
Organisation des ordinateurs

1. Le traitement de l'information

1.1. Qu'est-ce qu'un ordinateur ?

Un ordinateur est une machine capable de traiter des données, en suivant un programme préétabli.

1.2. La notion d'information

On appelle information, la connaissance qu'un observateur possède de l'état d'un système. L'information se transmet par l'intermédiaire de signaux, qui peuvent prendre des formes variées.

1.3. L'encodage de l'information

1.3.1. Les signaux continus

Un signal continu est un signal qui prend ses valeurs dans un domaine dense.

Ce qui n'est pas bien : l'information n'est pas transmise, émise et reçue avec fiabilité car la valeur de chaque signal est entachée d'imprécisions.

1.3.2. Les signaux discrets

Un signal discret est un signal ne possédant qu'un nombre fini de valeurs nominales.

Ce qui est bien : la transmission fiable de données est possible malgré la présence d'imprécisions. En effet, si l'amplitude des perturbations est suffisamment petite, alors les valeurs transmises peuvent toujours être correctement identifiées à leur réception.

a) Les signaux discrets binaires

Dans les ordinateurs modernes, l'information est transmise, traitée et mémorisée au moyen de signaux discrets binaires, c'est-à-dire possédant deux valeurs nominales.

Ce qui est bien : ils sont faciles à générer et à décoder, il présentent une bonne robustesse face aux perturbations et leur analyse est simple grâce à l'algèbre booléenne.

1.4. La quantité d'information

Pour pouvoir quantifier la quantité d'information transmise par un signal discret, diverses propriétés sont souhaitées :

- + la probabilité de recevoir une valeur est faible, + la quantité d'information est élevée
- Lorsqu'on combine des signaux indépendants, l'information doit s'additionner

La quantité d'information (en bits) transmise par une valeur discrète décodable de façon fiable est égale à $\log_2 \frac{1}{p}$, avec p = probabilité que cette valeur soit reçue.

Pour signaux binaires dont les deux valeurs sont équiprobables, la quantité d'information est de :

$$\log_2 \frac{1}{\frac{1}{2}} = \log_2 2 = 1.$$

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

De façon générale, si un signal possède N valeurs équiprobables, on a : $\log_2 \frac{1}{N} = \log_2 N$ bits, avec le bit = quantité d'information permettant de distinguer avec fiabilité deux valeurs équiprobables.

Les différentes unités employées : un octet (byte, B) = 8 bits d'information / un nibble = $\frac{1}{2}$ octet.

Les préfixes : K (kilo), M (mega), G (giga), T (tera), P (peta), ... ont une signification par rapport au contexte :

- En informatique : $2^{10}, 2^{20}, 2^{30}, 2^{40}, 2^{50}, \dots$ par approximation ($2^{10} = 1024 \approx 1000$)
- Par les fabricants : $10^3, 10^6, 10^9, 10^{12}, 10^{15} \dots \rightarrow$ erreur sur ce qui est annoncé

2. La représentation des données

2.1. Introduction

Les ordinateurs représentent l'information à l'aide de signaux discrets binaires. Par convention, les deux valeurs nominales de ces signaux sont notées 0 et 1 (d'autres notations sont possibles : T/F).

2.2. Les nombres entiers non signés (positifs)

2.2.1. La notation positionnelle

Pour représenter un entier non négatif à l'aide des seuls symboles 0 et 1, on peut utiliser un procédé que nous employons dans notre vie pour les nombres.

Principes :

- Chaque symbole de l'écriture du nombre est un chiffre $\in \{0,1,2,3,4,5,6,7,8,9\}$.
- Le poids de chaque chiffre est une puissance de 10 qui dépend de sa position.

Exemple pour mieux comprendre : la position se calcule de la droite vers la gauche.

La notation positionnelle se généralise à n'importe quelle base $r > 1$:

$$\begin{array}{r} \text{poids: } 10^2 \quad 10^1 \quad 10^0 \\ \text{position: } 2 \quad 1 \quad 0 \\ \boxed{1 \quad 2 \quad 3} \\ 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ = 1 \times 100 + 2 \times 10 + 3 \times 1 \\ = 123. \end{array}$$

- Les chiffres appartiennent à l'ensemble $\{0,1,\dots,r-1\}$.
- Le poids du chiffre situé à une position k de l'écriture d'un nombre est égal à r^k .

Exemple pour mieux comprendre (base 2) :

$$\begin{array}{r} \text{poids: } 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0 \\ \text{position: } 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \\ \boxed{1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1} \\ 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ = 64 + 32 + 16 + 8 + 2 + 1 \\ = 123. \end{array}$$

Le nombre encodé par la suite de bits $b_{n-1} b_{n-2} \dots b_0$, avec $n > 0$, est égal à : $\sum_{i=0}^{n-1} 2^i b_i$

Les bits b_{n-1} = bits de poids forts et b_0 = bits de poids faible. Pour déterminer la suite de bits qui représente un nombre donné, on procède de la façon suivante :

Propriétés :

- Le bit de poids faible = 0 (v pair) et = 1 (v impair).
- En supprimant le bit de poids faible d'une représentation de v , on a : $v = \left\lfloor \frac{v}{2} \right\rfloor$

L'algorithme est donc :

- Si v est pair \rightarrow afficher 0, sinon afficher 1.
- Remplacer v par $\left\lfloor \frac{v}{2} \right\rfloor$
- Si $v \neq 0$, recommencer l'étape 1.

Remarques :

- Cet algorithme génère les bits de la représentation de v en commençant par le bit de poids faible (c'est-à-dire de la droite vers la gauche).

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

- La suite de bits obtenue constitue la représentation la plus courte du nombre v . Des représentations plus longues s'obtiennent en préfixant le résultat d'un nombre quelconque de zéros de tête.

Exemple pour mieux comprendre :

Exemple: Représentation du nombre 123:

$v = 123$	impair	→ 1
$v = 61$	impair	→ 1
$v = 30$	pair	→ 0
$v = 15$	impair	→ 1
$v = 7$	impair	→ 1
$v = 3$	impair	→ 1
$v = 1$	impair	→ 1
$v = 0$.		

L'algorithme de calcul de la représentation d'un nombre v s'arrête après avoir produit n bits ou moins ssi $v < 2^n$.

Les nombres avec représentation binaire non signée sur n bits forment donc l'intervalle $[0, \dots, 2^n - 1]$

La représentation obtenue est donc **1111011**, à laquelle il est permis d'ajouter un nombre arbitraire de zéros de tête.

2.2.2. La représentation hexadécimale

En choisissant $r = 16$, on obtient la représentation hexadécimale. Elle est lisible et très facile à convertir vers et depuis la notation binaire. Un chiffre hexadécimal peut prendre 16 valeurs : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Exemple pour mieux comprendre :

$$\begin{array}{c} \text{poids: } 16^2 \quad 16^1 \quad 16^0 \\ \text{position: } 2 \quad 1 \quad 0 \\ \boxed{4 \quad | \quad 0 \quad 2} \\ 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0 \\ = 4 \times 256 + 13 \times 16 + 2 \times 1 \\ = 1234. \end{array}$$

2.2.2.1. Conversion hexadécimale ↔ binaire

Un chiffre hexadécimal = 4 bits d'information.

Il faut remplacer chaque chiffre par une séquence de 4 bits qui lui correspond.

Table de conversion :

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

2.2.3. Les opérations sur les entiers non signés

2.2.3.1. L'addition

On additionne un par un les bits de la représentation des opérandes, du bit de poids faible vers le bit de poids fort. Si le résultat est supérieur à 1, un report apparaît (encadré dans l'exemple : $123 + 456 = 579$).

$$\begin{array}{r} \boxed{1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1} \\ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ + 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \\ \hline 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \end{array}$$

2.2.3.2. La multiplication

Exemple : $34 \cdot 12 = 408$

$$\begin{array}{r} \times \quad 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \\ \quad 1 \ 1 \ 0 \ 0 \\ \hline \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + \quad 1 \ 0 \ 0 \ 0 \ 1 \ 0 \\ \hline 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \end{array}$$

2.3. Les nombres entiers signés (positifs et négatifs)

Il existe plusieurs façons de représenter en binaire les nombres entiers positifs ou négatifs. Les 3 représentations présentent des caractéristiques identiques : le bit de poids fort = bit de signe (0 pour nombres > 0 et 1 pour nombres < 0 et si la représentation commence par un bit de signe égal à 0, elle est égale à la représentation binaire non signée du même nombre.

2.3.1. La représentation par valeur signée

Principe : à la suite du bit de signe, on place la représentation binaire non signée de la valeur absolue du nombre représenté.

Exemple pour mieux comprendre : représentation sur 8 bits de $-42 = 10101010$.

Ce nombre est négatif car le bit de signe est 1 et sa représentation binaire sur 7 bits de $42 = 0101010$.

Par la représentation par valeur signée, un nombre v est représenté par : $v = (1 - 2b_{n-1}) \sum_{i=0}^{n-2} 2^i b_i$

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

Pour déterminer le nombre v , il y a deux cas à considérer :

- Si $b_{n-1} = 0$: on a $\sum_{i=0}^{n-2} 2^i b_i$
- Si $b_{n-1} = 1$: on a $-\sum_{i=0}^{n-2} 2^i b_i$

Avec n bits, la représentation par valeur permet d'encoder tous les nombres :

- Dans l'intervalle $[0 ; 2^{n-1} - 1]$ avec un bit de signe = 0
- Dans l'intervalle $[-2^{n-1} + 1 ; 0]$ avec un bit de signe égal à 1.
→ $[-2^{n-1} + 1 ; 2^{n-1} - 1]$

2.3.2. La représentation par complément à un

Principe : lorsque le bit de signe est = 1, on complémente l'encodage et donc on y remplace les bits = 0 par 1 et vice-versa.

Exemple : sur 8 bits de $-42 = 11010101$. Bit de signe = 1 car négatif. La représentation binaire non signée sur 7 bits de $42 = |-42|$ est 0101010, dont le complément est 1010101.

Pour déterminer le nombre v , il y a deux cas à considérer :

- Si $b_{n-1} = 0$: on a $\sum_{i=0}^{n-2} 2^i b_i = \sum_{i=0}^{n-1} 2^i b_i$
- Si $b_{n-1} = 1$: pour obtenir la représentation non signée sur $n-1$ bits de $|v|$, il faut complémer chaque bit b_i , pour $i \in [0 ; n - 2]$, càd remplacer b_i par $1 - b_i$: on a

On garde le même intervalle qu'à valeur signée.

$$\begin{aligned} &= \sum_{i=0}^{n-2} 2^i (1 - b_i) \\ &= \sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} 2^i b_i \\ &= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i b_i \\ &= 2^n - 1 - \sum_{i=0}^{n-1} 2^i b_i. \end{aligned}$$

2.3.2.1. Addition de deux nombres entiers signés

Pour appliquer l'addition, on va réutiliser l'algorithme pour la représentation non signée. Soit $w = b_{n-1}, \dots, b_0$ une suite de n bits avec $n \geq 1$. Le nombre non signé est $[w]_{ns}$ et le nombre par complément à un est $[w]_{c1}$.

Propriétés :

- Si $b_{n-1} = 0$, alors $[w]_{ns} = w_{c1}$
- Si $b_{n-1} = 1$, alors $[w]_{ns} = w_{c1} + 2^n - 1$

En d'autres termes, les représentations non signée et par complément à un sont égales à un certain décalage près, qui dépend du bit de signe.

Somme = $w + w' = w''$:

- Si pas de report en n : $[w'']_{ns} = [w]_{ns} + [w']_{ns}$
- Si report en n (et a été ignoré) : $[w'']_{ns} = [w]_{ns} + [w']_{ns} - 2^n$

Pour déterminer le rapport entre $[w]_{c1}$, $[w']_{c1}$ et $[w'']_{c1}$, il faut envisager plusieurs cas :

1. $b_{n-1} = 0$, $b'_{n-1} = 0$ et $b''_{n-1} = 0$
 $[w]_{ns} = [w]_{c1}$, $[w']_{ns} = [w']_{c1}$ et $[w'']_{ns} = [w'']_{c1}$
 On a : $[w'']_{c1} = [w'']_{ns}$
 $= [w]_{ns} + [w']_{ns}$
 $= [w]_{c1} + [w']_{c1}$.
2. $b_{n-1} = 0 / b'_{n-1} = 0 / b''_{n-1} = 1$
 $[w]_{ns} = [w]_{c1} \geq 0$, $[w']_{ns} = [w']_{c1} \geq 0$ et $[w]_{ns} + [w']_{ns} \geq 2^{n-1}$ car $b''_{n-1} = 1$
 Report en n , la somme n'est pas représentable par complément à un sur n bits
 → Dépassement arithmétique
3. $b_{n-1} = 0 / b'_{n-1} = 1 / b''_{n-1} = 0$
 $[w]_{ns} = [w]_{c1} / [w']_{ns} = [w']_{c1} + 2^n - 1 / [w'']_{ns} = [w'']_{c1}$ → Report en n
 $[w'']_{c1} = [w'']_{ns}$
 $= [w]_{ns} + [w']_{ns} - 2^n$
 $= [w]_{c1} + [w']_{c1} - 1$.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

$$4. \quad b_{n-1} = 0 / b'_{n-1} = 1 / b''_{n-1} = 1$$

- $[w]_{ns} = [w]_{c1} / [w']_{ns} = [w']_{c1} + 2^n - 1 / [w'']_{ns} = [w'']_{c1} + 2^n - 1$

$$\begin{aligned}[w'']_{c1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^n + 1 \\ &= [w]_{c1} + [w']_{c1}.\end{aligned}$$

$$5. \quad b_{n-1} = 1 / b'_{n-1} = 0 / b''_{n-1} = 0$$

- Idem que 3 en permuttant les deux opérations w et w'

$$\begin{aligned}[w'']_{c1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^n + 1 \\ &= [w]_{c1} + [w']_{c1}.\end{aligned}$$

$$6. \quad b_{n-1} = 1 / b'_{n-1} = 0 / b''_{n-1} = 1$$

- Idem que le 4

$$7. \quad b_{n-1} = 1 / b'_{n-1} = 1 / b''_{n-1} = 0$$

- $[w]_{ns} = [w]_{c1} + 2^n - 1$ et $[w]_{c1} \leq 0 / [w']_{ns} = [w']_{c1} + 2^n - 1$ et
 $[w']_{c1} \leq 0 / [w'']_{ns} = [w'']_{c1}$ et $[w'']_{c1} \geq 0 \rightarrow$ Report en n et dép. arith.

On a $[w]_{ns} + [w']_{ns} \leq 2^n + 2^{n-1} - 1$.

Or $[w]_{ns} + [w']_{ns} = [w]_{c1} + [w']_{c1} + 2^{n+1} - 2$.

Donc,

$$\begin{aligned}[w]_{c1} + [w']_{c1} &\leq 2^n + 2^{n-1} - 1 - 2^{n+1} + 2 \\ &= -2^{n-1} + 1.\end{aligned}$$

La somme $[w]_{c1} + [w']_{c1}$ n'est donc pas représentable sur n bits, sauf dans le cas particulier où elle vaut $-2^{n-1} + 1$.

$$8. \quad b_{n-1} = 1 / b'_{n-1} = 1 / b''_{n-1} = 1$$

- $[w]_{ns} = [w]_{c1} + 2^n - 1 / [w']_{ns} = [w']_{c1} + 2^n - 1 / [w'']_{ns} = [w'']_{c1} + 2^n - 1$
 \rightarrow Report en n

$$\begin{aligned}[w'']_{c1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^{n+1} + 1 \\ &= [w]_{c1} + [w']_{c1} - 1.\end{aligned}$$

Résumé des 8 cas :

- 1) Si les deux opérandes présentent le même bit de signe, et si celui-ci diffère de celui du résultat, alors il y a dépassement arithmétique.
- 2) Sinon :
 - Si pas de report en n, le résultat est correct.
 - Si report, le résultat est trop petit d'une unité.

2.3.3. La représentation par complément à deux

Idée : on décale la représentation des nombres négatifs d'une unité par rapport au complément à un, pour ne plus avoir de terme correctif lors des additions.

Principe : la représentation d'un nombre v sur n bits est égale à

- La représentation non signée de v sur n bits si $v \geq 0$,
- La représentation par complément à un de $v+1$ sur n bits si $v < 0$. (Si $v+1$ est nul, on prend représentation négative).

Exemple pour mieux comprendre : représentation sur 8 bits de $-42 = 11010110$.

Ce nombre est négatif donc la représentation est celle de -41 par complément à un, qui est 11010110 .

Le nombre v représenté vaut $-2^n b_{n-1} + \sum_{i=0}^{n-1} 2^i b_i$.

En effet :

- Si $b_{n-1} = 0$: on a $\sum_{i=0}^{n-1} 2^i b_i$ égale à non signée
- Si $b_{n-1} = 1$: on a la représentation de $v+1$ par complément à un.

On a donc :

$$v + 1 = -2^n + 1 + \sum_{i=0}^{n-1} 2^i b_i$$

$$v = -2^n + \sum_{i=0}^{n-1} 2^i b_i.$$

Avec n bits, la représentation par complément à deux permet d'encoder tous les nombres dans l'intervalle $[-2^{n-1} + 1 ; 2^{n-1} - 1]$.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

2.3.3.1. L'arithmétique par complément à deux

Notation : $[w]_{C2}$ = représentation par complément à deux.

Propriété :

- Si $b_{n-1} = 0$: $[w]_{ns} = [w]_{C2}$
- Si $b_{n-1} = 1$: $[w]_{ns} = [w]_{C2} + 2^n$.
 $\rightarrow [w]_{ns} =_2 [w]_{C2}$ où $=_k$ désigne l'égalité modulo k.

2.3.3.2. Quelques propriétés utiles

- ✓ On peut étendre la représentation d'un nombre vers davantage de bits en répétant le bit de signe. $w = b_{n-1}b_{n-2}\dots b_0$
 - Si $b_{n-1} = 0$: la propriété est évidente
 - Si $b_{n-1} = 1$: $b_0 = b_{n-1} = 1 \rightarrow -2^{n+1} + \sum_{i=0}^n 2^i b_i = -2^{n+1} + 2^n + \sum_{i=0}^{n-1} 2^i b_i = -2^n + \sum_{i=0}^{n-1} 2^i b_i$
- ✓ La représentation d'un nombre v se termine par k bits nuls si et seulement si v est divisible par 2^k .

En effet :

- Cette propriété est vraie pour les représentations non signées
- Les représentations non signée et par complément à deux sur n bits sont égales modulo 2^n
- On a $k \leq n$, donc deux nombres égaux modulo 2^n sont aussi égaux modulo 2^k
- ✓ L'opposé d'un nombre représenté par complément à deux s'obtient en inversant chaque bit de sa représentation, et en ajoutant 1 au résultat.

Soit $w = b_{n-1}b_{n-2}\dots b_0$, si l'on inverse chaque bit, le nombre représenté v satisfait : $v =_2 \sum_{i=0}^{n-1} 2^i (1 - b_i)$.

On a donc :

$$\begin{aligned} v + 1 &= 2^n 1 + \sum_{i=0}^{n-1} 2^i (1 - b_i) \\ &= 2^n 1 + \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i b_i \\ &= 2^n 2^n - \sum_{i=0}^{n-1} 2^i b_i \\ &= 2^n - \sum_{i=0}^{n-1} 2^i b_i \end{aligned}$$

2.3.3.3. L'addition par complément à deux

Quand calcule la somme non signée de deux suite de bits $w = b_{n-1}b_{n-2}\dots b_0$ et $w' = b'_n b'_{n-1} b'_{n-2} \dots b'_0$, on obtient $w'' = b''_{n-1} b''_{n-2} \dots b''_0$ tel que : $[w'']_{ns} =_2 [w]_{ns} + [w']_{ns}$

Par ailleurs, on a

$$\begin{aligned} [w]_{C2} &= 2^n - [w]_{ns}, \\ [w']_{C2} &= 2^n - [w']_{ns}, \\ [w'']_{C2} &= 2^n - [w'']_{ns}. \end{aligned}$$

Qui montre que le même algorithme peut être employé pour additionner les nombres non signés et représentés par complément à deux.

On en déduit

$$[w'']_{C2} = 2^n - [w]_{C2} - [w']_{C2},$$

Exemples :

Calcul de $12 + (-34)$:

$$\begin{array}{r} \boxed{1} \boxed{1} \boxed{1} \\ \begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\ + 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array} \end{array} \quad \begin{array}{r} \text{n. sign. compl. deux diff.} \\ 12 \quad \boxed{12} \quad 0 \\ 222 \quad -34 \quad 256 \\ \hline 234 \quad -22 \quad 256 \end{array}$$

Calcul de $34 + (-12)$:

$$\begin{array}{r} \boxed{1} \boxed{1} \\ \begin{array}{r} 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \end{array} \end{array} \quad \begin{array}{r} \text{n. sign. compl. deux diff.} \\ 34 \quad \boxed{34} \quad 0 \\ 244 \quad -12 \quad 256 \\ \hline 22 \quad 22 \quad 0 \end{array}$$

2.3.3.4. La multiplication par complément à deux

Principes :

- On procède de la même façon qu'avec les nombres non signés.
- Les opérandes et les produits partiels doivent être étendus sur le même nombre de bits (en répétant le bit de signe).

Calcul de $(-12) \times 34$ sur 12 bits:

$$\begin{array}{r} \times \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 0 \end{array} \end{array} \quad \begin{array}{r} \text{n. sign. compl. deux diff.} \\ \boxed{1} \quad \boxed{1} \quad 1 \quad 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array}$$

Calcul de $(-12) \times (-34)$ sur 12 bits:

$$\begin{array}{r} \times \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \\ \begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ + 0 \end{array} \end{array} \quad \begin{array}{r} \text{n. sign. compl. deux diff.} \\ \boxed{1} \quad \boxed{1} \quad 1 \quad 0 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \end{array}$$

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

Récapitulatif :

Bits	Non signée	Valeur signée	Compl. à un	Compl. à deux
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	0	-1

2.4. Les nombres réels

Pour représenter un réel arbitraire, il faut une quantité infinie d'information. On va donc approximer les représentations informatiques.

Les opérations manipulant les réels sont donc imprécises, les opérations arithmétiques augmentent en général le degré d'imprécision et lorsqu'on teste l'égalité de nombres réels, il faut tenir compte de cette imprécision.

2.4.1. La représentation en virgule fixe

Principe : on introduit un séparateur entre une partie entière et une partie fractionnaire, à une position fixée.

Exemple en base 10 :

$$\begin{array}{ccccccc} \text{poids: } & 10^2 & 10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} \\ \text{position: } & 2 & 1 & 0 & -1 & -2 & -3 \\ \hline & 1 & 2 & 3 & 4 & 5 & 6 \\ & , & & & & & \end{array}$$

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$$

$$= 1 \times 100 + 2 \times 10 + 3 \times 1 + 4 \times \frac{1}{10} + 5 \times \frac{1}{100} + 6 \times \frac{1}{1000}$$

$$= 123,456.$$

Exemple en base 2 (binaire) :

$$\begin{array}{ccccccc} \text{poids: } & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} \\ \text{position: } & 4 & 3 & 2 & 1 & 0 & -1 & -2 & -3 \\ \hline & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ & , & & & & & & & \end{array}$$

$$0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}$$

$$= 8 + 2 + \frac{1}{2} + \frac{1}{4}$$

$$= 10,75.$$

Propriété : s'il y a k bits après le séparateur, le nombre représenté vaut $\frac{1}{2^k}$ fois le nombre entier représenté par la même suite de bits.

Illustration : le nombre entier représenté par 01010110 vaut 86 : $10,75 = \frac{1}{8} \times 86$.

2.4.1.1. Les nombres signés en virgule fixe

La représentation en virgule fixe est donc équivalente à une représentation entière à un facteur près.

Pour représenter les entiers, on choisit :

- La représentation non signée pour les nombres non signés.
- La représentation par complément à deux pour les nombres signés.

Exemple : sur 8 bits avec 2 chiffres après la virgule, -24,25 se représente 10011111, en effet :

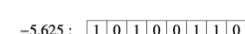
$$-24,25 = \frac{1}{2^2} (-97) = \frac{1}{2^2} (-2^8 + 159) \text{ et } [10011111]_{ns} = 159$$

2.4.1.2. L'addition en virgule fixe

Procédure :

1. On décale les opérandes de façon à faire coïncider leurs positions. Il s'effectue vers la droite et peut conduire à perdre les bits les moins significatifs.
2. On additionne les représentations alignées à l'aide du même algorithme que les entiers non signés.

5,5 : 

-5,625 : 

1. Décalage:





Rappel: Pour étendre les représentations vers la gauche, il faut en répéter le bit de signe.

2. Calcul de la somme:



Ce résultat représente donc le nombre -0,25.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

2.4.2. La représentation en virgule flottante

Lorsque la grandeur des nombres représentés est très variable, on fait appel à la virgule flottante.

On va dissocier la représentation des chiffres significatifs d'un nombre de celle de la grandeur de celui-ci. $v = m \cdot r^e$ (r est la base, m est la mantisse en virgule fixe et e est l'exposant).

Propriétés :

- La base = 10 pour notation scientifique et = 2 pour l'informatique.
- Les valeurs possibles de l'exposant = intervalle des valeurs représentables.
- Le nombre de bits choisi pour représenter la mantisse caractérise la précision avec laquelle les nombres sont représentés.

2.4.2.1. Le standard IEEE 754

Le standard définit plusieurs représentations :

- la simple précision:



Le champ s est le bit de signe (= 0 pour positifs et = 1 pour négatifs)

- la double précision:



2.4.2.1.1. L'encodage de l'exposant

Simple précision : encodé par la représentation entière non signée sur 8 bits du nombre $e + 127 \rightarrow$ intervalles des exposants = $[-127, \dots, 128]$

Double précision : encodé par la représentation entière non signée sur 11 bits du nombre $e + 1023 \rightarrow$ intervalles des exposants = $[-1023, \dots, 1024]$

2.4.2.1.2. L'encodage de la mantisse

Premier cas : l'exposant n'est pas égal à une valeur extrême

Simple précision :

L'exposant n'est pas égal à -127 ou 128 \rightarrow mantisse normalisée (on a $1 \leq |m| < 2$)

La mantisse m est représentée par $b_1 \dots b_n$ avec $n = 23$, vaut

$$|m| = 1 + \sum_{i=1}^n 2^{-i} b_i.$$

Double précision :

L'exposant n'est pas égal à -1023 ou 1024 \rightarrow mantisse normalisée (on a $1 \leq |m| < 2$)

La mantisse m est représentée par $b_1 \dots b_n$ avec $n = 52$, vaut

$$|m| = 1 + \sum_{i=1}^n 2^{-i} b_i.$$

Calcul de la représentation en simple précision de -7,5:

Exemple :

- Ce nombre est négatif, donc le bit de signe est égal à 1.
- Afin d'obtenir une mantisse normalisée, il faut choisir un exposant égal à 2. On obtient alors

$$|m| = \frac{7,5}{2^2} = 1,875,$$

qui satisfait bien $1 \leq |m| < 2$.

- La représentation de l'exposant est égale à la représentation entière non signée sur 8 bits du nombre $2 + 127 = 129$, soit 10000001.

- On a

$$1,875 = 1 + 2^{-1} + 2^{-2} + 2^{-3}.$$

La mantisse est donc représentée par la suite de bits

11100000000000000000000000000000.



Deuxième cas : l'exposant est égal à sa valeur minimale

Simple précision :

L'exposant est égal à -127 \rightarrow mantisse dénormalisée (on a $0 \leq |m| < 2$)

La mantisse m est représentée par $b_1 \dots b_n$, vaut

$$|m| = \sum_{i=1}^n 2^{-i+1} b_i.$$



NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

Double précision :

L'exposant est égal à -1023 → mantisse dénormalisée (on a $0 \leq |m| < 2$)

La mantisse m est représentée par $b_1 \dots b_n$, vaut

$$|m| = \sum_{i=1}^n 2^{-i+1} b_i.$$

Calcul de la représentation en simple précision de 2^{-140} .

Exemple :

- Le bit de signe est égal à $[0]$.
- Aucun exposant représentable ne conduit à une mantisse normalisée. On choisit donc un exposant égal à -127 , dont la représentation est $[00000000]$.
- On a

$$|m| = \frac{2^{-140}}{2^{-127}} = 2^{-13},$$

qui satisfait bien $0 \leq |m| < 2$.

- La mantisse est représentée par la suite de bits

$[000000000000100000000]$.



2.4.2.1.2.1. L'utilité des mantisses dénormalisées

Les mantisses dénormalisées permettent de représenter des nombres plus petits en valeur absolue qu'avec les mantisses normalisées, avec une diminution de précision.

Cas particulier : représentation de zéro :

- La mantisse est dénormalisée → exposant prend sa plus petite valeur et se représente 000...0
 - La mantisse = 0 et se représente 000...0
 - Le bit de signe est quelconque
- 2 représentations : 000...0 (positif) et 100...0 (négatif)

Troisième cas : l'exposant est égal à sa valeur maximale

Simple précision :

L'exposant est égal à 128

Double précision :

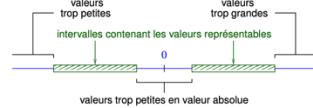
L'exposant est égal à 1024

Cette situation sert à encoder des valeurs spéciales qui ne sont pas des nombres réels :

- Si tous les bits de la mantisse sont égaux à 0 : il y a un dépassement
 - Vers les valeurs + si le bit de signe est 0 (infini positif)
 - Vers les valeurs - si le bit de signe est 1 (infini négatif)
- Si au moins un bit de la mantisse est égal à 1 : valeur indéfinie NaN (Not a Number)

2.4.2.1.3. Les nombres représentables

L'ensemble des réels représentables à l'aide d'un nombre de bits donné ne forme pas un intervalle: Comme les réels ne sont représentés qu'avec une précision limitée, l'ensemble des réels représentables n'est pas un continuum. Il est cependant utile de connaître les bornes des intervalles contenant les réels représentables. La situation est la suivante:



Plus grande valeur absolue représentable max_v :

- Simple précision: L'exposant est égal à 127 et la mantisse à $2 - 2^{-23}$.

$$\rightarrow max_v \approx 3,403 \times 10^{38}.$$

- Double précision: L'exposant est égal à 1023 et la mantisse à $2 - 2^{-52}$.

$$\rightarrow max_v \approx 1,798 \times 10^{308}.$$

Plus petite valeur strictement positive représentable min_v :

- Simple précision: L'exposant est égal à -127 et la mantisse à 2^{-22} .

$$\rightarrow min_v \approx 1,401 \times 10^{-45}.$$

- Double précision: L'exposant est égal à -1023 et la mantisse à 2^{-51} .

$$\rightarrow min_v \approx 4,941 \times 10^{-324}.$$

2.4.2.1.4. L'addition en virgule flottante

L'addition de deux nombres $v_1 = m_1 \times 2^{e_1}$ et $v_2 = m_2 \times 2^{e_2}$ s'effectue de la façon suivante (on suppose $|e_1| \leq |e_2|$):

1. On remplace e_1 par $e'_1 = e_2$, et m_1 par $m'_1 = m_1 2^{e_1 - e_2}$.

Note: Cela peut conduire à perdre un certain nombre de bits de la représentation de m_1 , ou à dénormaliser cette mantisse.

2. On remplace chaque mantisse négative par son complément à deux.

3. On calcule la somme m des deux mantisses en virgule fixe.

4. Si le résultat est négatif, on le remplace par son complément à deux.

5. On normalise $m \times 2^{e'_1}$ de manière à obtenir une mantisse normalisée ou dénormalisée.

Note: Cette opération peut conduire à détecter un dépassement.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

2.4.2.1.5. La multiplication en virgule flottante

La multiplication de deux nombres $v_1 = m_1 \times 2^{e_1}$ et $v_2 = m_2 \times 2^{e_2}$ s'effectue grâce à l'algorithme suivant:

1. On détermine le **signe** du produit.
2. On calcule la somme $e_1 + e_2$ en arithmétique entière.
Note: Cela peut conduire à détecter un dépassement, ou à effectuer un **arrondi vers zéro**.
3. On calcule le produit $m = m_1 \times m_2$ en virgule fixe.
4. On **normalise** si nécessaire le résultat $m \times 2^e$, de la même façon que pour l'addition.

2.5. La représentation de textes

2.5.1. Le code ASCII

Il est à la base d'une grande majorité des encodages actuellement utilisés.

Principes :

- Un caractère est encodé à l'aide de 7 bits d'information. On attribue donc à chaque symbole un code dans l'intervalle $[0, \dots, 127]$.
- Les codes de 0x00 à 0x1F représentent les caractères de contrôle.
- Les codes de 0x20 à 0x3F correspondent aux symboles mathématiques, à la ponctuation et aux chiffres. Le code du chiffre n est égal à $0x3n$. Remarque: La valeur d'un chiffre est donc égale aux quatre bits de poids faible de son code.
- Les codes de 0x40 à 0x5F contiennent les lettres majuscules et quelques symboles spéciaux. Les lettres sont classées par ordre alphabétique et possèdent des codes consécutifs, ce qui facilite les opérations de comparaison entre chaînes de caractères.
- Les codes de 0x60 à 0x7F contiennent les lettres minuscules, un caractère de contrôle (0x7F) et quelques symboles spéciaux.
- Note: Les codes d'une même lettre majuscule et minuscule partagent les mêmes cinq bits de poids faible.

20	30	0	40	@	50	P	60	'	70	p	
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D)
2E	.	3E	>	4E	N	5E	*	6E	n	7E	-
2F	/	3F	?	4F	O	5F	_	6F	o		

Table des caractères imprimables ASCII :

2.5.2. Le standard ISO 8859-1

Les ordinateurs manipulent les données par groupes de 8 bits. Par rapport à l'ASCII, un bit supplémentaire permet de représenter 128 caractères de plus.

Le standard ISO 8859 regroupe une quinzaine d'encodages couvrant les symboles les plus utilisés par les langues occidentales :

- Les 128 premiers caractères (bit de poids fort égal à 0) coïncident avec le code ASCII
- Les 128 caractères supplémentaires différent pour chaque variante du standard.
- La variante la plus utilisée, ISO 8859-1 ou ISO latin1, est un bon compromis pour les applications encodant chaque caractère sur un octet.

2.5.3. L'Unicode

Le standard Unicode a été introduit afin d'unifier la représentation de tous les systèmes d'écriture actuels et historiques. Exemple : « € » correspond à U+20AC.

Propriétés :

- La version actuelle d'Unicode définit 137929 symboles.
- Les 256 premiers codes sont ceux du standard ISO 8859-1.
- Les codes appartiennent à l'intervalle $[0, 0x10FFFF]$. La représentation d'un caractère nécessite donc 21 bits.
- Le symbole de code k est noté U + k, où k s'écrit en hexadécimal.

L'inconvénient d'Unicode : inefficace lorsque la majorité des caractères d'un texte sont représentables en ASCII ou en ISO 8859-1.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

La compression UTF-8 vise à pallier cet inconvénient, en encodant chaque symbole à l'aide d'un nombre variable d'octets. L'hypothèse est que les codes de petite valeur sont les plus fréquents.

Principe : la représentation du symbole U+k dépend de l'intervalle auquel appartient k :

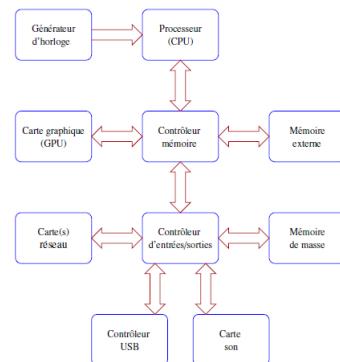
- Si $k \in [0, 0x7F]$: le caractère est représenté par l'octet $0b_6b_5...b_0$, où $0b_6b_5...b_0$ est l'encodage binaire non signé de k.
- Si $k \in [0x80, 0x7FF]$: le caractère est représenté par les deux octets $110b_{10}b_9...b_6$ et $10b_5b_4...b_0$, où $b_{10}b_9...b_0$ est l'encodage binaire non signé de k.
- Si $k \in [0x800, 0xFFFF]$: le caractère est représenté par les trois octets $1110b_{15}b_{14}b_{13}b_{12}$, $10b_{11}b_{10}...b_6$ et $10b_5b_4...b_0$, où $b_{15}b_{14}...b_0$ est l'encodage binaire non signé de k.
- Si $k \in [0x1000, 0x10FFFF]$: le caractère est représenté par les quatre octets $11110b_{20}b_{19}b_{18}$, $10b_{17}b_{16}...b_{12}$, $10b_{11}b_{10}...b_6$ et $10b_5b_4...b_0$, où $b_{20}b_{19}...b_0$ est l'encodage binaire non signé de k.

Exemple : pour U+20AC (€), on a $k = 0x20AC \in [0x800, 0xFFFF]$. En binaire, 0x20AC s'écrit 0010000010101100. Ce symbole est représenté par les 3 octets 11100010, 10000010, 10101100 c'est-à-dire E2, 82 et AC en hexadécimal.

3. La structure d'un ordinateur

3.1. Introduction

Un ordinateur contient : un ou plusieurs processeurs (CPU) qui exécute les programmes, de la mémoire pour retenir les programmes et les données, un générateur d'horloge imposant un rythme d'exécution des instructions, un ensemble de périphériques (carte graphique, carte réseau, carte son), des contrôleurs gérant le flux des données et des bus de communication.



3.2. La mémoire

Le terme de mémoire désigne tous les composants capables de retenir de l'information.

3.2.1. La mémoire vive

C'est un type de mémoire pour lequel la consultation et la modification de n'importe quelle partie de son contenu sont illimitées, son contenu est préservé tant que l'ordinateur reste sous tension, le taux de transfert est élevé et la latence¹ est faible.

Dans un ordinateur, elle est présente en tant que mémoire externe, registres, mémoire cache (accélérer les opérations impliquant mémoire externe) et dans les périphériques.

Deux technologies sont utilisées : mémoire statique (SRAM, info retenue par boucles de feedback) et mémoire dynamique (DRAM, info représentée par la charge de condensateur).

3.2.2. La mémoire morte (ROM)

C'est un type de mémoire dont le contenu ne peut pas être modifié durant son fonctionnement normal. Elle sert à mémoriser les données et les programmes qui ne doivent pas changer au cours de la vie du système comme : le programme de démarrage, les polices de caractères et le logiciel d'un système embarqué.

Les technologies de la mémoire morte : les composantes OTP sont programmables une seule fois, les EPROM et EEPROM peuvent être reprogrammées grâce à un mécanisme qui efface leur contenu, la mémoire flash est similaire à l'EEPROM mais implémente un mécanisme d'effacement plus flexible.

¹ Délai avec la fin d'une opération"

3.2.3. La mémoire de masse

Elle sert à retenir les données et les programmes qui doivent être préservés lorsque l'ordinateur est éteint et qui peuvent être changés.

Les 2 principales technologies sont : les disques durs et la mémoires flash.

3.2.4. L'adressage

La mémoire vive et morte sont organisées de la façon suivante : les données sont mémorisées dans des cellules (identifiées par leur adresse) de taille fixe et l'ensemble des adresses possibles forme l'espace d'adressage.

La notion d'adresse est liée au concept de pointeur : la cellule d'adresse 0x100 contient l'octet 0x12. On dit que 0x100 pointe vers 0x12.

0x103:	0x78
0x102:	0x56
0x101:	0x34
0x100:	0x12

Si une donnée doit être mémorisée sur plus d'une cellule, on la découpe en blocs placés dans des cellules consécutives de la mémoire. Les cellules d'adresse croissante énumèrent mes blocs :

- Depuis le poids faible vers le poids fort (petit-boutiste)
- Depuis le poids fort vers le poids faible (gros-boutiste)

Exemple : représentation de 0x12345678 sur des cellules de 8 bits à partir de l'adresse 0x100

0x103:	0x12
0x102:	0x34
0x101:	0x56
0x100:	0x78

0x103:	0x78
0x102:	0x56
0x101:	0x34
0x100:	0x12

Représentation petit-boutiste

Représentation gros-boutiste

L'alignement :

Même si la mémoire est organisée en octets, les échanges entre le processeur et la mémoire externe s'effectuent par blocs de plus grande taille. Il faut parfois en tenir compte lors de la programmation : une donnée représentée sur n octets, où n est une puissance de 2, est dite alignée si son adresse est un multiple de n. Certaines architectures interdisent les transferts de données non alignées. Pour d'autres (p.ex., x86-64), de tels transferts sont possibles, mais sont inefficaces.

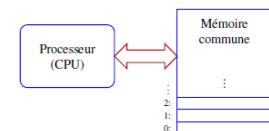
3.3. Le processeur

3.3.1. Introduction

C'est le composant responsable de l'exécution des programmes exprimés en code machine.

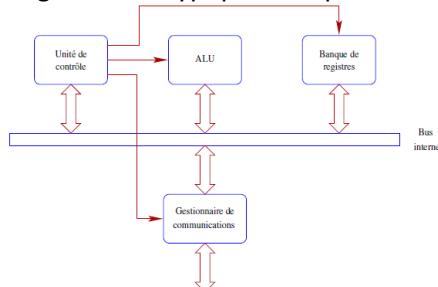
Deux modèles d'architectures existent :

- Modèle Von Neumann : les programmes et les données partagent le même espace d'adressage.
- Modèle Harvard : les programmes et les données sont placés dans des mémoires séparées.

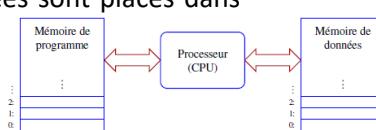


3.3.2. Structure d'un processeur

Organisation typique d'un processeur :



La banque de registres est une petite quantité de mémoire vive, utilisée comme espace de travail. L'unité arithmétique et logique (ALU) est le composant chargé de traiter l'information. Elle effectue les opérations arithmétiques sur les nombres entiers, le traitement de nombres réels et les opérations logiques. Le bus interne est un canal de communication entre les composants du processeur. Le gestionnaire de communications relie le bus interne à l'interface extérieur du processeur, gère aussi les échanges de données avec la mémoire et les périphériques. L'unité de contrôle est responsable de l'exécution des instructions en commandant les autres composants.



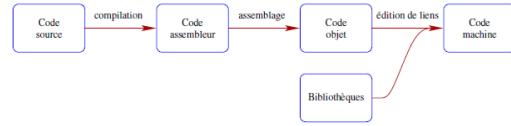
NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

3.3.3. Le code machine

Pour être directement exécutés, les programmes doivent être traduit en code machine.

Deux mécanismes :

- Un interpréteur traduit chaque instruction au moment où elle doit être exécutée
- Un compilateur traduit le programme en code machine une fois pour toute.



Production de code machine :

- 1) Compilation : Le code source est traduit en code assembleur par un compilateur. Le code assembleur est composé d'instructions que le processeur peut effectuer, mais il est exprimé dans un format lisible.
- 2) Assemblage : Le code assembleur est traduit en code objet. Celui-ci est similaire au code machine, mais peut contenir des références incomplètes vers du code extérieur, ainsi qu'être fragmenté en plusieurs modules.
- 3) Edition de liens : Le code objet est combiné avec du code issu de bibliothèques pour obtenir le code machine exécutable.

3.3.4. L'exécution des instructions

L'unité de contrôle gère deux registres :

- Registre d'instruction (IR) : contient code instruction (opcode) en cours d'exécution.
- Compteur de programme (RIP) : contient l'adresse en mémoire de la prochaine instruction à exécuter.

Elle effectue les opérations suivantes à un rythme d'horloge :

- 1) Charger dans IP l'opcode pointé par PC.
- 2) Décoder la valeur de IR.
- 3) Exécuter l'instruction correspondant en contrôlant les autres composants du processeur.
- 4) Recommencer à l'étape 1.

Illustration : exécution du programme suivant par un processeur x86-64, à partir de RIP = 0x1000

- 1) L'opcode 0x01 pointé par RIP est chargé dans IR.
- 2) Cet opcode est décodé. Il s'agit ici d'une opération d'addition.
- 3) Cette opération d'addition définit un octet d'opérandes. Celui-ci (0xD8) est chargé et décodé. Ici, l'addition est sur EAX et EBX et le résultat doit être écrit dans EAX.
- 4) RIP est incrémenté de 2 unités, pour le faire pointer vers l'instruction suivante (0x1002).
- 5) L'opération d'addition est exécutée :
 - a. Unité de contrôle demande à la banque de registre de transférer le contenu de EAX et EBX vers l'ALU via le bus.
 - b. L'ALU est piloté de façon à effectuer une addition, et à placer le résultat sur le bus.
 - c. La banque de registres charge le résultat dans EAX.
- 6) La procédure se répète à partir du point 1 pour l'instruction suivante.

0x1003:	⋮
0x1002:	0xC3
0x1001:	0xD8
0x1000:	0x01
0xFFFF:	⋮

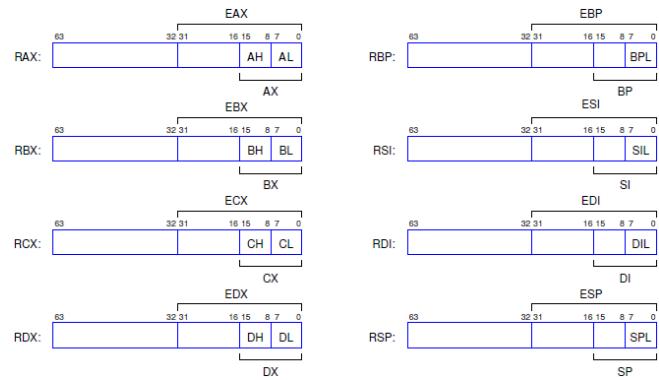
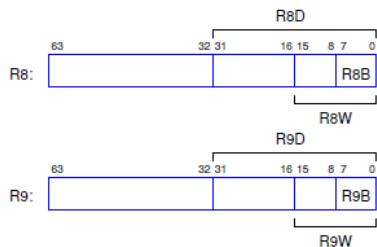
NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

3.4. L'architecture x86-64

3.4.1. Les registres

Les 16 registres généraux de 64 bits sont :
 RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8,
 R9, R10, R11, R12, R13, R14, R15.

Les parties des registres :



3.4.2. Les drapeaux

Le registre RFLAGS contient les drapeaux² suivants :

- CF : indique d'une opération arithmétique sur des nombres de n bits a produit un report à la position n.
- ZF : indique d'une opération a fourni un résultat nul.
- SF : correspond au bit de poids fort du résultat d'une opération (bit de signe pour -donnée signée).
- OF : indique un dépassement arithmétique pour des données signées.

3.4.3. Les modes d'adressage

Quand on exprime une instruction du processeur en langage d'assemblage, le mode d'assemblage, c'est ce qui indique comment les opérandes (paramètres) de l'opération vont être représentés.

En langage s'assemblage, l'écriture d'une instruction est composée

- D'une mnémonique : le nom conventionnel de l'instruction
- D'opérandes : source, destination ou les deux.

3.4.3.1. L'adressage registre (reg)

Quand on a effectué une opération, les sources et les destinations, ce sont les données qui sont dans le registre. Si un registre est une source, on va lire l'information en allant chercher dans le registre et s'il est une destination, on va placer le résultat dans le registre.

Exemple d'une opération complète (addition) : `ADD RAX, RBX`

Traduction : l'opération va prendre les informations qui se trouvent dans RAX (source et destination) et RBX (source), on va les additionner et mettre le résultat dans RAX et RBX garde sa valeur initiale.

3.4.3.2. L'adressage immédiat (imm)

Il utilise une constante comme source. On ne l'utilise jamais pour une destination (pas échanger 2 et 3).

Exemple (addition) : `ADD RDI, 0x10`

Traduction : l'opération additionne RDI et 0x10(une constante, ici 16 en hexadécimal) et met le résultat dans RDI.

3.4.3.3. L'adressage direct (mem)

Syntaxe : <taille> ptr [<adresse>] où taille = nbre de bits. Ce que l'on va écrire à la place de taille, ce sont ces mots clés : qword (64), dword (32), word (16) ou byte (8). Et adresse = pointeur vers emplacement de mémoire.

² Bits d'information mis à jour par certaines instructions

Indice : à chaque fois que l'on voit [] ou ptr → mémoire

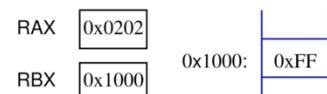
Exemple : `ADD dword ptr [0x1234], R8D` Traduction : adressage direct pour 1^{ère} opérande et adressage registre pour la 2^{ème}. Il faut que les deux opérandes aient la même taille. On va faire une lecture en mémoire d'une valeur de 32 bits à partir de l'adresse 1234 et lire la valeur dans le registre R8D et la somme va être mise dans la case 1234.

3.4.3.4. L'adressage indirect (mem)

On a remplacé l'adresse de la constante (adressage direct) par le nom d'un registre (64 bits).

Exemple : `ADD AH, byte ptr [RBX]` Traduction : on va chercher 8 bits d'informations à l'endroit donné par le registre RBX. Prendre la valeur du registre RBX, l'interpréter comme une adresse en mémoire, effectuer une lecture en mémoire d'un octet à cette adresse , ajouter cet octet à la valeur de AH et mettre résultat dans AH.

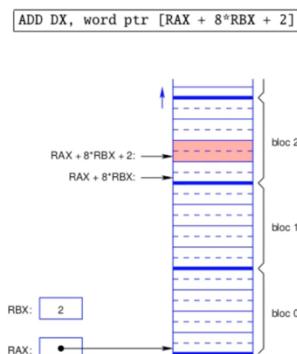
Exemple plus concret : On va calculer FF + 02. Après l'opération, RAX = 0x0102.



3.4.3.5. L'adressage indirect indexé (mem)

Syntaxe : <taille> ptr [<base> + <facteur> * <index> + <déplacement>] où base et index = registre 64 bits, facteur = 1,2,4 ou 8 et déplacement = constance (>0 ou <0).

Exemple concret : On a DX (registre 16 bits) et word (16 bits) donc on est bon pour les tailles. On va faire accès à la mémoire à l'adresse [] qui va falloir calculer. Quand on aura la valeur de l'adresse on l'additionne a DX et on met le résultat dans DX.



3.4.4. Les instructions de manipulation des données

Pour chaque instruction, les modes d'adressage(imm(pour constante comme argument), reg (pour utiliser le contenu d'un registre) ou mem (accès à la mémoire)) seront précisés ainsi que les drapeaux.

3.4.4.1. L'instruction MOV (Déplacement de données)

Exemple 1 : `MOV EBX, dword ptr [0x100]` Syntaxe similaire à ADD, sa mnémonique = MOV. Le 1^{er} argument = destination et le deuxième = source (ils doivent avoir même taille). Effet de l'instruction : lire l'information fournie par la source et l'écrire dans la destination.

Traduction : l'instruction va lire une donnée de 4 octets en mémoire à l'adresse 100, la donnée sera stockée sur 32 bits = 4 octets (dword), et ensuite, elle va l'écrire dans le registre EBX.

Exemple 2 : `MOV byte ptr [RAX + RSI - 4], 0xFF` On a un adressage imm à droite (la valeur que l'on va écrire sera une constante, ici FF) et un adressage mem indexé à gauche. Vu que l'on a byte, la donnée FF sera interprétée sur 8 bits. On prend valeur FF, l'écrire à l'endroit donné par [].

Il faut toujours faire attention à la taille des registres !

3.4.4.2. L'instruction XCHG (Échange du contenu des deux opérandes)

Exemple 1 : `XCHG AL, AH` On prend la valeur dans AL et celle de AH et on les permute. Si on a 0x1234 dans RAX, après l'opération, on aura 0x3412.

Exemple 2 (cas particulier) : `XCHG EAX, EAX` Vu qu'elle n'a aucun effet, on l'appelle NOP (No OPeration)

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

3.4.5. Les instructions arithmétiques

3.4.5.1. L'instruction ADD (Ajouter sa seconde opérande à sa première)

Exemple : `ADD R10, -1` On a un adressage reg à gauche et adressage imm à droite. On décrémente R10 de 1.

Drapeaux affectés : CF, ZF, SF et OF.

3.4.5.2. L'instruction SUB (Soustrait sa seconde opérande de sa première)

Exemple : `SUB R10, 1` On calcule la différence entre la première opérande et la deuxième et mettre le résultat dans la première opérande. Cette instruction ci fait exactement la même chose que celle au-dessus pour ADD.

Drapeaux affectés : les mêmes que ADD.

3.4.5.3. L'instruction CMP (Comparaison de deux opérandes)

On calcule la différence de la première et deuxième opérande

Exemple : `CMP EAX, EBX` On calcule la différence de EAX et EBX (comme SUB) mais on n'écrit le résultat nulle part. Pour connaître le résultat, il faut juste regarder l'état du drapeau.

Si les nombres sont signés, $\Delta = EAX - EBX$:

- ZF = 1ssi $\Delta = 0 \rightarrow EAX = EBX$
- SF = 1 (résultat < 0)ssi $\Delta < 0 \rightarrow EAX < EBX$

Drapeaux affectés : les mêmes que ADD et SUB.

CF	ZF	SF	OF
0	0	1	0
EAX	0xFFFFFFFF		
EBX	0x00000000		
Δ	0xFFFFFFF		
CF	ZF	SF	OF
1	0	0	0
EAX	0xFFFFFFFF		
EBX	0xFFFFFFFF0		
Δ	0x0000000F		
CF	ZF	SF	OF
1	1	0	0
EAX	0xFFFFFFFF		
EBX	0xFFFFFFFF		
Δ	0x00000000		

3.4.5.4. Les instructions INC et DEC (Incrémentation et décrémentation des opérandes)

Exemple : `INC byte ptr [RBX]` On a un adressage mem indirect basé sur registre RBX, byte = 1 octet. L'instruction va aller chercher l'octet qui se trouve à l'endroit donné par RBX, va ajouter 1 et va écrire le résultat là où on a pris l'octet.

Drapeaux affectés : CF est préservé et ZF, SF et OF sont mis à jour.

3.4.5.5. L'instruction MUL (Multiplication de deux nombres non-signés)

L'instruction n'admet qu'une seule opérande. La deuxième opérande et l'endroit du résultat sont imposés.

L'opération effectuée dépend de la taille de cette opérande :

- 8 bits : l'opérande est multipliée par AL et le résultat est placé dans AX (Quand on va écrire le résultat de l'opération dans AX, on va écraser AL).
- 16 bits : l'opérande est multipliée par AX et le résultat est placé dans DX:AX (Le résultat va être mis dans la paire DX:AX)
- 32 bits : l'opérande est multipliée par EAX et le résultat est placé dans EDX:EAX
- 64 bits : l'opérande est multipliée par RAX et le résultat est placé dans RDX:RAX

Exemple (non-signé) : `MUL dword ptr [0x1234]` Vu que l'on a un adressage, on va multiplier ce que l'on trouve à l'adresse 1234 (dword \rightarrow 32 bits) par quelque chose d'autre. On prend l'entier de 32 bits de l'adresse 1234, le multiplier par le contenu de EAX et mettre résultat dans la paire EDX(32 bits poids fort):EAX(32 bits de poids faible).

Drapeaux affectés : CF et OF = 0 si le résultat est représentable sur n bits. Les autres drapeaux sont modifiés de façon arbitraire.

Exemple plus concret :

CF	ZF	SF	OF	0x1237:	0xDD
0	0	0	0	0x1236:	0xCC
EAX	0x00001000			0x1235:	0xBB
EDX	????			0x1234:	0xAA
CF	ZF	SF	OF	0x1237:	0xDD
1	?	?	1	0x1236:	0xCC
EAX	0xCBA000			0x1235:	0xBB
EDX	0x00000DDC			0x1234:	0xAA

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

3.4.5.6. L'instruction IMUL (Multiplication de deux nombres signés)

C'est la même instruction de MUL sauf que l'on travaille avec des nombres signés.

	CF	ZF	SF	OF		CF	ZF	SF	OF		0x1237:	0xDD
	0	0	0	0	EAX	0x00001000				EDX	0xFFFFFDCC	
Exemple :	EDX	????			0x1237:	0xDD				0x1236:	0xCC	
					0x1236:	0xCC				0x1235:	0xBB	
					0x1235:	0xBB				0x1234:	0xAA	
					0x1234:	0xAA						

3.4.6. Les instructions logiques

3.4.6.1. Les instructions AND, OR et XOR

Ces instructions vont prendre 2 opérandes (sources), on va appliquer opération à ces 2 opérandes et le résultat va écraser la première opérande (destination), comme ADD.

Ces instructions permettent de forcer certains bits à 0 pour AND, à 1 pour OR et à inverser pour XOR. Les modes d'adressage sont identiques à ceux de ADD et les drapeaux affectés sont : CF et OF sont mis à 0 et ZF et SF sont mis à jour en fonction du résultat de l'opération.

L'instruction AND ou logique :

Le résultat est égal à un si les deux opérandes sont égales à 1 (1 = vrai et 0 = faux).

x	y	AND(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

Quand on va appliquer la fonction AND à deux opérandes, on va appliquer la fonction qui se trouve dans cette table individuellement à chaque bits, à chaque position (1, ... la fin des valeurs), pour chaque position, on va avoir un bit de résultat et on va les collecter dans la destination.

Dans l'exemple, ce sont les 2 bits de poids faibles que l'on force à 0.

Exemple plus concret :	AND byte ptr [0x100], 0xFC	0x100	10101010	0xAA	0x100	10101000	0xA8
		11111100	0xF0		11111100	0xF0	

L'instruction OR ou inclusif :

Le résultat est égal à 1 si au moins une des deux opérandes = 1.

x	y	OR(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

Exemple concret :

AL	10101010	0xAA	AL	11111010	0xFA
	11110000	0xF0		11110000	0xF0

L'instruction XOR ou exclusif :

Le résultat est égal à 1 si exactement 1 opérande = 1.

x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

Exemple concret :

RBX	1010...01010101010101010	0xA...AAAA	RBX	1010...00101010110101010	0xA...55AA
	0000...01111111000000000	0xFF00		0000...01111111000000000	0xFF00

3.4.6.2. L'instruction NOT

Cette instruction complémenté tous les bits d'une valeur (complément à un).

Table de vérité :	x	$NOT(x)$	DX	0000000001011010	0x005A	DX	111111110100101	0xFFA5
	0	1						
	1	0						

Il n'y a aucun drapeaux affectés.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

3.4.7. Les instructions de manipulation de la pile

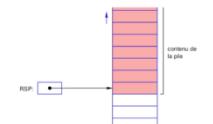
Une pile est une structure de données LIFO³ définissant deux opérations : empiler une valeur à son sommet (push) et dépiler une valeur depuis son sommet (pop).

Quand on retire une valeur, c'est toujours celle qui a été empilée en dernier lieu.

La pile sert aussi à :

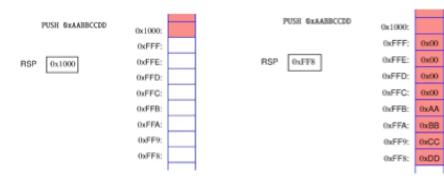
- mémoriser des données temporaires :
 - ... f: ... La flèche rouge démarre du haut, elle pointe des cases dans lesquelles il y a une exécution. Lorsqu'elle arrive en « f() », on va enregistrer α , l'adresse de l'exécution suivante dans la pile pour que la flèche rouge exécute d'abord « f : » et lorsqu'elle a fini, revienne en α pour continuer le programme et ensuite, effectue la même opération pour β .
 - ... β : ...
- sauvegarder le contenu de registres modifiés par une sous-routine.

Le contenu de la pile = cellules consécutives de la mémoire. Elle croît vers les adresses décroissantes. Rouge = contenu pile



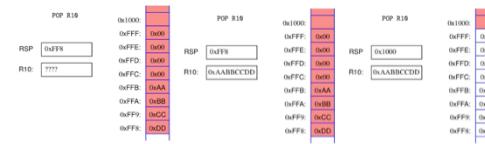
3.4.7.1. L'instruction PUSH

Elle empile une valeur de 64 bits (qword), donnée par son opérande. Les opérations réalisées sont décrémenter RSP de 8 unités et recopier l'endroit pointé par RSP.



3.4.7.2. L'instruction POP

Elle dépile une valeur de 64 bits et l'écrit à l'endroit spécifié par son opérande. Les opérations réalisées sont lire 8 octets depuis l'emplacement pointé par RSP, les recopier à l'endroit représenté par l'opérande et incrémenter RSP de 8 unités.



Exemple de séquences d'instructions :



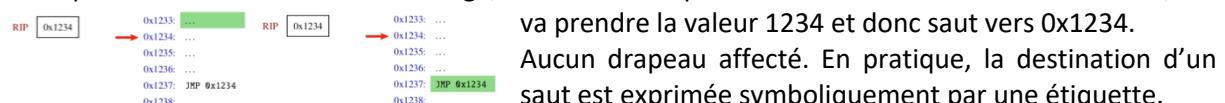
3.4.8. Les instructions de contrôle

Elles servent à modifier l'exécution séquentielle du programme, en transférant le contrôle, en continuant l'exécution), à un endroit arbitraire de celui-ci.

3.4.8.1. L'instruction JMP

Elle effectue un saut inconditionnel vers un emplacement de la mémoire de programme, donné par son opérande (source). L'instruction charge cette opérande dans le compteur de programme.

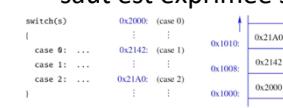
Exemple : Quand on est à la flèche rouge, on exécute ce qui est en vert. Quand on arrive à 0x1234, RIP



va prendre la valeur 1234 et donc saut vers 0x1234.

Aucun drapeau affecté. En pratique, la destination d'un saut est exprimée symboliquement par une étiquette.

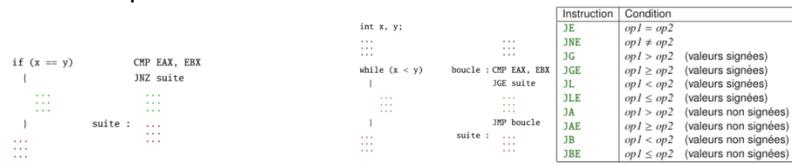
Autre exemple : `JMP qword ptr [8*RBX + 0x1000]`
obtient 1000. Si s = 1, on obtient 1008.
(hexa)



On considère que RBX = s. Si s = 0, on obtient 1000. Si s = 1, on obtient 1008. Si s = 2, on obtient 1016 (16 = 10 en hexa).

³ Last-In First-Out

Les instructions de saut conditionnel. Elle ne s'exécute pas automatiquement, juste s'il y a une condition particulière satisfaite.



3.4.8.2. L'instruction LOOP

Implémentation d'une boucle. Les opérations réalisées sont décrémenter RCX (toujours RCX) d'une unité et si la nouvelle valeur de RCX est non nulle, on effectue un saut.

Exemple :

	Mov RCX, 0x100	RCX	itération	Cas particulier (RCX = 0) :	RCX	itération
boucle:	NOP	0x100	1		0	1
	NOP	0xFF	2		0xFFFF...FF	2
	NOP	0xFE	3		0xFFFF...FE	3
	LOOP boucle	⋮	⋮		⋮	⋮
	0x02	255			0x0000...02	2 ⁶⁴ - 1
	0x01	256			0x0000...01	2 ⁶⁴

3.4.8.3. Les instructions CALL et RET

Elles permettent de programmer des sous-routines.

Les opérations réalisées par l'instruction CALL sont empiler la valeur courante de RIP, l'adresse où reprendre l'exécution du programme après la routine et effectuer un saut vers l'endroit donné par l'opérande.

Les opérations réalisées par l'instruction RET (pas argument) sont dépiler une valeur de 64 bits et effectuer un saut vers cette adresse.

Exemple (définition d'une fonction minswap, invoquée depuis le reste du programme) :

On va chercher deux valeurs (32 bits) à partir de l'adresse 100 et 104, que l'on place de ECX et EDX et on va ensuite appeler une fonction minswap. On va donc exécuter les opérations et quand on va exécuter RET, on va revenir après l'instruction CALL. La fonction minswap compare ECX et EDX. Si ECX < EDX (valeurs signées), on ne fait rien. Si ECX > EDX, on les permute.

```
MOV ECX, dword ptr [0x100]
MOV EDX, dword ptr [0x104]
CALL minswap
...
MOV ECX, dword ptr [0x108]
MOV EDX, dword ptr [0x10C]
CALL minswap
...
```

Lorsque l'on arrive à CALL minswap, on dit que $\alpha = \dots$ et on le place sur la pile. Ensuite, RET dépile α et on retourne à la position α .

4. La programmation en assembleur

4.1. Le langage d'assemblage

Il n'est pas n'est pas universel. Ici, on va utiliser l'architecture x86-64, le système d'exploitation Linux 64 bits et le compilateur GCC.

4.1.1. Un premier programme

```
.intel_syntax noprefix  
.text  
.global deep_thought  
.type deep_thought, @function  
deep_thought: MOV EAX, 42  
RET
```

Tout ce qui commence par un « . » sont des directives de compilations.

Traduction :

- « .intel_syntax noprefix » : on va se mettre dans la syntaxe du programme d'assemblage, intel.
 - « noprefix » : on va pouvoir écrire le nom des registres directement, sinon on doit mettre symbole spécial avant nom de chaque registre.
 - « .text » : on va se mettre au début du segment de code (instructions)
 - « .global deep_thought » : deep_thought = étiquette, elle est globale donc on pourra l'appeler à partir d'autres parties du programme
 - « type deep_thought, @function » : on nous dit que deep_thought représente une fonction
 - Opération : on va mettre 42 (valeur 42 en décimal) dans EAX et puis on effectue un retour de fonction. Ici, la fonction retourne la valeur 42 quand on l'appelle.

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

4.1.2. Les mécanismes de compilation et d'exécution

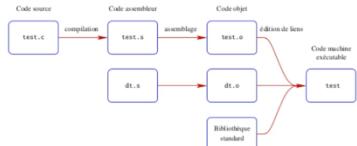
```
#include <stdio.h>
extern int deep_thought(void);
int main()
{
    printf("%d\n", deep_thought());
}
```

gcc -Wall -O3 -o test test.c dt.s

On déclare une fonction externe, `deep_thought`, qui ne prend pas d'argument et qui renvoie un entier.

Commande de compilation : `gcc` = compilateur, `-Wall` = avoir avertissements si erreur (warning), `-O3` = on veut optimiser et donc avoir le programme le plus efficace possible, `-O` test = on veut le résultat de la compilation dans le programme test, `test.c` = ce que l'on va compiler et on va fournir `dt.s`

Qu'est-ce qui s'est passé ?



Le fichier `test.c`, qui appelait `deep_thought` et `printf` a été compilé et on a obtenu un fichier `test.s`. Le fichier `test.s` a été traduit en un code machine et qui a été placé dans un fichier `test.o`. Ensuite, notre programme assembleur `dt.s` a été traduit aussi en code machine et le compilateur a aussi été chercher la fonction `printf` dans la bibliothèque standard. `test.o` contient `main`, `dt.o` contient l'implémentation de la fonction `deep_thought`. Le programme « éditeur de liens » a combiné les 3 morceaux de programme pour obtenir le fichier `test`, que l'on a exécuté.

4.1.3. Les étiquettes

En assembleur, une étiquette est une valeur représentée symboliquement. Deux formes possibles :

1. Étiquette d'adresse (Ex : `deep_thought` : EAX, 42) : `<étiquette>: <instruction> <opérande>` prend pour valeur l'adresse de l'instruction suivante, sera substituée en une valeur numérique par le programme d'assemblage, peut être utilisée avant sa définition et est soumise à certaines restrictions d'utilisations.
2. Étiquette définie par la directive (.equ boucle, 542) : à `.equ <étiquette>, <valeur>` partir de maintenant, `boucle` = abréviation pour 542. Référence à l'étiquette après sa définition.

Exemple : on a défini 3 étiquettes, 1(`answer = 42`), 2 (`deep_thought2`). Ici, on va commencer par

```
.intel_syntax noprefix
.text
.global deep_thought2
.type deep_thought2, @function
.equ answer, 42
.end:
deep.thought2: MOV EAX, answer
    RET
```

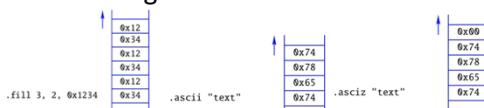
“`deep_thought2`”, elle met 42 dans EAX et ensuite, on fait un jump vers `end` qui lui nous retourne `RET`.

4.1.4. Le segment de données

Les directives sont :

- `.data` indique le début d'un segment de données.
- `.byte`, `.word`, `.int`, `.quad` suivent par une constante entière, définissent un entier codé sur 1, 2, 4 ou 8 octets
- `.fill <répétition>, <taille>, <valeur>` : remplit la mémoire avec `<répétition>` copies de `<valeur>`, encodées sur `<taille>` octets
- `.ascii` suivie d'une chaîne de caractères place cette chaîne en mémoire
- `.asciz` fait la même chose mais ajoute un octet nul à la fin de la chaîne
- `.balign <taille>` aligne l'adresse courante à un multiple de `<taille>`

Exemples :



Exemple de programme (émission de tickets pour une file d'attente) :

```
.intel_syntax noprefix
.data
    nb_tickets: .int 0
.text
.global ticket
.type ticket, @function
MOV EAX, dword ptr[nb_tickets]
    RET
```

```
#include <stdio.h>
extern int ticket(void);
int main()
{
    for(;;)
        printf("%d\n", ticket());
}
```

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

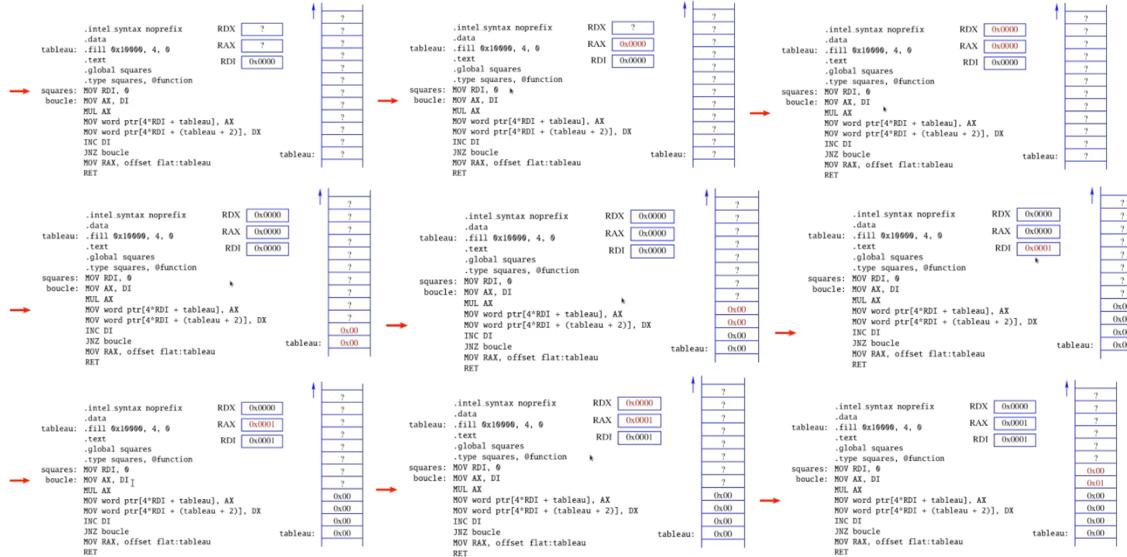
Pour une raison technique, les étiquettes d'adresse ne peuvent pas toujours directement figurer dans un adressage immédiat. L'instruction « MOV RAX, x » est invalide. La raison de cette restriction est que la valeur d'une étiquette d'adresse n'est généralement connue qu'au moment de l'édition de liens, à la compilation du programme. On écrira donc MOV RAX, offset :x

Exemple (fonction générant un tableau contenant la carré de tous les nombres entiers non signés encodés sur 16 bits). Au début, on a que des 0 dans le tableau. Le point d'entrée est squares.

```
.intel syntax noprefix
tableau:
    .data
    .fill 0x10000, 4, 0
    .text
    .global squares
    .type squares, @function
    MOV AX, 0
    MOV AX, DI
    MUL AX
    MOV word ptr[4*RDI + tableau], AX
    MOV word ptr[4*RDI + (tableau + 2)], DX
    INC DI
    JNZ boucle
    MOV RAX, offset flat:tableau
    RET

#include <stdio.h>
extern unsigned squares(void);
int main()
{
    unsigned i, *s;
    s = squares();
    for (i = 0; i < 0x10000; i++)
        printf("%u: %u\n", i, s[i]);
}
```

Détails de l'exemple :



4.2. L'appel d'une fonction

En plus du mécanisme de sauvegarde de l'adresse de retour par CALL et récupération de cette adresse par RET, il faut spécifier un protocole (convention d'appel) pour transmettre des arguments à une fonction appelée, récupérer de valeur de retour de cette fonction et permettre à cette fonction d'allouer des données temporaires.

Principes :

- Les 6 premiers arguments de l'appel sont fournis dans les registres RDI, RSI, RDX, RCX, R8 et R9.
- Les arguments suivants sont empilés dans l'ordre inverse de leur position.
- La valeur de retour est placée dans le registre RAX.
- La fonction appelée doit préserver RBX, RBP, R12, R13, R14 et R15.
- La fonction appelée doit maintenir le pointeur de pile RSP à une valeur multiple de 16, juste avant CALL.

Exemple : l'appel à la fonction ppcm, on va le faire en plaçant 10 dans les 32 bits de poids faible de

```
...
...
...
p = ppcm(10, 20);
...
...
```

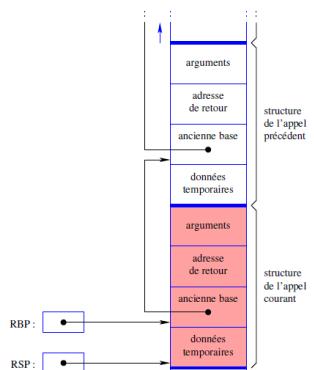
RDI, on va placer 20 dans RSI, et vu que ce sont que des 32 bits, on va utiliser les parties de poids faibles de ces registres.

4.3. La structure de pile

C'est une structure de données créée sur la pile à chaque appel de fonction et supprimée lorsque cet appel se termine. Cette structure est indexée à partie du registre RBP qui pointe vers sa base.

Composition de la structure dans l'ordre l'empilement :

- Les arguments de la fonction appelée à partir du 7^{ème} en ordre inverse de leur position, chaque argument occupe 8 octets.
- L'adresse de retour de la fonction.
- Un pointeur vers la base de la structure précédente (l'endroit où est placé ce pointeur constitue la base courante).
- Les données temporaires allouées par la fonction.

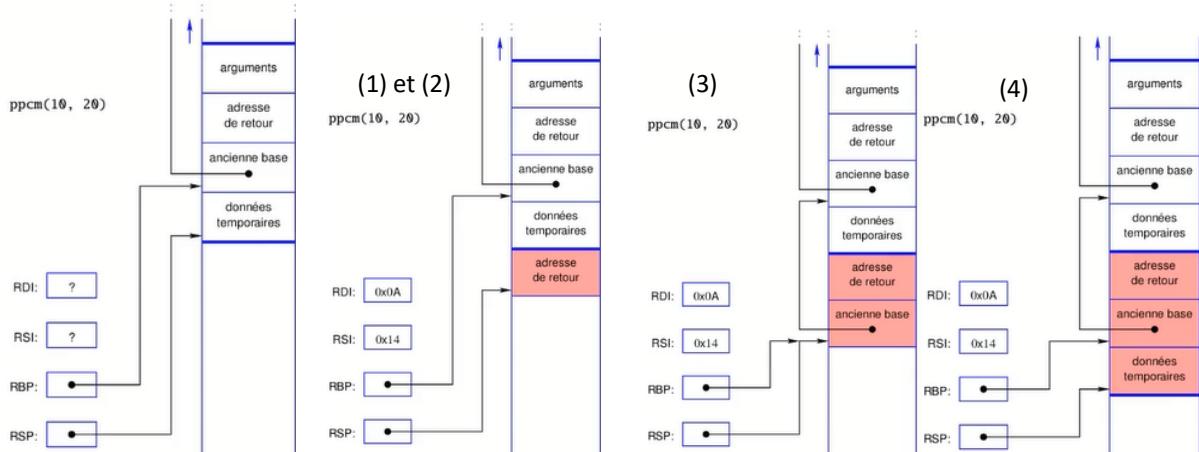


C'est la zone de la mémoire qui contient la pile. La pile croît vers adresses décroissantes (croissante de bas en haut). Ici, il y a deux stack frame (le rouge est plus récent que le bleu).

Par rapport au code ppcm : le stack frame blanc est celui de la fonction qui a appelé ppcm (à gauche) et le rouge est celui de ppcm (en cours d'appel, à droite). Si on avait un stack frame pour le pgcd, il aurait été mis en dessous du rouge. Dans le rouge : pas d'arguments (à partir du 7^{ème}), adresse de retour (comme pour la pile), données temporaires = g.

Les éléments de cette structure sont facilement accessibles par un adressage indirect indexé basé sur RBP. Accès aux arguments : « qword ptr [RBP + 16], qword ptr [RBP + 24]...on va avoir accès à l'argument 7 et 8 de la fonction. Accès aux données temporaires : « qword ptr [RBP - 8]... on a accès à l'octet le plus proche des données temporaires (début zone rouge).

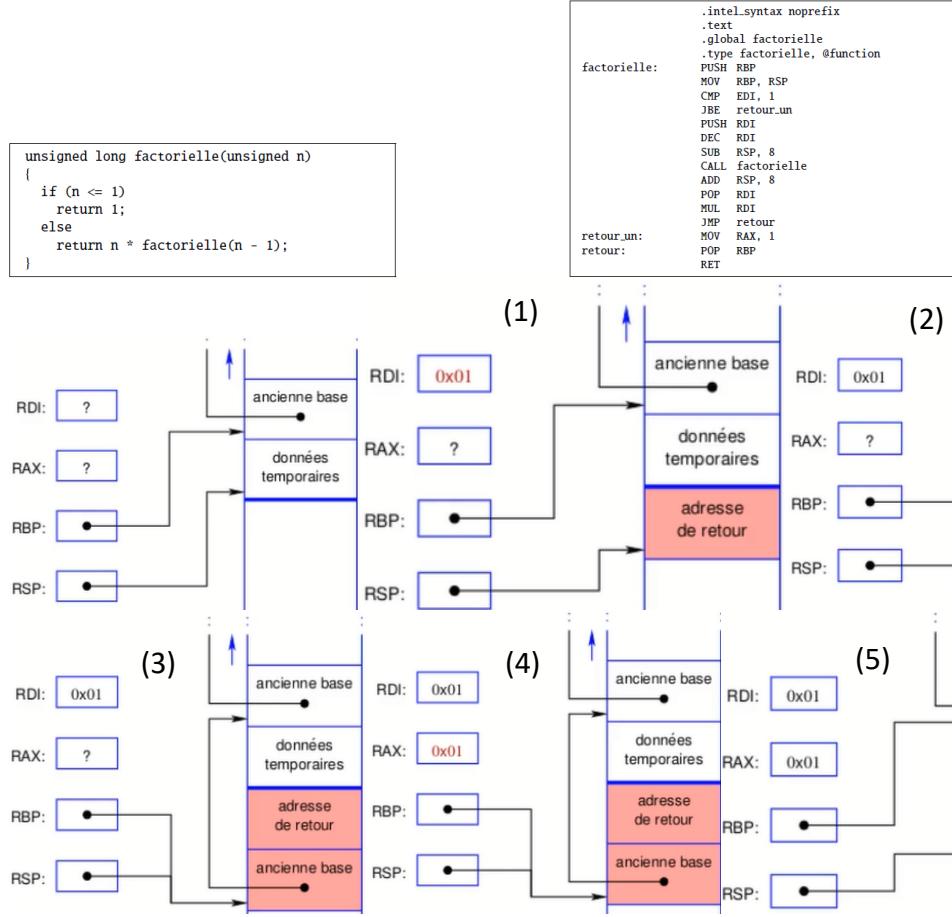
Que se passe-t-il quand on exécute le code « `p = ppcm(10,20)` ; » au niveau des stack frame.



La première chose que l'on fait, c'est de placer 10 (0x0A) dans RDI et 20 (0x14) dans RSI (1). Dans le code qui est en train d'appeler ppcm, je vais exécuter l'instruction CALL (2). CALL va empiler l'adresse de retour (adresse où il faudra revenir quand appel de fonction terminé). Maintenant, on va empiler l'ancienne base dans le nouveau stack frame pour avoir une nouvelle base (3), on a empilé l'ancienne valeur de RBP et donc le registre pointe maintenant vers la nouvelle base de la structure de pile. En dernier lieu, on empile les données temporaires (4).

Exemples de programmes pour mieux comprendre :

Exemple 1 : calcul récursif d'une factorielle



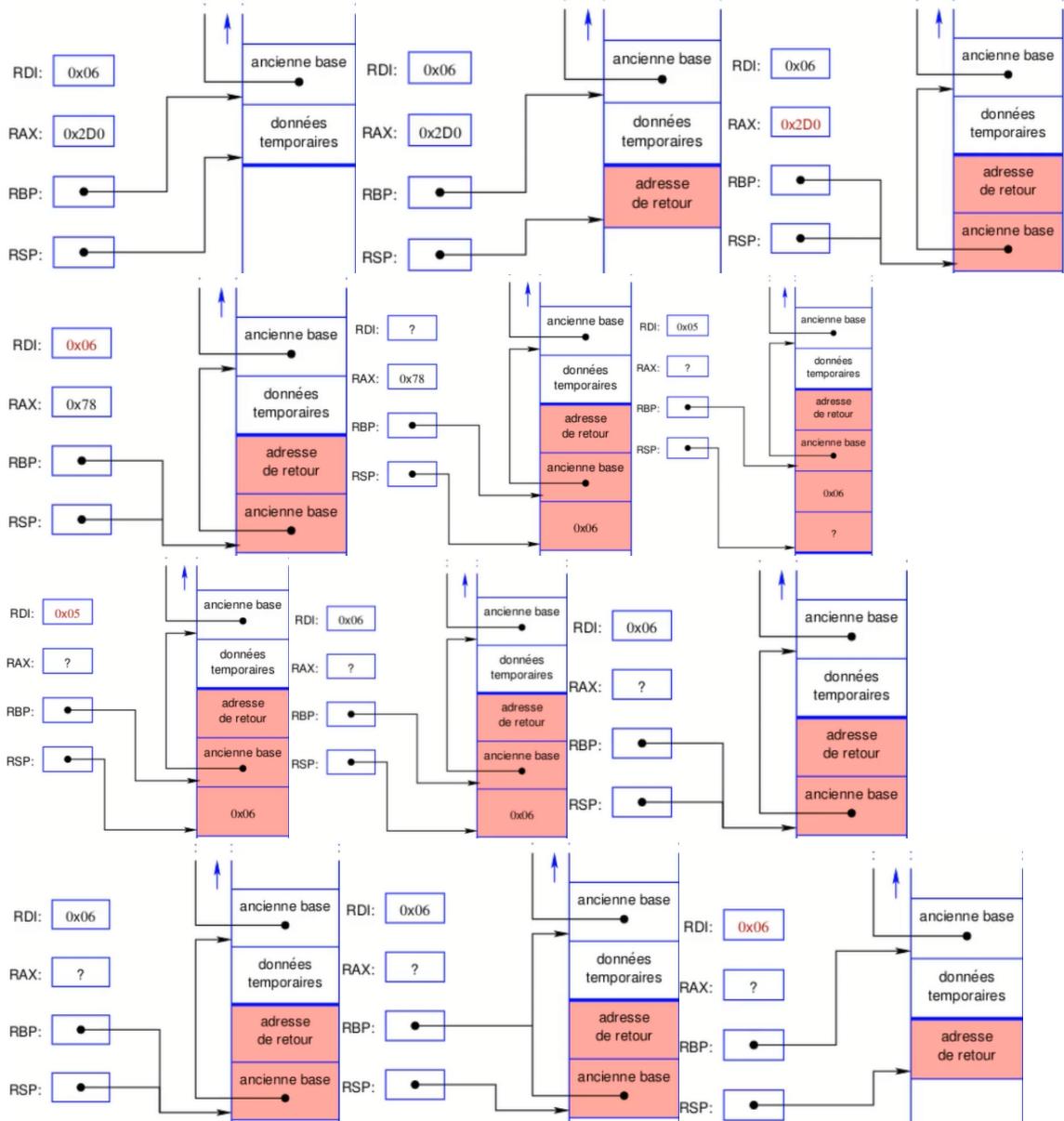
On est sur le point d'appeler factorielle. On a un bout du stack frame (le sommet) qui va appeler factorielle.

Factorielle de 1 : en appelant le code qui a appelé factorielle, on a placé 1 dans RDI (1^{ère} étape convention appel) et on a exécuté l'instruction CALL qui a placé l'adresse de retour sur la pile (1). Il me faut maintenant un pointeur vers l'ancienne base. Il faut donc empiler RBP (qui pointe vers l'ancienne base), on fait donc « PUSH RBP » (2). RBP (qui doit pointer vers le champ de RSP dans le stack frame courant) pointe toujours vers l'ancienne base, ce n'est pas bon, on met donc la valeur de RSP dans RBP (3). Comme il est mis dans le code c, on regarde donc si EDI < 1, ici c'est vrai donc on fait un saut vers « retour_un » et donc on va mettre 1 dans RAX (registre du résultat) (4). CALL sauve l'adresse de retour et RET récupère cette adresse de retour. Ici, le sommet de la pile contient l'ancienne base et non pas l'adresse de retour. On doit donc se débarrasser de l'ancienne base. On va donc retirer « ancienne base (en rouge) » et remettre dans RBP un pointeur vers « ancienne base (en blanc) », on va donc faire une instruction « POP RBP » (5). L'instruction « RET » va donc prendre « adresse de retour » et ensuite, le programme va continuer.

Factorielle de 6 : première chose on fait « PUSH RBP » pour garder l'ancienne base. Ensuite, on met le registre RSP dans RBP. On compare à nouveau EDI à 1, mais ici, EDI = 6 donc on ne fait pas de saut. On exécute ensuite un « PUSH RDI » pour mettre RDI au sommet de la structure. Vu que factorielle 6 = 6.factorielle(5), on va décrémenter RDI grâce à « DEC RDI », RDI vaut maintenant 5. Il faut que le pointeur de pile soit un multiple de 16. De la barre bleu foncé à « 0x06 », il y a 24 octets (8.3), ce n'est donc pas un multiple de 16. On va donc allouer 8 octets supplémentaires que l'on ne va pas utiliser. Si j'enlève 8 à RSP, on obtient 24+8 = 32, ce qui est bien un multiple de 16. Quand on fait « CALL factorielle », on calcule la factorielle de 5, RAX va contenir la factorielle de 5 (120 = 78 en hexa). RDI a maintenant une valeur que l'on ne connaît pas. Pour se débarrasser des 8 octets ajoutés, on va les

NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.

supprimer de RSP. Pour retrouver l'ancienne valeur de RDI, on fait « *POP RDI* » pour retrouver 6 et l'enlever de la structure. Quand on fait « *MUL RDI* », on multiplie RDI par RAX et on met le résultat dans RAX, ici $120 \cdot 6 = 720$ (0x2D0). On fait un saut « *retour* » pour arriver à la fin de la fonction et on doit rétablir la pile en effaçant la sauvegarde de RBP en faisant « *POP RBP* » et on finit par RET.



NE PAS OUBLIER : architecture x86-64 est petit-boutiste et aligner si nécessaire les données mémorisées sur plus d'une cellule.