Organisation des ordinateurs – Synthèse

Tom BERTRAND

Table des matières

Représentation des données

1 Information

1.1 Justification du signal binaire

Deux types de signaux sont comparés dans ce cours :

- Le signal analogique : prend des valeurs sur un domaine continu. Ce signal à pour désavantage qu'une erreur causée par un bruit parasite fausse totalement la valeur reçue ;
- Le signal numérique : prend ces valeurs dans un intervalle discret. Ce signal a pour défaut de n'avoir qu'un choix restreint des valeurs envoyables.

Il a été choisi d'utiliser la deuxième option et de permettre au signal de prendre deux valeurs, celles représentée par 1 bit, 0 et 1. Ces valeurs prennent la forme de tensions électriques dans les circuits internes d'un ordinateur. L'utilisation d'un tel signal permet de s'assurer que la valeur envoyée est bien reçue afin de ne pas corrompre des données.

1.2 Quantification de l'information

L'unité de l'information est le *bit*. Un bit correspond à l'information relative à l'issue d'un « pile ou face ». Plus précisément,

$$q = \log_2 \frac{1}{P}$$

où P est la probabilité qu'un événement équiprobable se produise.

Notons que 1 byte = 1 octet = 1 B = 8 bits = 2 nibbles même si cette dernière unité est très rarement utilisée.

Une information d'une taille de x bits sera représentée par une suite d'autant de bits valant chacun 0 ou 1. Cependant comme il n'est possible de représenter qu'un nombre entier de bits sur un ordinateur traditionnel, il faudra parfois arrondir la taille de l'information à l'unité supérieure.

2 Représentation binaire des entiers

2.1 Représentations d'entiers dans une base donnée

Une base $r \begin{cases} \in \mathbb{N} \\ > 1 \end{cases}$ utilise les chiffres $0, 1, 2, \dots, r-1$.

Et on représente le nombre entier N comme $N = \sum_{k=0} r^k b_k$ où chaque $b_k \in \{0, 1, 2, \dots, r-1\}$.

Dans le cas particulier de nombres entiers représentés en binaire, on a $N = \sum_{k=0} 2^k b_k$

2.2 Représentations d'entiers en binaire par valeur signée

Si le bit de poids fort vaut 0, le nombre est positif, sinon, négatif. Les bits restants sont analysés traditionnellement avec la formule abordée plus haut.

Pour une suite de n bits,

$$\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{vs} \rightarrow \begin{cases} b_{n-1} & \text{donne le signe} \\ b_{n-2}, \ldots, b_0 & \text{donnent la valeur absolue} \end{cases}$$

Par ailleurs, on a

$$N = (1 - 2b_{n-1}) \sum_{k=0}^{n-2} 2^k b_k \in [-2^{n-1} + 1\ ;\ 2^{n-1} - 1]$$

Remarquons également qu'il existe deux représentations distinctes de 0, le zéro positif et le zéro négatif.

$$\boxed{0\ 0\ 0\ \ldots\ 0\ 0}_{vs} = \boxed{1\ 0\ 0\ \ldots\ 0\ 0}_{vs} = 0$$

La représentation par valeur signée a le malheur de ne pas être efficace lors d'opérations arithmétiques simples.

2.3 Représentation d'entiers par complément à un

À partir de la représentation par valeur signée, si le bit de poids fort est nul (le nombre est donc positif), on garde la représentation. Si le bit de poids fort vaut 1, on complémente le reste des bits (i.e. on change les 0 en 1 et vice-versa).

Pour une suite de n bits,

$$\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c1} = \begin{cases} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{vs} & \text{si } b_{n-1} = 0 \\ \boxed{b_{n-1}\ 1 - b_{n-2}\ \ldots\ 1 - b_1\ 1 - b_0}_{vs} & \text{si } b_{n-1} = 1 \end{cases}$$

En comparaison avec des valeurs non signées, on a plutôt

$$\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c1} = \begin{cases} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} & \text{si } b_{n-1} = 0 \\ \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} - 2^n + 1 & \text{si } b_{n-1} = 1 \end{cases}$$

Par ailleurs, on a

$$N = (1 - 2^n) b_{n-1} + \sum_{k=0}^{n-1} 2^k b_k \in [-2^{n-1} + 1\ ;2^{n-1} - 1]$$

Remarquons qu'il existe également pour cette représentation deux expressions de 0 :

$$\boxed{1\ 1\ \ldots\ 1\ 1}_{c1} = \boxed{0\ 0\ \ldots\ 0\ 0}_{c1} = 0$$

Propriétés arithmétiques

$$\begin{array}{cc} & w_1 \\ + & w_2 \\ \hline & w_3 \end{array}$$

— Si l'addition n'entraine pas de report en position n,

$$(w'')_{ns} = (w)_{ns} + (w')_{ns}$$

— Si l'addition entraine un report en position n,

$$(w'')_{ns} = (w)_{ns} + (w')_{ns} - 2^n$$

Évaluation de la somme

Soient $b_{n-1}, b'_{n-1}, b''_{n-1}$, respectivement les bits de poids fort de w, w', w''.

— Si $b_{n-1} = b'_{n-1} \neq b''_{n-1} \Rightarrow$ dépassement arithmétique
— Sinon
  • Si l'on a un report à la position $n \Rightarrow (w'')_{c1} = (w)_{c1} + (w')_{c1} - 1$
  • Si l'on n'a pas de report à la position $n \Rightarrow (w'')_{c1} = (w)_{c1} + (w')_{c1}$

En pratique,

i. On additionne « bêtement »

ii. Si un report est généré en position n, on l'ignore mais on incrémente le résultat (écrit en complément à un) de 1

iii. Si les opérandes sont de même signe mais que le résultat a un signe différent, il y a dépassement arithmétique

2.4 Complément à deux

À partir de la représentation par complément à un, la conversion se fait très simplement :

$$\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c2} = \begin{cases} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c1} & \text{si } b_{n-1} = 0 \\ \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c1} - 1 & \text{si } b_{n-1} = 1 \end{cases}$$

En comparaison avec la représentation non signée, on a

$$\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c2} = \begin{cases} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} & \text{si } b_{n-1} \\ \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} - 2^n & \text{si } b_{n-1} \end{cases}$$

Ceci implique que $\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c2} =_{2^n} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns}$ [(1)]

Par ailleurs, on a

$$N = -2^n b_{n-1} + \sum_{k=0}^{n-1} 2^k b_k \in [-2^{n-1}\ ; 2^{n-1} - 1]$$

Propriétés arithmétiques

— Si l'on précède le bit de signe par un bit de même valeur, le nombre ne change pas
— $\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}\ \%2^k = 0 \Leftrightarrow$ les k derniers bits sont nuls
— L'opposé d'un nombre représenté en complément à deux est obtenu en calculant la valeur en complément à deux du complément de ce nombre incrémenté d'une unité
— La somme et le produit de nombres représentés en complément à deux se fait naturellement et le résultat est correct modulo-2^n.

[(1)] $\boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c2} =_{2^n} \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} \Leftrightarrow \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{c2} \%2^n = \boxed{b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0}_{ns} \%2^n$

3 Représentation binaire des « réels »

3.1 Représentation par virgule fixe

Dans la représentation à virgule fixe, un bit représente l'unité et les bits situés à gauche et à droite de celui-ci représentent les puissances à exposants respectivement positifs et négatifs de deux :

$$\begin{array}{|cccccccccc|}\hline 2^{n-m-1} & 2^{n-m-2} & \dots & 2^1 & 2^0 & 2^{-1} & \dots & 2^{-m+1} & 2^{-m} \\ \hline b_{n-1} & b_{n-2} & \dots & b_{m+1} & b_m & b_{m-1} & \dots & b_1 & b_0 \\ \hline \end{array} = \frac{\sum_{k=0}^{n-1} 2^k b_k}{2^m}$$

Le grand désavantage de cette représentation est que, comme son nom l'indique, la virgule est fixe. Ceci peut poser problème lors de l'addition de nombres dont la virgule n'est pas fixée à des bits de même poids. En effet, pour effectuer l'opération d'addition sur deux nombres, il faudra sacrifier de l'information en décalant la virgule d'un des deux nombres. Dans la majorité des cas, on sacrifiera les bits de poids faible afin de réduire l'erreur induite. Quoi qu'il en soit, ce phénomène est bien dérangeant.

3.2 Représentation par virgule flottante (IEEE 754)

La représentation IEEE 754 est une norme de représentation par virgule flottante des nombres. On représente un nombre comme

$$N = (1 - 2s)\ m\ 2^e$$

où s est le signe, m est la mantisse et e est un exposant.

On représente N en une suite de bits

signe	~exposant~	~mantisse~

La composante de signe fonctionne comme pour la représentation par valeur signée. Cette composante contient un bit d'information dont la valeur fixe le signe du nombre. Si le bit vaut 1, le nombre est négatif, sinon, positif.

Ensuite, la valeur de l'exposant e est telle que $e = \sim\text{exposant}\sim - 127$ en simple précision ou bien $e = \sim\text{exposant}\sim - 1023$ en double précision. Et comme ~exposant~ est stocké sur 8 bits en simple précision et 11 en double précision, la valeur de e appartient à $[-127\ ;128]$ en simple précision et, à $[-1023\ ;1024]$ en double précision.

Enfin, la mantisse est exprimée sur 23 bits en simple précision et sur 52 bits en double précision. L'interprétation de la mantisse dépend de la valeur de e :

Mantisse normalisée

Si e n'est pas égal à une des bornes de son intervalle de valeurs ni à 0, la mantisse est dite normalisée. On a alors

$$\sim\text{mantisse}\sim = \begin{array}{|cccccc|}\hline 2^{-1} & 2^{-2} & \dots & 2^{-(n-1)} & 2^{-n} \\ \hline b_1 & b_2 & \dots & b_{n-1} & b_n \\ \hline \end{array}$$

$$\Rightarrow m = 1 + \sim\text{mantisse}\sim = 1 + \sum_{k=1}^{n} 2^{-k} b_k \begin{cases} \in \left[1\ ;2 - \dfrac{1}{2^{23}}\right] \text{ en simple précision} \\ \in \left[1\ ;2 - \dfrac{1}{2^{52}}\right] \text{ en double précision} \end{cases}$$

et on utilise la formule

$$N = (1 - 2s)\ m\ 2^e$$

Si, à l'inverse, on cherche à traduire N en format IEEE 754, on cherche e tel que

$$1 \leq \frac{|N|}{2^e} < 2$$

qui est trouvé facilement par utilisation de $\log_2$. Ensuite, m est donné par

$$m = \frac{|N|}{2^e}$$

Il suffit ensuite de transformer e sous la forme ~exposant~ et m sous la forme ~mantisse~ et d'encoder ces représentations dans le modèle IEEE 754.

Mantisse dénormalisée

— Si $e = \begin{cases} -127 & \text{en simple précision} \\ -1023 & \text{en double précision} \end{cases}$

on a alors

$$m = \sim\text{mantisse}\sim = \overset{\begin{matrix} 2^0 & 2^{-1} & \dots & 2^{n-2} & 2^{n-1} \end{matrix}}{\boxed{\begin{matrix} b_0 & b_1 & \dots & b_{n-2} & b_{n-1} \end{matrix}}} \begin{cases} \in \left[0\ ; 2 - \dfrac{1}{2^{22}}\right] & \text{en simple précision} \\ \in \left[0\ ; 2 - \dfrac{1}{2^{51}}\right] & \text{en double précision} \end{cases}$$

et on calcule N avec

$$N = (1-2s)\ m\ 2^e = \begin{cases} \pm m\ 2^{-127} \\ \pm m\ 2^{-1023} \end{cases}$$

— Si $e = 0$

on a alors

$$m = \sim\text{mantisse}\sim = \overset{\begin{matrix} 2^0 & 2^{-1} & \dots & 2^{n-2} & 2^{n-1} \end{matrix}}{\boxed{\begin{matrix} b_0 & b_1 & \dots & b_{n-2} & b_{n-1} \end{matrix}}} \begin{cases} \in \left[0\ ; 2 - \dfrac{1}{2^{22}}\right] & \text{en simple précision} \\ \in \left[0\ ; 2 - \dfrac{1}{2^{51}}\right] & \text{en double précision} \end{cases}$$

et si $m = 0 = e$, on atteint les deux représentations de 0 (dépendant du signe).

— Si $e = \begin{cases} 128 & \text{en simple précision} \\ 1024 & \text{en double précision} \end{cases}$

$$m = 0 \Rightarrow (1-2s) \cdot \infty$$
$$m \neq 0 \Rightarrow NaN \text{ (not a number)}$$

Valeurs limites

$$N_{\max} = \begin{cases} \left(2 - \dfrac{1}{2^{23}}\right) 2^{127} \approx 3{,}403 \cdot 10^{38} & \text{en simple précision} \\ \left(2 - \dfrac{1}{2^{52}}\right) 2^{1023} \approx 1{,}798 \cdot 10^{308} & \text{en double précision} \end{cases}$$

$$|N|_{\min} = \begin{cases} 2^{-22} \cdot 2^{-127} \approx 1{,}401 \cdot 10^{-45} & \text{en simple précision} \\ 2^{-51} \cdot 2^{-1023} \approx 4{,}941 \cdot 10^{-1074} & \text{en double précision} \end{cases}$$

3.3 Opération d'addition avec virgules flottantes

Soient les termes de l'addition, $|v_1| = m_1\ 2^{e_1}$ et $|v_2| = m_2\ 2^{e_2}$, tels que $|e_1| \leq |e_2|$.

i. On pose $|v_1| = m_1'\ 2^{e_2}$ (donc $m_1' = m_1\ 2^{e_1 - e_2}$) afin d'aligner les virgules. Ceci peut engendrer des erreurs, à savoir des erreurs d'arrondi ou une dénormalisation de la mantisse
ii. On complémente à deux les mantisses des nombres négatifs
iii. On additionne « bêtement » les deux mantisses
iv. Si le résultat est négatif, on le remplace par son complément à deux et on fixe le bit de signe à 1. Si ce n'est pas le cas, il est donc fixé à 0
v. On multiplie le résultat par 2^{e_2}

3.4 Multiplication avec virgules flottantes

Soient les facteurs de la multiplication, $v_1 = (1 - 2s)\ m_1\ 2^{e_1}$ et $v_2 = (1 - 2s)\ m_2\ 2^{e_2}$

i. On calcule $s =_2 s_1 + s_2$
ii. On calcule $e = e_1 + e_2$
iii. On calcule $m = m_1 m_2$
iv. On normalise $m\ 2^e$

4 Représentation binaire des caractères

4.1 Code ASCII

On code un caractère sur 7 bits en ASCII (128 valeurs). Quelques repères :

— De 0x00 à 0x1F : caractères spéciaux « invisibles » tels '\t', '\n', '\0', etc.
— De 0x20 à 0x3F : symboles mathématiques, ponctuation et chiffres. Remarquons par ailleurs que le chiffre N du système décimal se trouve en position $0x(30 + N)$
— De 0x40 à 0x7F : alphabet et caractères spéciaux d'écriture ('@', '~', ']', …). Notons ici que le code ASCII d'une minuscule est donné par l'addition de 0x20 et du code ASCII de la même lettre en majuscule. Aussi, sauf mention contraire, il est également conseillé de retenir que le caractère 'A' porte le code 0x41.

4.2 Normes ISO 5589

On code un caractère sur 8 bits en ISO. Si le premier bit est nul, les 7 bits suivants représentent un caractère ASCII. S'il vaut 1, les 7 bits suivant sont interprétés selon une norme ISO 5589 particulière dépendant des besoins de l'utilisateur.

4.3 Unicode

La norme Unicode est un autre format d'encodage de très nombreux caractères et symboles. En Unicode, un caractère est codé sur un segment de 32 bits dont 21 bits servent réellement à l'encodage. Un code Unicode est compris entre 0x00 et 0x10FFFF. Notons que, par convention, la notation $U + k$ se réfère au code $0xk$ de la norme Unicode (avec k, un nombre hexadécimal).

4.4 UTF-8

La norme UTF-8 a pour vocation d'optimiser l'encodage des caractères en leur associant des codes plus ou moins longs selon leur fréquence d'utilisation. Le nombre de bits nécessaires à l'encodage d'un caractère est donc variable.

Soit le code k où k est un nombre hexadécimal. Si $k \in$ …

— [0x00 ; 0x7F], on utilise un octet de la forme $\boxed{0\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}$ dont les 7 bits de fin sont traduits en ASCII

— [0x80 ; 0x7FF], on utilise un segment composé de deux octets et dont la forme est donnée par $\boxed{1\ 1\ 0\ b_{12}\ b_{11}\ b_{10}\ b_9\ b_8}\boxed{1\ 0\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}$

— [0x800 ; 0xFFFF], on utilise un segment composé de trois octets et dont la forme est donnée par $\boxed{1\ 1\ 1\ 0\ b_{19}\ b_{18}\ b_{17}\ b_{16}}\boxed{1\ 0\ b_{13}\ b_{12}\ b_{11}\ b_{10}\ b_9\ b_8}\boxed{1\ 0\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}$

— [0x10000 ; 0x10FFFF], on utilise un segment composé de quatre octets de la forme $\boxed{1\ 1\ 1\ 1\ 0\ b_{26}\ b_{25}\ b_{24}}\boxed{1\ 0\ b_{21}\ b_{20}\ b_{19}\ b_{18}\ b_{17}\ b_{16}}\boxed{1\ 0\ b_{13}\ b_{12}\ b_{11}\ b_{10}\ b_9\ b_8}\boxed{1\ 0\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}$

Remarquons qu'à partir de deux bytes, le nombre de chiffres 1 successifs au début du segment d'information représente le nombre de bytes utilisés. Ensuite, chaque nouveau byte faisant partie de la représentation affiche le code $\boxed{1\ 0}$ comme premiers bits.

Fonctionnement d'un ordinateur

1 Composants d'un ordinateur

1.1 Processeur/CPU (exécute)

Introduisons avant tout deux types d'architectures de processeur :

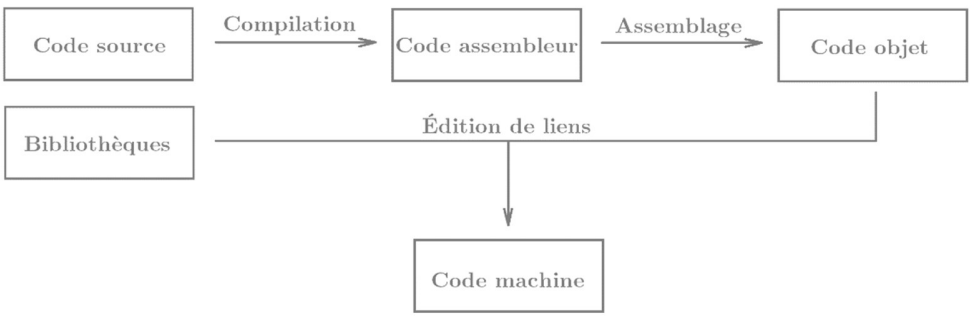
- Le modèle Von Neumann, les instructions et les données sont stockées dans un même espace de stockage commun. C'est celui utilisé par une majorité de systèmes ;
- Le modèle Harvard, on observe une séparation de la mémoire en une partie réservée aux données et en une autre réservée aux instructions.

Le processeur peut être divisé en cinq composants majeurs (ceci est bien-sûr une simplification) :

- Banque de registres, stocke de l'info à très court terme. Elle reçoit des instructions de l'unité de contrôle, à savoir lire ou écrire dans les registres. On notera également que l'instruction qui consiste à ne rien faire est un ordre que la banque de registres réceptionne ;
- Arithmetic Logic Unit/ALU, effectue des opérations arithmétiques ;
- Bus interne, porte l'information à travers un canal commun à tout le processeur ;
- Gestionnaire de communication, établit un lien avec les composants extérieurs au processeur ;
- Unité de contrôle, envoie les différents ordres aux divers composants du processeur. Ce composant possède également deux registres, IR et $PC(= RIP$ en x86-64) qui contiennent respectivement l'adresse de l'instruction courante et celle de la suivante. Le traitement d'une instruction suit la routine chargement – décodage – exécution.

Le processeur a la capacité d'exécuter une série d'instructions codées par un utilisateur. La traduction et l'évolution de l'expression de la tâche demandée peut se faire de deux manières :

- Interprétation, la tâche demandée est traduite et effectuée étape par étape par le processeur ;
- Compilation, la tâche est comprise dans son ensemble par le processeur et, après traitement, est effectuée d'une traite. Cette méthode demande cependant un traitement de l'ensemble des instructions qui est représentée ci-dessous.



1.2 Mémoire (stocke)

Il existe différents types de stockage. On notera

- La mémoire vive (RAM), très flexible et rapide, son contenu est cependant réinitialisé lors d'une mise hors tension ;
- La mémoire morte (ROM), employée pour stocker de l'information même hors tension mais qui n'est pas destinée à être modifiée (ou très faiblement) ;
- La mémoire de masse, mémoire modifiable et dont le contenu peut être préservé même hors tension. Il existe deux tels types de mémoires largement répandus : le disque dur, composé d'un empilement de disques, et la mémoire flash, qui consiste en un circuit électronique permettant de stocker l'information.

1.3 Générateur d'horloge (rythme)

Le générateur d'horloge dicte le rythme dans tout l'ordinateur afin de synchroniser tous les composants.

1.4 Périphériques (diversifient)

Les périphériques sont divers en nature. En effet, on retrouve dans cette section les cartes graphique, réseau, son, etc.

1.5 Contrôleurs (aiguillent)

Les contrôleurs permettent d'aiguiller le flux d'information vers les composants visés.

1.6 Bus (transfert)

Le bus est un canal de communication mis en commun pour les différents composants. Vu sa nature, les composants sont obligés de partager ce canal et, quand une transaction est déjà en cours, doivent attendre pour pouvoir l'utiliser.

2 Conventions de la mémoire

2.1 Composition

La mémoire d'un ordinateur peut être vue comme un ensemble de cases contenant, en général, 8 octets d'information chacune. Ces cases sont identifiées par une adresse allant de 0 à $2^{64} - 1$.

2.2 Alignement

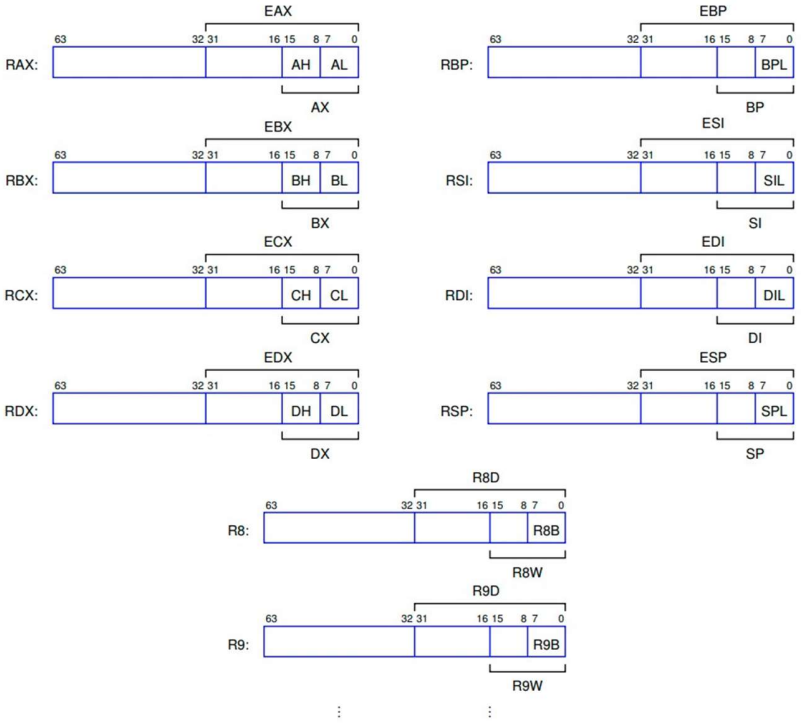
Le stockage de l'information dans la mémoire fonctionne en utilisant le principe de l'alignement. En effet, lors du stockage d'une information de n octets on préfèrera optimiser le système de stockage en plaçant cette information dans un espace dont l'adresse est un multiple de n.

2.3 Disposition des bits

Enfin, dans l'architecture dans laquelle nous travaillerons, l'information sera stockée en encodage petit-boutiste. C'est-à-dire que les bits de poids le plus faible sont stockés aux adresses les plus faibles. L'encodage grand-boutiste s'effectue dans l'ordre inverse.

3 Utilisation des registres

Nous serons amenés à utiliser différents registres de la banque de registres. Les registres principaux sont au nombre de 16, à savoir *RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8, R9, ..., R15*. Voici une représentation de ces registres :



On notera que de nombreuses conventions sont à respecter lors de l'utilisation de ces registres. Ces restrictions seront explicitées par la suite.

Cette image ne reprend pas le registre *RFLAGS* qui contient les drapeaux CF, ZF, SF et OF. Quand ces drapeaux sont actualisés par une instruction, ils permettent d'obtenir des informations quant au déroulement de celle-ci :

- *CF (Carry Flag)* indique quand une opération sur des nombres de n bits a effectué un report à la position n ;
- *ZF (Zero Flag)* indique quand une opération a fourni un résultat nul ;
- *SF (Sign Flag)* correspond au bit de poids fort du résultat d'une opération ;
- *OF (Overflow Flag)* signale un dépassement arithmétique lorsque l'on traite avec des données signées.

Code assembleur

1 Modes d'adressage

1.1 Adressage registre

Le simple fait de nommer un registre fait appel à son contenu. Par exemple, additionner RAX et RBX revient à additionner les valeurs contenues dans ces registres.

1.2 Adressage immédiat

L'utilisation d'une constante sans référence à l'espace mémoire est parfois autorisé.

1.3 Adressage direct

On extrait une valeur de taille donnée à partir d'une adresse constante de la mémoire. On notera dans ce cas `<taille> ptr [<constante>]`. Remarquons que `<taille>` peut prendre différentes valeurs, à savoir

- `byte` pour 8 bits
- `word` pour 16 bits
- `dword` pour 32 bits
- `qword` pour 64 bits

1.4 Adressage indirect

Ce mode fonctionne similairement à l'adressage direct mais, cette fois-ci, l'adresse ne doit plus être passée par une constante mais peut l'être via la valeur contenue dans un registre. On notera donc `<taille> ptr [<registre>]` où `<taille>` peut prendre les valeurs reprises ci-dessus.

1.5 Adressage indexé

Ce mode d'adressage est très similaire aux deux précédents mais permet cette fois-ci d'effectuer une opération entre valeurs constantes et valeurs contenues dans des registres lors de la détermination de l'adresse. On note `<taille> ptr [<base> + <facteur> * <index> + <déplacement>]` où `<taille>` fonctionne comme cela a été expliqué plus tôt. Cette expression peut certes paraitre un peu complexe mais voici comment l'utiliser :

- `<base>` et `<index>` sont des registres de 64 bits dont on extrait la valeur
- `<facteur>` vaut 1, 2, 4, ou 8
- `<déplacement>` est une constante signée sur 32 bits

L'opération contenue dans les crochets droits fournira alors l'adresse à laquelle nous désirons faire référence.

2 Gestion des piles

Le code assembleur possède un système de gestion de piles. Une pile est un moyen de stocker des données selon la politique LIFO d'une manière homologue à celle évoquée lors du cours d'introduction à l'informatique. Une pile servira à stocker des adresses, des valeurs temporaires, des arguments passés à des fonctions, etc. Notons que l'empilement se fait dans l'ordre décroissant des adresses et que l'adresse de l'élément situé tout en haut de la pile (dernier ajout) est stocké dans le registre *RSP*. De plus amples informations détaillant son utilisation et sa structure seront fournies à la fin de cette synthèse.

3 Instructions en assembleur

3.1 Convention d'écriture

En code assembleur, une instruction est composée

- d'une mnémonique, nom conventionnel de l'opération à effectuer ;
- d'opérandes, les potentiels éléments que l'on fournit en entrée à l'opération.

On notera en pratique

```
<mnémonique> <opérande 1>, <opérande 2>, …
```

Au vu de la proximité du langage assembleur avec le processeur, ses conventions d'utilisation dépendent de nombreuses variables comme l'architecture, les outils utilisés et le système d'exploitation.

Remarquons enfin qu'il est souvent nécessaire que les deux potentiels opérandes associés à une mnémonique soient de même taille. Aussi, il est judicieux de noter que tout mode d'adressage n'est pas toujours permis selon l'instruction utilisée.

3.2 Instructions de gestion de données

- `MOV <op1>, <op2>` qui place la valeur représentée par `<op2>` dans `<op1>`
- `XCHG <op1>, <op2>` qui échange les valeurs représentées par `<op1>` et `<op2>`

3.3 Instructions arithmétiques

- `ADD <op1>, <op2>` qui place le résultat de la somme des valeurs représentées par `<op1>` et `<op2>` dans `<op1>`. Les drapeaux CF, ZF, SF et OF sont tous affectés
- `SUB <op1>, <op2>` qui effectue la même chose que `ADD` mais dans le cas d'une soustraction
- `CMP <op1>, <op2>` qui compare `<op1>` et `<op2>` en mettant à jour les drapeaux CF, ZF, SF et OF. En analysant la valeur de ces drapeaux, il est ensuite possible de tirer des conclusions quant à la comparaison et de l'exploiter (dans le cas d'un saut conditionnel par exemple)

— `INC <op1>` qui incrémente la valeur représentée par `<op1>` d'une unité en mettant à jour les drapeaux ZF, SF et OF

— `DEC <op1>` qui effectue la même chose mais en décrémentant d'une unité

— `MUL <op1>` qui, sous l'hypothèse que ces nombres ne soient pas signés, multiplie `<op1>` (de taille n) par `<facteur>` et stocke le résultat dans `<produit>` où ces deux derniers termes diffèrent selon la valeur de n :

Valeur de n	`<facteur>`	`<produit>`
8 bits	AL	AX
16 bits	AX	AX et DX
32 bits	EAX	EDX et EAX
64 bits	RAX	RDX et RAX

Cette opération affecte les drapeaux CF et OF

— `IMUL <op1>` qui effectue la même chose mais sous l'hypothèse que `<op1>` et `<facteur>` soient signés

3.4 Instructions logiques

— `AND <op1>, <op2>` qui crée une suite de bits et la stocke dans `<op1>`. Cette suite est composée en analysant les bits de même poids des deux opérandes en suivant la table de vérité du « ET logique » :

Bit de poids N d'`<op1>`	Bit de poids N d'`<op2>`	Bit de poids N placé dans `<op1>`
0	0	0
0	1	0
1	0	0
1	1	1

Cette opération affecte les drapeaux CF, OF, ZF et SF (les deux premiers étant tous deux mis à zéro) ;

— `OR <op1>, <op2>` qui effectue la même chose mais en suivant la table de vérité du « OU inclusif logique » :

Bit de poids N d'`<op1>`	Bit de poids N d'`<op2>`	Bit de poids N placé dans `<op1>`
0	0	0
0	1	1
1	0	1
1	1	1

— `XOR <op1>, <op2>` qui effectue la même chose mais en suivant la table de vérité du « OU exclusif logique » :

Bit de poids N d'`<op1>`	Bit de poids N d'`<op2>`	Bit de poids N placé dans `<op1>`
0	0	0
0	1	1
1	0	1
1	1	0

— `NOT <op1>` qui complémente `<op1>` à un tout en n'affectant aucun drapeau

3.5 Instructions de gestion de pile

— `PUSH <op1>` qui place la valeur de 64 bits représentée par `<op1>` en haut de la pile et décrémente *RSP* de 8 unités afin que ce dernier reste à jour

— `POP <op1>` qui place la valeur du haut de la pile dans `<op1>` et incrémente *RSP* de 8 unités afin que ce dernier reste à jour

3.6 Instructions de contrôle

— `JMP <op1>` qui place `<op1>` dans le registre *PC* (*RIP* en x86-64) et fait donc de la prochaine opération à exécuter par le processeur celle dont l'adresse est donnée par `<op1>`. Pour cette instruction, on utilisera souvent le principe d'étiquette pour déterminer `<op1>`. Une étiquette fait référence à un endroit particulier du code (“ `<étiquette> : <code>` ” ou “ `.equ <étquette>, <adresse>` ”). Il est également possible d'effectuer un saut conditionnel dépendant de l'état des flags (premier tableau) ou dépendant sur le résultat d'une comparaison entre deux opérandes (via l'instruction `CMP`) qui précède directement l'instruction (deuxième tableau) :

Mnémonique	Condition
`JC`	`CF` $= 1$
`JNC`	`CF` $= 0$
`JZ`	`ZF` $= 1$
`JNZ`	`ZF` $= 0$
`JS`	`SF` $= 1$
`JNS`	`SF` $= 0$
`JO`	`OF` $= 1$
`JNO`	`OF` $= 0$

Mnémonique	Condition (par rapport au `CMP`)
`JE`	`<op1>` $=$ `<op2>`
`JNE`	`<op1>` $\neq$ `<op2>`
`JG`	`<op1>` $>$ `<op2>` (valeurs signées)
`JGE`	`<op1>` $\geq$ `<op2>` (valeurs signées)
`JL`	`<op1>` $<$ `<op2>` (valeurs signées)
`JLE`	`<op1>` $\leq$ `<op2>` (valeurs signées)
`JA`	`<op1>` $>$ `<op2>` (valeurs non signées)
`JAE`	`<op1>` $\geq$ `<op2>` (valeurs non signées)
`JB`	`<op1>` $<$ `<op2>` (valeurs non signées)
`JBE`	`<op1>` $\leq$ `<op2>` (valeurs non signées)

— `LOOP <op1>` qui effectue un " `DEC RCX` " suivi, à condition que $RCX \neq 0$, d'un "`JMP <op1>`"
— `CALL <op1>` qui fait appel à une fonction dont la première instruction est stockée à l'adresse représentée par `<op1>`. À cette fin, on stocke la valeur stockée dans PC (RIP en x86-64) dans une pile puis on effectue un `JMP <op1>`. Le stockage de la valeur de PC dans la pile prend tout son sens car l'opération `JMP <op1>` altère la valeur de PC
— `RET` qui quitte la fonction courante et retourne au code appelant. Pour ce faire, on effectue un `JMP` vers l'adresse qui avait été empilée par le précédent `CALL` tout la dépilant

4 Forçage

Le forçage permet de modifier une chaine de bits d'une manière donnée en utilisant les instructions logiques.

4.1 Forçage par `AND`

Le forçage par `AND` permet de forcer à 0 des bits de poids choisis dans une suite de bits. Pour ce faire, on passe en deuxième opérande de l'instruction `AND` une suite de bits de même taille celle que l'on souhaite forcer. Ce deuxième opérande est composé de 0 aux bits de poids correspondant à ceux que l'on souhaite forcer à 0 dans le premier opérande. Le reste des bits du deuxième opérande sont mis égaux à 1. En pratique, on a de manière imagée pour une suite de n bits

	$b_{n-1}\ b_{n-2}\ \ldots\ b_i\ \ldots\ b_1\ b_0$
AND	$1\quad 1\quad \ldots\ 0\ \ldots\ 1\ 1$
	$b_{n-1}\ b_{n-2}\ \ldots\ 0\ \ldots\ b_1\ b_0$

4.2 Forçage par `OR`

Le forçage par `OR` effectue l'opération inverse. C'est-à-dire que les bits égaux à 1 dans le deuxième opérande forcent à 1 les bits de poids correspondant dans le premier opérande. Les bits nuls n'ont en revanche aucun effet. De manière imagée

	$b_{n-1}\ b_{n-2}\ \ldots\ b_i\ \ldots\ b_1\ b_0$
OR	$0\quad 0\quad \ldots\ 1\ \ldots\ 0\ 0$
	$b_{n-1}\ b_{n-2}\ \ldots\ 1\ \ldots\ b_1\ b_0$

4.3 Forçage par `XOR`

Le forçage par `XOR` complémente à un les bits du premier opérande qui sont du même poids que les bits égaux à 1 dans le deuxième opérande. Le reste des bits restent inchangés. De manière imagée

	$b_{n-1}\ b_{n-2}\ \ldots\ b_i\ \ldots\ b_1\ b_0$
XOR	$0\ \ 0\ \ldots\ 1\ \ldots\ 0\ \ 0$
	$b_{n-1}\ b_{n-2}\ \ldots\ 1-b_i\ \ldots\ b_1\ b_0$

4.4 Forçage par `NOT`

Le forçage par `NOT` complémente simplement une suite de bits à un.

5 Segment de données

Il est possible d'inscrire en mémoire différents types de données en utilisant les commandes détaillées plus bas. Il est possible de récupérer l'adresse d'un segment de données en plaçant une étiquette référençant sa création.

- `.data` qui indique le début d'un segment de données
- `.byte` ou `.word` ou `.int` ou `.quad` qui, suivi d'un entier constant, stocke l'entier sur 1, 2, 4, ou 8 bytes respectivement
- `.fill <répétition>, <taille>, <valeur>` qui remplit un segment de données de `<répétition>` fois successives la valeur `<valeur>` encodée sur `<taille>` bits
- `.ascii <chaine de caractères>` qui stocke la chaine de caractères dans le segment de données
- `.asciiz <chaine de caractères>` qui stocke la chaine de caractères suivie du symbole terminateur (‘\0’) dans le segment de données
- `.balign <valeur>` qui fixe l'adresse courante du segment à un multiple de `<valeur>`

6 Directives de compilation

D'autres instructions commençant également par un point peuvent, placées en début de code, spécifier au processeur des informations concernant le code que nous écrivons. Les quatre commandes qui suivent sont souvent utilisées dans cet ordre en début de code. Aussi, un segment de données peut se définir entre la première et la deuxième de ces instructions :

- `.intel_syntax noprefix` qui indique la variante syntaxique que nous utiliserons dans le code. Ici, nous utilisons la variante « noprefix »
- `.text` qui indique que le nous désirons écrire du code
- `.global <étiquette>` qui permet de faire appel à l'étiquette depuis un autre programme. En C, on utilisera `extern <type> <étiquette>` où `<type>` fait référence à la valeur retournée
- `.type <étiquette>, @fuction` qui définit l'étiquette comme faisant référence à une fonction

7 Précision concernant les étiquettes

Les étiquettes sont des repères qui peuvent permettre d'effectuer des sauts d'une instruction à l'autre dans le code assembleur ou de façon plus rudimentaire de simplement stocker l'adresse d'une instruction ou d'agir comme une variable. Cependant il y a un détail auquel nous nous devons de prêter attention.

```
<étiquette> :   <instruction1>
                <instruction2>
                JMP <étiquette>
```

Dans cet exemple imagé de code, l'instruction de saut n'a pas réellement la valeur de l'adresse d'`<instruction1>` pour opérande mais bien la valeur du déplacement qu'il faut effectuer dans la mémoire pour passer de l'instruction actuelle à celle que l'on référence avec `<étiquette>`. Si l'on désire avoir accès à la véritable adresse absolue d'`<instruction1>`, pour la stocker par exemple, on utilisera alors le mot-clef `offset_flat : <étiquette>` qui fournit l'adresse absolue à laquelle l'étiquette fait référence.

8 Appel de fonction/routine

8.1 Propriétés de la fonction

Une fonction doit pouvoir

- admettre des arguments
- allouer de l'espace mémoire pour de potentielles variables locales
- retourner une valeur au code appelant

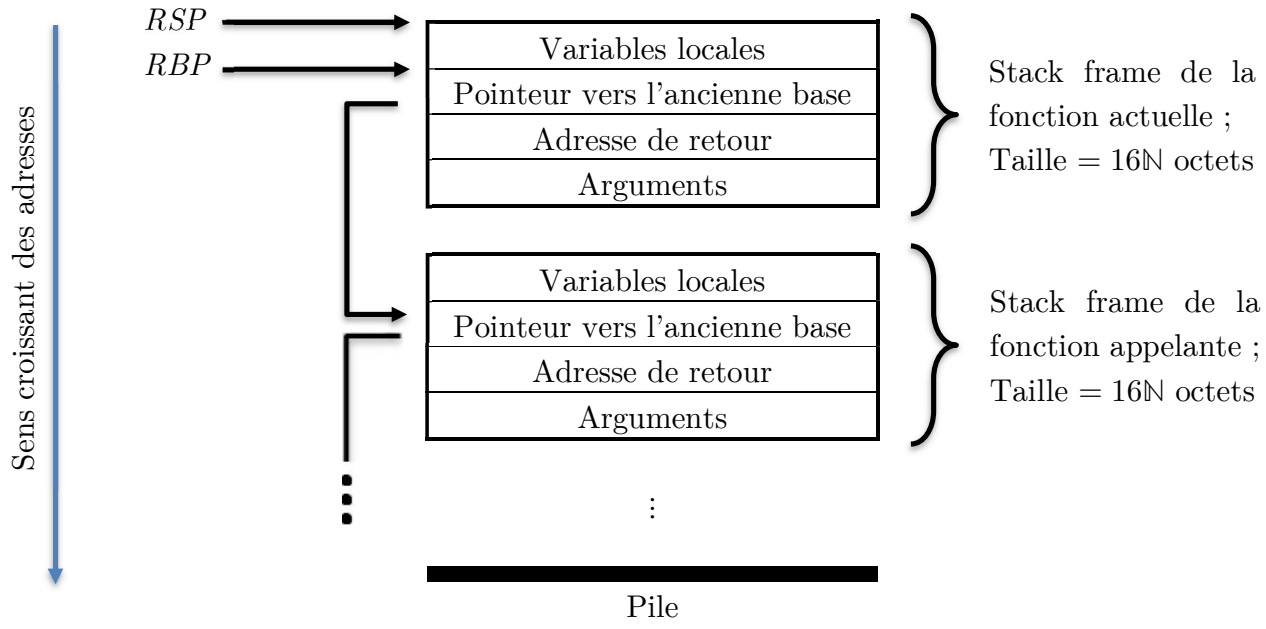
8.2 Conventions d'appel de fonction

Afin de savoir comment toutes ces opérations peuvent être réalisées, nous établissons les conventions d'appel de fonction :

- Les six potentiels premiers arguments sont passés par l'intermédiaire des registres *RDI*, *RSI*, *RDX*, *RCX*, *R8* et *R9* (dans cet ordre)
- Les potentiels arguments supplémentaires sont empilés dans l'ordre inverse de l'ordre dans lequel ils sont passés. Cela permettra notamment de les dépiler dans l'ordre dans lequel ils ont été passés
- La potentielle valeur de retour d'une fonction est placée dans le registre *RAX*
- Il est demandé à la fonction de préserver les registres *RBX*, *RBP*, *R12*, *R13*, *R14* et *R15*
- *RSP* (qui est le pointeur de pile) doit être un multiple de 16 avant l'appel de la fonction

8.3 Structure de pile

Nous venons de le voir, les appels de fonction suivent des conventions qui font quelques fois appel à l'utilisation de pile. Cette utilisation de pile jouit également de règles conventionnelles. Voici la structure de la pile lors d'appels de fonctions :



Étant donné cette organisation, nous pouvons extraire quelques propriétés :

- on peut accéder à la $k^{\text{ème}}$ variable locale avec " `qword ptr[RBP - 8 * k]` "
- le $n^{\text{ème}}$ argument passé à la fonction et qui est stocké dans la pile (donc en réalité le $(n+6)^{\text{ème}}$ argument de la fonction) est accessible avec `qword ptr [RBP + 8 * n + 8]`

Note : Les variables locales, pointeurs vers la base précédente, adresses de retour, arguments sont tous stockés sur 64 bits chacun.