Chapitre 1
Bases de numération

Introduction

Toute information traitée par ordinateur est stockée dans les **registres du processeur**, en **mémoire principale**, sur les **registres des périphériques** ou sur **les mémoires de masse**.

Les technologies actuelles se prêtent au stockage d'information en **logique binaire**.

L'information est codée sur des **ensembles de bits (8 bits, 16 bits, 32 bits ou 64 bits)**, selon la précision désirée.

Les données traitées par l'ordinateur sont essentiellement **des instructions, des caractères, des nombres entiers et des nombres réels.**

Les **caractères** sont généralement représentés sur **8 bits** (octet ou byte), les **entiers** sur 16 bits ou **32 bits** et les nombres **réels** sur **32 bits** ou **64 bits**

La numération

De tout temps, l'homme a cherché à compter avec plus ou moins de réussite. Dès la préhistoire, on retrouve sur des os des entailles pouvant avoir servi à compter des animaux ou des objets. S'il est relativement simple de compter avec des traits jusqu'à 5, il devient plus difficile de compter au-delà. Imaginez le nombre 100 représenté avec des entailles, l'erreur est assurée... Les romains ont mis en place un système de numération basé sur des symboles littéraux.

Cette numération très utile à cette époque présente de nos jours l'inconvénient majeur de ne pas pouvoir autoriser les opérations de base (addition, soustraction) ce qui est gênant. Essayez d'ajouter de tête MCDIV + CCLXXI. Problème n'est-ce pas ? La réponse est : MDCLXXV (1675).

C'est vers l'an 750 en Inde, que les indiens ont mis en place la numération que nous utilisons aujourd'hui avec tous ses chiffres. Les Indiens les ont ensuite transmis aux Arabes qui les ont transmis aux Européens vers l'an 1200. Et ce n'est qu'environ trois siècles plus tard que les chiffres se généraliseront, un peu. C'est pour cette raison que l'on parle de chiffres arabes. Si les Arabes ne sont pas à l'origine des chiffres, ils sont par contre à l'origine des mathématiques. C'est l'astronome et mathématicien Al Khuwârizmî qui établira les fondements des règles du calcul algébrique.

La numération

La numération fait appel à deux principes fondamentaux que l'on retrouvera dans toutes les bases de calcul.

- Le premier principe fondamental est le principe de position, on associe à un chiffre qui a une position dans un nombre une valeur parfaitement définie. Par exemple 1 signifie que l'on a une fois l'unité. Le même chiffre placé dans le nombre 1254 signifie qu'il y a une fois mille.

- Le deuxième principe fondamental est le principe du zéro. Le zéro matérialise une position où il y a absence d'éléments. Pour que le nombre 10 signifie dix, il faut placer le un sur la colonne des dizaines et matérialiser l'absence d'unité par un zéro. Le zéro donne son sens au nombre en positionnant le 1 dans la colonne des dizaines.

Bases de calcul

Les bases de calcul sont nombreuses même si peu sont réellement utilisées. On peut citer un certain nombre de bases qui ont eu ou qui ont toujours une très grande importance :

- La base 2 ou base binaire très utilisée en A.I.I. est la base de la logique booléenne ou algèbre de Boole.
- La base 4. Cette base a été l'objet de nombreuses polémiques et aurait pu devenir la base universelle.
- La base 5 est une base très intéressante qui permet facilement de compter jusqu'à 30 avec ses dix doigts. La première main comptant les unités et la deuxième comptant les cinquaines.
- La base 8 (base octale). Utilisée il y a un certain temps en informatique lorsque les machines utilisées étaient peu gourmande en puissance de bus.
- La base 10 (base décimale). Elle est considérée comme la base universelle. de ce fait nous l'utilisons presque tout le temps.

Bases de calcul

- La base 12 qui est une base religieuse établie sur les douze signes du zodiaque. Cette base a été utilisée principalement dans le commerce (c'est à cause de l'utilisation de cette base que l'on parle encore d'une douzaine d'œufs ou d'une douzaine d'huîtres, dans une journée il y a 24 heures divisées en deux fois douze heures, etc.)
- La base 16 (base hexadécimale). Cette base est très utilisée dans le monde de la micro-informatique et des automates. Elle permet de coder un mot (16 bits) sur 4 variables hexadécimales. Cette base fait intervenir tous les chiffres de la base 10 complétée par les 6 premières lettres de l'alphabet.
- La base 20 cette base a été construite à partir des dix doigts des mains et des pieds de l'homme. Cette base a été très utilisée et il en subsiste quelques traces dans notre numération actuelle (quatre-vingts, quatre-vingt-dix).
- La base 60 cette base a été construite sur des concepts religieux. D'un maniement complexe, elle est toujours utilisée de nos jours: sur un cercle, il y a 360° qui sont divisés chacun en 60 minutes divisées chacune en 60 secondes. Il en va de même pour chacune des heures de la journée qui sont divisées chacune en 60 minutes, elles-mêmes divisées en 60 secondes. Mais notons que la la division des secondes se fait en centième de seconde.

Bases de calcul

- En informatique, chaque signal n'ayant que deux états possibles, état 0 et état 1, les bases que nous seront amenés à utiliser pour coder des signaux seront des multiples de la base binaire (Binaire, Octale et Hexadécimale).
- L'objet de ce chapitre portera plus particulièrement sur les problèmes de changement de base (10) en base (2) ou (16) ou l'inverse, ainsi que le passage de base (10) en base (16).
    - Codage en base 2 ® 0 ; 1.
    - Codage en base 8 ® 0 ; ...; 7.
    - Codage en base 10 ® 0;...; 9.
    - Codage en base 16 ® 0; ...; F.

Codage des informations : Le codage binaire

Le bit

Bit (noté b avec une minuscule dans les notations) signifie "binary digit", c'est-à-dire 0 ou 1 en numérotation binaire. C'est la plus petite unité d'information manipulable par une machine numérique. Il est possible de représenter physiquement cette information binaire :

- par un signal électrique ou magnétique, qui, lorsqu'elle atteint une certaine valeur, correspond à la valeur 1,
- par des aspérités géométriques dans une surface,
- grâce à des bistables, c'est-à-dire des composants électroniques qui ont deux états d'équilibre.

Avec un bit il est ainsi possible d'obtenir deux états : soit 1, soit 0.

Deux bits rendent possible l'obtention de quatre états différents (2 × 2).

Trois bits permet d'obtenir huit états différents (2 × 2 × 2).

Par extension, n bits permet d'obtenir 2n états différents.

Codage des informations : Le codage binaire

L'octet (byte)

L'octet (en anglais byte, noté B avec une majuscule dans les notations) est une unité d'information composée de 8 bits. Il permet de stocker un caractère, telle qu'une lettre, un chiffre ...

Ce regroupement de nombres par série de 8 permet une lisibilité plus grande, au même titre que l'on apprécie, en base décimale, de regrouper les nombres par trois pour pouvoir distinguer les milliers.

Par exemple le nombre 1 256 245 est plus lisible que 1256245.

Une unité d'information composée de 16 bits est généralement appelée mot (word)

Une unité d'information de 32 bits de longueur est appelée double mot (double word ou dword).

Pour un octet, le plus petit nombre est 0 (représenté par huit zéros 00000000), le plus grand est 255 (représenté par huit chiffre "un" 11111111), ce qui représente 256 possibilités de valeurs différentes

Unités binaires, … dans le langage courant

KiloOctets, MégaOctets, …

L'informatique se singularise par l'utilisation d'une base différente de celle de 10 pour les unités du système international. Dans le **langage courant** les unités suivantes sont utilisées:

- Un kilooctet (ko) = 1024 octets
- Un Mégaoctet (Mo) = 1024 Ko = 1 048 576 octets
- Un Gigaoctet (Go) = 1024 Mo = 1 073 741 824 octets
- Un Téraoctet (To) = 1024 Go = 1 099 511 627 776 octets

Il est également utile de noter que la communauté internationale dans son ensemble utilise préférentiellement le nom de "byte" plutôt que le terme "octet", ce dernier étant francophone.
Les notations précédentes sont donc remplacées par kilobyte (KB), mégabyte (MB), gigabyte (GB) et terabyte (TB) avec un B majuscule pour différencier Byte de bit.

Unités binaire, … dans la norme officielle

KiOctets, MeOctets, …

Toutefois, la **norme IEE 1541** a redéfini les multiplicateurs afin d'éviter la confusion avec la base 10. Ainsi les unités suivantes sont formellement définies:

- Un kibi (symbole Ki) = 1024 bytes (octets)
- Un mebi (symbole Mi) = 1024 Ki = 1 048 576 bytes
- Un gibi (symbole Gi) = 1024 Mi = 1 073 741 824 bytes
- Un tebi (symbole Ti) = 1024 Gi = 1 099 511 627 776 bytes

Codage des informations : Codage des nombres entiers

Représentation d'un nombre entier non signé (toujours **positif**)

Un entier non signé est un entier positif ou nul.
Le nombre de bits à utiliser dépend de la plage des nombres que l'on désire représenter.
Pour représenter un nombre A10 entier non signé en base 10 il suffit de trouver les coefficients du polynôme selon la relation

$$A_{10} = \sum_{i=0}^{n-1} a_i 2^i = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2^1 + a_0 2^0$$

$$a_i \in \{0,1\}$$

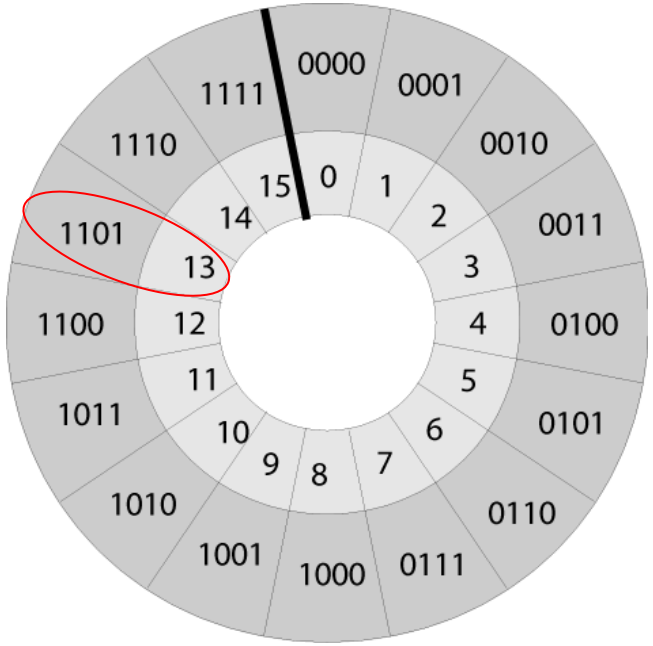
$$A_{10} \in [0 \ldots 2^n - 1]$$

Codage des informations : Codage des nombres entiers

Représentation d'un nombre entier non signé (**positif**)

Par exemple, pour un nombre donné en base 2 comprenant 4 bits

$$A_{10} = (1101)_2 = \underbrace{1 \cdot 2^3}_{8} + \underbrace{1 \cdot 2^2}_{4} + 0 \cdot 2^1 + \underbrace{1 \cdot 2^0}_{1} = 13_{10}$$



Codage des informations : Codage des nombres entiers

Représentation de nombre **en compléments à deux** d'un nombre entier signé

La représentation de nombres en compléments à deux permet **d'effectuer des additions et des soustractions** sans devoir tester s'il s'agit de nombres positifs ou négatifs.

Les nombres positifs sont représentés comme attendu; par contre les nombres négatifs sont obtenus de la manière suivante :

- On inverse les bits de l'écriture binaire de sa valeur absolue (opération binaire NOT), on fait ce qu'on appelle le complément à un.
- On ajoute 1 au résultat (les dépassements sont ignorés).

La représentation en complément à deux permet de simplifier la structure des unités arithmétiques se trouvant à l'intérieur des microprocesseurs

Codage des informations : Codage des nombres entiers

Cette opération correspond au calcul de

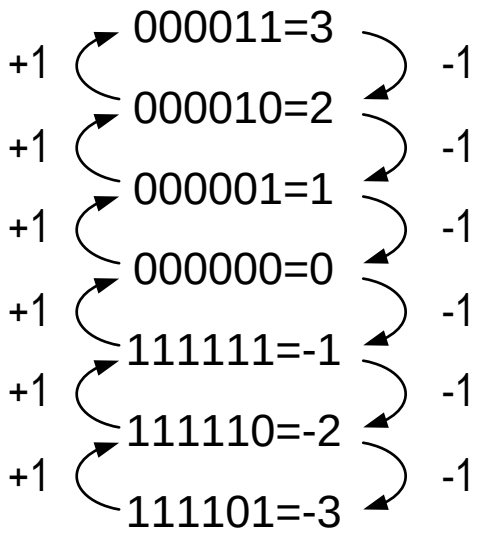
$$2^n - |X|$$

où *n* est la longueur de la représentation.

Ainsi -1 s'écrit comme 256-1=255=11111111_2, pour les nombres sur 8 bits.

La représentation en complément à deux permet de simplifier la structure des unités arithmétiques se trouvant à l'intérieur des microprocesseurs

Représentation des nombres négatifs



Codage des informations : Codage des nombres entiers

Définition:

On dit que les composants électroniques qui effectuent des **opérations sur n bits** travaillent en **arithmétique modulo 2^n**.

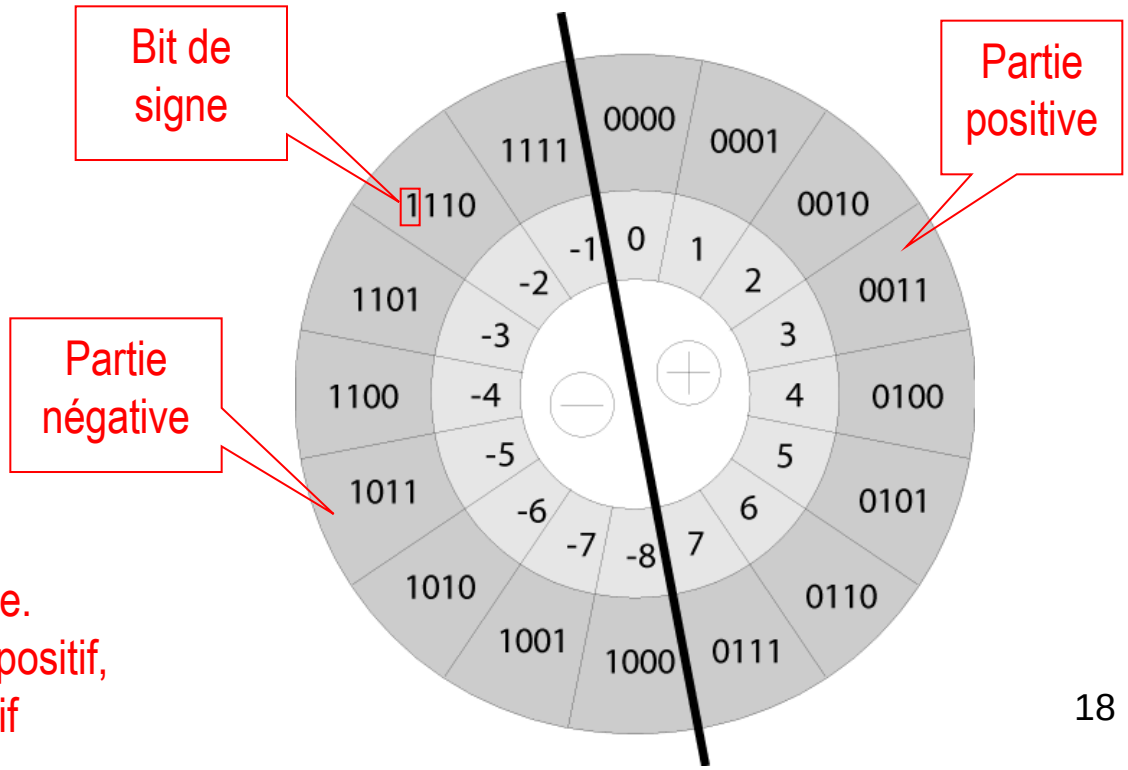
$$-A_2 = 2^n - A_2 \pmod{2^n}$$

Cette relation signifie que pour chacun des $2^{n-1}-1$ nombres positifs il est possible de trouve un nombre négatif correspondant

Codage des informations : Codage des nombres entiers

Représentation de nombre en compléments à deux d'un nombre entier

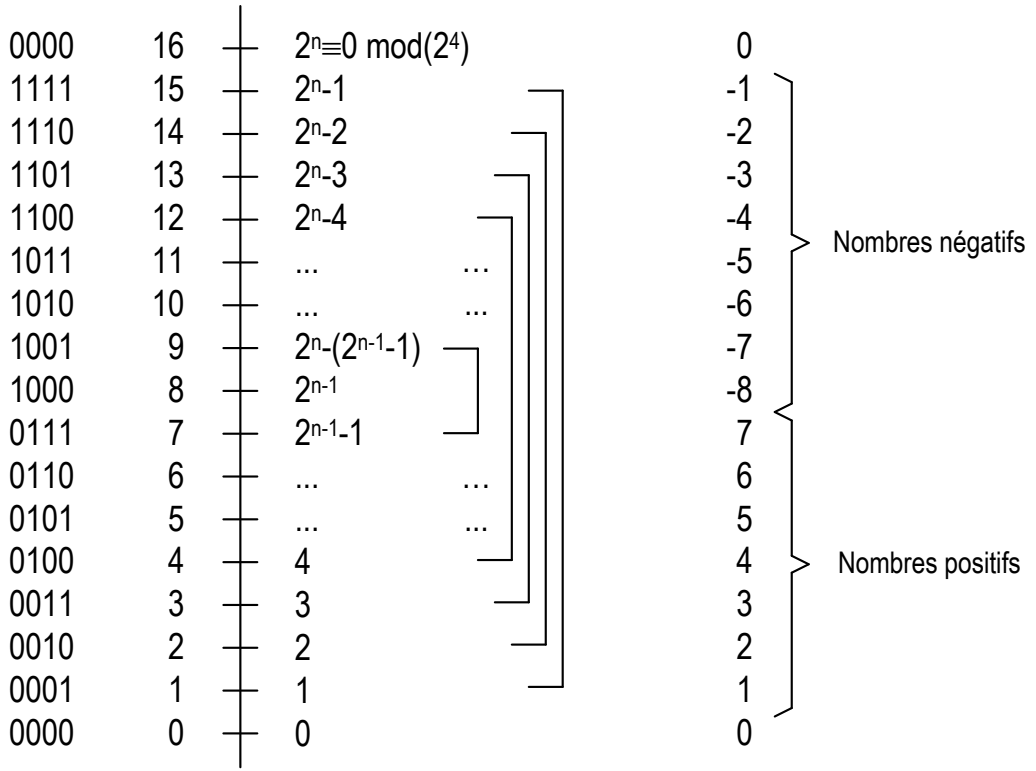
Cette représentation des nombres peut être également visualisée sur un cercle.
On remarque que sur n bits seuls 2^n bits peuvent être codés. Les nombres situés dans l'intervalle [0 .. 2^{n-1}-1] sont positifs et ceux situés dans l'intervalle [2^{n-1} … 2^n-1] sont négatifs



Le bit de poids fort est le bit de signe.
Si ce bit est égal à 0 le nombre est positif,
s'il est égal à 1 le nombre est négatif

Codage des informations : Codage des nombres entiers

Intervalle des nombres en compléments à deux sur 4 bits



Complement à 2	Décimal
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

Si on ne considère que des nombres positifs on peut (avec 4 bits) numéroter de 0 à 15 .
Avec des nombres entiers positifs et négatifs on numérote de -8 à 7 .
En d'autres termes, on « décale » de -8 .

Codage des informations : Codage des nombres entiers

Représentation de nombre en compléments à deux d'un nombre entier

La représentation en complément à deux se sert du bit le plus significatif comme signe.
Pour représenter un nombre signé en complément à deux en base 10, il suffit de modifié le premier coefficient du polynôme représentatif des nombres entier non signé

$$A_{10} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2^1 + a_0 2^0$$

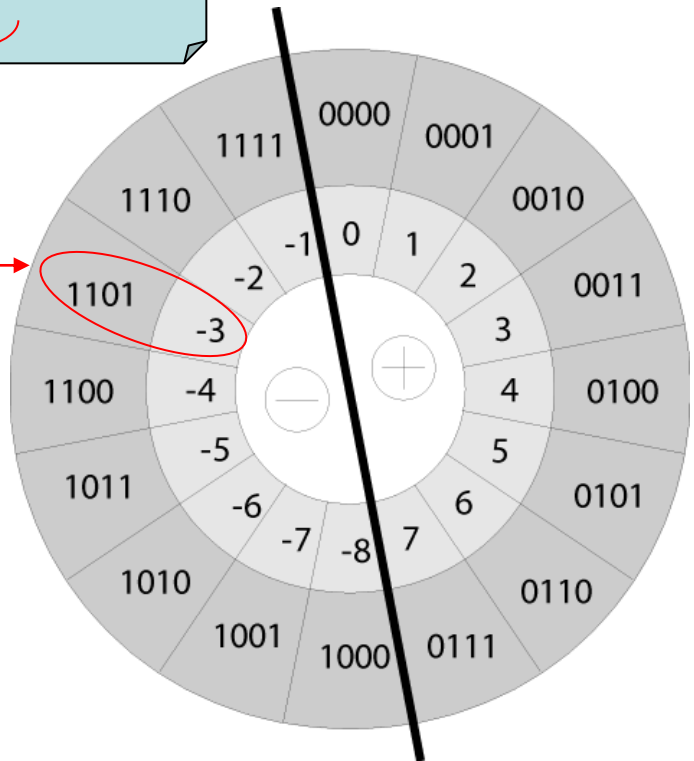
Par exemple …

Codage des informations : Codage des nombres entiers

Représentation de nombre en compléments à deux d'un nombre entier

Par exemple, pour un nombre donné de 4 bits

$$A_{10} = (1101)_2 = \underbrace{-1 \cdot 2^3}_{-8} + \underbrace{1 \cdot 2^2}_{4} + 0 \cdot 2^1 + \underbrace{1 \cdot 2^0}_{1} = -3_{10}$$



Codage des informations : Codage des nombres entiers

Changement de signe

En partant de la définition d'un nombre négatif, on peut trouver la manière de réaliser un changement de signe

$$-A_{10} = 2^n - A_{10} = 2^n - \left(\underbrace{-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i}_{A_{10}} \right) = \underbrace{-2^{n-1} + \underbrace{\sum_{i=0}^{n-2} 2^i}_{2^{n-1}} + 1}_{2^n} + a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

$$= -(1-a_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1-a_i)2^i + 1 = -\overline{a}_{n-1}2^{n-1} + \sum_{i=0}^{n-2}\overline{a}_i 2^i + 1$$

Le **changement de signe d'un nombre en complément à 2** peut être réalisé en prenant le **complément à 1 de chaque bit et en ajoutant 1**

Codage des informations : Codage des nombres entiers

Extension du signe

Une règle importante, pour les nombres représentés en complément à 2, est l'extension du signe.
Soit un nombre A_{10} exprimé en binaire. Si A_{10} est un nombre positif, il est clair que les bits supplémentaires de poids fort (à gauche) sont des « 0 ».
Si le nombre est négatif on peut se demander quelles sont les valeurs des bits supplémentaires.
(ce sont des 1 …)

Soit un nombre binaire négatif pouvant être représenté par n bits au minimum, et soit m > n.

$$A_{10} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i$$

Pour un nombre négatif, on a $a_{n-1}=a_{m-1}=1$

Codage des informations : Codage des nombres entiers

Extension du signe

Démonstration

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

$$\underbrace{2^{m-1}}_{1+\sum_{i=0}^{m-2}2^i} = \underbrace{2^{n-1}}_{1+\sum_{i=0}^{n-2}2^i} + \sum_{i=n-1}^{m-2}2^i$$

$$\Rightarrow -2^{n-1}+\sum_{i=0}^{n-2}a_i 2^i = -2^{n-1}-\underbrace{\sum_{i=n-1}^{m-2}2^i+\sum_{i=0}^{m-2}a_i 2^i}_{\sum_{i=0}^{n-2}a_i 2^i}$$

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

$$\Rightarrow \sum_{i=n-1}^{m-2}2^i = \sum_{i=n-1}^{m-2}a_i 2^i \Rightarrow a_{m-2}=a_{m-3}=\ldots=a_n=a_{n-1}=1$$

Codage des informations : Codage des nombres entiers

Extension du signe

On voit donc que pour un nombre négatif, les bits de poids forts supplémentaires doivent être mis à « 1 ». On parle d'extension du signe.

Cette caractéristique peut être illustrée par un exemple.
Soit le nombre -27 exprimé à l'aide de 5 et 7 bits

$$-27 \begin{cases} \underbrace{-1\cdot 2^5}_{-32} + \underbrace{0\cdot 2^4}_{0} + \underbrace{0\cdot 2^3}_{0} + \underbrace{1\cdot 2^2}_{4} + \underbrace{0\cdot 2^1}_{0} + \underbrace{1\cdot 2^0}_{1} \\ \underbrace{\underbrace{-1\cdot 2^7}_{-128} + \underbrace{1\cdot 2^6}_{64} + \underbrace{1\cdot 2^5}_{32}}_{-32} + \underbrace{0\cdot 2^4}_{0} + \underbrace{0\cdot 2^3}_{0} + \underbrace{1\cdot 2^2}_{4} + \underbrace{0\cdot 2^1}_{0} + \underbrace{1\cdot 2^0}_{1} \end{cases}$$

Codage des informations : Codage des nombres entiers

Opérations arithmétiques en complément à deux

Les opérations d'addition et de soustraction en complément à deux fonctionnent parfaitement bien en arithmétique modulo 2^n.
Seule la somme de deux nombres positifs ou de deux nombre négatifs peut conduire à un dépassement de capacité

Exemple : soustraction sur 4 bits (mod 16 ou 2^4) en complément à deux

Pour A>B

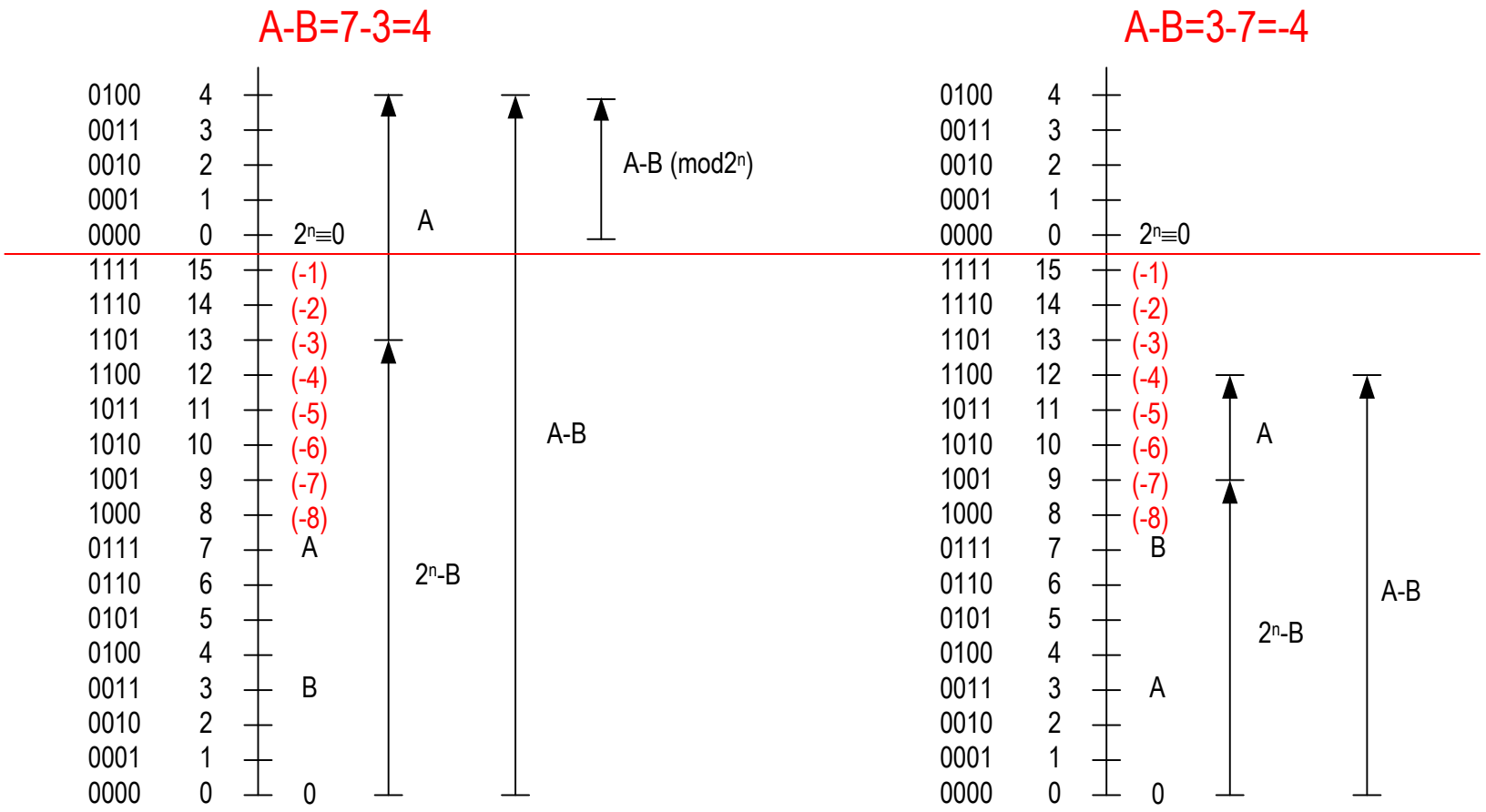
$$\left.\begin{array}{l} A = 7 \\ B = 3 \end{array}\right\} A - B = A + 2^4 - B = 20 \,(\mathrm{mod}\, 16) = 4$$

Pour A<B

$$\left.\begin{array}{l} A = 3 \\ B = 7 \end{array}\right\} \begin{array}{l} A - B = A + 2^4 - B = 12 \,(\mathrm{mod}\, 16) \\ \qquad = -4 \,(\textit{complément à deux sur 4 bits}) \end{array}$$

Codage des informations : Codage des nombres entiers

Opérations arithmétiques en complément à deux



Système hexadécimal

- Le système hexadécimal nécessite l'introduction de 16 chiffres, représentant les 16 premiers entiers positifs :
  0; 1; 2; 3; 4; 5; 6; 7; 8; 9; A; B; C; D; E; F.

- Un entier est écrit comme la concaténation de ces chiffres, et sa lecture s'effectue de droite à gauche. Sa valeur vaut la somme des chiffres affectés de poids correspondant aux puissances successives du nombre 16.

- Par exemple, 4D5 vaut 5 + 13*16 + 4*16*16.

- Ainsi 15AACF7 se convertit en décimal en calculant:
  $1\times166 + 5\times165 + 10\times164 + 10\times16^3 + 12\times16^2 + 15\times161 + 7\times160 = 22719735$.

- Souvent on indique les nombres hexadécimaux par la forme:
  0xHHHH ou HHHHh ex. 0xA2EB ou A2EBh (ici un nombre de16 bits)

Conversion de binaire en hexadécimal

- Ce format est largement utilisé en informatique car il permet une conversion facile avec le système binaire.
- En outre, la notation hexadécimale est plus compacte que la notation binaire, et utilise jusqu'à quatre fois moins de chiffres que ce dernier pour représenter le même nombre.
- La conversion de binaire en hexadécimal se fait en regroupant les chiffres (les bits) quatre par quatre, ou inversement en remplaçant chaque chiffre hexadécimal par 4 chiffres binaires.
- Exemple:

binaire	1.0101.1010.1010.1100.1111.0111						
regroupé par 4	1	0101	1010	1010	1100	1111	0111
regroupé en hexadécimal	1	5	A	A	C	F	7
hexadécimal	15A.ACF7						

Conversion de hexadécimal en binaire

- Il suffit de convertir la valeur de chacun des chiffres sous leur forme binaire.
- 0x1A2F va s'écrire
  - 1
  - $10_{=8+2}$
  - 2
  - $15_{=8+4+2+1}$
- Soit: $1\ 1010\ 0010\ 1111_2$

Table des valeurs des groupements de chiffres binaires

Binaire	Décimal	Octal	Hexadécimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7

Binaire	Décimal	Octal	Hexadécimal
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

Exercices

- Effectuer les exercices au chapitres 1 et 2 de la note d'exercices

Codage des informations : Codage des nombres réels en **virgule fixe**

Représentation d'un nombre réel en virgule fixe

La représentation d'un nombre en virgule fixe est un type de donnée correspondant à un nombre qui possède (en base deux ou en base dix) un nombre fixe de chiffres après la virgule.

Les nombres en virgule fixe sont utiles pour représenter des quantités fractionnaires dans un format utilisant le complément à deux quand le processeur de l'ordinateur n'a aucune unité de calcul en virgule flottante ou quand une virgule fixe permet d'augmenter la vitesse d'exécution ou d'améliorer l'exactitude des calculs.

Les petits processeurs et microcontrôleurs ne possèdent pas d'unité de calcul permettant de travailler en virgule flottante.

Si les algorithmes à exécuter requièrent des calculs en nombres réels, il est possible dans certaines circonstances de représenter les nombres réels par des nombres en virgule fixe.

Codage des informations : Codage des nombres réels en **virgule fixe**

Représentation d'un nombre réel en virgule fixe

Dans ce cas, il faut s'assurer que le résultat de chacune des opérations désirées soit programmée proprement et présente la précision voulue.

Les paramètres suivants dépendent du type de problème à traiter :

- ⇨ nombre de bits requis,
- ⇨ choix de l'emplacement de la virgule dans le champ de bits représentant le nombre,
- ⇨ vitesse de calcul.

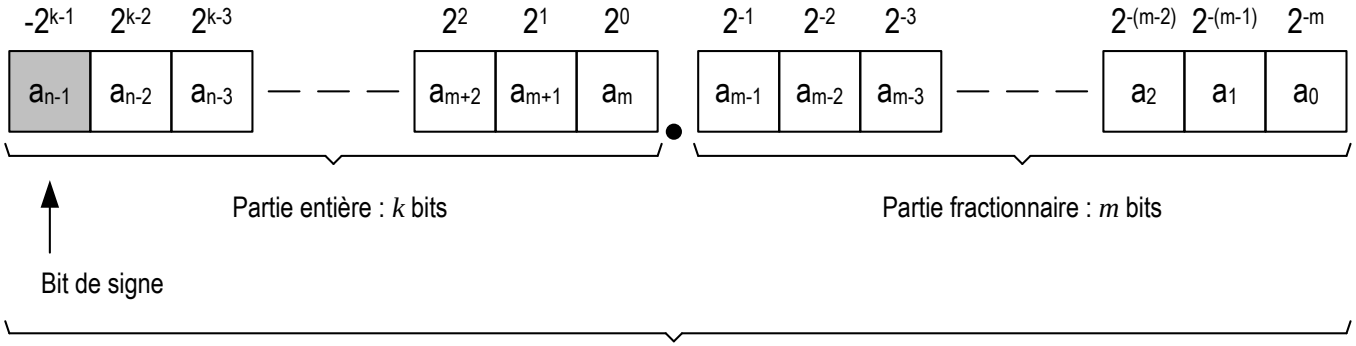
Codage des informations : Codage des nombres réels en virgule fixe

Représentation d'un nombre réel en virgule fixe

Un nombre réel en virgule fixe est donné par la relation suivante

$$A_{10} = -a_{n-1}2^{n-m-1} + \sum_{i=m}^{n-m-2} a_i 2^{m-i} + \sum_{k=0}^{m-1} a_i 2^{-(m-k)}$$

ou sous une forme un peu plus explicite



Nombre fractionnaire en virgule fixe de n (avec $n=k+m$) bits au format $k.m$

En pratique: pour convertir à la main les nombres négatifs, il peut être plus aisé de décomposer le nombre en une partie intière signée et une partie fractionnaire toujours positive:

Exemple: $-2.4 = -3 + 0.6$

Codage des informations : Codage des nombre réel en virgule flottante

La représentation en virgule flottante

La représentation en virgule flottante permet de résoudre de nombreux problèmes de représentation.
Un nombre en virgule fixe possède une fenêtre de représentation fixe :

⇨ selon la position de la virgule, on peut représenter des nombres très petits ou très grands.

La représentation en virgule flottante met en œuvre une sorte de fenêtre glissante de précision en relation avec la valeur à représenter.
C'est donc un système de représentation des nombres dans lequel l'intervalle des nombres exprimables est indépendant du nombre de chiffres significatifs (la précision).
Cette représentation est basée sur la représentation des nombres en notation scientifique

$$Signe \cdot Fraction_2 \cdot 2^{\text{exposant}-\text{excès}}$$

Codage des informations : Codage des nombre réel en **virgule flottante**

La représentation en virgule flottante

Les nombres à virgule flottante sont les nombres les plus souvent utilisés dans un ordinateur pour représenter des valeurs non entières. Ce sont des approximations de nombres réels.

⇨ Les nombres à virgule flottante possèdent
- un signe *s* (dans {-1, 1}),
- une mantisse *m* (aussi appelée significande)
- et un exposant *e*.

Un tel triplet représente un nombre réel

$$s.m.b^{e}$$

où *b* est la base de représentation (ici 2)

En faisant varier *e*, on fait « flotter » la virgule décimale.
Généralement, *m* est d'une taille fixée.

Ceci s'oppose à la représentation en **virgule fixe**, où l'exposant *e* est fixé.

Codage des informations : Codage des nombre réel en virgule flottante

La représentation en virgule flottante

$$Signe \cdot Fraction_2 \cdot 2^{\text{exposant}-\text{excès}}$$

- Le **signe** indique si le **nombre** est **positif** ou **négatif**.
- La **valeur absolue** du nombre est donnée par **$Fraction_2 \cdot 2^{exposant-excès}$**.
- Selon la précision exigée, on utilise
  - une représentation **16 bits (précision réduite)**,
  - sur **32 bits (simple précision)**
  - ou sur **64 bits (double précision)**.
- Excès vaut -127 en format 32 bits et -1023 en format 64 bits

Codage des informations : Codage des nombre réel en virgule flottante

- En général on travaille en simple ou en double précision.

- La **partie fractionnelle** des nombres en représentation en virgule flottante est **normalisée**, c'est-à-dire que pour un format de nombre donné, elle aura toujours la même forme:
  (**1.xxx** ou **0.xxx** selon le format).

- En représentation virgule flottante sur 32 bits, selon le format **IEEE 754**, la mantisse prend toujours la forme **1.xxx**.

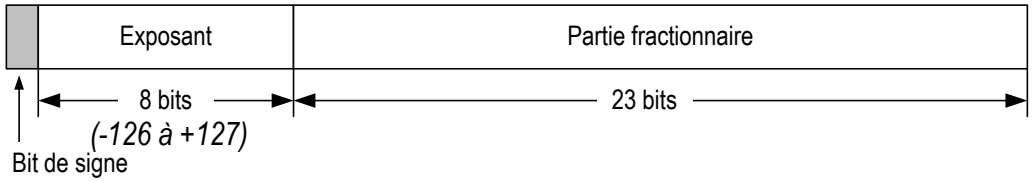
- Comme le chiffre de poids fort est toujours égal à 1, **ce bit n'apparaît pas**
- (redondance de l'information), il s'agit donc d'un **bit implicite**.

Codage des informations : Codage des nombre réel en virgule flottante

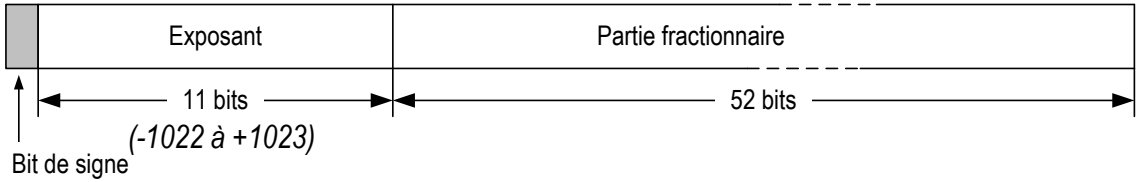
Le format virgule flottante IEEE (ANSI-IEEE 754)

Le standard IEEE 754 est le standard le plus utilisé aujourd'hui pour représenter les nombres en virgule flottante.
Le standard IEEE définit trois formats de représentation des nombres flottants :

⇨ la simple précision (sur 32 bits),



⇨ la double précision (sur 64 bits),



⇨ la précision étendue (sur 80 bits).

Ce dernier format est surtout destiné à résoudre les erreurs d'arrondis

Codage des informations : Codage des nombre réel en virgule flottante

Le format virgule flottante IEEE (ANSI-IEEE 754)

	Encodage	**S**igne	**E**xposant	**M**antisse	Valeur d'un nombre
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$

avec $1 \leq M < 2$

Codage des informations : Codage des nombre réel en virgule flottante

Le format virgule flottante IEEE (ANSI-IEEE 754)

Valeurs réservées

Le standard IEEE 754 prévoit quelques valeurs réservées qui permettent d'exprimer le zéro, les infinis et les nombres non réels.

- Le **zéro** est indiqué par un **champ d'exposant à 0** et une **partie fractionnaire à 0**.

- **S**i l'exposant est égal à $\mathbf{2^e - 1}$, et si la mantisse est **nulle,**
  le nombre est **±infini** (selon le bit de signe),
  e représente le nombre de bits de l'exposant.

- Lorsque la **valeur représentée** n'est **pas réelle**, on dit **NaN** (Not a Number), le **champ d'exposant est rempli de 1** et la **partie fractionnaire** est **non nulle**.

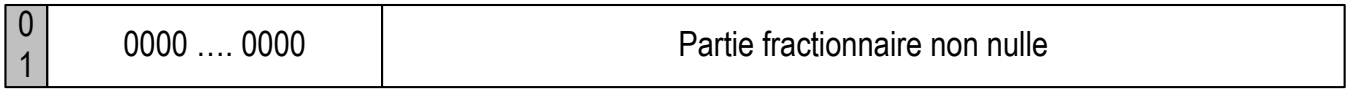
Codage des informations : Codage des nombre réel en virgule flottante

Nombres dénormalisés

Les **nombres dénormalisés** ont leur **exposant qui vaut zéro** mais leur **partie fractionnaire** est **différente de zéro**.

Sur ces nombres, la **mantisse n'est plus normalisée**, c'est à dire qu'il n'y a **plus de bit implicite à 1** devant la virgule, **mais plutôt un zéro**.

Cette **dénormalisation** du format est nécessaire pour exprimer des **nombres encore plus petits**.



$$Signe \cdot 0.Fraction_2 \cdot 2^{-\text{excès}}$$

excès vaut dans ce cas **-126** en format 32 bits et **-1022** en format 64 bits

Codage des nombre réel en virgule flottante: **résumé**

Type	Exposant	Mantisse
Zéros	0	0
Nombres dénormalisés	0	différente de 0
Nombres normalisés	1 à $2^e - 2$	quelconque
Infinis	$2^e - 1$	0
NaNs	$2^e - 1$	différente de 0

e représente le nombre de bits de l'exposant (8 ou 11 bits).

Exercices

- Effectuer les exercices de la série 1.2.1 et 1.2.2

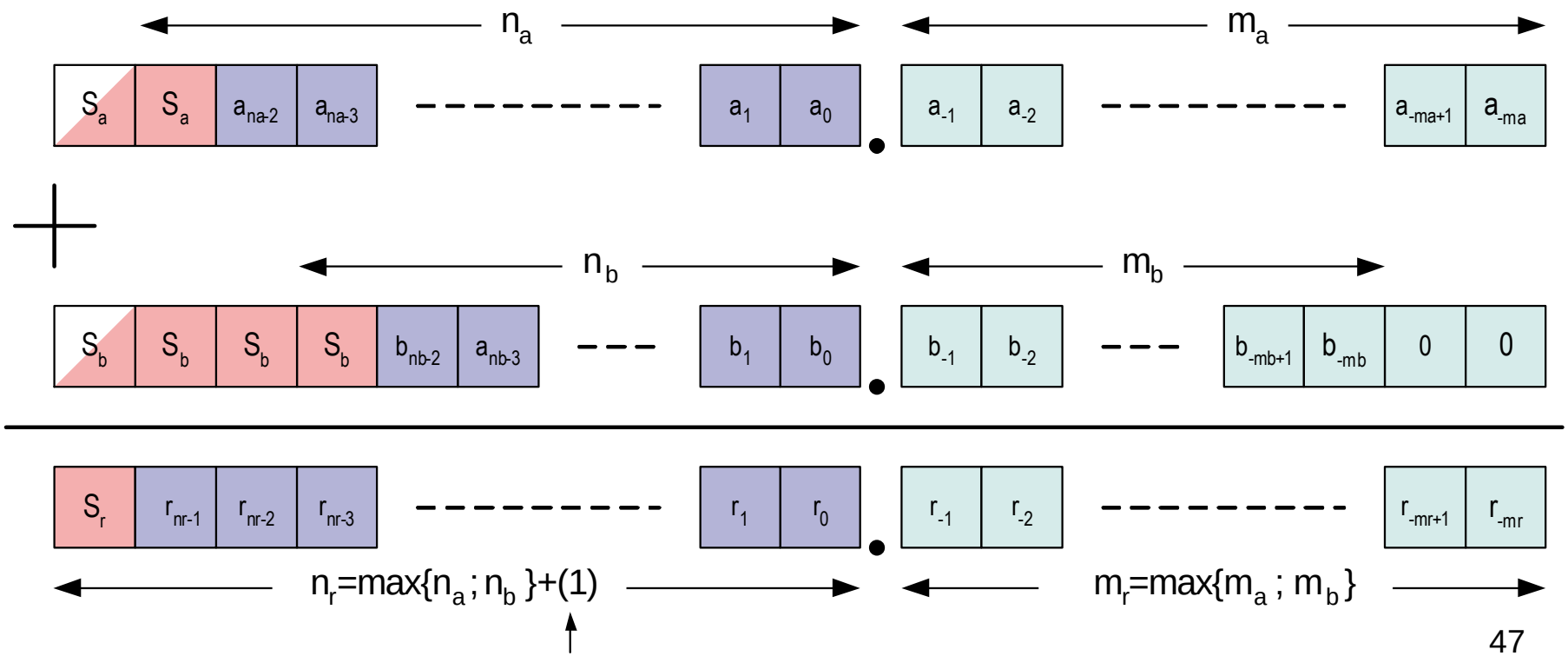
Opérations arithmétiques binaires

Opération arithmétique : addition entre deux nombres fractionnaires

Choix d'un format commun $n_c=\max(n_a,n_b)$, $m_c=\max(m_a,m_b)$

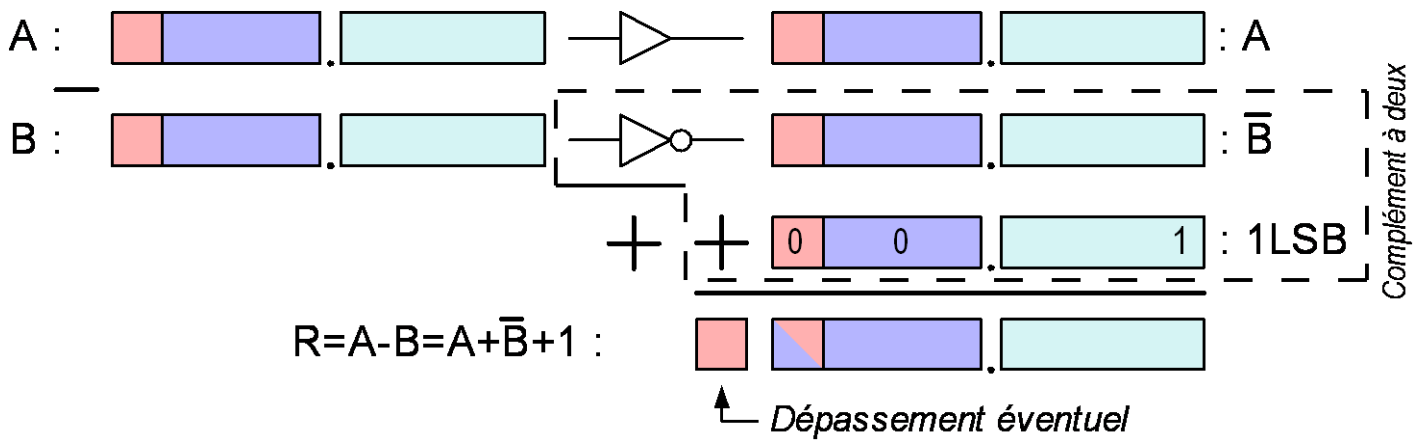
Alignement de la virgule

Extension des bits des opérandes a et b selon les valeurs de (n_c, m_c)



Opération arithmétique : soustraction entre deux nombres fractionnaires

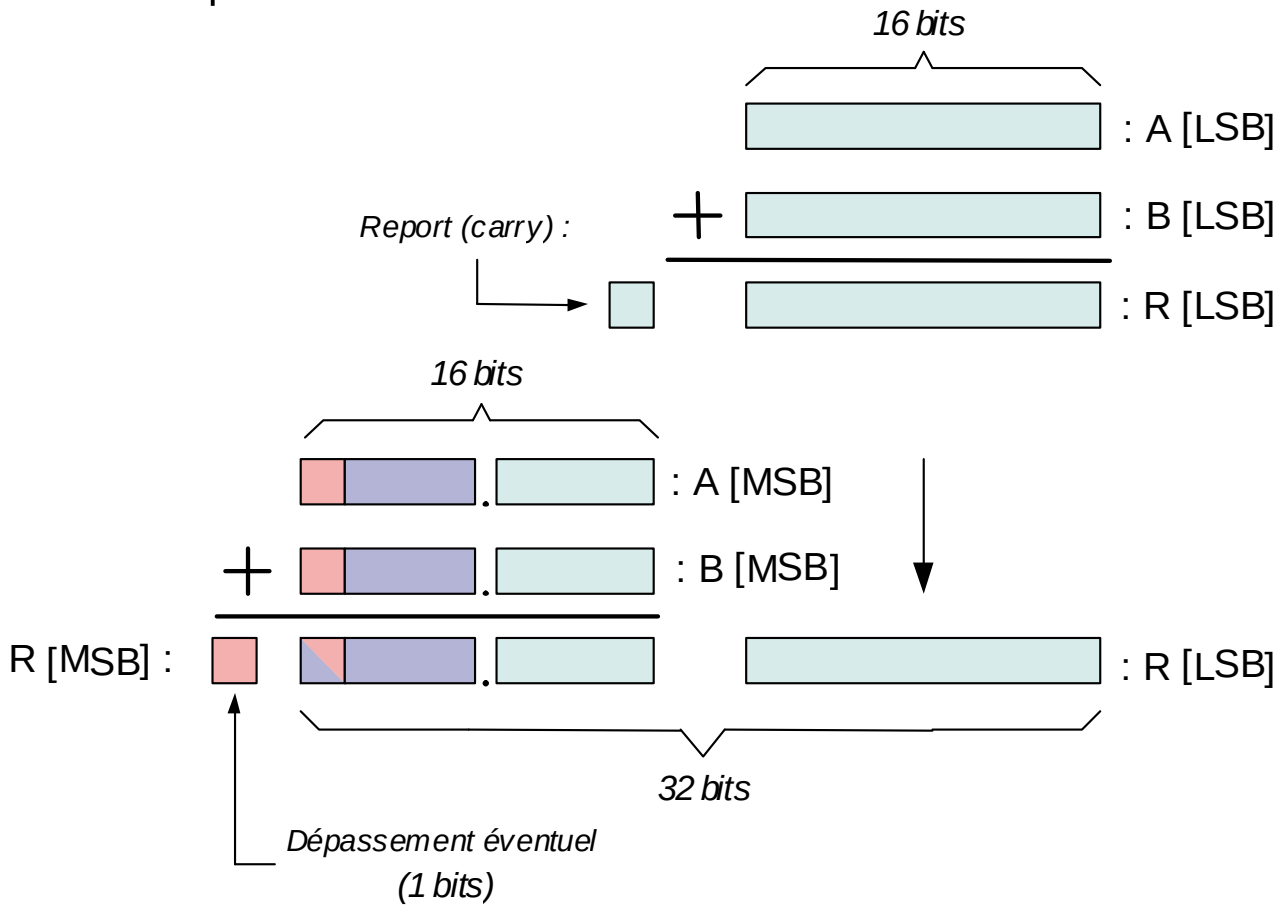
Exemples de calcul



Une soustraction se résume à une addition avec le complément à deux du soustracteur

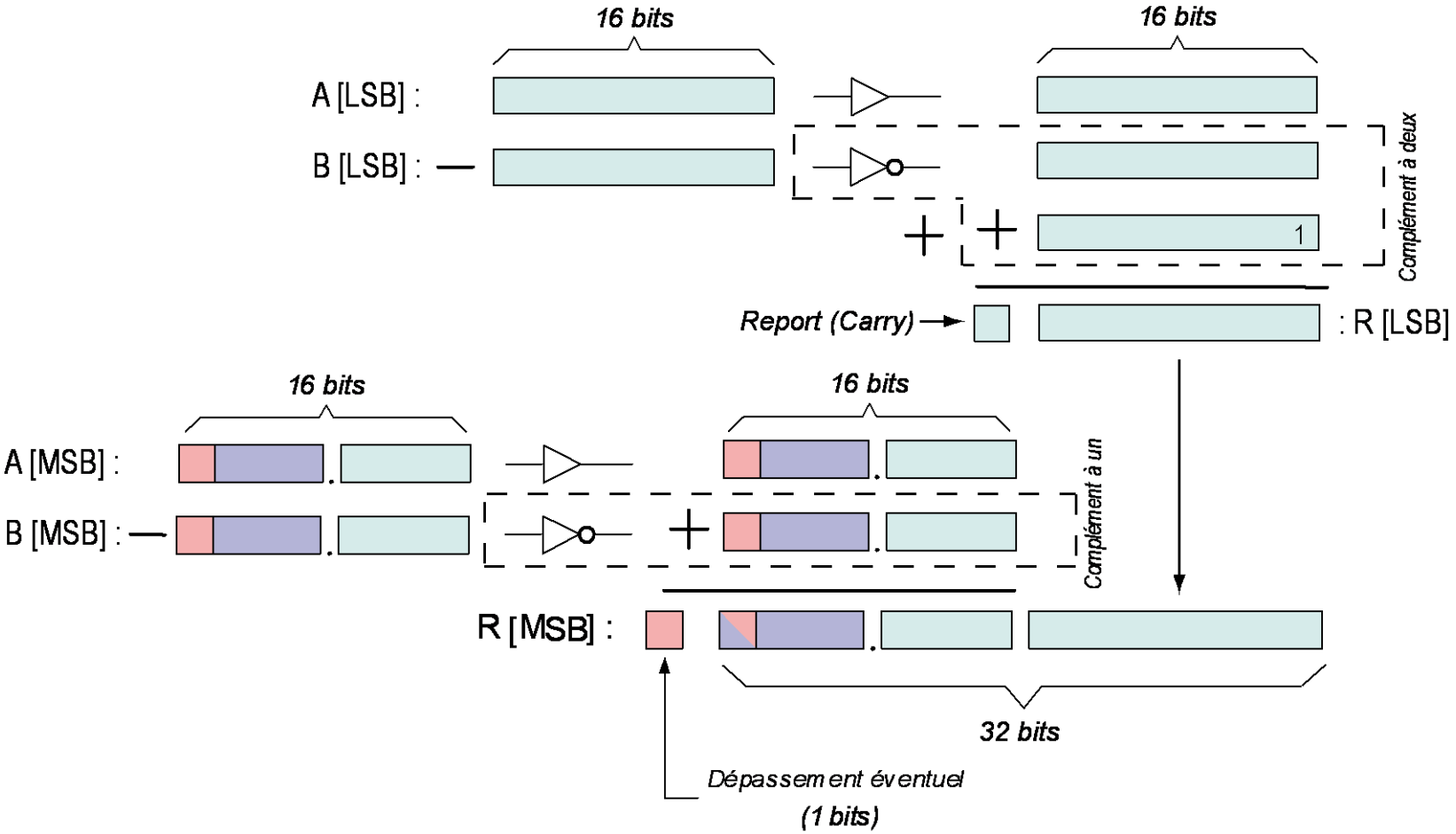
Addition double précision entre deux nombres entiers et fractionnaires

Addition 32 bits par mot de 16 bits



Soustraction double précision entre deux nombres entiers et fractionnaires

Soustraction 32 bits par mot de 16 bits



Multiplication entre deux nombres entiers non signés

Multiplicande $B^{[U]}$ non signé, multiplicateur $A^{[U]}$ non signé

Exemple: 8 bits × 8 bits = résultat sur 16 bits

		2	2	0
	x	1	0	5
	1	1	0	0
0	0	0	0	
+ 2	2	0		
2	3	1	0	0
		↓		
2	**3**	**1**	**0**	**0**

								Multiplicande B :		1	1	0	1	1	1	0	0
								Multiplicateur A :	x	0	1	1	0	1	0	0	1
							0		0	1	1	0	1	1	1	0	0
						0	0		0	0	0	0	0	0	0	0	
					0	0	0		0	0	0	0	0	0	0		
				0	0	0	0		0	1	1	0	1	1	1	0	0
			0	0	0	0	0		0	0	0	0	0				
		0	0	0	1	1	0		1	1	0	0					
+	0	0	1	1	0	1	1		1	0	0						
←	0	1	0	1	1	0	1		0	0	0	1	1	1	1	0	0

Si on a de la difficulté avec les retenues, on peut effectuer ces sommes deux par deux.

Multiplication entre deux nombres entiers

Multiplicande B[S] signé, multiplicateur A[U] non signé

Exemple: 8 bits × 8 bits = résultat sur 16 bits
Il faut **étendre le signe** sur tous les bits (colonnes) concernés.

```
      − 3 6
    x   1 0 5
    ---------
      − 1 8 0   [1]
  −     0 0 0
−   0   3 6
  ---------
    − 3 7 8 0   [1]

                [1]
                [1]
        ↓

  ---------
    − 3 7 8 0
  =========
```

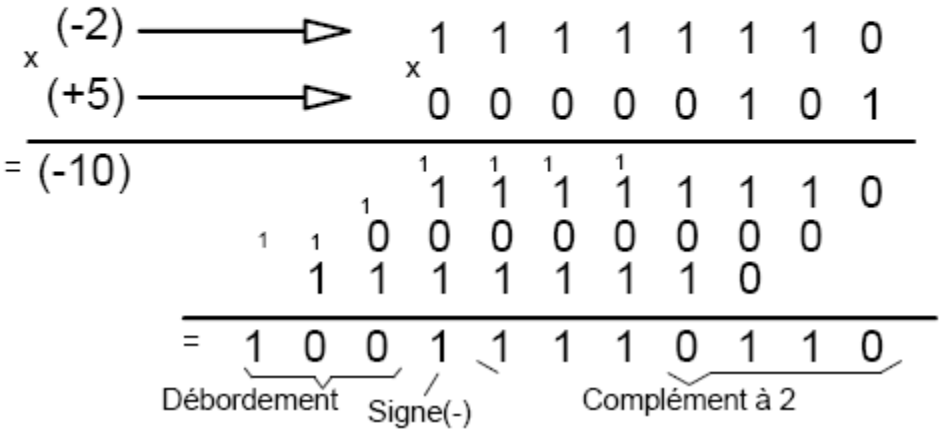
Multiplicande : B	**[1]**	1	1	1	1	1	1	1	1	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**
Multiplicateur : A	x									**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**
		1	***1***	***1***	***1***	***1***	***1***	***1***	***1***	1	1	0	1	1	1	0	0
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
		0	0	0	0	0	0	0	0	0	0	0	0	0	0		
		1	***1***	***1***	***1***	***1***	1	1	0	1	1	1	0	0			
		0	0	0	0	0	0	0	0	0	0	0	0				
		1	***1***	***1***	1	1	0	1	1	1	0	0					
		1	***1***	1	1	0	1	1	1	0	0						
		0	0	0	0	0	0	0	0	0							
	←	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**

[1] *Extension du signe pour chaque multiplication intermédiaire*

Multiplication entre deux nombres entiers signés

Autre exemple: 4 bits × 4 bits = résultat sur 8 bits

(-2) x (+5) : le résultat étant négatif , le (-2) sera représenté en complément à 2 :
$(+5)_{10} \rightarrow$ (0 0 0 0 0 1 0 1)$_2$
$(+2)_{10} \rightarrow$ (0 0 0 0 0 0 1 0) $_2$ son complément à 2 : (1 1 1 1 1 1 1 0) .



Ici , l'opération à été volontairement poursuivie dans la partie débordement.
Dans la pratique, cette partie de l'opération ne peut s'effectuer (à cause de la longueur des registres) .
Le bit le plus significatif du résultat à 8 bits est 1 (bit de signe), il s'agit onc d'un nombre négatif, représenté en complément à 2 . Soit (1 1 1 1 0 1 1 0) qui est le complément à 2 de (0 0 0 0 1 0 1 0) = 10, le signe étant négatif , nous trouvons pour résultat : (-10)

Multiplication entre deux nombres entiers signés

Multiplicande $B^{[S]}$ signé, multiplicateur $A^{[S]}$ signé

Exemple: 4 bits × 4 bits = résultat sur 8 bits
Il faut étendre le signe sur tous les bits (colonnes) concernés,
donc en pratique effectuer la multiplication sur 8 bits

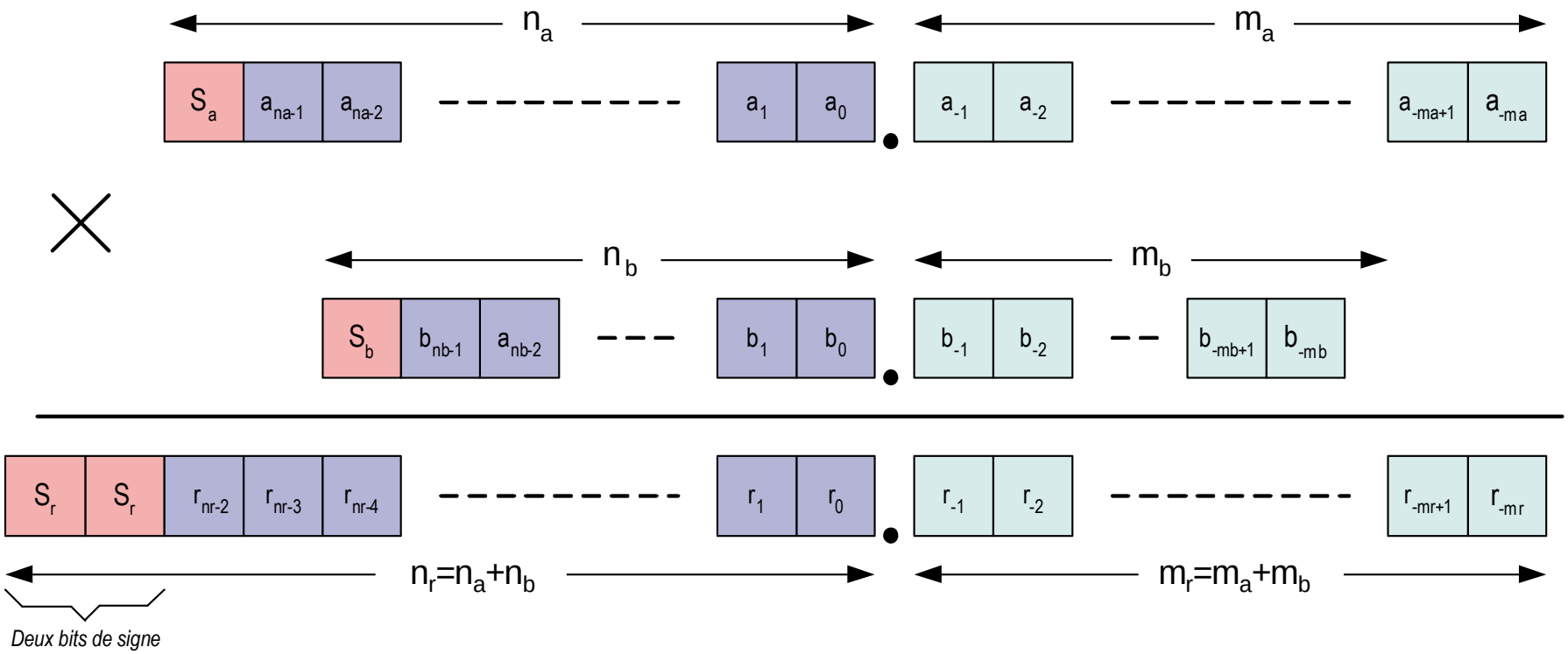
	— 3	***Multiplicande : B***	[1]	1	1	1	1	**1**	**1**	**0**	**1**
x	— 5	***Multiplicateur : A***	[1]	1	1	1	1	**1**	**0**	**1**	**1**
			[1]	1	1	1	1	1	1	0	1
			[1]	1	1	1	1	1	0	1	
				0	0	0	0	0	0		
			[1]	1	1	1	0	1			
				1	1	0	1				
				1	0	1					
				0	1						
				1							
	+ **1 5**	←		**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

[1] *Extension du signe pour chaque multiplication intermédiaire*

Exercices

- Effectuer à la main en binaire
  (4 bits x 4 bits): mais **opération** et résultat sur 8 bits
  - $(-1) \times 2$
  - $1 \times (-2)$
  - $(-1) \times (-2)$

Multiplication entre deux nombres fractionnaires



Multiplication entre deux nombres fractionnaires

Format du résultat

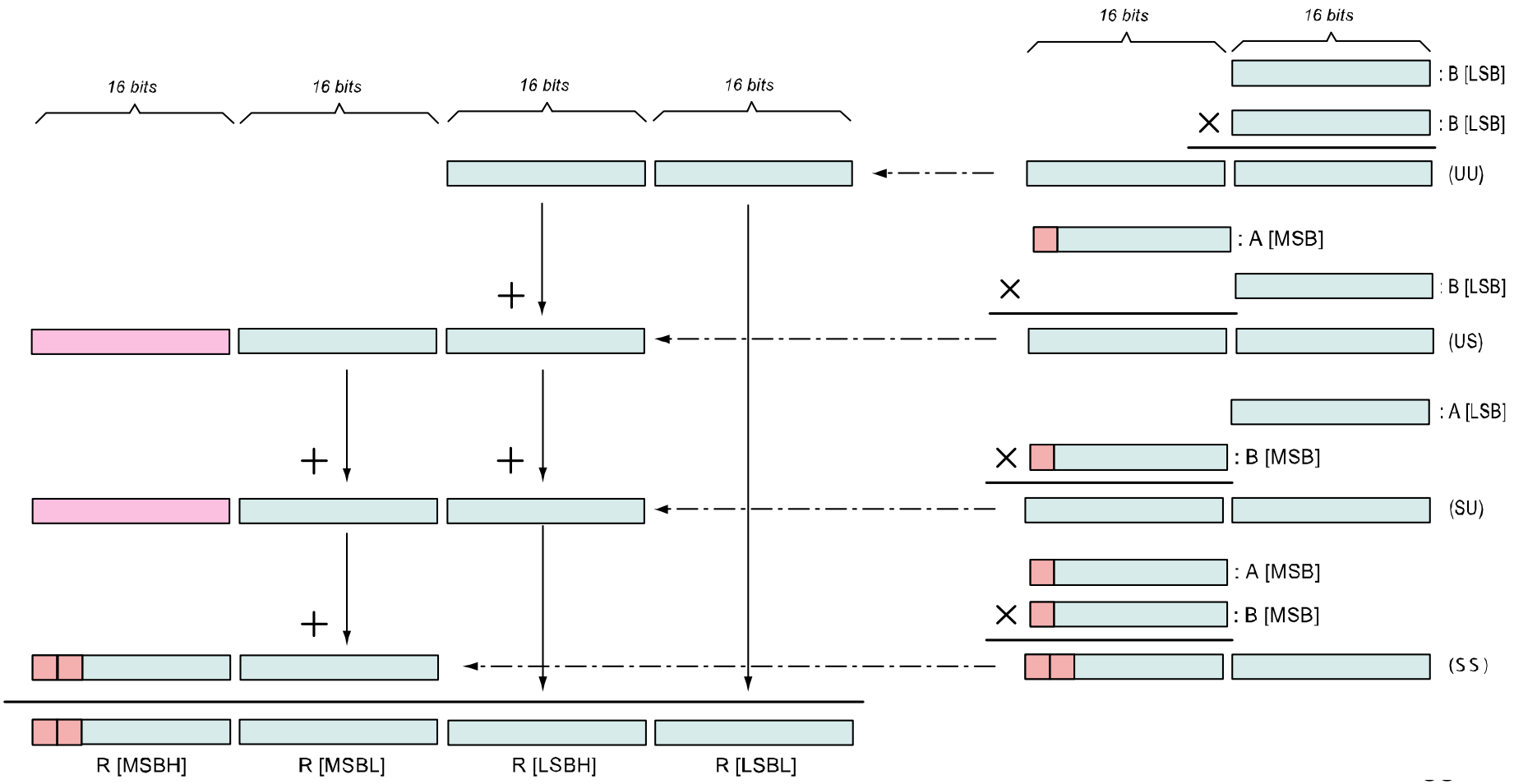
En effet, soit deux nombres :

x en format (n.m) compris dans la plage $[-2^{(n-1)} \ldots 2^{(n-1)}-2^{(-m)}]$
y en format (j.k) compris dans la plage $[-2^{(j-1)} \ldots 2^{(j-1)}-2^{(-k)}]$

Le résultat de la multiplication de x*y donne en principe le format (n+j).(m+k)

$z=x*y$ en format $[-2^{(n+j-1)} \ldots 2^{(n+j-1)}-2^{-(m+k)}]$

Multiplication entre deux nombres fractionnaires double précision

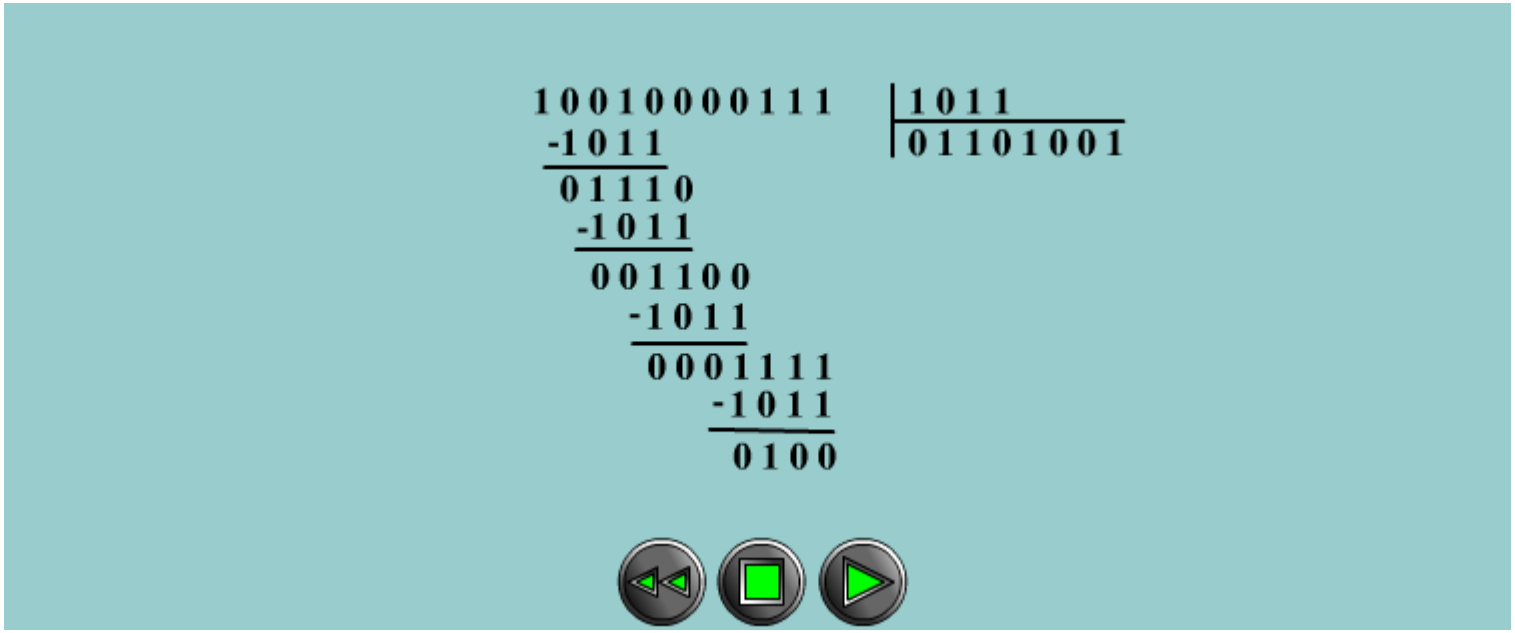


Division entre deux nombres binaires

La division binaire s'effectue à l'aide de soustractions et de décalages, comme la division décimale, sauf que les digits du quotient ne peuvent être que 1 ou 0.

Le bit du quotient est 1 si on peut soustraire le diviseur, sinon il est 0.

Voici une animation de la division du nombre 10010000111_2 par $1011_2 = 1101001_2$ reste 100_2, c'est-à-dire 1159 / 11 = 105, reste 4.



http://php.iai.heig-vd.ch/~lzo/micro/cours/Division.swf

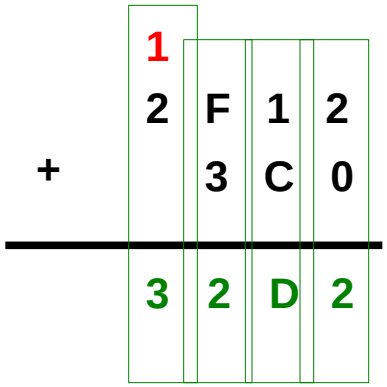
Opérations arithmétiques hexadécimales

Pour effectuer les opérations binaires, il est souvent plus rapide et moins sujet à l'erreur de travailler en hexadécimal.

L'addition en Hexadécimal

Cette addition s'effectue comme en décimal, sauf que qu'on génère une retenue lorsqu'une somme partielle dépasse 16 au lieu de 10.

Exemple : $(2.F12)_{16} + (3C0)_{16}$



Résultat : $(3.2D2)_{16}$

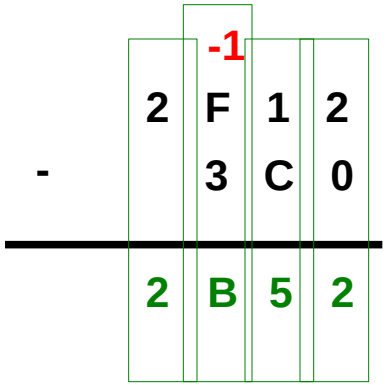
Outil :

0
1
2
3
4
5
6
7
8
9
A (10)
B (11)
C (12)
D (13)
E (14)
F (15)
10
11
12
13
14
15
16
...

La soustraction en Hexadécimal

La même chose vaut pour la soustraction. Quand le nombre du bas dépasse celui du haut, on fait un emprunt au chiffre de gauche et on ajoute 16 au nombre du haut.

Exemple : $(2.F12)_{16} - (3C0)_{16}$



Résultat : $(2.B52)_{16}$

Outil :

0
1
2
3
4
5
6
7
8
9
A (10)
B (11)
C (12)
D (13)
E (14)
F (15)
10
11
12
13
14
15
16
...

La multiplication en Hexadécimal

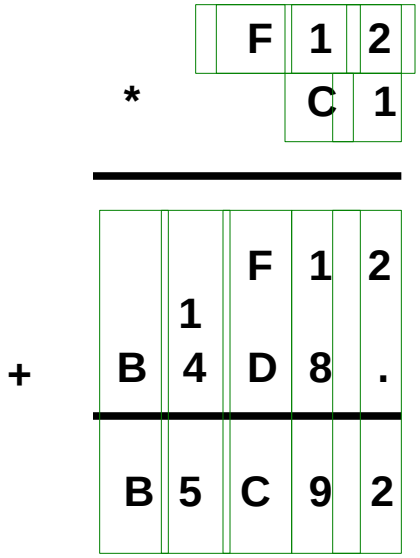
La multiplication binaire de nombres assez longs est difficile en raison du grand nombre de retenues dans la somme finale et on se trompe très souvent.

On peut l'effectuer plus facilement en hexadécimal si on dispose de la table de multiplication appropriée ci-dessous.

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

La multiplication en Hexadécimal

Exemple : $(F12)_{16} * (C1)_{16}$



Si on a de la difficulté avec les retenues, on peut effectuer ces sommes deux par deux :

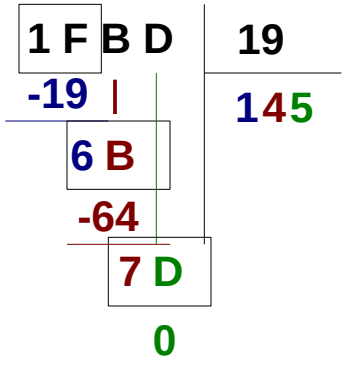
Résultat : $(B5.C92)_{16}$

Outil :

0
1
2
3
4
5
6
7
8
9
A (10)
B (11)
C (12)
D (13)
E (14)
F (15)
10
11
12
13
14
15
16
17
18
...

La division en Hexadécimal

Exemple : $(1.FBD)_{16}$ / $(19)_{16}$



Résultat : $(145)_{16}$ reste $(0)_{16}$

Outil :

1 * 19 = 19
2 * 19 = 32
3 * 19 = 4B
4 * 19 = 64
5 * 19 = 7D
6 * 19 = 96
7 * 19 = AF
8 * 19 = C8
9 * 19 = E1
A * 19 = FA
B * 19 = 113
C * 19 = 12C
D * 19 = 145
E * 19 = 15E
F * 19 = 177
10 * 19 = 190

Le code ASCII

- Le code ASCII (American Standard Code for Information Interchange) est le codage utilisé en informatique pour communiquer tout caractère alphanumérique et en particulier pour la transmission de données entre le clavier d'un micro-ordinateur et l'unité centrale.
- Il y a deux codes ASCII, le code ASCII standard et le code ASCII étendu. Le clavier est équipé d'un microprocesseur du type Intel 8048 qui scrute les circuits du clavier en permanence. Chaque touche possède un code distinct.
- Le code ASCII standard possède 127 caractères. Le code ASCII étendu en possède 255.
- Il faudra donc pour coder l'ensemble des caractères utiliser 7 bits (du bit 0 au bit 6) pour le code ASCII standard et 8 bits pour le code ASCII étendu (du bit 0 au bit 7).
- Le code ASCII différencie les lettres majuscules des lettres minuscules. Par exemple, pour écrire "A", le microprocesseur du clavier envoie à l'unité centrale le code $41_{(16)}$, pour écrire "a", le microprocesseur envoie à l'unité centrale le code $61_{(16)}$.

Tableau du code ASCII

				B6	0	0	0	0	1	1	1	1
				B5	0	0	1	1	0	0	1	1
				B4	0	1	0	1	0	1	0	1
B3	B2	B1	B0	Exemple : E = 100 0101$_{(2)}$ = 69$_{(10)}$. Appuyez sur "ALT", saisissez 69 et relâchez "ALT". Convaincu !								
0	0	0	0		NUL	DLE	SP	0	@	P	`	p
0	0	0	1		SOH	DC1	!	1	A	Q	a	q
0	0	1	0		STX	DC2	~	2	B	R	b	r
0	0	1	1		ETX	DC3	#	3	C	S	c	s
0	1	0	0		EOT	DC4	$	4	D	T	d	t
0	1	0	1		ENQ	NAK	%	5	E	U	e	u
0	1	1	0		ACK	SYN	&	6	F	V	f	v
0	1	1	1		BEL	ETB	'	7	G	W	g	w
1	0	0	0		BS	CAN	(	8	H	X	h	x
1	0	0	1		HT	EM	)	9	I	Y	i	y
1	0	1	0		LF	SUB	*	:	J	Z	j	z
1	0	1	1		VT	ESC	+	;	K	[	k	{
1	1	0	0		FF	FS	,	<	L	\	l	|
1	1	0	1		CR	GS	-	=	M	]	m	}
1	1	1	0		SO	RS	.	>	N	^	n	~
1	1	1	1		SI	US	/	?	O	_	o	DEL

Applets: liens sur la page « Supports de cours »

- Complément à 2
- Conversion d'un décimal en virgule flottante
- Analyse d'une variable en virgule flottante 32-bit et conversion en décimal
- Analyse d'une variable en virgule flottante 64-bit et conversion en décimal

Ces applets sont utiles, en particulier, comme aide et vérification aux exercices de numérotation. Elles ne remplacent pas la solution par calcul, qui reste **l'exercice qui sera demandé aux TE**.

Exercices

- Effectuer les exercices au chapitres 3 et 4 de la note d'exercices

Travail personnel

- Wikipédia en français
  - Traitement numérique (microprocesseurs)
  - Complément à deux
  - Virgule flottante
  - Système hexadécimal

- *Lire tous ces sujets,*
  noter par écrit ce qui n'est pas (encore) clair, pour des questions et approfondissements successifs en classe.