Chapitre 1
Arithmétique des ordinateurs

Version du 03/10/2024

Table des matières

I. Bases de numération....3
1. Définitions....3
1.1. Base 10 ou système décimal....3
1.2. Base 2 ou système binaire....4
1.3. Base 16 ou système hexadécimal....5
1.4. Base quelconque....6
2. Conversions....7
2.1. Conversion d'entiers entre 0 et 15....7
2.2. Conversion d'une base quelconque vers la base 10....7
2.2.1. Méthode....7
2.2.2. Exemples....8
2.3. Conversion de la base 10 vers une base quelconque....8
2.3.1. Méthode de conversion pour la partie entière (divisions successives)....8
2.3.2. Méthode de conversion pour la partie fractionnaire (multiplications successives)....8
2.3.3. Exemples....9
2.4. Conversion de la base 2 vers une base en puissance de deux....9
2.4.1. Méthode....9
2.4.2. Exemples....10
2.5. Conversion d'une base quelconque vers une base quelconque....10
2.5.1. Méthode....10
2.5.2. Exemples....10
II. Opérations fondamentales....11
1. Addition binaire....11
2. Soustraction binaire....11
3. Multiplication binaire....12
4. Division binaire....13
5. Addition hexadécimale....16
III. Bits, mots et octets....17
1. Définitions....17
2. Complément à un....18
3. Complément à deux....18
IV. Encodage d'entiers....20
1. Entiers non signés....20
1.1. Introduction....20
1.2. Encodage d'entiers non signés....20
1.3. Décodage d'entiers non signés....20

1.4. Dépassement non signé....21
2. Entiers signés....23
2.1. Introduction....23
2.2. Soustraction à l'aide du complément à deux....24
2.3. Encodage d'entiers signés....25
2.4. Décodage d'entiers signés....25
2.5. Dépassement signé....26
2.6. Extension de signe....28
V. Autres types d'encodage....29
1. BCD....29
2. Code Gray ou binaire réfléchi....29
3. ASCII....30

I. Bases de numération

1. Définitions

1.1. Base 10 ou système décimal

Du fait de nos dix doigts, nous représentons les nombres en **base 10** (aussi appelé **système décimal**).

La base 10 utilise **dix symboles (chiffres)** : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Ces symboles se placent côte à côte et peuvent être séparés par une virgule. Par exemple, un nombre en base 10 peut se représenter de la manière suivante :

1 5 7 , 6 2 5

La virgule est utilisée comme séparateur décimal et sert à séparer la partie entière d'un nombre (157 dans l'exemple ci-dessus) de sa partie fractionnaire (0,625).

Chaque symbole possède une position numérotée (ou colonne).

2	1	0		–1	–2	–3
1	5	7	,	6	2	5

La position d'un chiffre est positive quand il se trouve à gauche de la virgule et négative quand il se trouve à sa droite. La première position à gauche de la virgule est la position 0 (ou colonne 0).

Dans cette *notation positionnelle*, la valeur d'un symbole dépend de sa position. Chaque chiffre doit être multiplié par un *poids*. Le poids d'un symbole est 10 levé à la puissance de la position du symbole.

10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	← Poids
1	5	7	,	6	2	5	

Colonne des centaines
Colonne des dizaines
Colonne des unités
Colonne des millièmes
Colonne des centièmes
Colonne des dixièmes

Ce nombre peut se représenter comme une somme de produits :

$$\mathbf{157{,}625} = \mathbf{1}\times 10^2 + \mathbf{5}\times 10^1 + \mathbf{7}\times 10^0 + \mathbf{6}\times 10^{-1} + \mathbf{2}\times 10^{-2} + \mathbf{5}\times 10^{-3}$$

1.2. Base 2 ou système binaire

Un ordinateur mémorise et manipule uniquement des 0 et des 1. Par conséquent, lorsque l'on souhaite étudier la représentation de nombres au sein d'un ordinateur, la **base 2** (aussi appelé **système binaire**) semble plus appropriée que le système décimal.

La base deux utilise **deux symboles** : 0 et 1.

Tout comme le système décimal, ces symboles se placent côte à côte et peuvent être séparés par une virgule. Chaque symbole possède une position numérotée (ou colonne). Par exemple, un nombre binaire peut se représenter de la manière suivante :

7	6	5	4	3	2	1	0		–1	–2	–3
1	0	0	1	1	1	0	1	,	1	0	1

Dans le système binaire, le poids d'un symbole est 2 levé à la puissance de la position du symbole :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
1	0	0	1	1	1	0	1	,	1	0	1

ou

128	64	32	16	8	4	2	1		.5	.25	.125
1	0	0	1	1	1	0	1	,	1	0	1

Ce nombre peut se représenter comme une somme de produits :
$10011101{,}101_2 = \mathbf{1}\times2^7 + \mathbf{0}\times2^6 + \mathbf{0}\times2^5 + \mathbf{1}\times2^4 + \mathbf{1}\times2^3 + \mathbf{1}\times2^2 + \mathbf{0}\times2^1 + \mathbf{1}\times2^0 + \mathbf{1}\times2^{-1} + \mathbf{0}\times2^{-2} + \mathbf{1}\times2^{-3}$
$10011101{,}101_2 = 128 + 16 + 8 + 4 + 1 + 0{,}5 + 0{,}125$
$10011101{,}101_2 = 157{,}625_{10}$

Remarques :

- Quand nous manipulons plusieurs bases à la fois, les valeurs des différentes bases doivent être placées en indice à droite des nombres de façon explicite.
- Si une base n'est pas précisée explicitement, le contexte doit être clair.

Afin de pouvoir manipuler facilement les nombres binaires, il est conseillé de connaître par cœur les premières puissances de deux positives et négatives.

Puissance de 2 positive	
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1 024
2^{11}	2 048
2^{12}	4 096
2^{13}	8 192
2^{14}	16 384
2^{15}	32 768
2^{16}	65 536

Puissance de 2 négative	
2^{-1}	0,5
2^{-2}	0,25
2^{-3}	0,125
2^{-4}	0,0625
2^{-5}	0,03125

1.3. Base 16 ou système hexadécimal

Le système binaire est indispensable pour représenter les nombres dans un ordinateur, mais il faut dire qu'il n'est pas très pratique à manipuler pour l'espèce humaine. En effet, beaucoup de chiffres sont rapidement nécessaires ; ce qui prête rapidement à confusion. C'est la raison pour laquelle, la **base 16** (aussi appelé **système hexadécimal**) est couramment utilisée. De plus, il est très facile de représenter un nombre binaire sous sa forme hexadécimale (ce point sera décrit un peu plus loin).

La base 16 utilise **seize symboles** : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Les lettres A, B, C, D, E et F sont utilisées pour représenter respectivement les valeurs décimales 10, 11, 12, 13, 14 et 15.

Tout comme précédemment, ces symboles se placent côte à côte et peuvent être séparés par une virgule. Chaque symbole possède une position numérotée (ou colonne). Par exemple, un nombre hexadécimal peut se représenter de la manière suivante :

$$\overset{1}{9}\ \overset{0}{D}\ ,\ \overset{-1}{A}$$

Dans le système hexadécimal, le poids d'un symbole est 16 levé à la puissance de la position du symbole :

16^1	16^0		16^{-1}
9	D	,	A

Ce nombre peut se représenter comme une somme de produits :
$9D.A_{16} = \mathbf{9}\times16^1 + \mathbf{13}\times16^0 + \mathbf{10}\times16^{-1}$
$9D.A_{16} = 144 + 13 + 0{,}625$
$9D.A_{16} = 157{,}625_{10}$

Afin de pouvoir manipuler facilement les nombres hexadécimaux, il est conseillé de connaître par cœur les premières puissances de seize positives et négatives.

Puissance de 16 positive	
16^0	1
16^1	16
16^2	256
16^3	4 096
16^4	65 536

Puissance de 16 négative	
16^{-1}	0,0625

1.4. Base quelconque

Plus généralement, un nombre peut être représenté dans n'importe quelle base. La valeur d'une base peut être n'importe quel nombre entier supérieur ou égal à deux.

Une base *b* utilise *b* symboles. Si les dix chiffres ne suffisent plus, les lettres de l'alphabet sont utilisées (en commençant par la lettre 'A').

Ces symboles se placent côte à côte et peuvent être séparés par une virgule. En supposant que a_i représente un symbole d'une base *b*, n'importe quel nombre (*N*) de cette base peut s'écrire :

b^n	b^{n-1}	b^{n-2}		b^2	b^1	b^0		b^{-1}	b^{-2}		b^{-m+1}	b^{-m}	← Poids
a_n	a_{n-1}	a_{n-2}	…	a_2	a_1	a_0	,	a_{-1}	a_{-2}	…	a_{-m+1}	a_{-m}	

Partie entière — Séparateur — Partie fractionnaire

Enfin, ce nombre peut également se représenter comme une somme de produits : $N = \sum_{i=-m}^{i=n} a_i b^i$

2. Conversions

2.1. Conversion d'entiers entre 0 et 15

Afin d'effectuer les conversions décimales, binaires et hexadécimales le plus efficacement possible, il est conseillé de pouvoir convertir d'emblée les entiers entre 0 et 15. Le tableau ci-dessous doit donc être mémorisé par cœur :

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

Notez qu'au maximum, seulement quatre chiffres sont nécessaires pour représenter un nombre binaire entre 0 et 15. Les poids peuvent valoir uniquement 1, 2, 4 et 8, ce qui facilite les conversions.

Par exemple : $1101_2 = 8 + 4 + 1 = 13_{10}$

2.2. Conversion d'une base quelconque vers la base 10

2.2.1. Méthode

Convertir un nombre en base *b* vers sa représentation décimale :

- Écrire le nombre en base *b* comme une somme de produits : $\sum_{i=-m}^{i=n} a_i b^i$ (*Cf.* I.1.4. Base quelconque)
- Calculer le résultat.

2.2.2. Exemples

- **$4213{,}04_5$ → base 10**
  $= 4\times5^3 + 2\times5^2 + 1\times5^1 + 3\times5^0 + 0\times5^{-1} + 4\times5^{-2}$
  $= 4\times125 + 2\times25 + 1\times5 + 3\times1 + 4\times0{,}04$
  $= 500 + 50 + 5 + 3 + 0{,}16$
  $= 558{,}16_{10}$

- **$1\ 0110\ 1001{,}0101_2$ → base 10**
  $= 1\times2^8 + 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}$
  $= 256 + 64 + 32 + 8 + 1 + 0{,}25 + 0{,}0625$
  $= 361{,}3125_{10}$

- **$B3C{,}A_{16}$ → base 10**
  $= 11\times16^2 + 3\times16^1 + 12\times16^0 + 10\times16^{-1}$
  $= 11\times256 + 3\times16 + 12 + 10\times0{,}0625$
  $= 2\ 816 + 48 + 12 + 0{,}625$
  $= 2\ 876{,}625_{10}$

2.3. Conversion de la base 10 vers une base quelconque

La méthode permettant de convertir la partie entière d'un nombre est différente de celle permettant de convertir sa partie fractionnaire. Il faut donc convertir ces deux parties séparément.

2.3.1. Méthode de conversion pour la partie entière (divisions successives)

Pour convertir la partie entière d'un nombre en base 10 vers une base *b*.

- Diviser successivement la partie entière par *b* jusqu'à obtenir un quotient égal à 0.
- **La partie entière du nombre en base *b* sera constituée des restes des divisions.** En commençant par le dernier reste, écrire tous les restes côte à côte de gauche à droite (le dernier reste doit être le chiffre le plus à gauche du nombre en base *b*). Si *b* est supérieur à dix, les valeurs de certains restes peuvent être représentées par une lettre.

2.3.2. Méthode de conversion pour la partie fractionnaire (multiplications successives)

Pour convertir la partie fractionnaire d'un nombre en base 10 vers une base *b*.

- Multiplier la partie fractionnaire du nombre en base 10 par *b*.
- Multiplier la partie fractionnaire du produit ainsi obtenu par *b*.
- Répéter l'opération jusqu'à obtenir une précision suffisante ou une partie fractionnaire égale à 0.
- **La partie fractionnaire du nombre en base *b* sera constituée des parties entières des produits.** En commençant par le premier produit, écrire toutes les parties entières côte à côte de gauche à droite (la dernière partie entière doit être le chiffre le plus à droite du nombre en base *b*). Si *b* est supérieur à dix, les valeurs de certaines parties entières peuvent être représentées par une lettre.

2.3.3. Exemples

- **$715{,}40625_{10}$ → base 2**

715 / 2 = 357 Reste = **1** ↑
357 / 2 = 178 Reste = **1**
178 / 2 = 89 Reste = **0**
89 / 2 = 44 Reste = **1**
44 / 2 = 22 Reste = **0**
22 / 2 = 11 Reste = **0**
11 / 2 = 5 Reste = **1**
5 / 2 = 2 Reste = **1**
2 / 2 = 1 Reste = **0**
1 / 2 = 0 Reste = **1** ⊥ **$715_{10} = 10\ 1100\ 1011_2$**

0,40625 × 2 = **0**,8125 ⊤
0,8125 × 2 = **1**,625
0,625 × 2 = **1**,25
0,25 × 2 = **0**,5
0,5 × 2 = **1** ↓ **$0{,}4025_{10} = 0{,}01101_2$**

Par conséquent : **$715{,}40625_{10} = 10\ 1100\ 1011{,}01101_2$**

- **$41\ 695{,}78125_{10}$ → base 16**

41 695 / 16 = 2 605 Reste = **15** ↑
2 605 / 16 = 162 Reste = **13**
162 / 16 = 10 Reste = **2**
10 / 16 = 0 Reste = **10** ⊥ **$41\ 695_{10} = A2DF_{16}$**

0,78125 × 16 = **12**,5 ⊤
0,5 × 16 = **8** ↓ **$0{,}78125_{10} = 0.C8_{16}$**

Par conséquent : **$41\ 695{,}78125_{10} = A2DF{,}C8_{16}$**

2.4. Conversion de la base 2 vers une base en puissance de deux

2.4.1. Méthode

Chaque chiffre d'un nombre en base 2^n est constitué de n chiffres en base 2. Par conséquent, chaque chiffre d'un nombre en base 2^n peut être converti indépendamment en binaire. Et inversement, un nombre binaire peut être découpé en groupes de n chiffres convertibles indépendamment en base 2^n.

2.4.2. Exemples

- **$10010110110011111,11100101001_2$ → base 16 (2^4)**
  = $0001\ 0010\ 1101\ 1001\ 1111,1110\ 0101\ 0010_2$ ← *Groupes de 4 chiffres*
  = $12D9F,E52_{16}$

- **$10010110110011111,11100101001_2$ → base 8 (2^3)**
  = $010\ 010\ 110\ 110\ 011\ 111,111\ 001\ 010\ 010_2$ ← *Groupes de 3 chiffres*
  = $226637,7122_8$

- **$3A271E65,7B3C_{16}$ → base 2**
  = $0011\ 1010\ 0010\ 0111\ 0001\ 1110\ 0110\ 0101,0111\ 1011\ 0011\ 1100_2$ ← *Groupes de 4 chiffres*
  = $11101000100111000111100110010 1,01111011001111_2$

- **$4270,634_8$ → base 2**
  = $100\ 010\ 111\ 000,110\ 011\ 100_2$ ← *Groupes de 3 chiffres*
  = $100010111000,1100111_2$

2.5. Conversion d'une base quelconque vers une base quelconque

2.5.1. Méthode

Si les deux bases sont des puissances du même entier (*n*), les nombres doivent être convertis en passant par la base *n*. Dans le cas contraire, il faut passer par la base 10.

2.5.2. Exemples

- **$3D7,5_{16}$ → base 8** ← *$16 = 2^4$ et $8 = 2^3$ (via base 2)*
  = $0011\ 1101\ 0111,0101_2$
  = $001\ 111\ 010\ 111,010\ 100_2$
  = $1727,24_8$

- **23_5 → base 8**

  $23_5 = 2\times5 + 3$
  $23_5 = 13_{10}$

  13 / 8 = 1 Reste = **5**
  1 / 8 = 0 Reste = **1**

  $23_5 = 15_8$

II. Opérations fondamentales

1. Addition binaire

Les additions en représentation binaire suivent les mêmes règles que celles en représentation décimale. Par exemple, effectuons l'addition suivante : 1010110 + 1011101

Étape 1

$$\begin{array}{rr} & 1\ 0\ 1\ 0\ 1\ 1\ \mathbf{0} \\ + & 1\ 0\ 1\ 1\ 1\ 0\ \mathbf{1} \\ \hline & \mathbf{1} \end{array}$$

Étape 2

$$\begin{array}{rr} & 1\ 0\ 1\ 0\ 1\ \mathbf{1}\ 0 \\ + & 1\ 0\ 1\ 1\ 1\ \mathbf{0}\ 1 \\ \hline & \mathbf{1}\ 1 \end{array}$$

Étape 3

$$\begin{array}{rr} & \mathbf{1} \\ & 1\ 0\ 1\ 0\ \mathbf{1}\ 1\ 0 \\ + & 1\ 0\ 1\ 1\ \mathbf{1}\ 0\ 1 \\ \hline & \mathbf{0}\ 1\ 1 \end{array}$$

Étape 4

$$\begin{array}{rr} & \mathbf{1}\ \mathbf{1} \\ & 1\ 0\ 1\ \mathbf{0}\ 1\ 1\ 0 \\ + & 1\ 0\ 1\ \mathbf{1}\ 1\ 0\ 1 \\ \hline & \mathbf{0}\ 0\ 1\ 1 \end{array}$$

Étape 5

$$\begin{array}{rr} & \mathbf{1}\ \mathbf{1}\ 1 \\ & 1\ 0\ \mathbf{1}\ 0\ 1\ 1\ 0 \\ + & 1\ 0\ \mathbf{1}\ 1\ 1\ 0\ 1 \\ \hline & \mathbf{1}\ 0\ 0\ 1\ 1 \end{array}$$

Étape 6

$$\begin{array}{rr} & \mathbf{1}\ 1\ 1 \\ & 1\ \mathbf{0}\ 1\ 0\ 1\ 1\ 0 \\ + & 1\ \mathbf{0}\ 1\ 1\ 1\ 0\ 1 \\ \hline & \mathbf{1}\ 1\ 0\ 0\ 1\ 1 \end{array}$$

Étape 7

$$\begin{array}{rr} & \mathbf{1}\ \ 1\ 1\ 1 \\ & \mathbf{1}\ 0\ 1\ 0\ 1\ 1\ 0 \\ + & \mathbf{1}\ 0\ 1\ 1\ 1\ 0\ 1 \\ \hline & 1\ \mathbf{0}\ 1\ 1\ 0\ 0\ 1\ 1 \end{array}$$

Étape 8

$$\begin{array}{rr} & \mathbf{1}\ \ 1\ 1\ 1 \\ & 1\ 0\ 1\ 0\ 1\ 1\ 0 \\ + & 1\ 0\ 1\ 1\ 1\ 0\ 1 \\ \hline & \mathbf{1}\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \end{array}$$

- **Étape 1 :** 0 + 1 = 1
- **Étape 2 :** 1 + 0 = 1
- **Étape 3 :** 1 + 1 = 10 (= 0 avec une retenue de 1)
- **Étape 4 :** 1 + 0 + 1 = 10 (= 0 avec une retenue de 1)
- **Étape 5 :** 1 + 1 + 1 = 11 (= 1 avec une retenue de 1)
- **Étape 6 :** 1 + 0 + 0 = 1
- **Étape 7 :** 1 + 1 = 10 (= 0 avec une retenue de 1)
- **Étape 8 :** 1 + 0 + 0 = 1

Par conséquent : 1010110 + 1011101 = 10110011

2. Soustraction binaire

Les soustractions en représentation binaire suivent les mêmes règles que celles en représentation décimale. Par exemple, effectuons la soustraction suivante : 1101001 – 110110

Étape 1

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & 0 & \mathbf{1} \\ - & 0 & 1 & 1 & 0 & 1 & 1 & \mathbf{0} \\ \hline & & & & & & & \mathbf{1} \end{array}$$

Étape 2

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & {}_{1}\mathbf{0} & 1 \\ - & 0 & 1 & 1 & 0 & {}_{1+}1 & \mathbf{1} & 0 \\ \hline & & & & & & \mathbf{1} & 1 \end{array}$$

Étape 3

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & {}_{1}\mathbf{0} & {}_{1}0 & 1 \\ - & 0 & 1 & 1 & {}_{1+}0 & {}_{1+}\mathbf{1} & 1 & 0 \\ \hline & & & & & \mathbf{0} & 1 & 1 \end{array}$$

Étape 4

$$\begin{array}{cccccccc} & 1 & 1 & 0 & \mathbf{1} & {}_{1}0 & {}_{1}0 & 1 \\ - & 0 & 1 & 1 & {}_{1+}\mathbf{0} & {}_{1+}1 & 1 & 0 \\ \hline & & & & \mathbf{0} & 0 & 1 & 1 \end{array}$$

Étape 5

$$\begin{array}{cccccccc} & 1 & 1 & {}_{1}\mathbf{0} & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & 0 & {}_{1+}1 & \mathbf{1} & {}_{1+}0 & {}_{1+}1 & 1 & 0 \\ \hline & & & \mathbf{1} & 0 & 0 & 1 & 1 \end{array}$$

Étape 6

$$\begin{array}{cccccccc} & 1 & {}_{1}\mathbf{1} & {}_{1}0 & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & {}_{1+}0 & {}_{1+}\mathbf{1} & 1 & {}_{1+}0 & {}_{1+}1 & 1 & 0 \\ \hline & & \mathbf{1} & 1 & 0 & 0 & 1 & 1 \end{array}$$

Étape 7

$$\begin{array}{cccccccc} & \mathbf{1} & {}_{1}1 & {}_{1}0 & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & {}_{1+}\mathbf{0} & {}_{1+}1 & 1 & {}_{1+}0 & {}_{1+}1 & 1 & 0 \\ \hline & \mathbf{0} & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

- **Étape 1 :** 1 – 0 = 1
- **Étape 2 :** 0 – 1 → on emprunte 1 → 10 – 1 = 1
- **Étape 3 :** 0 – (1 + 1) = 0 – 10 → on emprunte 1 → 10 – 10 = 0
- **Étape 4 :** 1 – (1 + 0) = 1 – 1 = 0
- **Étape 5 :** 0 – 1 = 1 → on emprunte 1 → 10 – 1 = 1
- **Étape 6 :** 1 – (1 + 1) = 1 – 10 → on emprunte 1 → 11 – 10 = 1
- **Étape 7 :** 1 – (1 + 0) = 1 – 1 = 0

Par conséquent : 1101001 – 110110 = 110011

3. Multiplication binaire

Les multiplications en représentation binaire suivent les mêmes règles que celles en représentation décimale. Par exemple, effectuons la multiplication suivante : 10110 × 101

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & 0 \\ \times & & & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 0 & 0 & 0 \\ + & 1 & 0 & 1 & 1 & 0 \\ \hline 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}$$

← Multiplicande
← Multiplicateur
← **Étape 1 :** Le chiffre le plus à droite du multiplicateur vaut 1 : écrire le multiplicande.
← **Étape 2 :** Le prochain chiffre du multiplicateur vaut 0 : écrire des 0 décalés à gauche.
← **Étape 3 :** Le prochain chiffre vaut 1 : écrire le multiplicande décalé à gauche.
← **Étape 4 :** Additionner les produits partiels.

Par conséquent : 10110 × 101 = 1101110

4. Division binaire

Les divisions en représentation binaire suivent les mêmes règles que celles en représentation décimale. Par exemple, effectuons la multiplication suivante : 11111001 / 110

Étape 1 :

1 1 1 1 1 0 0 1	1 1 0
– 1 1 0	**1**
1	

- Le diviseur (110) va une fois dans les trois premiers chiffres du dividende (111).
- Inscrire ‘1’ dans le quotient.
- Soustraire le diviseur (110) des trois premiers chiffres du dividende (111).

Étape 2 :

1 1 1 **1** 1 0 0 1	1 1 0
– 1 1 0 ↓	1 **0**
1 **1**	

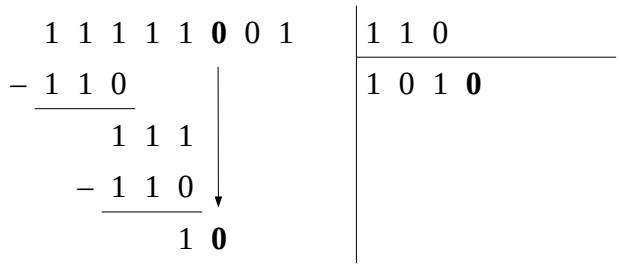
- Descendre le prochain chiffre du dividende (1) jusqu’au reste.
- Le nouveau reste (11) est plus petit que le diviseur (110).
- Inscrire ‘0’ dans le quotient.

Étape 3 :

1 1 1 1 **1** 0 0 1	1 1 0
– 1 1 0 ↓	1 0 **1**
1 1 1	
– 1 1 0	
1	

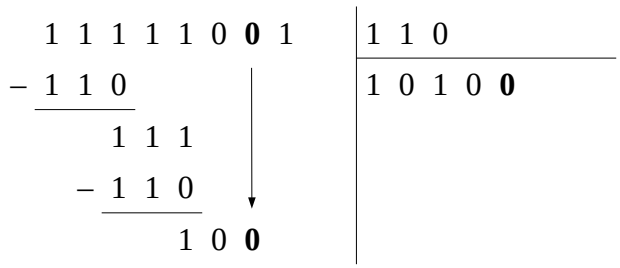
- Descendre le prochain chiffre du dividende (1).
- Le diviseur va une fois dans le nouveau reste (111).
- Inscrire ‘1’ dans le quotient.
- Soustraire le diviseur.

Étape 4 :



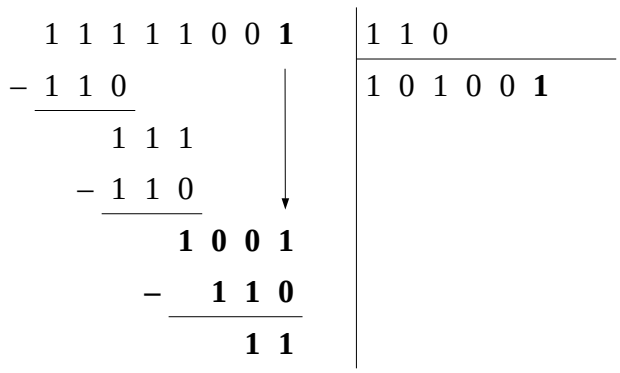
- Descendre le prochain chiffre du dividende (0).
- Le nouveau reste (10) est plus petit que le diviseur.
- Inscrire ‘0’ dans le quotient.

Étape 5 :



- Descendre le prochain chiffre du dividende (0).
- Le nouveau reste (100) est toujours plus petit que le diviseur.
- Inscrire ‘0’ dans le quotient.

Étape 6 :



- Descendre le prochain chiffre du dividende (1).
- Le diviseur va une fois dans le nouveau reste (1001).
- Inscrire ‘1’ dans le quotient.
- Soustraire le diviseur.

Étape 7 :

```
  1 1 1 1 1 0 0 1   | 1 1 0
– 1 1 0             | 1 0 1 0 0 1 ,
    1 1 1           |
  – 1 1 0           |
      1 0 0 1       |
    –   1 1 0       |
        1 1 0       |
```

- Tous les chiffres du dividende ont été descendus.
- Ajouter une virgule au quotient.
- Ajouter un 0 au reste.

Étape 8 :

```
  1 1 1 1 1 0 0 1   | 1 1 0
– 1 1 0             | 1 0 1 0 0 1 , 1
    1 1 1           |
  – 1 1 0           |
      1 0 0 1       |
    –   1 1 0       |
        1 1 0       |
      – 1 1 0       |
          0         |
```

- Le diviseur va une fois dans le nouveau reste (110).
- Inscrire ‘1’ dans le quotient.
- Soustraire le diviseur.
- Le dernier reste vaut 0.

Par conséquent : 11111001 / 110 = 101001,1

5. Addition hexadécimale

Pour effectuer les additions hexadécimales, additionnez chaque colonne avec les valeurs décimales des chiffres. Ensuite, convertissez le résultat d'une colonne en hexadécimal et propagez la retenue si elle existe.

Exemple:

$$\begin{array}{rcccc} & {\scriptstyle 1} & & {\scriptstyle 1} & \\ & 3 & B & 5 & \\ & C & 1 & A & \\ + & D & 2 & F & \\ \hline 1 & C & F & E & \end{array}$$

Étape 1:

- $5_{16} + A_{16} + F_{16} = 5_{10} + 10_{10} + 15_{10} = 30_{10}$
- $30_{10} = 1E_{16}$ (retenue = 1)

Étape 2:

- $1_{16} + B_{16} + 1_{16} + 2_{16} = 1_{10} + 11_{10} + 1_{10} + 2_{10} = 15_{10}$
- $15_{10} = F_{16}$ (pas de retenue)

Étape 3:

- $3_{16} + C_{16} + D_{16} = 3_{10} + 12_{10} + 13_{10} = 28_{10}$
- $28_{10} = 1C_{16}$ (retenue = 1)

III. Bits, mots et octets

1. Définitions

Un *bit*, qui est la contraction de *binary digit*, est la plus petite unité d'information qu'un ordinateur peut manipuler. Un bit possède la valeur 0 ou 1.

Un *mot* est un regroupement d'un ou de plusieurs bits pouvant être manipulés comme un tout. La taille d'un mot (son nombre de bits) est fixe. Par exemple :

- Un mot de 1 bit possède deux combinaisons possibles : 0 et 1.
- Un mot de 2 bits possède quatre combinaisons possibles : 00, 01, 10 et 11.
- Un mot de 3 bits possède huit combinaisons possibles : 000, 001, …, 110, 111.
- Un mot de 4 bits possède seize combinaisons possibles : 0000, 0001, …, 1111.
- **Un mot de *n* bits possède 2^n combinaisons possibles.**

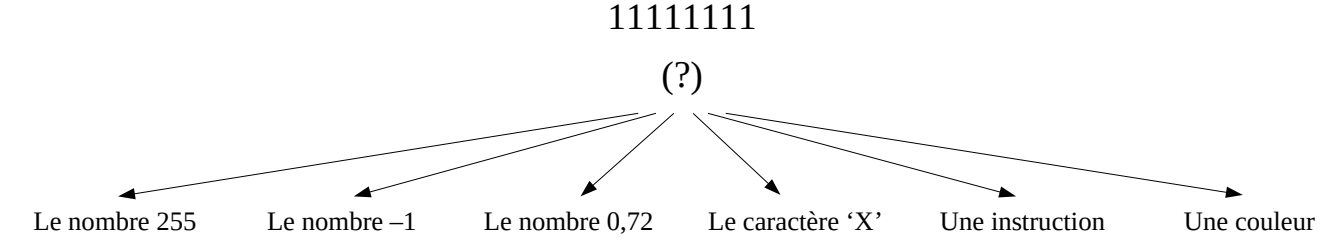
Dans certains contextes, la taille d'un mot peut être implicite. Par exemple, lorsque l'on programme un microprocesseur de la famille des 68000, la taille par défaut d'un mot est de 16 bits : le terme « mot » signifie « mot de 16 bits » et le terme « mot long » signifie « mot de 32 bits ». Par contre, lorsque l'on programme un microprocesseur de la famille des ARM 32 bits, la taille par défaut d'un mot est de 32 bits : le terme « mot » signifie « mot de 32 bits » et le terme « demi-mot » signifie « mot de 16 bits ». Hors contexte, la taille d'un mot est indéfinie et doit être précisée.

Un *octet* est un mot de 8 bits. Il possède 256 combinaisons possibles. De nos jours, les microprocesseurs sont conçus pour manipuler des octets ou des groupes d'octets.

Le bit le plus à gauche d'un mot est le **bit de poids fort** ; souvent abrégé **MSB** (*Most Significant Bit*). Le bit le plus à droite est le **bit de poids faible** ; souvent abrégé **LSB** (*Least Significant Bit*). Les bits d'un mot de *n* bits sont numérotés de 0 à *n – 1* : le bit 0 est le LSB ; le bit *n – 1* est le MSB.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← Numéro du bit
Exemple d'un mot de 16 bits :	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	
	↑															↑	
	MSB															**LSB**	

Maintenant que vous savez ce qu'est un mot, une question importante doit être posée : qu'est-ce qu'un mot représente au sein d'un ordinateur ? Par exemple, que représente l'octet suivant ?



En fait, il n'est pas possible de répondre à cette question, car le contexte n'est pas précisé. **Un mot en lui-même ne représente rien.** Sa représentation dépend d'un contexte. Hors contexte, tout ce que l'on sait concernant un octet, c'est qu'il possède 256 combinaisons de 0 et de 1. Par conséquent, en fonction du contexte, un programmeur sera capable d'associer un sens particulier à chaque combinaison. Par exemple, l'octet 11111111_2 peut représenter la valeur 255 si l'encodage utilisé est celui des nombres non signés, ou encore la valeur –1 si l'encodage utilisé est celui des nombres signés. Autrement dit, l'octet 11111111_2 peut représenter n'importe quoi en fonction du contexte.

Pour résumer, un ordinateur peut manipuler uniquement des groupes de 0 et de 1. Si l'on souhaite manipuler d'autres types de données (par exemple, des entiers, des caractères, des images, etc.). Nous devons les encoder avec des 0 et des 1.

Par souci de commodité, un mot est souvent représenté dans sa forme hexadécimale, Par exemple, l'octet 11111111_2 est équivalent à l'octet FF_{16}.

2. Complément à un

Le complément à un d'un mot s'obtient en inversant chacun de ses bits.

Quelques exemples :

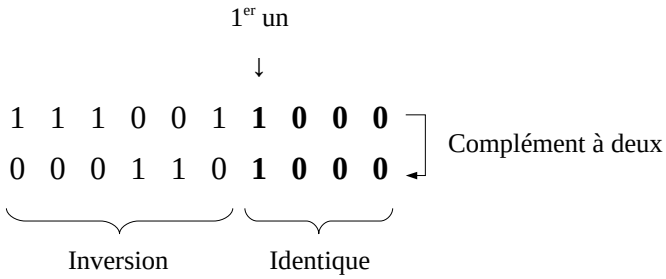
Mot		**Complément à 1 du mot**	
Binaire	**Hexadécimal**	**Binaire**	**Hexadécimal**
00000000	00	11111111	FF
00110110	36	11001001	C9
10011000	98	01100111	67
10111100	BC	01000011	43
11111111	FF	00000000	00

3. Complément à deux

Le complément à deux d'un mot s'obtient en inversant chacun de ses bits et en lui additionnant un. Autrement dit, il s'agit du complément à un plus un.

Par exemple, le complément à un du mot de 10 bits 1110011000_2 est $0001100111_2 + 1_2 = 0001101000_2$

Le complément à deux s'obtient également en conservant la valeur des bits de poids faible (du LSB jusqu'au premier 1) et en inversant les autres bits.



Quelques exemples :

Mot		**Complément à 2 du mot**	
Binaire	**Hexadécimal**	**Binaire**	**Hexadécimal**
00000000	00	00000000	00
00110110	36	11001010	CA
10011000	98	01101000	68
10111100	BC	01000100	44
11111111	FF	00000001	01

Le complément à deux est principalement utilisé pour l'encodage des entiers signés.

IV. Encodage d’entiers

1. Entiers non signés

1.1. Introduction

Il existe plusieurs façons d’encoder les nombres non signés. Nous étudierons la plus couramment utilisée en informatique. C’est aussi la plus naturelle et la plus évidente.

Chaque combinaison d’un mot représente **directement** la valeur d’un nombre binaire positif comme si le mot lui-même était ce nombre. C’est-à-dire que **la représentation binaire du mot est identique à la représentation binaire du nombre**. Par exemple, l’encodage du nombre binaire 101_2 sur 8 bits non signés donnera : 00000**101**.

Étant donné qu’un mot de n bits possède 2^n combinaisons, il est possible de représenter des entiers compris entre 0 et $2^n - 1$.

1.2. Encodage d’entiers non signés

Pour convertir un entier non signé en un mot binaire, il faut utiliser la méthode de conversion de la partie entière d’un nombre de la base 10 vers la base 2. *Cf.* I.2.3. Conversion de la base 10 vers une base quelconque.

Exemple : Encodons le nombre 50 en un mot binaire de 8 bits.

50 / 2 = 25 Reste = **0**
25 / 2 = 12 Reste = **1**
12 / 2 = 6 Reste = **0**
6 / 2 = 3 Reste = **0**
3 / 2 = 1 Reste = **1**
1 / 2 = 0 Reste = **1**

$50_{10} = 00110010_2$

1.3. Décodage d’entiers non signés

Pour convertir un mot binaire en un entier non signé, il faut utiliser la méthode de conversion d’un nombre de la base 2 vers la base 10. *Cf.* I.2.2. Conversion d’une base quelconque vers la base 10.

Exemple : Décodons le mot de 6 bits 100110_2.

$100110_2 = 32 + 4 + 2 = 38_{10}$

1.4. Dépassement non signé

Lorsqu'une opération sur des mots binaires à taille fixe est effectuée, certains résultats ne peuvent plus être encodés correctement, car le nombre de bits est insuffisant. L'encodage du résultat ne correspond donc pas au résultat correct attendu. Nous disons alors qu'un **dépassement non signé** s'est produit.

Par exemple, considérons l'addition sur 8 bits suivante (les deux opérandes et le résultat sont sur 8 bits) :

$$
\begin{array}{cccccccccl}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & {\scriptstyle 1} & & \\
 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \leftarrow 250_{10} \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \leftarrow 10_{10} \\
\hline
 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \leftarrow 4_{10}\ (\cancel{260}_{10}) \\
 & \uparrow & & & & & & & & & \\
 & \text{Retenue} & & & & & & & & &
\end{array}
$$

Le résultat attendu est 260 et nécessite 9 bits pour être encodé. En fait, le neuvième bit est **la retenue**. Par conséquent, le résultat sur 8 bits est 4 et ne représente pas le résultat attendu (la retenue est ignorée).

Un octet permet d'encoder des valeurs entre 0 et 255. Manifestement, le résultat attendu (260) est trop grand et ne peut pas être encodé sur 8 bits.

Prenons comme exemple pratique le programme en langage C suivant :

```c
#include <stdio.h>

int main()
{
    // Declare 3 variables de 8 bits non signés.
    unsigned char a, b, c;

    // Initialise les opérandes.
    a = 250;
    b = 10;

    // Effectue addition.
    c = a + b;

    // Affiche les 3 variables.
    printf("%u + %u = %u", a, b, c);

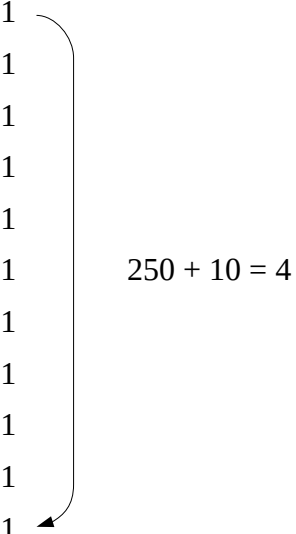
    return 0;
}
```

Ce programme affiche le texte suivant dans la fenêtre console :

```
250 + 10 = 4
```

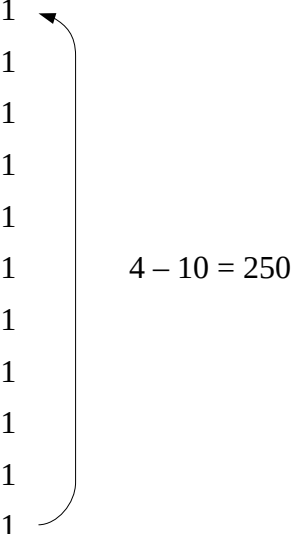
En fait, l'incrémentation d'un mot est cyclique :

Décimal	Binaire	
248	11111000	
249	11111001	+ 1
250	11111010	+ 1
251	11111011	+ 1
252	11111100	+ 1
253	11111101	+ 1
254	11111110	+ 1
255	11111111	+ 1
0	00000000	+ 1
1	00000001	+ 1
2	00000010	+ 1
3	00000011	+ 1
4	00000100	+ 1
5	00000101	+ 1



Même raisonnement pour la soustraction :

Décimal	Binaire	
248	11111000	
249	11111001	– 1
250	11111010	– 1
251	11111011	– 1
252	11111100	– 1
253	11111101	– 1
254	11111110	– 1
255	11111111	– 1
0	00000000	– 1
1	00000001	– 1
2	00000010	– 1
3	00000011	– 1
4	00000100	– 1
5	00000101	– 1



Nous pouvons noter que :

- 255 + 1 = 0
- 0 – 1 = 255

Remarque :
Le terme de retenue (*carry / borrow*) est souvent utilisé pour décrire un dépassement non signé.

2. Entiers signés

2.1. Introduction

Il existe plusieurs façons d'encoder les nombres signés. Nous étudierons la plus couramment utilisée en informatique. Elle se fonde sur l'utilisation du **complément à deux**.

Les différentes combinaisons d'un mot se divisent en deux parties. La première moitié est utilisée pour représenter les entiers positifs et la seconde pour représenter les entiers négatifs.

Étant donné qu'un mot de n bits possède 2^n combinaisons, il est possible de représenter des entiers compris entre -2^{n-1} et $2^{n-1} - 1$.

Par exemple, un mot de 4 bits peut représenter les entiers compris entre –8 et 7 :

Entier	Mot de 4 bits	
–8	1000	Entiers négatifs (MSB = 1)
–7	1001	
–6	1010	
–5	1011	
–4	1100	
–3	1101	
–2	1110	
–1	1111	
0	0000	Entiers positifs (MSB = 0)
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	

Nous pouvons remarquer que **le bit de poids fort vaut 0 quand le nombre est positif et 1 quand il est négatif**. C'est pourquoi ce bit de poids fort est aussi appelé le **bit de signe**.

Dans ce système d'encodage, le complément à deux d'un mot représente l'opposé du nombre.

Par exemple, $C2\,(3_{10}) = -3_{10}$:
- $3_{10} = 0011_2$
- $C2\,(0011_2) = 1101_2 = -3_{10}$

Notons qu'un octet représente les entiers compris entre –128 et 127.

2.2. Soustraction à l'aide du complément à deux

Étant donné que le complément à deux représente l'opposé d'un nombre, une soustraction peut être remplacée par une addition (la retenue devra être ignorée).

Par exemple, soit la soustraction sur 8 bits suivante : $115_{10} - 62_{10}$

Nous savons que :
- $115_{10} = 01110011_2$
- $62_{10} = 00111110_2$
- $C2\,(62_{10}) = 11000010_2$

Soustraction

	0	1	1	1	0	0	1	1	← 115_{10}
–	0	0	1	1	1	1	1	0	← 62_{10}
	0	0	1	1	0	1	0	1	← 53_{10}

Addition

	1	1					1		
	0	1	1	1	0	0	1	1	← 115_{10}
+	1	1	0	0	0	0	1	0	← $C2\,(62_{10}) = -62$
	~~1~~	0	0	1	1	0	1	0	1 ← 53_{10}
	↑								
	~~Retenue~~								

Les résultats sur 8 bits sont identiques : $00110101_2 = 53_{10}$

Par conséquent : $115_{10} - 62_{10} = 115_{10} + C2\,(62_{10}) = 53_{10}$

2.3. Encodage d’entiers signés

La méthode de conversion d’un entier signé dépend du signe de l’entier.

Si l’entier est positif, il peut être encodé de la même façon qu’un entier non signé.
Cf. IV.1.2. Encodage d’entiers non signés.

Si l’entier est négatif :
- Convertir sa valeur absolue en binaire.
- Déterminer le complément à deux de sa valeur absolue.

Exemple : Encodons le nombre –17 en un mot binaire de 8 bits.

17 / 2 = 8 Reste = **1**
8 / 2 = 4 Reste = **0**
4 / 2 = 2 Reste = **0**
2 / 2 = 1 Reste = **0**
1 / 2 = 0 Reste = **1**

$17_{10} = 00010001_2$
$-17 = C2\,(00010001_2)$
$-17 = 11101111_2$

2.4. Décodage d’entiers signés

La méthode de conversion dépend du bit de signe du mot (c’est-à-dire du bit de poids fort).

Si le bit de signe vaut 0, l’entier est positif et peut être décodé comme un entier non signé.
Cf. IV.1.3. Décodage d’entiers non signés.

Si le bit de signe vaut 1, l’entier est négatif :
- Déterminer le complément à deux du mot.
- Convertir le complément à deux vers sa forme décimale et placer le signe moins à sa gauche.

Exemple : Décodons le mot de 6 bits 100110_2.

Le bit de signe vaut 1 :
$C2(100110_2) = 011010_2$
$011010_2 = 16 + 8 + 2 = 26_{10}$
$100110_2 = -26_{10}$

2.5. Dépassement signé

Tout comme les entiers non signés, lorsqu'une opération à taille fixe est effectuée, il peut arriver que le résultat attendu ne puisse pas être encodé correctement. Nous disons alors qu'un **dépassement signé** s'est produit.

Par exemple, considérons l'addition sur 8 bits suivante :

$$
\begin{array}{rcl}
 & 1\ 1\ 1\ 1 & \\
 & 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0 & \leftarrow\ 120_{10} \\
+ & 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 & \leftarrow\ 10_{10} \\
\hline
 & \mathbf{1}\ 0\ 0\ 0\ 0\ 0\ 1\ 0 & \leftarrow\ -126_{10}\ (\cancel{130}_{10}) \\
 & \uparrow & \\
 & \text{Négatif} &
\end{array}
$$

Un octet permet d'encoder des valeurs entre –128 et 127. Manifestement, le résultat attendu (130) est trop grand et ne peut pas être encodé sur 8 bits. Nous pouvons également remarquer que le résultat erroné est négatif (–126).

Prenons comme exemple pratique le programme en langage C suivant :

```c
#include <stdio.h>

int main()
{
    // Declare 3 variables de 8 bits signés.
    signed char a, b, c;

    // Initialise les opérandes.
    a = 120;
    b = 10;

    // Effectue addition.
    c = a + b;

    // Affiche les 3 variables.
    printf("%i + %i = %i", a, b, c);

    return 0;
}
```

Ce programme affiche le texte suivant dans la fenêtre console :

```
120 + 10 = -126
```

En fait, l'incrémentation d'un mot est cyclique :

Décimal	Binaire	
118	01110110	
119	01110111	+ 1
120	01111000	+ 1
121	01111001	+ 1
122	01111010	+ 1
123	01111011	+ 1
124	01111100	+ 1
125	01111101	+ 1
126	01111110	+ 1
127	01111111	+ 1
–128	10000000	+ 1
–127	10000001	+ 1
–126	10000010	+ 1
–125	10000011	+ 1

120 + 10 = –126

Même raisonnement pour la soustraction :

Décimal	Binaire	
118	01110110	
119	01110111	– 1
120	01111000	– 1
121	01111001	– 1
122	01111010	– 1
123	01111011	– 1
124	01111100	– 1
125	01111101	– 1
126	01111110	– 1
127	01111111	– 1
–128	10000000	– 1
–127	10000001	– 1
–126	10000010	– 1
–125	10000011	– 1

– 126 – 10 = 120

2.6. Extension de signe

L'extension de signe sert à augmenter le nombre de bits utilisés pour encoder un entier signé. Il s'agit de dupliquer le bit de signe vers la gauche.

Par exemple, si nous voulons étendre l'encodage sur 8 bits d'un entier signé vers un encodage sur 16 bits, le bit de signe du mot de 8 bits (bit 7) doit être dupliqué sur tous les nouveaux bits de gauche (bits 8 à 15).

Exemple pour un nombre positif :

Représentation décimale	104
Mot binaire sur 8 bits	01101000
Mot binaire sur 16 bits	**00000000**01101000

Extension de signe

Exemple pour un nombre négatif :

Représentation décimale	–115
Mot binaire sur 8 bits	10001101
Mot binaire sur 16 bits	**11111111**10001101

Extension de signe

V. Autres types d'encodage

Il existe un grand nombre d'encodages différents utilisés par toute une variété d'application. Nous en verrons uniquement quelques-uns.

1. BCD

BCD sont les initiales de *Binary-Coded Decimal*. Cet encodage est principalement utilisé pour l'affichage de nombre dans le système décimal.

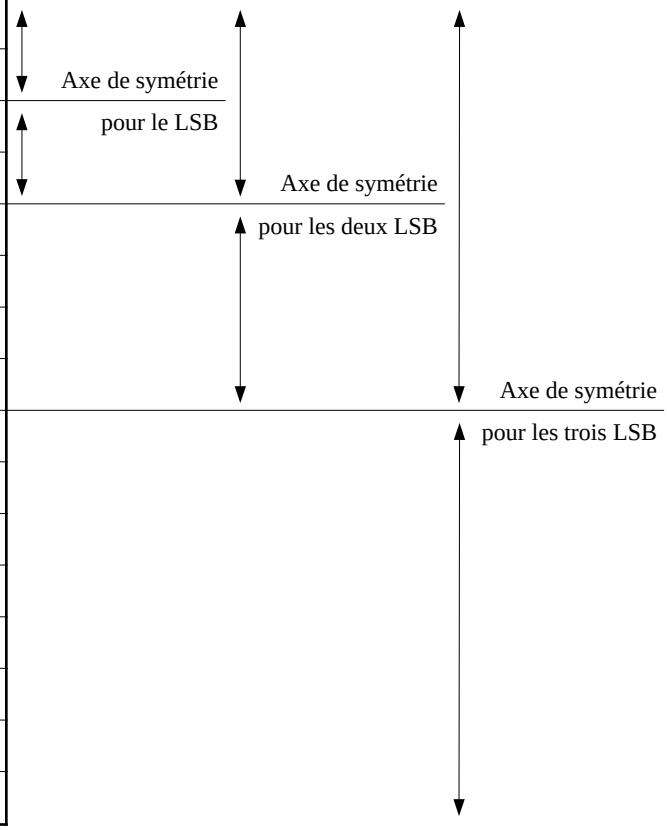
Chaque chiffre décimal est encodé sur 4 bits. Par exemple : $3\,768_{10} = 0011\ 0111\ 0110\ 1000_{BCD}$

3	7	6	8	← Décimal
0011	0111	0110	1000	← BCD

2. Code Gray ou binaire réfléchi

Le code Gray est une autre façon d'encoder les entiers non signés. Le tableau suivant en donne les seize premières représentations.

Décimal	Binaire naturel	Code Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000



Le code Gray est également appelé **binaire réfléchi**, car les bits de poids faibles possèdent un axe de symétrie. Contrairement au binaire naturel, les chiffres ne possèdent pas de poids.

La principale caractéristique du code Gray est qu'un seul bit change d'une représentation à l'autre. C'est la raison pour laquelle cet encodage est souvent utilisé dans certains algorithmes, dans des applications particulières, dans la correction d'erreurs ou encore dans les tableaux de Karnaugh. Ces derniers sont utilisés pour réduire les expressions logiques au maximum ; nous les étudierons dans un prochain chapitre.

3. ASCII

ASCII est l'acronyme de *American Standard Code for Information Interchange*.

Le code ASCII est utilisé pour l'encodage des caractères. Par exemple :

- '0' = 30_{16} = 00110000_2
- 'A' = 41_{16} = 01000001_2
- 'a' = 61_{16} = 01100001_2