Notions d'architecture et de système

Eliana Carozza

01/09/2025

1 Arithmétique des ordinateurs

Dans un ordinateur, toutes les informations (données ou programmes) sont représentées à l'aide de deux chiffres 0 et 1, appelés chiffres binaires ou Binary Digits en anglais (ou plus simplement bits).

Dans la mémoire d'un ordinateur (RAM, ROM, registres des micro-processeurs, etc.), ces chiffres binaires sont regroupés en octets (c'est-à-dire par « paquets » de 8, qu'on appelle *bytes* en anglais) puis organisés en mots machine (en dit *words* en anglais) de 2, 4 ou 8 octets (pour les machines les plus courantes).

Par exemple, une machine dite de 64 bits est un ordinateur qui manipule directement des mots de 8 octets ($8 \times 8 = 64$ bits) lorsqu'il effectue des opérations (en mémoire ou dans ses calculateurs).

Ce regroupement des bits en octets ou mots machine permet de représenter (et manipuler) d'autres données que des 0 ou des 1, comme par exemple des nombres entiers, des (approximations de) nombres réels, des caractères alpha-numériques ou des textes. Néanmoins, il est nécessaire d'inventer des encodages pour représenter ces informations.

1.1 Représentation des entiers

1.1.1 Encodage des entiers naturels.

L'encodage le plus simple est celui des nombres entiers naturels. Il consiste simplement à interpréter un octet ou un mot machine comme un entier écrit en base 2. Dans cette base, les chiffres (0 ou 1) d'une séquence sont associés à un poids 2^i (puissance de 2 qui dépend de la position i des chiffres dans la séquence), de manière similaire à l'encodage en base 10.

Par exemple, l'octet `01001101` peut être représenté en colonnes de la manière suivante :

séquence	0	1	0	0	1	1	0	1
positions	7	6	5	4	3	2	1	0
poids	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Vu comme un entier composé de huit chiffres binaires, cet octet correspond au nombre N calculé de la manière suivante :

$N = 0\times 2^7 + 1\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 64+8+4+1 = 77.$

Plus généralement, une séquence $b_{n-1}b_{n-2}\dots b_1 b_0$ de n bits b_i correspond au nombre N suivant :

$$N = \sum_{0\leq i<n} b_i \times 2^i$$

Le chiffre b_{n-1} est appelé le *bit de poids fort* et le chiffre b_0 est appelé le *bit de poids faible*.

Cet encodage des entiers naturels par des séquences de n chiffres binaires permet donc de représenter tous les entiers de 0 à $2^n - 1$.

Écriture en base 16. Une autre base fréquemment utilisée est la base 16, dite hexadécimale. Puisqu'il faut pouvoir écrire 16 chiffres hexadécimaux, on utilise les chiffres de 0 à 9 pour les 10 premiers, puis les lettres A, B, C, D, E et F pour les 6 derniers. La valeur de chaque lettre est donnée par le tableau de correspondance ci-dessous :

lettre	valeur
A	10
B	11
C	12
D	13
E	14
F	15

Par exemple, la séquence `2A4D` correspond à la représentation en colonnes suivante :

séquence	2	A	4	D
positions	3	2	1	0
poids	16^3	16^2	16^1	16^0

La valeur de cette séquence correspond donc au nombre :

$$N = 2\times 16^3 + 10\times 16^2 + 4\times 16^1 + 13\times 16^0 = 10829$$

On peut facilement passer d'un nombre en base 2 à un nombre en base 16 en regroupant les chiffres binaires par 4. Par exemple, la séquence de bits `1010010111110011` correspond au nombre hexadécimal `A5F3`, comme on peut le voir simplement de la manière suivante :

A	5	F	3
1010	0101	1111	0011

Boutisme (Endianness). La représentation en machine des entiers naturels sur des mots de 2, 4 ou 8 octets se heurte au problème de l'ordre dans lequel ces octets sont organisés en mémoire. Ce problème est appelé le *boutisme* (ou *endianness* en anglais).

Prenons l'exemple d'un mot de 2 octets (16 bits) comme `4CB6`. Il y a deux organisations possibles d'un tel mot en mémoire :

— **Gros boutisme** (*big endian*) : l'octet de poids fort est placé en premier, c'est-à-dire à l'adresse mémoire la plus petite :

$$\ldots\ 4C\ B6\ \ldots$$

— **Petit boutisme** (*little endian*) : au contraire, l'octet de poids faible est placé en premier :

$$\ldots\ B6\ 4C\ \ldots$$

En principe, la représentation petit ou gros boutiste est transparente à l'utilisateur, car elle est gérée au niveau du système d'exploitation. Elle prend toutefois de l'importance lorsqu'on accède directement aux octets, soit en mémoire, soit lors d'échanges d'informations sur un réseau.

Les avantages et inconvénients des deux représentations sont multiples : la lisibilité est plus simple pour un humain en gros boutiste, tandis que les opérations arithmétiques se font plus facilement en petit boutiste.

1.1.2 Encodage des entiers relatifs

Principe de base. L'encodage des entiers relatifs est plus délicat. L'idée principale est d'utiliser le bit de poids fort d'un mot mémoire pour représenter le signe d'un entier :

$$0 \Rightarrow \text{entier positif}, \qquad 1 \Rightarrow \text{entier négatif}.$$

De cette manière, l'encodage des entiers naturels ne change pas. Par exemple, pour des mots binaires sur 4 bits :

- le mot `0011` représente l'entier 3, - le mot `1101` représente l'entier -5.

Un mot binaire de n bits permet alors de représenter les entiers relatifs dans l'intervalle :

$$-(2^{n-1}-1) \ \leq N \leq \ 2^{n-1}-1.$$

Limites de cet encodage. Cet encodage simpliste souffre de deux problèmes :

1. le nombre zéro possède deux représentations. Par exemple, avec des mots de 4 bits, `0000` et `1000` codent tous deux 0. 2. les opérations arithmétiques sont compliquées.

En effet, pour additionner deux entiers relatifs encodés de cette manière, il faut choisir entre addition ou soustraction selon les signes. Ainsi, l'addition binaire de `0101` (5 en base 10) et `1101` (–5 en base 10) donne :

$$0101 + 1101 = 0010 \quad \text{(en ignorant la retenue)}$$

mais `0010` (2 en base 10) n'est pas la représentation de 0.

Complément à 2. La solution la plus courante est d'utiliser l'encodage dit par *complément à 2*. Dans cet encodage, le bit de poids fort représente toujours le signe, mais il est maintenant interprété comme ayant la valeur -2^{n-1} pour un entier écrit sur n bits.

Ainsi, la séquence de bits $b_{n-1}b_{n-2}\dots b_1b_0$ est interprétée comme le nombre :

$$N = -b_{n-1} \times 2^{n-1} + \sum_{0 \leq i < n-1} b_i \times 2^i.$$

La représentation des nombres positifs ou nuls est inchangée : le nombre 5 s'écrit toujours `0101` sur 4 bits. En revanche, -5 s'écrit désormais `1011`.

Avec cet encodage, un mot binaire de n bits permet de représenter les entiers relatifs dans l'intervalle :

$$-2^{n-1} \quad \leq N \leq \quad 2^{n-1} - 1.$$

Par exemple : - sur 4 bits : de –8 à 7, - sur 8 bits : de –128 à 127, - sur 16 bits : de –32768 à 32767.

Avantage du complément à 2. Avec le complément à 2, l'addition de deux mots binaires m_1 et m_2 s'effectue simplement comme une addition binaire $m_1 + m_2$, sans se soucier des signes des entiers encodés.

Exemple (sur 4 bits) :

$$0101 + 1011 = 0000 \quad \text{(sans tenir compte de la retenue finale)}$$

Entiers signés et non signés En informatique, on parle d'*entiers signés* pour les entiers relatifs, et d'*entiers non signés* pour les entiers naturels.

Il est important de comprendre que, dans la machine, rien n'indique qu'un ou plusieurs octets représentent un entier signé ou non signé. C'est le contexte qui détermine l'interprétation, notamment lors des opérations de conversion depuis et vers des chaînes de caractères.

2.1.2 Représentation approximative des nombres réels

Un ordinateur n'est pas en mesure de représenter les nombres réels. Même en se limitant à des opérations arithmétiques très simples, on peut montrer que ce n'est pas possible. Pour cette raison, un ordinateur n'offre qu'une approximation des nombres réels, le plus souvent sous la forme de nombres flottants.

L'encodage des nombres flottants est inspiré de l'écriture scientifique des nombres décimaux, qui se compose d'un signe (+ ou –), d'un nombre décimal m, appelé *mantisse*, compris dans l'intervalle $[1, 10[$ (1 inclus et 10 exclu) et d'un entier relatif n appelé *exposant*.

Ainsi, de manière générale, l'écriture scientifique d'un nombre décimal est de la forme :

$$\pm m \times 10^n$$

avec m la mantisse et n l'exposant. Le nombre 0 ne peut pas être représenté avec cette écriture.

Norme IEEE 754. La représentation des nombres flottants et les opérations arithmétiques qui les accompagnent ont été définies dans la norme internationale IEEE 754.

Selon la précision ou l'intervalle de représentation souhaité, la norme définit : - un format de données sur 32 bits, appelé *simple précision* ou `binary32`, - un format de données sur 64 bits, appelé *double précision* ou `binary64`.

Les différences entre la norme IEEE 754 et l'écriture scientifique sont :

1. la base choisie est la base 2,
2. la mantisse est donc dans l'intervalle $[1, 2[$,
3. l'exposant n est *décalé* (ou biaisé) d'une valeur d qui dépend du format choisi (32 ou 64 bits).

Format 32 bits. Dans le format 32 bits, représenté par le schéma ci-dessous, le bit de poids fort est utilisé pour représenter le signe s (avec 0 pour le signe $+$), les 8 bits suivants stockent la valeur de l'exposant n, et les 23 derniers bits décrivent la mantisse :

signe	exposant	mantisse (fraction)
1	8	23

Pour représenter des exposants positifs et négatifs, la norme IEEE 754 n'utilise pas le complément à 2, mais un encodage *décalé* sous forme d'entier non signé. Dans le format 32 bits, l'exposant est décalé avec $d = 127$, ce qui permet de représenter des exposants signés dans l'intervalle $[-127, 128]$. Les valeurs 0 et 255 étant réservées pour des cas particuliers, les exposants valides sont donc ceux de l'intervalle $[-126, 127]$.

Mantisse normalisée. La mantisse m est toujours comprise dans l'intervalle $[1, 2[$, c'est-à-dire un nombre de la forme $1, xx \dots xx$, commençant nécessairement par le chiffre 1.

Ainsi, pour gagner 1 bit de précision, les 23 bits dédiés à la mantisse servent uniquement à représenter la partie fractionnaire (après la virgule).

Si les 23 bits de mantisse sont $b_1 b_2 \dots b_{23}$, alors la mantisse est donnée par :

$$m = 1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_{23} \times 2^{-23}.$$

Exemple. Considérons le mot de 32 bits suivant :

$$\underbrace{1}_{\text{signe}}\ \underbrace{10000110}_{\text{exposant}}\ \underbrace{10101101100000000000000}_{\text{fraction}}$$

Le calcul associé est le suivant :

$$\text{signe} = (-1)^1 = -1, \qquad \text{exposant} = (2^7 + 2^2 + 2^1) - 127 = (128 + 4 + 2) - 127 = 7,$$

$$\text{mantisse} = 1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-7} = 1{,}677734375.$$

Ainsi, ce mot représente le nombre décimal :

$$-1{,}677734375 \times 2^7 = -214{,}75.$$

Résumé des formats. La différence entre les encodages 32 et 64 bits réside dans la valeur d du décalage pour l'exposant et dans le nombre de bits alloués à la fraction f et à l'exposant n.

	exposant (e)	fraction (f)	valeur
32 bits	8 bits	23 bits	$(-1)^s \times 1.f \times 2^{(e-127)}$
64 bits	11 bits	52 bits	$(-1)^s \times 1.f \times 2^{(e-1023)}$

En simple précision (32 bits), les flottants positifs couvrent approximativement l'intervalle $[10^{-38}, 10^{38}]$, tandis qu'en double précision (64 bits), l'intervalle représentable est $[10^{-308}, 10^{308}]$.

Valeurs spéciales. Les encodages particuliers définis par la norme sont donnés ci-dessous :

signe	exposant	fraction	valeur spéciale
0	0	0	$+0$
1	0	0	-0
0	255	0	$+\infty$
1	255	0	$-\infty$
$*$	255	$\neq 0$	NaN

Nombres dénormalisés. Les nombres représentés précédemment sont les nombres flottants *normalisés*. Avec cet encodage, le plus petit nombre décimal positif représentable est 2^{-126} (soit $\approx 10^{-38}$).

Puisque la mantisse est implicitement de la forme $1, f$, il n'existe pas de nombres représentables dans l'intervalle $[0, 2^{-126}[$, alors qu'il y en a 2^{23} dans l'intervalle $[1 \times 2^{-126}, 2 \times 2^{-126}] = [2^{-126}, 2^{-125}]$.

Afin de rééquilibrer la représentation des nombres autour de 0, la norme IEEE 754 permet d'encoder des nombres de la forme :

$$(-1)^s \times 0{,}f \times 2^{-126}$$

avec une mantisse $0, f$ commençant implicitement par un 0. Ces nombres flottants, appelés *nombres dénormalisés*, correspondent à des flottants avec un exposant égal à 0 et une mantisse non nulle.

Ainsi, la plus petite valeur représentable est $2^{-23} \times 2^{-126} = 2^{-149}$.

Arrondis. Lorsqu'une valeur ne peut pas être représentée exactement, il faut l'*arrondir* en choisissant le flottant le plus adapté. La norme IEEE 754 définit quatre modes d'arrondi :

— au plus près : le flottant le plus proche de la valeur exacte (en cas d'égalité, on choisit le flottant pair, c'est-à-dire avec une mantisse se terminant par 0) ;
— vers zéro : le flottant le plus proche de 0 ;

— vers $+\infty$: le plus petit flottant supérieur ou égal à la valeur exacte ;
— vers $-\infty$: le plus grand flottant inférieur ou égal à la valeur exacte.

Flottants dans les langages de programmation. Les langages de programmation, et notamment C et OCaml, proposent des nombres flottants encodés suivant la norme IEEE 754. En C, le type `float` correspond au format 32 bits (simple précision), tandis que le type `double` correspond au format 64 bits (double précision). En OCaml, tous les flottants sont en double précision.

On peut utiliser la notation décimale ou scientifique pour les définir, avec le point comme séparateur décimal. Lorsqu'on imprime un nombre flottant, on précise le nombre de chiffres après la virgule. En cas de débordement ou d'erreur arithmétique, on obtient silencieusement des valeurs infinies ou NaN.

Propriétés des nombres flottants. Il faut être très prudent dans l'usage des flottants :
— $1,2$ n'a pas de représentation exacte ;
— l'addition et la multiplication ne sont pas associatives ;
— la multiplication n'est pas distributive par rapport à l'addition ;
— il est dangereux d'utiliser des tests d'égalité entre flottants dans un programme.

2 Modèle de von Neumann

Au milieu des années 40, une équipe de chercheurs de l'université de Pennsylvanie conçoit un ordinateur dans lequel les programmes à exécuter étaient stockés au même endroit que les données qu'ils devaient manipuler, à savoir dans la mémoire de l'ordinateur.

Cette architecture est appelée *modèle de von Neumann.*

2.1 Composants d'un ordinateur

On distingue quatre grands types de composants :

1. une unité arithmétique et logique,
2. une unité de contrôle,
3. la mémoire de l'ordinateur,
4. les périphériques d'entrée-sortie.

Unité centrale de traitement (CPU). Les deux premiers composants (unité arithmétique et logique, et unité de contrôle) sont habituellement rassemblés dans un ensemble de circuits électroniques appelé *Unité Centrale de Traitement*, ou plus simplement *processeur (CPU).*

Unité arithmétique et logique. C'est un circuit électronique qui effectue : - des opérations arithmétiques sur les nombres entiers et flottants, - des opérations logiques sur les bits.

Unité de contrôle. Elle joue le rôle de chef d'orchestre de l'ordinateur. Sa tâche est de récupérer en mémoire la prochaine instruction à exécuter ainsi que les données sur lesquelles elle doit opérer.

Mémoire de l'ordinateur. La mémoire contient à la fois les programmes et les données.

Mémoire vive (volatile) : elle perd son contenu dès que l'ordinateur est éteint. Son avantage principal est la rapidité d'accès aux données.

Mémoire non volatile : elle conserve ses données même sans alimentation électrique, mais elle est en général beaucoup plus lente que la RAM.

Périphériques d'entrée-sortie. Il existe un très grand nombre de périphériques permettant la communication entre l'ordinateur et le monde extérieur (clavier, écran, disque, imprimante, etc.).

Exemples de périphériques.
- claviers, souris,
- manettes de jeu,
- scanners, appareils photos, webcams,
- écrans et vidéo-projecteurs,
- imprimantes,
- haut-parleurs,
- périphériques à la fois d'entrée et de sortie : lecteurs de disques (CD, Blu-Ray, etc.), disques durs, clés USB, cartes SD, cartes réseaux.

Unité de contrôle. Elle est constituée de trois sous-composants :
- le *registre d'instruction (IR)* contient l'instruction courante à décoder et exécuter,
- le *pointeur d'instruction (IP)* indique l'emplacement mémoire de la prochaine instruction,
- le *micro-programme* contrôle presque tous les mouvements de données entre la mémoire, l'ALU et les périphériques d'entrée-sortie.

Unité arithmétique et logique (ALU). Elle est composée de plusieurs registres, dits registres de données, et d'un registre spécial appelé *accumulateur*, dans lequel s'effectuent les calculs. L'ALU peut également envoyer des signaux pour indiquer des erreurs de calcul.

Mémoire. Les échanges de données entre l'unité centrale de traitement et la mémoire se font à travers un médium de communication appelé *bus*. Les périphériques d'entrée-sortie peuvent également lire et écrire directement dans la mémoire par ce bus, sans passer par le CPU. Cet accès direct est réalisé par un circuit spécialisé appelé *contrôleur DMA*.

Dispositifs d'entrée-sortie. Ils sont connectés à l'ordinateur par des circuits électroniques appelés *ports d'entrée-sortie*. L'accès à ces ports se fait habituellement via des adresses mémoires prédéfinies.

Pour connaître l'état d'un périphérique, le CPU peut :
- soit lire périodiquement les adresses correspondantes,

— soit être directement averti par le périphérique grâce à un mécanisme d'*interruption*.
Lorsqu'une interruption survient, le CPU peut alors lire le contenu des ports.

2.2 Organisation de la mémoire

La mémoire d'un ordinateur peut être vue comme un tableau de cases appelées *mots mémoire*. Chaque case possède une adresse unique. Traditionnellement, ce tableau mémoire est représenté verticalement.

L'espace mémoire d'un programme actif (appelé *processus*) est découpé en quatre segments principaux :

— le **segment de code** : contient les instructions du programme ;
— le **segment de données** : contient les données dont l'adresse et la valeur sont connues dès l'initialisation ;
— le **segment de pile (stack)** : contient l'espace mémoire alloué dynamiquement lors des appels de fonctions (paramètres et variables locales) ;
— le **segment du tas (heap)** : contient toutes les données allouées dynamiquement par un programme.

Accès aux mots de la mémoire. L'adresse d'un mot est donnée par :

$$\text{adresse} = \text{adresse du segment} + \text{déplacement dans le segment}.$$

Gestion du segment de pile. La pile permet de gérer facilement l'espace mémoire des fonctions. Grâce à l'imbrication des appels, une fonction f qui appelle une fonction g ne reprend son exécution qu'après la fin de g. La pile conserve l'adresse de retour dans f pour reprendre l'exécution après l'appel.

Un trop grand nombre d'appels imbriqués peut provoquer un *débordement de pile* (*stack overflow*).

Gestion du tas. Le tas est la zone mémoire où sont placées les données allouées dynamiquement. Dans certains langages, le programmeur utilise une instruction explicite pour allouer et libérer la mémoire. Dans d'autres (comme OCaml), l'allocation est automatique.

La principale difficulté réside dans la libération des zones mémoire : libérer trop tôt une zone encore utilisée entraîne des erreurs d'exécution.

En OCaml, la libération est gérée automatiquement par un mécanisme de *ramasse-miettes* (garbage collector).

2.3 Limitation du modèle de von Neumann.

Ce modèle impose un va-et-vient constant entre le CPU et la mémoire.

La différence de vitesse entre les microprocesseurs et la mémoire est telle qu'aujourd'hui, avec cette architecture, les microprocesseurs modernes passeraient tout leur temps à attendre des données venant de la mémoire. C'est ce qu'on appelle le *goulot d'étranglement du modèle de von Neumann*.

Pour y remédier, les fabricants ont introduit les *mémoires caches*. Ce sont des composants très rapides (mais coûteux) qui s'intercalent entre la mémoire principale et le CPU.

Architectures parallèles. D'autres pistes ont également été explorées, en particulier les architectures dites *parallèles*, dans lesquelles plusieurs opérations peuvent être exécutées simultanément sur différentes données.

Exemples :
- **SIMD** : *Single Instruction Multiple Data*,
- **MIMD** : *Multiple Instructions Multiple Data.*

2.4 Langage machine et assembleur

Les programmes stockés dans la mémoire centrale d'un ordinateur sont constitués d'instructions de bas niveau, directement compréhensibles par le CPU. Ces instructions ne sont rien d'autre que de simples octets.

Pour progresser dans l'exécution d'un programme, l'unité de contrôle de l'ordinateur réalise en continu, à un rythme imposé par une horloge globale, un *cycle d'exécution d'une instruction*, constitué de trois étapes :

1. **Chargement.** À l'adresse mémoire indiquée par son registre *IP*, l'unité de contrôle récupère le mot binaire qui contient la prochaine instruction à exécuter et le stocke dans son registre *IR*.
2. **Décodage.** La suite de bits contenue dans le registre *IR* est décodée pour déterminer quelle instruction doit être exécutée et sur quelles données. C'est également à cette étape que sont chargées les données.
3. **Exécution.** L'instruction est exécutée : - par l'ALU s'il s'agit d'une opération arithmétique ou logique, - par l'unité de contrôle s'il s'agit d'une opération de branchement.

Il est parfois nécessaire d'écrire directement des programmes en langage machine. Cependant, il n'est pas raisonnable de manipuler les mots binaires bruts. On utilise alors un *langage d'assemblage* (ou *assembleur*), qui constitue le langage le plus bas niveau d'un ordinateur lisible par un humain.