Nombres et octets

12 septembre 2024

A connaître :

1. Numération en base 2, 10 et 16. Définition, conversions, *endianess* ;
2. Représentation des entiers signés, non-signés ;
3. Représentation des flottants.

Compétences :

1. Effectuer des conversions d'une base a vers une base b ;
2. Interpréter une série d'octets par la représentation donnée ;
3. Retrouver la valeur d'un nombre IEEE 754 .
4. Proposer un protocole de données pour représenter des données sous forme de nombre.

1 Rappels de numération

Le monde en général utilise le système décimal comme base pour compter. Il comporte dix chiffres : $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Ce choix a été fait naturellement car nous avons 10 doigts. Les chiffres d'un nombre sont ordonnés. En comptant à partir de la droite :

— Le premier chiffre est le chiffre des unités (1) ;

— Le deuxième chiffre est celui des dizaines (10) ;

— Le troisième chiffre est celui des centaines (100) ;

— etc.

Ainsi, un nombre est la somme d'unités, de dizaines, de centaines, etc. On note également que les valeurs des chiffres sont toutes des puissances de 10 :

— $1 = 10^0$;

— $10 = 10^1$;

— $100 = 10^2$;

— etc.

$3892 = 3 \times 10^3 + 8 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$

— Avec 1 chiffre, on peut représenter 10 nombres : de 0 à 9 ;

— Avec 2 chiffres, on peut représenter 100 nombres : de 0 à 99 ;

— Avec 3 chiffres, on peut représenter 1000 nombres : de 0 à 999 ;

— etc.

Propriété 1.1: Nombre de combinaisons

Avec n chiffres ($n \in \mathbb{N}^*$) [a]*, on peut représenter 10^n nombres décimaux : de 0 à $10^n - 1$.*

a. $\mathbb{N}$ est l'ensemble des entiers naturels $(0, 1, 2,$ etc). $\mathbb{N}^*$ est l'ensemble des entiers naturels non-nuls, c'est-à-dire différents de 0.

1.1 Compter dans une autre base

En 1847, George Boole publie un article décrivant un système algébrique basé sur deux états logiques, vrai ou faux, et trois opérations élémentaires, ET, OU et NON. Malheureusement, ce système est tombé dans l'oubli, jusqu'en 1937, où Claude Shannon, étudiant au MIT, l'utilise pour l'électronique en remarquant que l'on peut faire correspondre l'état du circuit avec une variable booléenne.

Les premiers programmes informatiques consistaient à brancher et débrancher des prises. Ainsi, un circuit a deux états logiques possibles :

— un état logique haut, où la tension appliquée est proche de la tension d'alimentation : on le représente par le chiffre 1 ;

— un état logique bas, où la tension appliquée est proche de 0 volt : on le représente par le chiffre 0.

En combinant plusieurs circuits, il est alors possible de représenter des situations plus complexes, comme créer un circuit additionneur, soustracteur, etc.

En regroupant les états des circuits, nous obtenons un nombre composé d'états binaires, c'est à dire de 0 ou de 1, que l'on appellera nombre binaire. Comme il n'y a que deux chiffres, on dit que l'on compte en base 2.
Nous reprenons ensuite les mêmes propriétés de représentation que pour la base 10.

Propriété 1.2: Nombre de possibilités

Avec n chiffres ($n \in \mathbb{N}$), on peut représenter 2^n nombres binaires : de 0 à $2^n - 1$.

Théorème 1.3: Unicité de la représentation positionnelle

Tout entier naturel n peut se représenter par une somme de puissances de 2.
$n = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 ... + a_k \times 2^k$ *(où $k \in \mathbb{N}$, et $a_0, a_1, ..., a_k \in \{0; 1\}$)* [a]

a. $\{0; 1\}$ est l'ensemble de nombres contenant uniquement 0 et 1. Ainsi, les nombres $a_0, a_1, ..., a_k$ valent soit 0, soit 1.

Dans la pratique, nous préférerons écrire cette somme dans l'autre sens, comme pour la base 10, pour exhiber les chiffres binaires.

— $7 = 4 + 2 + 1 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (111)_2$

— $38 = 32 + 4 + 2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (10\ 0110)_2$

Note : pour ne pas confondre un nombre binaire et un nombre décimal, nous inscrivons en indice la base du nombre considéré. On utilise d'ailleurs cette propriété pour convertir un nombre binaire en un nombre décimal :

— $(111)_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$

— $(1\ 0011)_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 2 + 1 = 19$

Si trouver une décomposition en puissances de 2 est assez facile pour des petits nombres, il existe également un algorithme permettant de retrouver la représentation binaire d'un nombre décimal.

Propriété 1.4: Algorithme de conversion en base 2

A partir d'un nombre entier, on effectue une division euclidienne par 2. On obtient alors un quotient et un reste (qui est soit 0, soit 1). On récupère le quotient, on le divise par 2 et on répète l'opération jusqu'à obtenir un quotient égal à 0.
A la fin, on lit les restes dans l'ordre inverse de leur obtention (de bas en haut) pour obtenir notre nombre binaire.

Nous allons convertir le nombre décimal 38 en un nombre binaire.

— 38 / 2 = 19 reste 0

— 19 / 2 = 9 reste 1

— 9 / 2 = 4 reste 1

— 4 / 2 = 2 reste 0

— 2 / 2 = 1 reste 0

— 1 / 2 = 0 reste 1

Ainsi : $38 = (10\ 0110)_2$

Nous pouvons également réfléchir d'une autre manière pour effectuer la conversion : *Quelle est la plus grande puissance de 2 ?*

Ainsi, pour convertir le nombre 51 en binaire, on sait que $2^5 = 32$, et que $2^6 = 64$, qui est plus grand que 51. On écrit : $51 = 2^5 + 19$. On recommence avec la différence, 19. $2^4 = 16$, donc $51 = 2^5 + 2^4 + 3$. $3 = 2 + 1 = 2^1 + 2^0$ d'où $51 = 2^5 + 2^4 + 2^1 + 2^0$.

Propriété 1.5: Algorithme de décomposition en puissances de 2

A partir d'un nombre entier, on cherche la plus grande puissance de 2 inférieure ou égale. On fait la différence entre les deux nombres, et on recommence ces deux opérations (avec la différence) jusqu'à ce que la différence soit nulle. A la fin, nous obtenons la décomposition en sommes de puissances de 2 du nombre donné en entrée.

Il est possible, à partir de cette décomposition, d'en déduire l'écriture binaire du nombre. Cette volonté de chercher à chaque fois la valeur la plus grande qui correspond au critère donné a un nom particulier. On appelle cet algorithme un **algorithme glouton**. Par exemple, le rendu de monnaie est un problème qui peut se résoudre avec un algorithme glouton, car on cherche à chaque fois la pièce ou billet avec la plus grande valeur.

La représentation en base 2 amène cependant plusieurs inconvénients :

— Un nombre binaire nécessite plus de chiffres pour être écrit qu'un nombre décimal ;

— Notre société utilise la base 10, donc il n'est pas aisé de donner naturellement la valeur décimale d'un nombre binaire : seriez-vous capable de lire **instantanément** le nombre $(1001\ 1101)_2$?

Ainsi, pour réduire la quantité de chiffres nécessaires pour représenter un nombre binaire, les informaticiens (et également mathématiciens) ont eu pour idée d'utiliser d'autres bases avec plus de chiffres, et ayant un lien fort avec la base binaire : la base octale (base 8) et la base hexadécimale (base 16). La première n'étant plus vraiment d'actualité, nous nous concentrerons uniquement sur la seconde.

Si vous avez suivi attentivement le cours, vous avez peut-être remarqué que les nombres binaires sont ordonnées par groupes de quatre chiffres. Cela permet, tout comme nous séparons nos nombres décimaux par tranches de trois chiffres, d'améliorer la lisibilité.
On se souvient que $2^4 = 16$. Donc si on regroupe un nombre binaire par groupes de quatre chiffres, et qu'on associe à chaque combinaison de ces quatre chiffres un nombre, alors nous pouvons compter dans une nouvelle base : la base hexadécimale.
Pour représenter les valeurs 0 à 9, nous utilisons les mêmes chiffres que la base 10, puis pour représenter les valeurs 10 à 15, nous utiliserons les six premières lettres de notre alphabet : A,B,C,D,E et F.

Base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 2	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
Base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Propriété 1.6: Unicité de représentation en base 16

Tout entier naturel n peut se représenter par une somme de puissances de 16.
$n = a_0 \times 16^0 + a_1 \times 16^1 + a_2 \times 16^2 ... + a_k \times 16^k$ *(où* $k \in \mathbb{N}$*, et* $a_0, a_1, ..., a_k \in [\![0; 15]\!]$ *)* [a]

a. $[\![0; 15]\!]$ est l'ensemble des nombres entiers compris entre 0 et 15, inclus tous les deux.

Les conversions entre les bases 16 et 2 sont relativement aisées :

— Pour passer de la base 2 à la base 16 : on regroupe les chiffres par 4, et on convertit chaque groupe de 4 chiffres binaires en 1 chiffre hexadécimal ;

— Pour passer de la base 16 à la base 2 : on convertit chaque chiffre hexadécimal en un groupe de 4 chiffres binaires.

$(A2)_{16} = (1001\ 0010)_2$
$(101\ 0111\ 1011)_2 = (57B)_{16}$

Les algorithmes de conversions entre la base 16 et la base 10 sont sensiblement identiques à ceux liant les bases 2 et 10 :

— Pour passer de la base 10 à la base 16, on utilise le même algorithme que pour convertir en base 2, sauf que l'on divise par 16 et non par 2. On remplace bien évidemment les nombres supérieurs à 9 par leur chiffre hexadécimal correspondant.

— Pour passer de la base 10 à la base 16 : on utilise la propriété même de l'écriture hexadécimale, où chaque chiffre représente une puissance de 16.

1.2 Les nombres relatifs

En apprenant l'existence des nombres relatifs comme étant la distance algébrique (avec un sens) par rapport au nombre 0, nous avons décidé de représenter ce sens par

deux signes :

— + si le nombre est positif ("à droite de 0") ;

— − si le nombre est négatif ("à gauche de 0").

Comme il y a deux possibilités, l'introduction d'un bit de signe fait tout de suite sens.

Définition 1.7: Poids fort et faible

On appelle bit de poids fort (resp. faible) le chiffre qui représente la puissance de 2 la plus élevée (resp. basse). Même chose pour les octets : on sera amenés à parler d'octet de poids faible et poids fort.

Définition 1.8: Nombres signés

Un entier signé (en base 2) est un nombre qui se décompose en deux parties :

— *Le bit de poids fort donne le signe du nombre : 0 pour positif, 1 pour négatif ;*

— *Tous les autres bits représentent la valeur absolue de cet entier.*

On veut représenter le nombre $(-38)_{10}$ en base 2 signée, avec 8 bits. On a donc 1 bit de signe, qui sera égal à 1 ici, et 7 bits représentant la valeur absolue du nombre. $(38)_{10} = (010\ 0110)_2$. Ainsi, $(-38)_{10} = (1010\ 0110)_2$

Afin de ne pas confondre la représentation entre un entier naturel et un entier signé, on précisera en amont la représentation que l'on utilise pour le nombre.

Propriété 1.9: Nombre de possibilités

Avec n bits, on peut représenter $2^n - 1$ nombres, compris entre $-2^{n-1} - 1$ et $2^{n-1} - 1$.

Trouver la représentation de l'opposé d'un nombre est relativement aisé puisqu'il suffit de modifier le bit de signe. Mais cela pose en réalité un grave problème : il existe deux représentations différentes d'un même nombre, le zéro. En effet, toujours en restant sur une base de 8 bits signés, les nombres $(0000\ 0000)_2$ et $(1000\ 0000)_2$ représentent deux versions du zéro, une positive et une négative. Il faut donc ajouter une opération qui permet de ne garder qu'un seul zéro (celui dont l'écriture est positive) après un calcul (somme par ex.). De plus, les opérations de base comme l'addition ou la soustraction nécessitent des précautions particulières pour s'effectuer.

1.3 Complément à 2^n

Comme pour la représentation signée, il nous faut au préalable fixer un nombre de bits utilisés pour représenter un nombre relatif.

Définition 1.10: Complément d'un nombre

Soit N et K deux entiers naturels. On appelle "complément" à N le nombre $N - K$.

4 est le complément à 10 de 6 car $10 - 6 = 4$.

Cela revient à se poser la question suivante : *"A partir d'une quantité initiale, combien dois-je ajouter pour obtenir ma valeur souhaitée ?"*.

Propriété 1.11: Représentation des nombres avec le complément à 2^n

- *Si le bit de poids fort est égal à 0, alors le nombre est positif. On garde alors la représentation classique en base 2. Avec un nombre de n bits, il me reste alors $n-1$ bits pour coder mes nombres positifs. La valeur maximale positive est alors $2^7 - 1 = 127$.*
- *Si le bit de poids fort est 1, alors le nombre est négatif. Pour retrouver la valeur absolue du nombre, nous pouvons utiliser l'algorithme suivant :*
  - *On inverse tous les bits du nombre ;*
  - *On lui ajoute 1.*

  Cet algorithme permet, à partir de l'écriture d'un nombre k, d'obtenir l'écriture du nombre $-k$.

Pour illustrer la méthode du complément à deux, nous utiliserons des nombres codés avec 8 bits. Elle fonctionne comme ceci :

$(0000\ 1111)_{C2}$ est un nombre positif car le bit de poids fort vaut 0. Sa valeur absolue est donc celle de l'écriture classique en base 2 : $(0000\ 1111)_{C2} = 2^0 + 2^1 + 2^2 + 2^3 = (15)_{10}$

Je veux maintenant représenter le nombre -15. Pour cela, j'utilise l'algorithme du complément à deux :

- J'inverse les bits du nombre : $0000\ 1111 \rightarrow 1111\ 0000$;
- J'ajoute 1 : $1111\ 0000 \rightarrow 1111\ 0001$;
- $(-15)_{10} = (1111\ 0001)_{C2}$.
- Juste par simple curiosité, nous notons que $(1111\ 0001)_2 = (241)_{10}$.

Qu'en est-il du problème des deux zéros ?

- Le nombre $(0000\ 0000)_{C2}$ vaut bien 0, en suivant les règles données.
- Le nombre $(1000\ 0000)_{C2}$ est négatif. Avec l'algorithme, retrouvons sa valeur absolue :
  $1000\ 0000 \rightarrow 0111\ 1111 \rightarrow 1000\ 0000$ et $(1000\ 0000)_2 = (128)_{10}$
  Ainsi, $(1000\ 0000)_{C2} = (-128)_{10}$.

On admettra pour l'instant que -128 (ou encore -2^{n-1} est la valeur minimale pour un nombre à 8 bits. En observant attentivement les représentations en base 2 d'un nombre et son complément, on remarque que leur somme vaut 256 :

- $241 + 15 = 256$
- $128 + 128 = 256$

Nous comprenons alors l'appellation de "complément" : nous déterminons en fait le complément à 2^n du nombre considéré.

1.4 Nombres décimaux

Idée : si on peut représenter des entiers par une somme de puissances de deux, alors on peut représenter la partie décimale d'un nombre par une somme de puissances négatives de deux.

Exemple : $0.875 = 0.5 + 0.25 + 0.125 = 2^{-1} + 2^{-2} + 2^{-3}$

Comme on travaille en puissances de deux, la multiplication par deux fonctionnera très bien, et fait l'objet d'un algorithme qui permet de donner une écriture binaire.

Propriété 1.12: Algorithme donnant la représentation binaire d'un nombre décimal

Soit x un nombre décimal compris entre 0 et 1.

- *On multiplie x par deux. On note le résultat sous la forme $n + k$, où n est la partie entière, et k la partie décimale.*
- *On réitère cette opération avec k, tant que $k > 0$, ou que k a une valeur qui a déjà été obtenue.*
- *Les valeurs successives de n obtenues forment l'écriture binaire de x*

On souhaite avoir la représentation binaire du nombre 0.25.

- $0.25 \times 2 = 0 + 0.5$
- $0.5 \times 2 = 1 + \mathbf{0.0}$

Ainsi, $(0.25)_{10} = (0,01)_2$.

On souhaite avoir la représentation binaire du nombre 0.1.

- $0.1 \times 2 = 0 + \mathbf{0.2}$
- $0.2 \times 2 = 0 + 0.4$
- $0.4 \times 2 = 0 + 0.8$
- $0.8 \times 2 = 1 + 0.6$
- $0.6 \times 2 = 1 + \mathbf{0.2}$

On a obtenu à nouveau 0.2 à l'issue de notre algorithme. Nous savons alors que le motif 0011 se répète, après avoir écrit un premier 0. Nous obtenons alors un développement binaire avec un nombre infini de chiffres, comme pour le nombre $\frac{1}{3}$. Dans ce cas, nous tronquons alors à un certain nombre de chiffres, car l'ordinateur ne peut disposer d'une mémoire infinie. Nous proposons ici une précision de 10 chiffres binaires :
$(0.1)_{10} = (0,0001100110)_2$

Nous remarquons alors l'inconvénient majeur de cette écriture, elle ne donne pas toujours la valeur exacte du nombre, et on doit donc travailler avec une marge d'erreur que l'on doit minimiser pour éviter que l'erreur ne se propage trop vite. Nous allons voir comment les scientifiques se sont accordés sur la question.

1.5 Norme IEEE-754

Les nombres décimaux sont en général représentés avec la norme IEEE-754. Les deux formats les plus utilisés sont :

- simple précision, pour un total de 32 bits ;
- double précision, pour un total de 64 bits.

Quelque soit le format utilisé, un nombre IEEE-754 est découpé de la même manière :

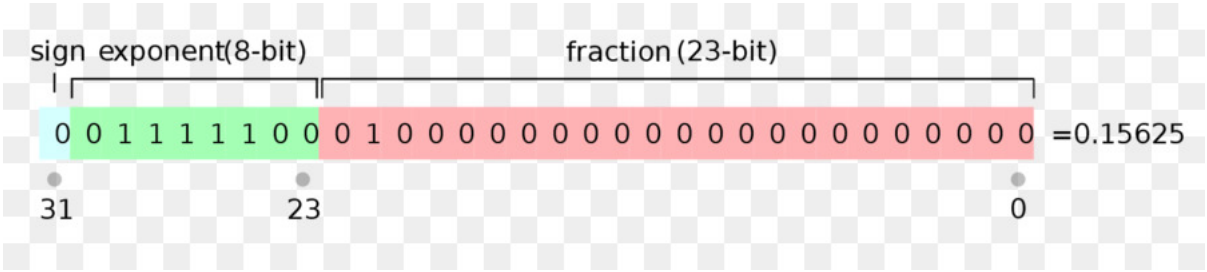

FIGURE 1 – Schéma décomposant les éléments d'un nombre IEEE754

— un bit de signe, noté s ;
— une partie qui représente l'exposant, que l'on note e ;
— une partie qui représente la mantisse, que l'on note m.

Un nombre IEEE-754, que l'on notera x, sera alors égal à :

$$x = s \times 2^{e-127} \times (1 + m)$$

Cela ressemble fortement à l'écriture scientifique d'un nombre, à la différence qu'on travaille en puissances de 2 et non de 10. m est la "partie binaire", contenant la somme de puissances de 2 négatives de ce nombre.

1.6 Les nombres en programmation

Les langages de programmation utilisent des types communs (en général) pour représenter les nombres. Ces types peuvent être signés ou non selon les langages.

Définition 1.13: Endianness

Un nombre big endian représente son octet de poids fort en premier, et un nombre little endian représente son octet de poids faible en premier.

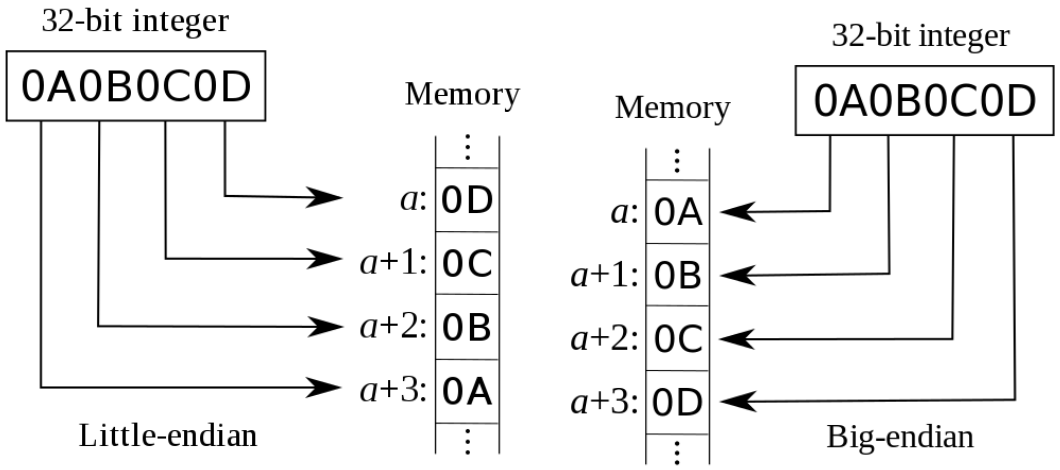

FIGURE 2 – Représentation en mémoire des nombres selon leur *endianness*

Type	Taille (octets)	Remarque
byte	1	
short	2	
int	4	
long	8	
float	4	IEEE754
double	8	IEEE754

Dans un cadre de programmation bas niveau, il peut arriver qu'on ait besoin de manipuler les octets voire des bits d'un nombre.

On peut être amené à travailler sur une certaine série de bits présente dans un nombre. Pour travailler dessus sans toucher aux autres bits, on utilise un autre entier, appelé **masque** dont les seuls bits à manipuler valent 1.

```
1 int mask13 = 1 << 13; // uniquement bit 13 à 1
2 int mask17 = 1 << 17; // uniquement bit 17 à 1
3 int mask13_17 = mask13 | mask17; // bits 13 et 17 à 1
4 int mask13to17 = 0b11111 << 13 // bits 13 à 17 à 1
```

Exercice 1 — Représentation des nombres

1. Convertir les nombres 214 et 2022 en binaire puis en hexadécimal. On ajoutera autant de 0 nécessaires à gauche pour compléter l'octet en cours (si le nombre obtenu a 6 chiffres binaires, on ajoutera deux 0 pour former un octet.

Opérateur	Action	Remarques / exemples
x	inversion/complément	Le complément de 0b11110000 est 0b00001111
x <<y	décalage à gauche	0b00001100 <<3 donnera le nombre 0b01100000
x >>y	décalage à droite	0b00001100 >>3 donnera le nombre 0b00000001
x & y	conjonction / ET logique	0b01110111 ET 0b11101110 donnera 0b01100110
x | y	disjonction / OU logique	0b01110111 OU 0b11101110 donnera 0b11111111
x ^ y	disjonction exclusive / OU exclusif	0b01110111 XOR 0b11101110 donnera 0b10011001

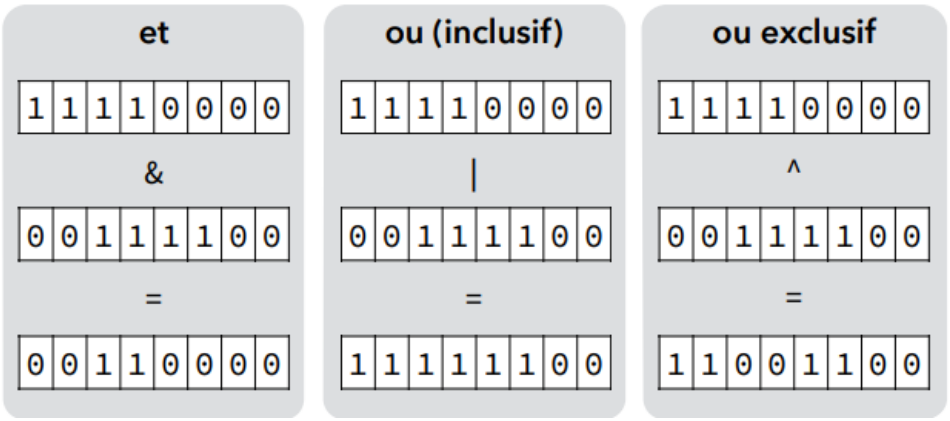

FIGURE 3 – Illustration des opérations *bitwise*.

Source: https://cs108.epfl.ch/

2. Convertir les nombres 92, -51 et -114 en représentation en complément à 2 sur 8 bits.
3. Convertir en binaire puis en décimal les nombres hexadécimaux $15FF$, $024B8E$ et $E41F$.
4. A l'aide d'une machine, convertir le nombre 2 104 845 en binaire sur 4 octets, et donner sa représentation *little endian* et *big endian*.
5. On souhaite convertir un entier n quelconque en base k. Combien faut-il de chiffres pour pouvoir le représenter ?

Exercice 2 — Interprétation d'octets

On reçoit la série de bits suivante : `0110 1101 0110 1111`.
Interpréter cette série :

1. En *big endian :*
   (a) comme un entier à 2 octets non-signé (`unsigned short int, uint16`) ;
   (b) comme un entier à 2 octets signé (`short int, int16`) ;
   (c) comme deux entiers 8 bits signés (`int8`) ;

(d) comme deux entiers 8 bits non-signés (`uint8`) ;

(e) comme deux caractères ASCII.

2. La même chose mais en *little endian*.

Exercice 3 — Bourrage d'octets

L'entier (`int`) tel qu'on le connaît a une taille de 32 bits, et un booléen sollicite 8 bits (alors que l'information peut théoriquement rentrer dans un seul bit).

Proposer une solution pour stocker un ensemble de trois nombres et deux booléens dans un seul entier, en sachant que les trois nombres sont compris entre 0 et 500 ; puis une solution pour lire les informations contenues dans un tel entier. On utilisera exclusivement les opérateurs *bitwise*.

On rédigera un algorithme, qu'on mettra en oeuvre avec Python ou Java.

Exercice 4 — Les couleurs

Les couleurs sont souvent décomposées en composantes rouge, verte et bleue en informatique, et représentées par une série de réels compris entre 0 et 1 (inclus).

1. Si on utilise des `double` pour représenter les canaux, combien d'octets sont nécessaire pour encoder les informations d'une couleur ?
2. On se propose d'utiliser une représentation plus compacte : pour chaque canal, on associe un entier compris entre 0 et 255. Combien d'octets sont désormais nécessaires ?
3. Proposer une fonction prenant en entrée les valeurs des canaux de couleur et qui les empaquette dans un `int` dont le détail est donné dans la table 1, puis la fonction qui effectue l'opération inverse.

31 ... 24	23 ... 16	15 ... 8	7 ... 0
/	r_7 ... r_0	g_7 ... g_0	b_7 ... b_0

TABLE 1 – Détail des bits d'un `int` contenant les informations de couleur

Exercice 5 — Mini-projet de conception protocole

On souhaite utilise le moins de mémoire possible pour enregistrer l'entrée d'une manette pour une *frame*. On souhaite enregistrer :

— la position de deux joysticks analogiques (deux coordonnées à valeurs dans $[0; 1]$ sur les axes x et y) ;

— l'appui de 9 boutons d'action différents,

Proposer un protocole de sérialisation utilisant le moins d'octets possibles pour représenter ces informations.

Exercice 6 — Application

Pour cet exercice, on considère uniquement les flottants en simple précision.

1. Donner la mantisse et l'exposant, puis la représentation flottante des nombres 0,625 ; -34,06125 et -483,375.
2. Extraire la mantisse, l'exposant et le signe des séries d'octets ci-dessous. Donner ensuite la représentation décimale.

1. 01000100000100101001000010000000
2. 10111111001110000000000000000000

Exercice 7 — Numération

Examen L3SPI 2023-2024

1. Vérifier que le nombre IEEE-754 de la figure 1 vaut bien 0,15625.
2. Convertir le nombre 174,125 en nombre flottant IEEE 754 simple précision *big endian*. Pour ce faire, on calculera d'abord ses valeurs de s, e et m, puis on écrira la série de bits.

Exercice 8 — Prédiction et réconciliation

Examen L3 SPI 2023-2024

Dans une application en ligne, où il y a un besoin de modéliser des entités et de les faire se déplacer, la difficulté augmente lorsqu'il faut partager ces informations avec les utilisateurs en temps réel, mais aussi que leur expérience soit satisfaisante, c'est-à-dire que les déplacements et animations présentés soient fluides.

On rappelle que dans une application client-serveur autoritaire, le serveur détient la vérité, et le client doit s'y plier.

Dans cet exercice, on considère que l'état du serveur est rafraîchi 10 fois par seconde. On peut alors signer par un nombre chaque rafraîchissement des états du monde : toutes les 100 ms, on incrémente la signature.

Un utilisateur contrôle une entité dans un plan à deux dimensions x et y muni d'un repère orthonormé [1]. L'axe des x est orienté à droite et l'axe des y en haut. Le client envoie des paquets à hauteur de 10 fois par seconde sur l'état de son clavier : les touches Z,Q,S,D donnent les ordres de déplacement respectivement en haut, à gauche, en bas et à droite. Un ordre de déplacement permet de se déplacer d'une unité en $100ms$: si à $t = 0ms$ en $(0; 0)$, j'envoie un ordre de me déplacer vers le haut, alors à $t = 100ms$, l'entité sera en $(0, 1)$

Pour transférer les données aux clients, le protocole de sérialisation binaire correspond à la transmission de ces données, toutes encodées en *big-endian*, dans l'ordre :

1. Le numéro de signature de l'état (entier 32 bits non-signé) ;
2. Le nombre d'utilisateurs (1 octet non-signé) ;
3. Pour chaque utilisateur :
   (a) son identifiant (1 octet non-signé) ;
   (b) sa coordonnée sur x (short int signé) ;
   (c) sa coordonnée sur y (short int signé).

Interpréter le paquet reçu par le client ci-dessous :

```
00000000 00000000 00000000 00011010
00000011 10011101 00000100 11000000
10000000 00001111 01011001 10000000
00011110 10000000 00000010 11110110
00001000 00000000 00000000 00001000
```

1. Rappel : les axes sont perpendiculaires et de même unité