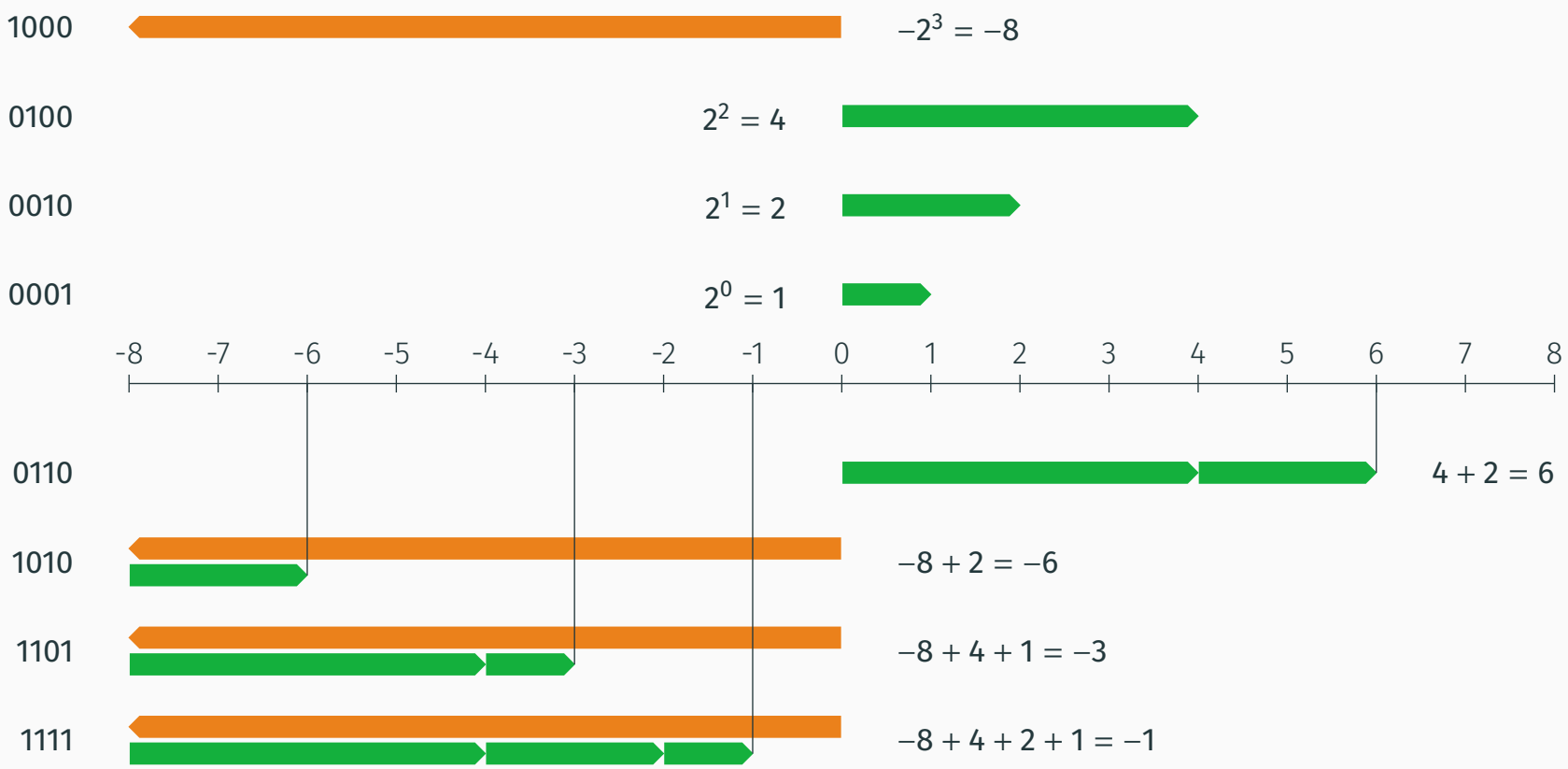
Visualiation du complément à 2 (mots de 4 bits)

Visualiation du complément à 2 (mots de 4 bits)

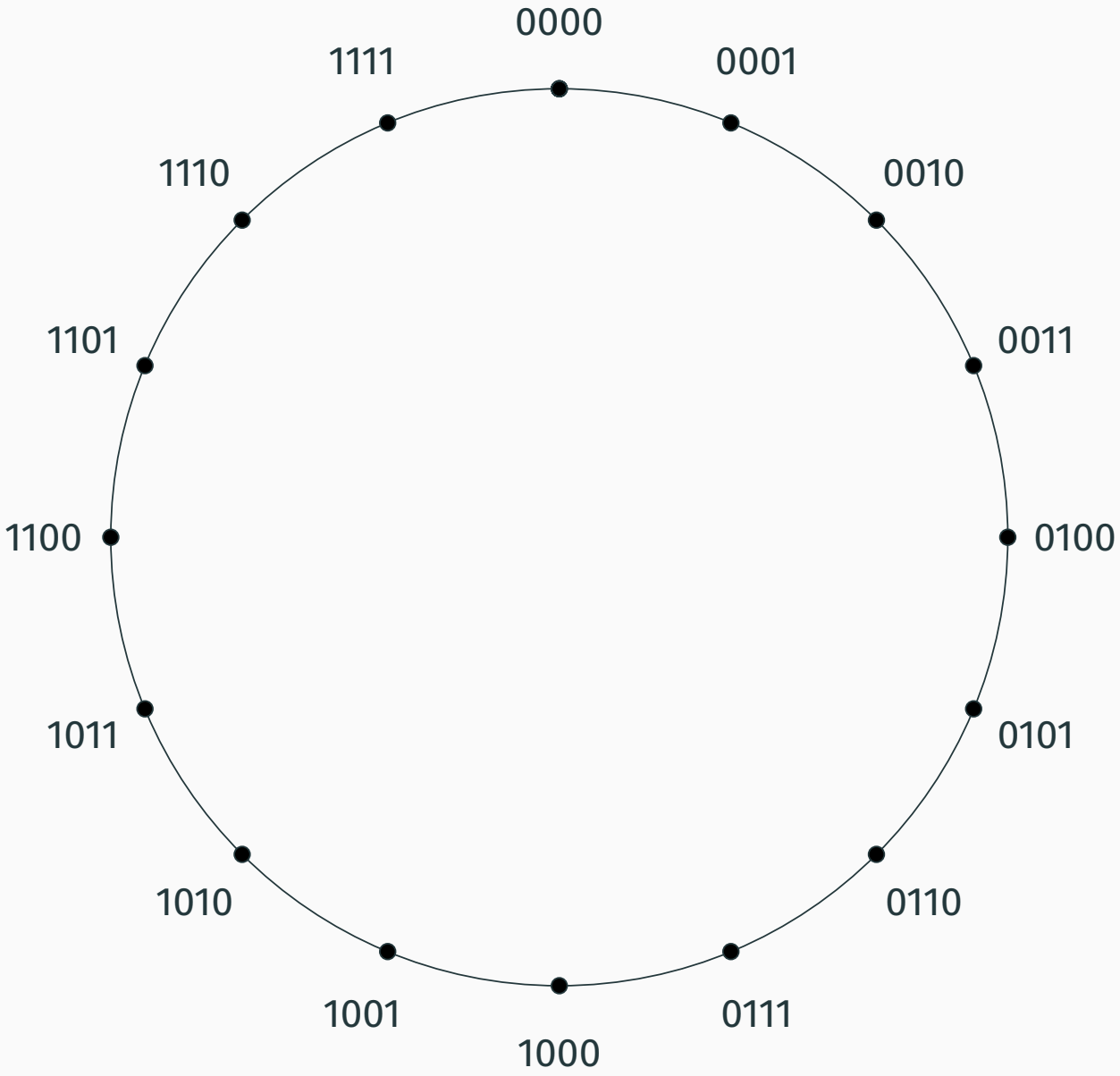
Visualiation du complément à 2 (mots de 4 bits)



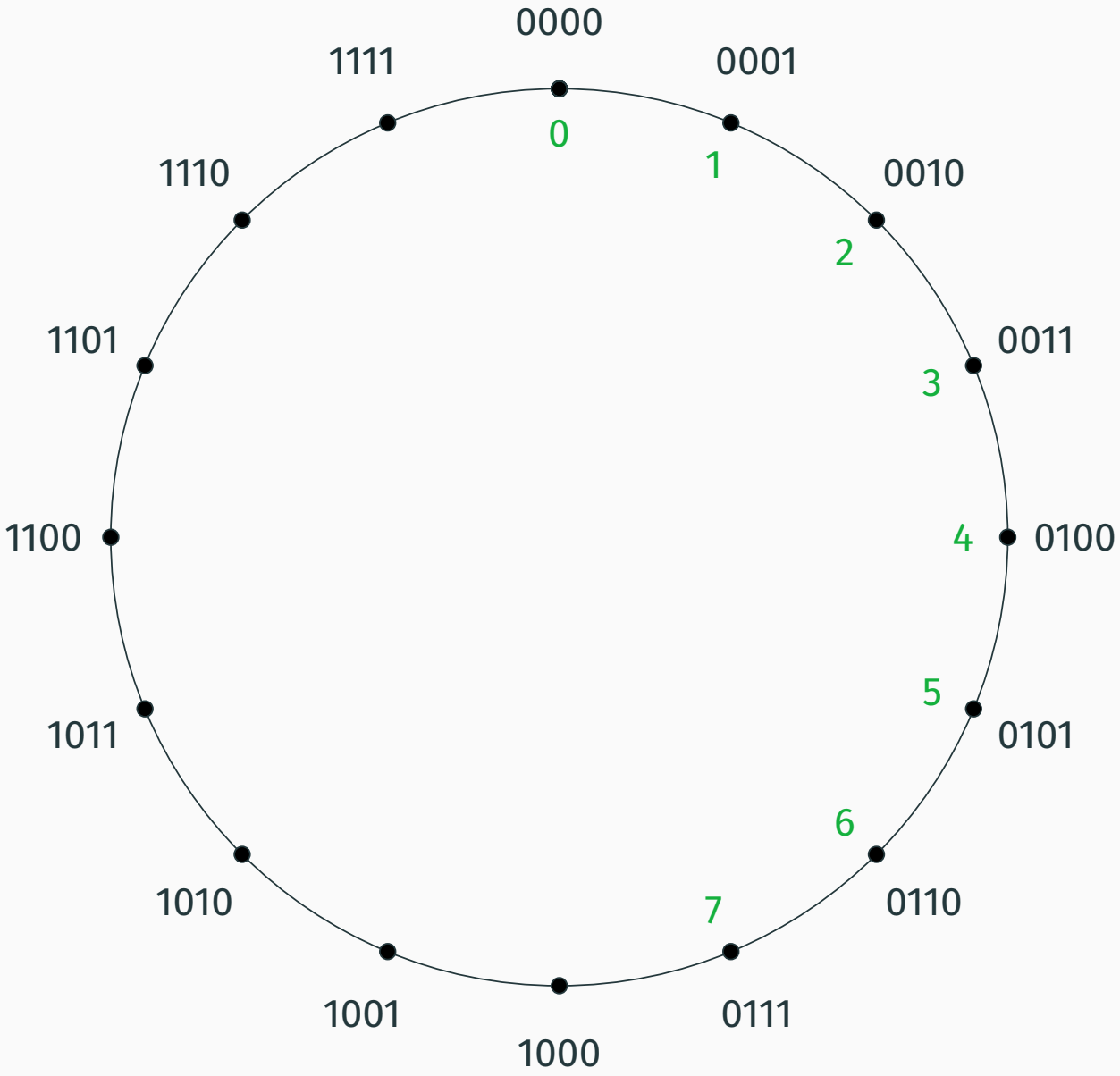
Visualiation du complément à 2 (mots de 4 bits)



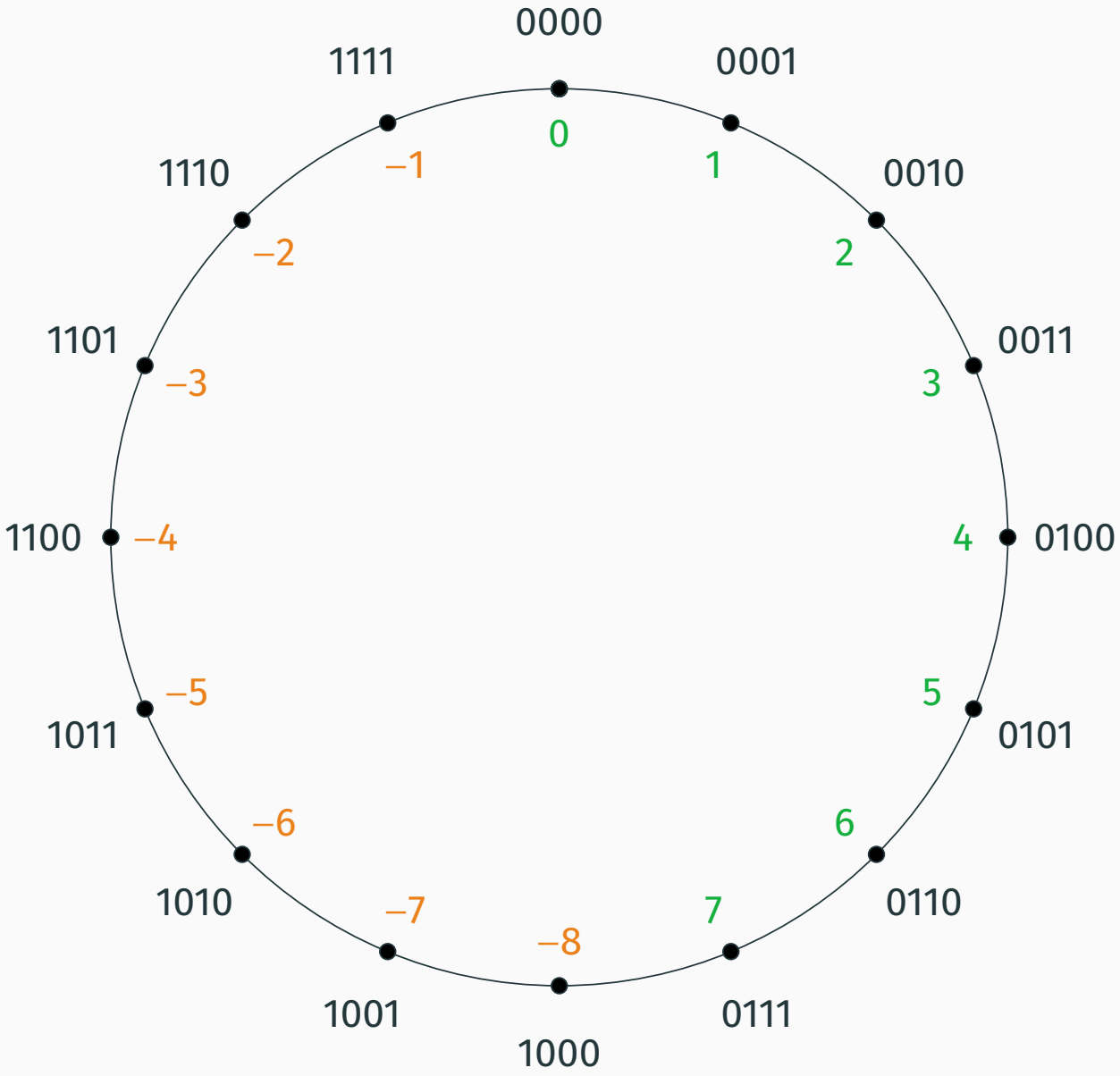
Autre visualiation du complément à 2 (mots de 4 bits)



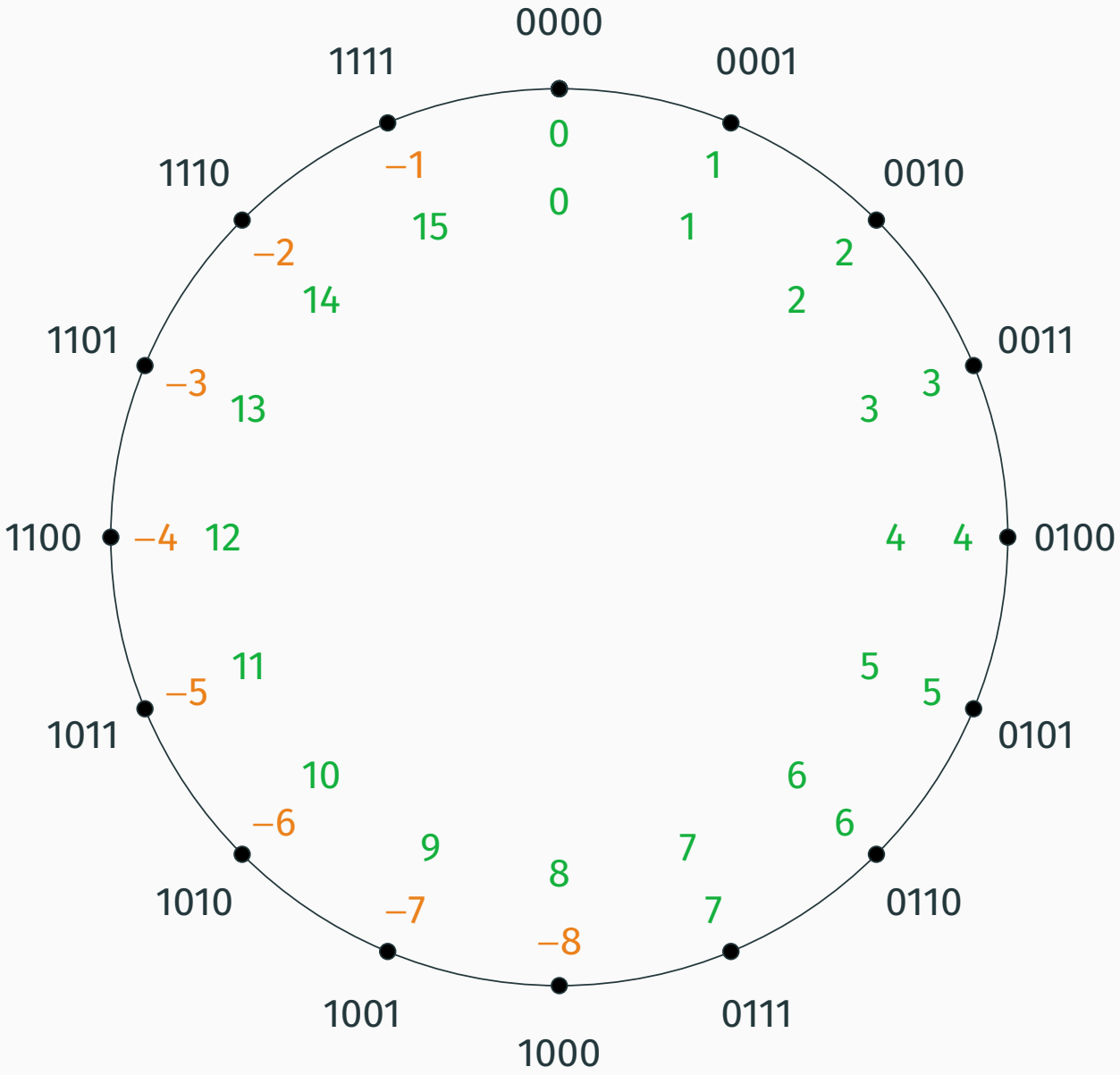
Autre visualiation du complément à 2 (mots de 4 bits)



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



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



Machines et complément à 2

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 !

Arithmétique en complément à 2

Avec des mots de 4 bits :

```
6 + 3 == -7
```

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

On a bien $-7 \equiv 9 \mod 2^4$.

```
-5 + (-6) = 5
```

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

On a bien $-11 \equiv 5 \mod 2^4$.

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