

Organisation des Ordinateurs

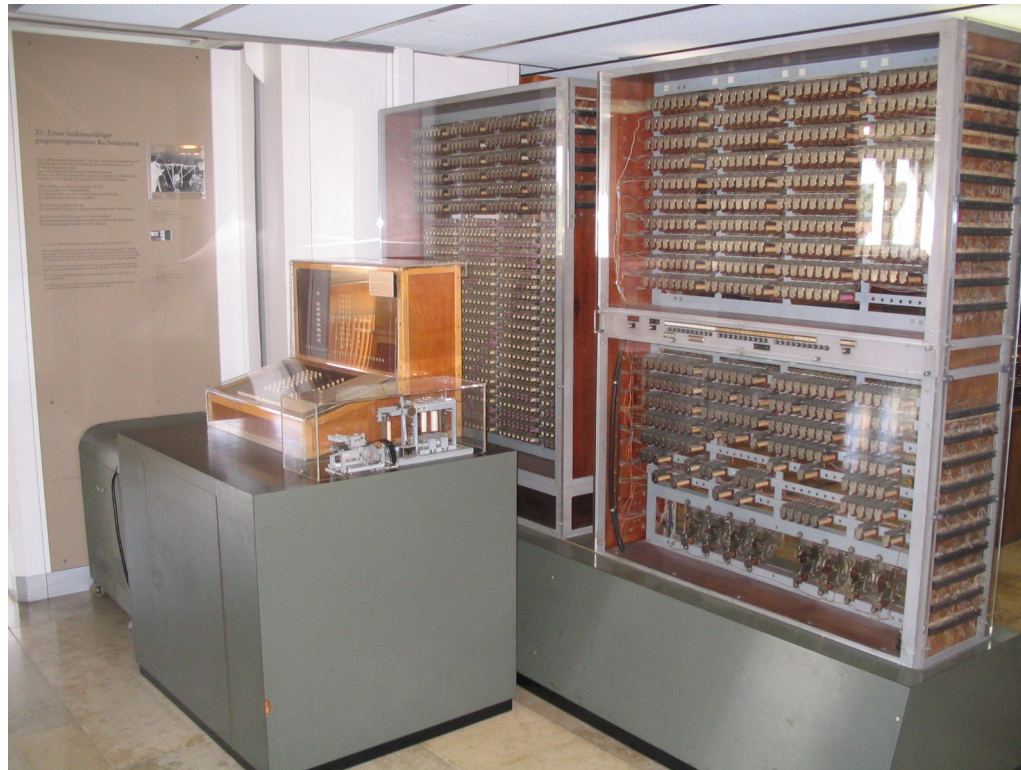
Bernard Boigelot

E-mail : bernard.boigelot@uliege.be
WWW : <https://people.montefiore.uliege.be/boigelot/>
<https://people.montefiore.uliege.be/boigelot/cours/org/>

Chapitre 1

Le traitement de l'information

Les ordinateurs



(Source: [http://en.wikipedia.org/wiki/Z3_\(computer\)](http://en.wikipedia.org/wiki/Z3_(computer)))

Définition:

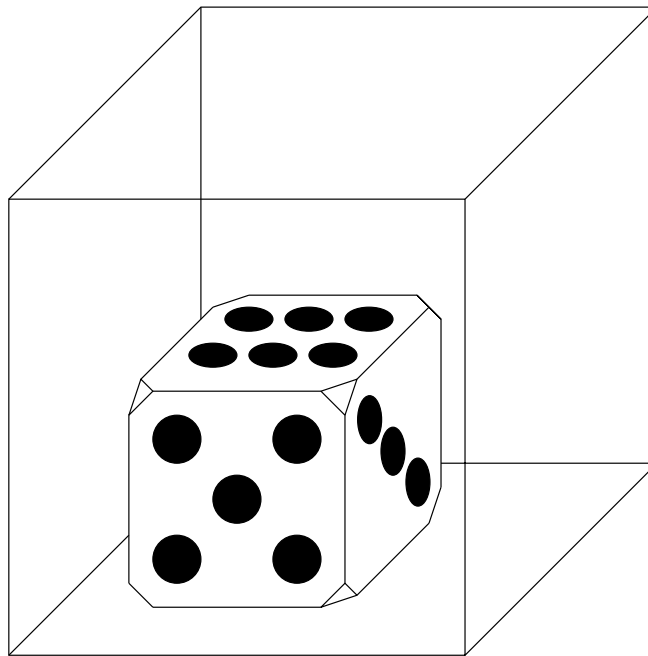
Un ordinateur est une machine capable de **traiter des données**, en suivant un **programme** préétabli.

Objectif du cours: Etudier leurs principes de fonctionnement.

L'information

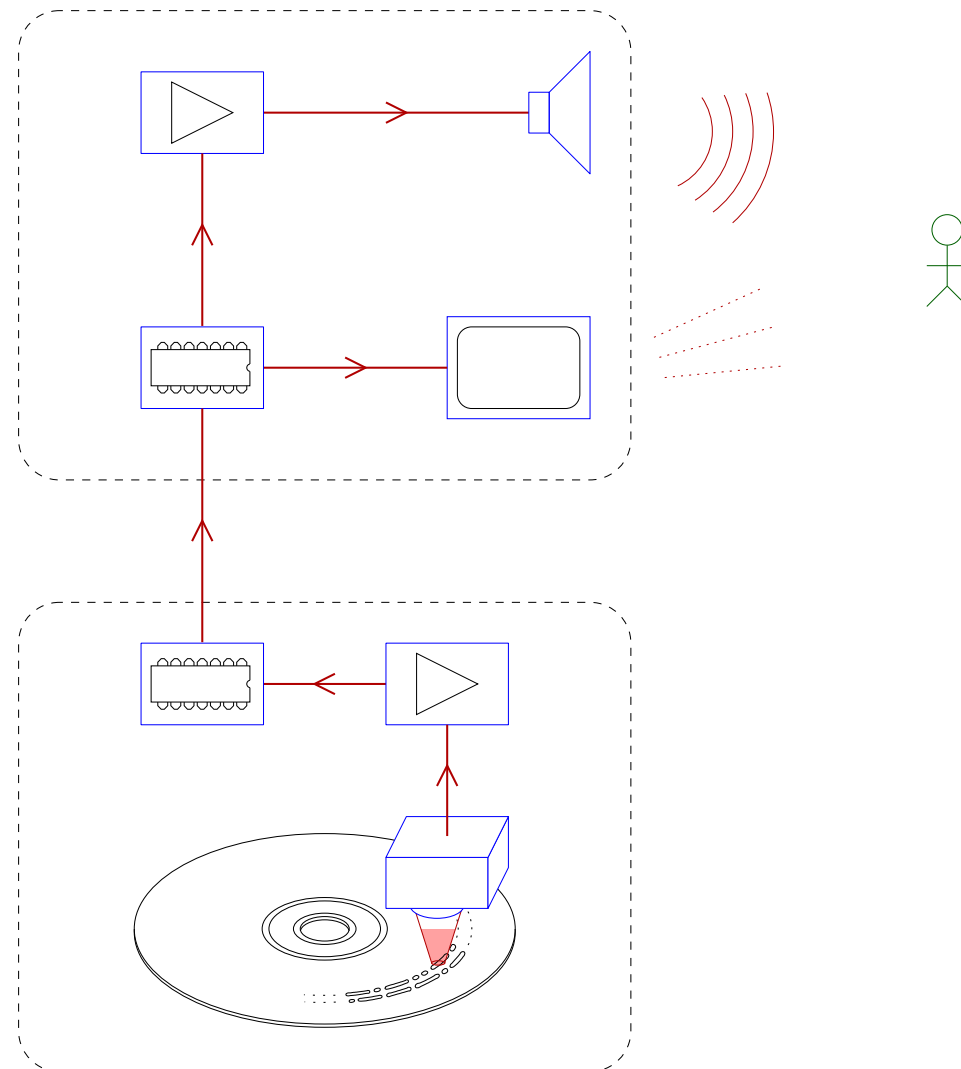
La notion d'information correspond à la connaissance que l'on a de l'état d'un système.

Exemple:



L'information se transmet par l'intermédiaire de **signaux**, qui peuvent prendre des formes variées.

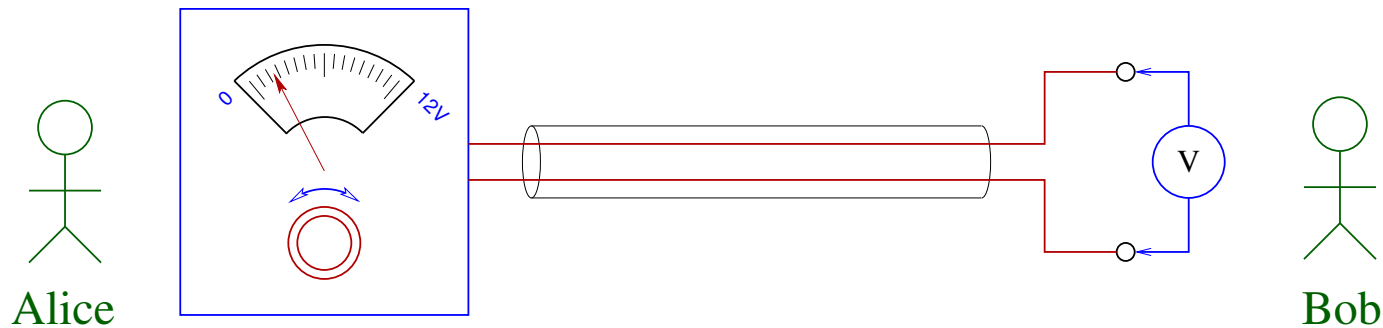
Exemple: Lecteur de DVD.



Les signaux continus

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

Exemple:



Inconvénient: L'information n'est pas **transmise fidèlement**, car la valeur de chaque signal est entachée d'**imprécisions**.

Les signaux discrets

Définition: Un **signal discret** est un signal possédant **un nombre fini** de valeurs nominales.

Avantage: La **transmission fiable** de données est possible malgré la présence d'imprécisions.

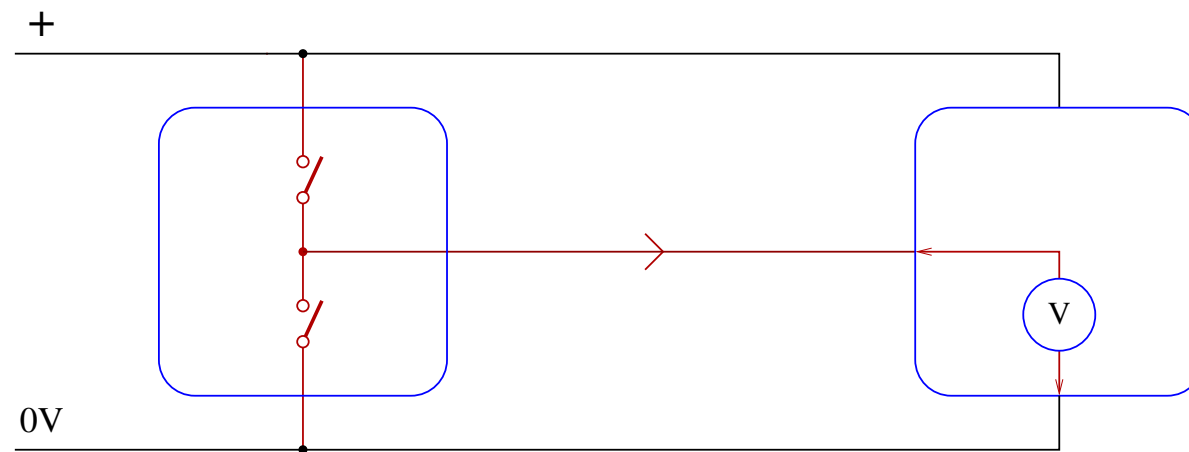
En effet, si l'amplitude des perturbations est suffisamment petite, alors les valeurs transmises peuvent toujours être **correctement identifiées** à leur réception.

Les signaux binaires

Dans les ordinateurs, l'information est transmise, traitée et mémorisée au moyen de signaux discrets **binaires**.

Avantages:

- Ils sont faciles à **générer** et à **décoder**.



- Ils présentent une bonne **robustesse** face aux perturbations.
- Leur analyse est simple grâce à l'**algèbre booléenne**.

La quantité d'information

Comment peut-on **quantifier** l'information transmise par un signal discret?

Propriétés souhaitées:

- Plus la **probabilité** de recevoir une valeur est faible, plus la quantité d'information est élevée.

$$q \left(\text{dice} \right) > q \left(\text{coin} \right)$$

- Lorsqu'on **combine des signaux indépendants**, l'information doit s'additionner.

$$q \left(\text{coin} \text{ coin} \text{ coin} \right) = 3 \times q \left(\text{coin} \right)$$

Définition: La **quantité d'information** transmise par une valeur discrète décodable de façon fiable est égale à

$$\log_2 \frac{1}{p},$$

où p dénote la probabilité que cette valeur soit reçue.

Cette quantité d'information s'exprime en **bits** (binary digits, b).

Par conséquent, la quantité d'information contenue dans un signal pouvant prendre N **valeurs équiprobables** (décodables de façon fiable) vaut

$$\log_2 N.$$

Un bit représente donc la quantité d'information permettant de distinguer fiablement deux valeurs équiprobables.

Exemples

On transmet une **lettre de l'alphabet** au moyen d'un signal de tension: A = 0 V, B = 0,04 V, C = 0,08 V, ..., Y = 0,96 V, Z = 1 V.

Situation 1: Les **26 valeurs** peuvent être fiablement reconnues.

- Si les probabilités de recevoir un E et un Z sont (resp.) égales à 0,17 et 0,0012, alors la quantité d'information transmise par les signaux correspondants vaut (resp.)

$$\log_2 \frac{1}{0,17} \approx 2,56 \text{ bits}$$

et

$$\log_2 \frac{1}{0,0012} \approx 9,70 \text{ bits.}$$

- Si les 26 lettres ont la même probabilité d'être reçues, alors la quantité d'information contenue dans un signal vaut

$$\log_2 26 \approx 4,70 \text{ bits.}$$

Situation 2: On ne peut distinguer que les tensions **supérieures ou inférieures à 0,5 V.**

Si les 2 valeurs sont équiprobables, alors la quantité d'information transmise par un signal vaut

$$\log_2 2 = 1 \text{ bit.}$$

Les unités de quantité d'information

- Un *octet (byte, B)* représente 8 bits d'information.
- Un *nibble* est un demi-octet.
- Les préfixes *K (kilo)*, *M (mega)*, *G (giga)*, *T (tera)*, *P (peta)*, ... signifient suivant le contexte
 - soit 2^{10} , 2^{20} , 2^{30} , 2^{40} , 2^{50} , ...
 - soit 10^3 , 10^6 , 10^9 , 10^{12} , 10^{15} , ...

(En effet $2^{10} = 1024 \approx 1000$.)

Exemple: La capacité d'un disque dur vendu comme contenant 4 TB n'est en réalité que de

$$\frac{4 \cdot 10^{12}}{2^{40}} \approx 3,64 \text{ TB.}$$

Chapitre 2

La représentation des données

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: *False / True, L / H, ...*

Question: A l'aide des seuls symboles 0 et 1, comment peut-on représenter des **données plus complexes**?

Les nombres entiers non signés

Dans la vie quotidienne, on représente les nombres naturels à l'aide de la **notation positionnelle**.

Principes:

- L'ensemble des chiffres est $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- Le **poids** de chaque chiffre est une puissance de 10 qui dépend de sa **position**.

Exemple:

poids :	10^2	10^1	10^0
position :	2	1	0

1	2	3
---	---	---

$$\begin{aligned} & 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ &= 1 \times 100 + 2 \times 10 + 3 \times 1 \\ &= 123. \end{aligned}$$

La représentation binaire

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

- L'ensemble des chiffres est $\{0, 1, \dots, r - 1\}$.
- Le poids du chiffre à la position k est r^k .

Exemple: base $r = 2$ (binaire).

poids: 2^6 2^5 2^4 2^3 2^2 2^1 2^0
position: 6 5 4 3 2 1 0

1	1	1	1	0	1	1
---	---	---	---	---	---	---

$$\begin{aligned} & 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 64 + 32 + 16 + 8 + 2 + 1 \\ &= 123. \end{aligned}$$

Le nombre encodé par la suite de bits $b_{n-1}b_{n-2} \dots b_0$ est donc égal à

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

Note: Les bits b_{n-1} et b_0 sont respectivement appelés **bit de poids fort** et **bit de poids faible**.

Calcul de la représentation d'un nombre

La représentation d'un nombre v peut se calculer grâce aux deux propriétés suivantes:

- Le **bit de poids faible** est égal à 0 si v est pair, et à 1 si v est impair.
- En retirant le bit de poids faible d'une représentation de v , on obtient une représentation de $\left\lfloor \frac{v}{2} \right\rfloor$.

On a donc l'algorithme suivant:

1. Si v est pair, afficher 0. Sinon, afficher 1.
2. Remplacer v par $\left\lfloor \frac{v}{2} \right\rfloor$.
3. Si $v \neq 0$, recommencer à 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**).
- 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: 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
$v = 0$			

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

Les valeurs représentables

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

Les nombres possédant une représentation binaire non signée sur n bits forment donc l'intervalle

$$[0, \dots, 2^n - 1].$$

Pour $n = 8, 16, 32$, on a donc les bornes supérieures 255, 65535 et 4294967295.

La représentation hexadécimale

La notation positionnelle n'est pas limitée aux bases $r = 2$ et $r = 10$. En choisissant $r = 16$, on obtient la représentation **hexadécimale**, très utilisée en informatique.

Avantages: Cette représentation 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:

poids :	16^2	16^1	16^0
position :	2	1	0

4	D	2
---	---	---

$$\begin{aligned} & 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0 \\ &= 4 \times 256 + 13 \times 16 + 2 \times 1 \\ &= 1234. \end{aligned}$$

La conversion hexadécimal ↔ binaire

Un chiffre hexadécimal représente exactement **4 bits** d'information.

Pour convertir un nombre hexadécimal en binaire, il suffit donc de remplacer chaque chiffre par la séquence de 4 bits qui lui correspond. La conversion réciproque est similaire.

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

Exemple: 4D2 ↔ 0100 1101 0010 .

Note: Si la base utilisée pour représenter les nombres n'est pas évidente à déduire du contexte, il faut la préciser explicitement. Cela peut se faire:

- À l'aide d'un **indice**:

$$1234_{10} = 4D2_{16} = 10011010010_2.$$

- Avec un **suffixe**:

$$1234\mathbf{d} = 4D2\mathbf{h} = 10011010010\mathbf{b}.$$

- Avec un **préfixe**:

$$1234 = \mathbf{0x}4D2 = \mathbf{0b}10011010010.$$

L'arithmétique binaire non signée

Le calcul de la **somme de deux nombres entiers signés** peut s'effectuer selon les règles du calcul écrit.

Les tables d'addition binaire sont les suivantes (les **reports** sont entourés):

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

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

L'opération d'addition s'effectue bit par bit, du bit de poids faible vers celui de poids fort.

Exemple: Calcul de la somme $123 + 456 = 579$ sur 10 bits:

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

La multiplication de nombres binaires non signés

Le calcul d'un **produit** s'effectue selon des règles analogues à celle du calcul décimal:

1. Des **produits partiels** sont successivement calculés pour chaque bit du multiplicateur, et convenablement alignés.
2. Ces produits partiels sont ensuite additionnés.

La **table de multiplication binaire** est triviale:

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

Exemple: Calcul du produit $34 \cdot 12 = 408$:

				1	0	0	0	1	0	
×							1	1	0	0
<hr style="border: 0.5px solid blue;"/>										
				0	0	0	0	0	0	0
			0	0	0	0	0	0	0	
		1	0	0	0	1	0			
+	1	0	0	0	1	0				
<hr style="border: 0.5px solid blue;"/>										
	1	1	0	0	1	1	0	0	0	

La représentation des nombres entiers signés

Il existe plusieurs procédés permettant de représenter des nombres entiers **positifs et négatifs**:

- La représentation par **valeur signée**.
- La représentation par **complément à un**.
- La représentation par **complément à deux**.

Ces trois méthodes possèdent des points communs:

- Le signe d'un nombre est représenté par le bit de poids fort (ici appelé **bit de signe**). Celui-ci est égal à
 - 0 pour les nombres positifs.
 - 1 pour les nombres négatifs.
- La représentation d'un nombre positif est toujours identique à sa représentation binaire **non signée** de même taille.

La représentation par valeur signée

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

Exemple: La représentation sur 8 bits du nombre -42 est égale à $\boxed{10101010}$. En effet

- Ce nombre est négatif, donc le bit de signe est égal à $\boxed{1}$.
- La représentation binaire non signée sur 7 bits de $42 = |-42|$ est $\boxed{0101010}$.

Selon ce procédé, le nombre v représenté par le groupe de bits $b_{n-1}b_{n-2} \dots b_0$ est égal à

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

Les valeurs représentables

A l'aide de n bits, la représentation par valeur signée permet d'encoder

- tous les éléments de l'intervalle $[0, \dots, 2^{n-1} - 1]$ (bit de signe égal à 0), et
- tous les éléments de l'intervalle $[-2^{n-1} + 1, \dots, 0]$ (bit de signe égal à 1).

L'ensemble des valeurs représentables est donc l'intervalle

$$[-2^{n-1} + 1, \dots, 2^{n-1} - 1].$$

Remarques:

- Le nombre 0 possède deux représentations distinctes.
- Ce procédé rend difficile le calcul des opérations arithmétiques.

La représentation par complément à un

Principe: La représentation d'un nombre est similaire à sa représentation par valeur signée, mais les bits qui suivent le bit de signe sont **complémentés** (0 est remplacé par 1, et vice-versa).

Exemple: La représentation sur 8 bits du nombre -42 est égale à $\boxed{11010101}$. En effet

- Ce nombre est négatif, donc le bit de signe est égal à $\boxed{1}$.
- La représentation binaire non signée sur 7 bits de $42 = |-42|$ est 0101010, dont le complément est $\boxed{1010101}$.

L'ensemble des nombres représentables à l'aide de n bits est identique à celui de la représentation par **valeur signée**, c'est-à-dire l'intervalle

$$\boxed{[-2^{n-1} + 1, \dots, 2^{n-1} - 1]}.$$

Selon ce procédé, le nombre v représenté par la suite de bits $b_{n-1}b_{n-2} \dots b_0$ vaut

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

En effet,

- Si $v > 0$, on a $b_{n-1} = 0$ et $v = \sum_{i=0}^{n-1} 2^i b_i$.

- Si $v < 0$, on a $b_{n-1} = 1$ et $|v| = \sum_{i=0}^{n-2} 2^i (1 - b_i)$
 $= \sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} 2^i b_i$
 $= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i b_i$
 $= 2^n - 1 - \sum_{i=0}^{n-1} 2^i b_i.$

Remarques:

- Le nombre 0 possède les deux représentations $000\dots 0$ (zéro positif) et $111\dots 1$ (zéro négatif).
- Pour calculer l'opposé d'un nombre, il suffit d'inverser tous les bits de sa représentation.

L'arithmétique par complément à un

Notations: Soit $w = b_{n-1}b_{n-2} \dots b_0$ une suite de bits ($n \geq 1$).

- Le nombre **non signé** représenté par w est noté $[w]_{ns}$.
- Le nombre représenté par w **par complément à un** est noté $[w]_{c_1}$.

Exemples:

$$\begin{aligned} [11010101]_{ns} &= 213 \\ [11010101]_{c_1} &= -42. \end{aligned}$$

Propriété:

- Si $b_{n-1} = 0$, alors $[w]_{ns} = [w]_{c_1}$.
- Si $b_{n-1} = 1$, alors $[w]_{ns} = [w]_{c_1} + 2^n - 1$.

(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.)

L'addition par complément à un

Lorsqu'on calcule la **somme non signée** de deux suites de bits $w = b_{n-1}b_{n-2} \dots b_0$ et $w' = b'_{n-1}b'_{n-2} \dots b'_0$, on obtient un résultat $w'' = b''_{n-1}b''_{n-2} \dots b''_0$ tel que:

- Si aucun report n'est apparu à la position n , alors

$$[w'']_{ns} = [w]_{ns} + [w']_{ns}.$$

- Si un report est apparu à la position n (et a été ignoré), alors

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

Question: Quelle est la relation qui lie $[w]_{c_1}$, $[w']_{c_1}$ et $[w'']_{c_1}$?

Il y a plusieurs cas à considérer:

Cas 1: Si $b_{n-1} = 0$, $b'_{n-1} = 0$ et $b''_{n-1} = 0$:

- $[w]_{ns} = [w]_{c_1}$.
- $[w']_{ns} = [w']_{c_1}$.
- $[w'']_{ns} = [w'']_{c_1}$.
- L'addition n'a produit aucun report à la position n .

On a donc:

$$\begin{aligned} [w'']_{c_1} &= [w'']_{ns} \\ &= [w]_{ns} + [w']_{ns} \\ &= [w]_{c_1} + [w']_{c_1}. \end{aligned}$$

Cas 2: Si $b_{n-1} = 0$, $b'_{n-1} = 0$ et $b''_{n-1} = 1$:

- $[w]_{ns} = [w]_{c_1} \geq 0$.
- $[w']_{ns} = [w']_{c_1} \geq 0$.
- $[w]_{ns} + [w']_{ns} \geq 2^{n-1}$ car $b''_{n-1} = 1$.

La somme $[w]_{c_1} + [w']_{c_1}$ n'est donc pas représentable par complément à un sur n bits.

→ dépassement arithmétique!

Cas 3: Si $b_{n-1} = 0$, $b'_{n-1} = 1$ et $b''_{n-1} = 0$:

- $[w]_{ns} = [w]_{c_1}$.
- $[w']_{ns} = [w']_{c_1} + 2^n - 1$.
- $[w'']_{ns} = [w'']_{c_1}$.
- L'addition a obligatoirement produit un report à la position n .

On a donc:

$$\begin{aligned} [w'']_{c_1} &= [w'']_{ns} \\ &= [w]_{ns} + [w']_{ns} - 2^n \\ &= [w]_{c_1} + [w']_{c_1} - 1. \end{aligned}$$

Cas 4: Si $b_{n-1} = 0$, $b'_{n-1} = 1$ et $b''_{n-1} = 1$:

- $[w]_{ns} = [w]_{c_1}$.
- $[w']_{ns} = [w']_{c_1} + 2^n - 1$.
- $[w'']_{ns} = [w'']_{c_1} + 2^n - 1$.
- L'addition ne peut pas avoir produit un report à la position n .

On a donc:

$$\begin{aligned} [w'']_{c_1} &= [w'']_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^n + 1 \\ &= [w]_{c_1} + [w']_{c_1}. \end{aligned}$$

Cas 5: Si $b_{n-1} = 1$, $b'_{n-1} = 0$ et $b''_{n-1} = 0$:

Equivalent au cas 3, en permutant les deux opérandes w et w' .

Cas 6: Si $b_{n-1} = 1$, $b'_{n-1} = 0$ et $b''_{n-1} = 1$:

Idem vis à vis du cas 4.

Cas 7: Si $b_{n-1} = 1$, $b'_{n-1} = 1$ et $b''_{n-1} = 0$:

- $[w]_{ns} = [w]_{c_1} + 2^n - 1$ et $[w]_{c_1} \leq 0$.
- $[w']_{ns} = [w']_{c_1} + 2^n - 1$ et $[w']_{c_1} \leq 0$.
- $[w]''_{ns} = [w]''_{c_1}$ et $[w]''_{c_1} \geq 0$.
- L'addition a produit un report à la position n .

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

Or $[w]_{ns} + [w']_{ns} = [w]_{c_1} + [w']_{c_1} + 2^{n+1} - 2$.

Donc,

$$\begin{aligned} [w]_{c_1} + [w']_{c_1} &\leq 2^n + 2^{n-1} - 1 - 2^{n+1} + 2 \\ &= -2^{n-1} + 1. \end{aligned}$$

La somme $[w]_{c_1} + [w']_{c_1}$ n'est donc pas représentable sur n bits, sauf dans le cas particulier où elle vaut $-2^{n-1} + 1$.

→ dépassement arithmétique!

Cas 8: Si $b_{n-1} = 1$, $b'_{n-1} = 1$ et $b''_{n-1} = 1$:

- $[w]_{ns} = [w]_{c_1} + 2^n - 1$.
- $[w']_{ns} = [w']_{c_1} + 2^n - 1$.
- $[w]''_{ns} = [w]''_{c_1} + 2^n - 1$.
- L'addition a produit un report à la position n .

On a donc:

$$\begin{aligned} [w]''_{c_1} &= [w]''_{ns} - 2^n + 1 \\ &= [w]_{ns} + [w']_{ns} - 2^{n+1} + 1 \\ &= [w]_{c_1} + [w']_{c_1} - 1. \end{aligned}$$

Résumé

- Si $b_{n-1} = b'_{n-1} \neq b''_{n-1}$: **Dépassement arithmétique.**
- Sinon:
 - Si l'addition a produit un report à la position n :

$$[w'']_{c_1} = [w]_{c_1} + [w']_{c_1} - 1.$$

- Sinon:

$$[w'']_{c_1} = [w]_{c_1} + [w']_{c_1}.$$

Algorithme d'addition par complément à un

Pour calculer la somme de deux nombres représentés par complément à un sur n bits:

1. Additionner les deux suites de bits comme si elles représentaient des nombres **non signés**.
2. Si l'opération précédente produit un report à la position n (ignoré), effectuer une deuxième addition pour **ajouter 1** au résultat.
3. Si le signe des deux opérandes est identique et ne correspond pas à celui du résultat, signaler un **dépassement arithmétique**.

Exemples:

- Calcul de $12 + (-34)$:

			1	1	1					n. sign.	compl. un	diff.
	0	0	0	0	1	1	0	0		12	12	0
+	1	1	0	1	1	1	0	1		221	-34	255
	1	1	1	0	1	0	0	1		233	-22	255

- Calcul de $34 + (-12)$:

			1	1	1			1		n. sign.	compl. un	diff.
	0	0	1	0	0	0	1	0		34	34	0
+	1	1	1	1	0	0	1	1		243	-12	255
	0	0	0	1	0	1	0	1		21	21	0
+	0	0	0	0	0	0	0	1		1	1	0
	0	0	0	1	0	1	1	0		22	22	0

Note: Les étapes 2 et 3 de l'algorithme sont effectuées dans cet ordre pour gérer correctement le cas d'un résultat égal à $-2^{n-1} + 1$:

Exemple: Calcul de $(-120) + (-7)$:

1									n. sign.	compl. un	diff.
	1	0	0	0	0	1	1	1	135	-120	255
+	1	1	1	1	1	0	0	0	248	-7	255
	0	1	1	1	1	1	1	1	127	127	0
+	0	0	0	0	0	0	0	1	1	1	0
	1	0	0	0	0	0	0	0	128	-127	255

La représentation par complément à deux

Idée: Par rapport au complément à un, **décaler d'une unité** la représentation des nombres négatifs permet d'éviter l'étape de correction dans l'algorithme d'addition.

Principes: La représentation d'un nombre v sur n bits est égale

- à la **représentation entière non signée** de v sur n bits si $v \geq 0$.
- à la **représentation (négative) par complément à un** de $v + 1$ sur n bits si $v < 0$.

Exemples:

- La représentation sur 8 bits du nombre -42 est égale à 11010110.
- La représentation sur n bits du nombre -1 est égale à 111...1.

En complément à deux, le nombre v représenté par la suite de bits $b_{n-1}b_{n-2} \dots b_0$ vaut

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

En effet,

- Si $b_{n-1} = 0$: Alors, $v = \sum_{i=0}^{n-1} 2^i b_i$.
- Si $b_{n-1} = 1$: Alors, $b_{n-1}b_{n-2} \dots b_0$ est la représentation de $v + 1$ par complément à un.

On a donc

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

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

L'arithmétique par complément à deux

Notation: On note $[w]_{c_2}$ le nombre dont $w = b_{n-1}b_{n-2} \dots b_0$ (avec $n \geq 1$) est la représentation par complément à deux.

Propriété:

- Si $b_{n-1} = 0$, alors $[w]_{ns} = [w]_{c_2}$.
- Si $b_{n-1} = 1$, alors $[w]_{ns} = [w]_{c_2} + 2^n$.

Par conséquent, on a

$$\boxed{[w]_{ns} =_{2^n} [w]_{c_2}},$$

où “ $=_k$ ” désigne l'égalité modulo k .

Notes:

- L'ensemble des nombres représentables par complément à deux sur n bits forme l'intervalle

$$[-2^{n-1}, \dots, 2^{n-1} - 1] .$$

- Le nombre zéro possède une seule représentation $000 \dots 0$.

Quelques propriétés utiles

- On peut étendre la représentation d'un nombre vers davantage de bits en répétant son bit de signe.

Considérons la représentation $w = b_{n-1}b_{n-2} \dots b_0$.

- Si $b_{n-1} = 0$, la propriété est évidente.
- Si $b_{n-1} = 1$, alors le nombre représenté par $11b_{n-2}b_{n-3} \dots b_0$ vaut

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

qui est bien la valeur encodée par w .

- *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

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

On a donc

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

qui est bien l'opposé du nombre représenté par w .

L'addition par complément à deux

Lorsqu'on calcule la **somme non signée** de deux suites de bits $w = b_{n-1}b_{n-2} \dots b_0$ et $w' = b'_{n-1}b'_{n-2} \dots b'_0$, on obtient un résultat $w'' = b''_{n-1}b''_{n-2} \dots b''_0$ tel que

$$[w'']_{ns} = 2^n [w]_{ns} + [w']_{ns}.$$

Par ailleurs, on a

$$\begin{aligned} [w]_{c_2} &= 2^n [w]_{ns}, \\ [w']_{c_2} &= 2^n [w']_{ns} \text{ et} \\ [w'']_{c_2} &= 2^n [w'']_{ns}. \end{aligned}$$

On en déduit

$$[w'']_{c_2} = 2^n [w]_{c_2} + [w']_{c_2},$$

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

Exemples:

- Calcul de $12 + (-34)$:

			1	1	1								
										n. sign.	compl. deux	diff.	
	0	0	0	0	1	1	0	0		12	12	0	
+	1	1	0	1	1	1	1	0		222	-34	256	
	1	1	1	0	1	0	1	0		234	-22	256	

- Calcul de $34 + (-12)$:

			1	1	1								
										n. sign.	compl. deux	diff.	
	0	0	1	0	0	0	1	0		34	34	0	
+	1	1	1	1	0	1	0	0		244	-12	256	
	0	0	0	1	0	1	1	0		22	22	0	

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 en répétant le bit de signe).

Exemples

- Calcul de $(-12).34$ sur 12 bits:

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

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

			1	1	1	1	1	1	1	1	0	1	0	0
	×		1	1	1	1	1	1	0	1	1	1	1	0
<hr/>														
				1		1								
1	1		1	1	1	1		1						
			0	0	0	0	0	0	0	0	0	0	0	0
			1	1	1	1	1	1	1	0	1	0	0	
			1	1	1	1	1	1	0	1	0	0		
			1	1	1	1	1	0	1	0	0			
			0	0	0	0	0	0	0					
			1	1	0	1	0	0						
			1	0	1	0	0							
			0	1	0	0								
			1	0	0									
			0	0										
		+	0											
<hr/>														
			0	0	0	1	1	0	0	1	1	0	0	0

Récapitulatif

Le tableau suivant reprend les différentes représentations des nombres entiers sur 4 bits:

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

La représentation des nombres réels

Problème: Pour représenter un réel arbitraire (même borné), une **quantité infinie d'information** est nécessaire.

Solution: Les représentations informatiques des réels seront **approximées**.

Conséquences:

- Les opérations manipulant les réels sont inévitablement imprécises.
- Les opérations arithmétiques augmentent en général le degré d'imprécision.
- Il faut tenir compte de cette imprécision lorsqu'on teste, par exemple, **l'égalité de nombres réels**.

La représentation en virgule fixe

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

Exemple (base 10):

poids: 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3}
position: 2 1 0 -1 -2 -3

1	2	3	4	5	6
---	---	---	---	---	---

,

$$\begin{aligned} & 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3} \\ &= 1 \times 100 + 2 \times 10 + 3 \times 1 + 4 \times \frac{1}{10} + 5 \times \frac{1}{100} + 6 \times \frac{1}{1000} \\ &= 123,456. \end{aligned}$$

La virgule fixe en binaire

Exemple:

poids: 2^4 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3}
position: 4 3 2 1 0 -1 -2 -3

0	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

,

$$\begin{aligned} & 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} \\ &= 8 + 2 + \frac{1}{2} + \frac{1}{4} \\ &= 10,75. \end{aligned}$$

Propriété: S'il y a k bits après le séparateur, alors 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, et l'on a bien

$$10,75 = \frac{1}{8} \cdot 86.$$

Les nombres signés en virgule fixe

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

Le choix du procédé utilisé pour représenter les entiers reste libre. En pratique, 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,

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

L'addition en virgule fixe

Procédure:

1. On **décale** les opérandes de façon à faire coïncider leurs positions.

Ce décalage s'effectue toujours **vers la droite**, et peut conduire à perdre les bits les moins significatifs des représentations.

2. On additionne les représentations alignées à l'aide du même algorithme que pour les entiers non signés.

Exemple: Calcul sur 8 bits de 5,5 (2 bits après la virgule) + (-5,625) (4 bits après la virgule):

5,5 :

0	0	0	1	0	1	1	0
---	---	---	---	---	---	---	---

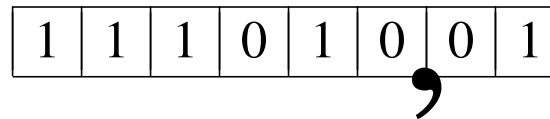
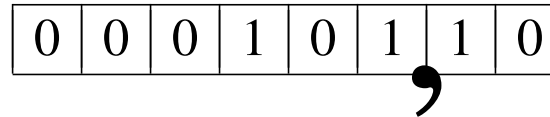
,

-5,625 :

1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---

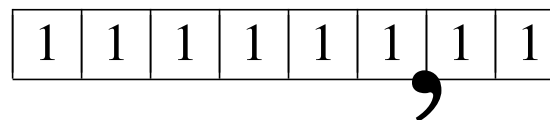
,

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 $-0,25$.

La virgule flottante

Problème: La virgule fixe n'est pas adaptée aux applications où la **grandeur des nombres** représentés est très variable.

Exemple: Masse d'un corps exprimée en kg:

- ≈ 40 chiffres **après la virgule** en physique des particules.
- ≈ 40 chiffres **avant la virgule** en astronomie.

Solution: Dissocier la représentation des **chiffres significatifs** d'un nombre de celle de la grandeur de celui-ci. Un nombre réel v sera exprimé sous la forme

$$v = m \cdot r^e,$$

où

- r est la base,
- m est la **mantisse** (en virgule fixe),
- e est l'**exposant** (entier).

Exemple: Masse d'un électron:

$$\approx 9,109 \cdot 10^{-31} \text{ kg}$$

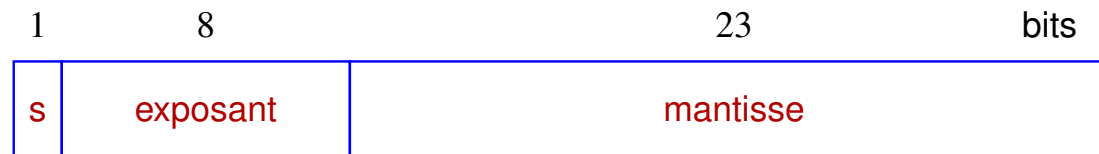
Propriétés:

- La base est égale à 10 pour la notation scientifique usuelle, et à 2 pour les représentations informatiques.
- Les valeurs possibles de l'exposant déterminent l'intervalle des valeurs représentables.
- Le nombre de bits choisi pour représenter la mantisse caractérise la précision avec laquelle les nombres sont représentés.

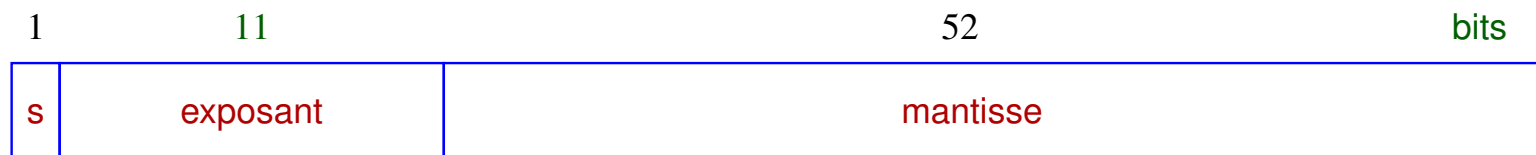
Le standard IEEE 754

Ce standard, très utilisé, définit plusieurs procédés de représentation, dont

- la **simple précision**:



- la **double précision**:



Le champ s est un **bit de signe**. Comme dans le cas des entiers, il vaut 0 pour les nombres positifs et 1 pour les nombres négatifs.

L'encodage de l'exposant

L'exposant d'un nombre est représenté de la façon suivante:

- **Simple précision:** Un exposant e est encodé par la représentation entière non signée sur 8 bits du nombre $e + 127$.

L'intervalle des exposants représentables est donc

$$[-127, \dots, 128].$$

- **Double précision:** Un exposant e est encodé par la représentation entière non signée sur 11 bits du nombre $e + 1023$.

L'intervalle des exposants représentables est donc

$$[-1023, \dots, 1024].$$

L'encodage de la mantisse

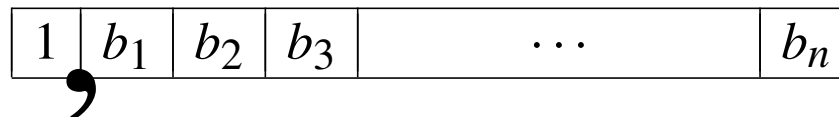
Le procédé d'encodage de la **mantisse** diffère suivant la valeur de l'exposant.

Premier cas: L'exposant n'est pas égal à une valeur extrême (-127 ou 128 pour la simple précision, -1023 ou 1024 pour la double précision).

On dit alors que la mantisse est **normalisée**.

Dans ce cas, la mantisse m représentée par $b_1 b_2 \dots b_n$ (avec $n = 23$ pour la simple précision et $n = 52$ pour la double précision) vaut

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



Corollaire: Pour une mantisse normalisée m , on a $1 \leq |m| < 2$.

Exemple

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

- Ce nombre est **néglatif**, donc le bit de signe est égal à $\boxed{1}$.
- 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 $\boxed{10000001}$.
- On a

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

La mantisse est donc représentée par la suite de bits

$$\boxed{1110000000000000000000000000}.$$

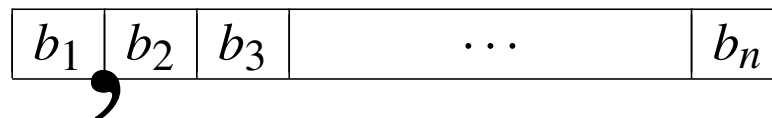
Les mantisses dénormalisées

Deuxième cas: L'exposant est égal à sa valeur minimale (-127 pour la simple précision et -1023 pour la double précision).

On dit alors que la mantisse est **dénormalisée**.

Dans ce cas, la mantisse m représentée par $b_1 b_2 \dots b_n$ vaut

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



Corollaire: Pour une mantisse dénormalisée m , on a $0 \leq |m| < 2$.

Exemple

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

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

- 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

$\boxed{00000000000000001000000000}$.

L'utilité des mantisses dénormalisées

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

Cas particulier: Représentation du nombre zéro:

- La mantisse est nécessairement dénormalisée. L'exposant prend donc sa **plus petite valeur** possible, et se représente $\boxed{000 \dots 0}$.
- La mantisse est égale à 0, et se représente $\boxed{000 \dots 0}$.
- Le bit de signe est quelconque.

Il y a donc **deux représentations** de zéro:

- $\boxed{000 \dots 0}$ (zéro positif), et
- $\boxed{100 \dots 0}$ (zéro négatif).

Les valeurs spéciales

Troisième cas: L'exposant est égal à sa valeur maximale (128 pour la simple précision et 1024 pour la double précision).

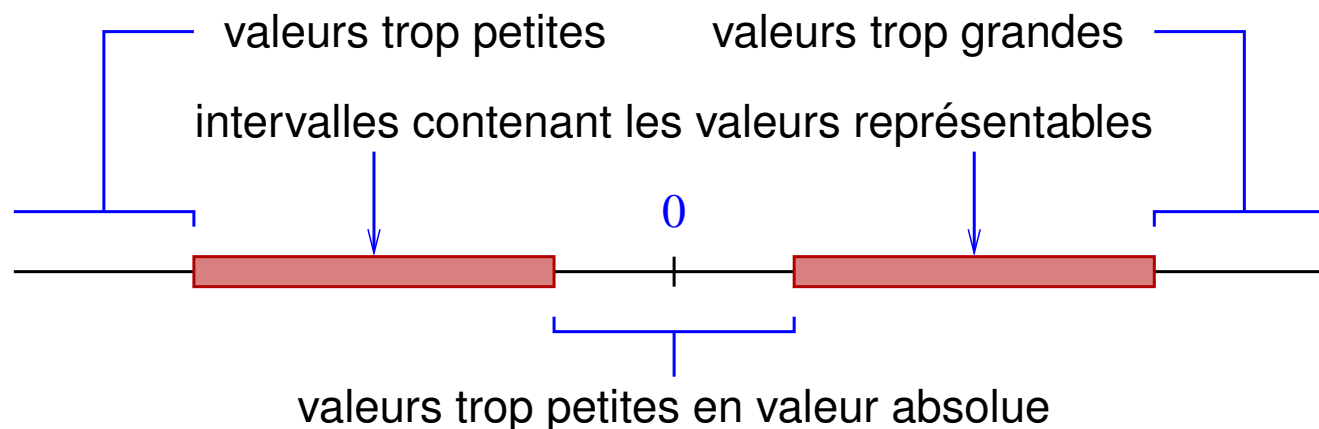
Cette situation sert à encoder des **valeurs spéciales**, qui représentent des résultats qui ne correspondent pas à un nombre réel:

- **Si tous les bits de la mantisse sont égaux à 0:** La représentation indique un **dépassement**
 - vers les valeurs positives si le bit de signe est 0 (**infini positif**).
 - vers les valeurs négatives si le bit de signe est 1 (**infini négatif**).
- **Si au moins un bit de la mantisse est égal à 1:** La représentation correspond à une **valeur indéfinie**: NaN (*Not a Number*).

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 $2 - 2^{-23}$.
→ $max_v \approx 3,403 \cdot 10^{38}$.
- **Double précision:** L'exposant est égal à 1023 et la mantisse à $2 - 2^{-52}$.
→ $max_v \approx 1,798 \cdot 10^{308}$.

Plus petite valeur strictement positive représentable min_v :

- **Simple précision:** L'exposant est égal à -127 et la mantisse à 2^{-22} .
→ $min_v \approx 1,401 \cdot 10^{-45}$.
- **Double précision:** L'exposant est égal à -1023 et la mantisse à 2^{-51} .
→ $min_v \approx 4,941 \cdot 10^{-324}$.

L'addition en virgule flottante

L'addition de deux nombres $v_1 = m_1 .2^{e_1}$ et $v_2 = m_2 .2^{e_2}$ 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** $m .2^{e'_1}$ de manière à obtenir une mantisse normalisée ou dénormalisée.

Note: Cette opération peut conduire à détecter un dépassement.

La multiplication en virgule flottante

La multiplication de deux nombres $v_1 = m_1 .2^{e_1}$ et $v_2 = m_2 .2^{e_2}$ 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 $m = m_1 . m_2$ en virgule fixe.
4. On **normalise** si nécessaire le résultat $m .2^e$, de la même façon que pour l'addition.

La représentation de textes

Il existe plusieurs standards d'encodage des caractères alphanumériques.

Le code ASCII

Ce standard 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, ..., 127]**.
- Les codes **de 0x00 à 0x1F** représentent des caractères de contrôle. Leur interprétation peut dépendre du système utilisé.

- 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**.

Table des caractères imprimables ASCII:

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		

Le standard ISO 8859-1

Les ordinateurs modernes 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 diffèrent 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.

Unicode

L'écriture de certaines langues nécessite **plus de 256 symboles** (p.ex.: chinois, japonais, coréen, ...).

Le standard **Unicode** a été introduit afin d'unifier la représentation de tous les systèmes d'écriture actuels et historiques.

Propriétés:

- La version actuelle d'Unicode (16.0.0, septembre 2024) définit **154998 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 est écrit en hexadécimal.

Exemple: le symbole “€” correspond à $U+20AC$.

La compression UTF-8

L'inconvénient d'Unicode est que cet encodage est **inefficace** lorsque la majorité des caractères d'un texte sont représentables en ASCII ou en ISO 8859-1.

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

$$\boxed{0b_6b_5 \dots b_0},$$

où $b_6b_5 \dots b_0$ est l'encodage binaire non signé de k .

- Si $k \in [0x80, 0x7FF]$: Le caractère est représenté par les deux octets

$$\boxed{110b_{10}b_9 \dots b_6} \boxed{10b_5b_4 \dots b_0},$$

où $b_{10}b_9 \dots b_0$ est l'encodage binaire non signé de k .

- Si $k \in [0x800, 0xFFFF]$: Le caractère est représenté par les trois octets

$$\boxed{1110b_{15}b_{14}b_{13}b_{12}} \boxed{10b_{11}b_{10}\dots b_6} \boxed{10b_5b_4\dots b_0},$$

où $b_{15}b_{14}\dots b_0$ est l'encodage binaire non signé de k .

- Si $k \in [0x10000, 0x10FFFF]$: Le caractère est représenté par les quatre octets

$$\boxed{11110b_{20}b_{19}b_{18}} \boxed{10b_{17}b_{16}\dots b_{12}} \boxed{10b_{11}b_{10}\dots b_6} \boxed{10b_5b_4\dots b_0},$$

où $b_{20}b_{19}\dots b_0$ est l'encodage binaire non signé de k .

Exemple: Pour **U+20AC** (“€”), on a $k = 0x20AC \in [0x800, 0xFFFF]$. En binaire, $0x20AC$ s'écrit $0010\ 0000\ 1010\ 1100$. Ce symbole est donc représenté par les trois octets

$$\boxed{1110\ 0010} \boxed{1000\ 0010} \boxed{1010\ 1100},$$

c'est-à-dire $\boxed{E2}\ \boxed{82}\ \boxed{AC}$ en hexadécimal.

Chapitre 3

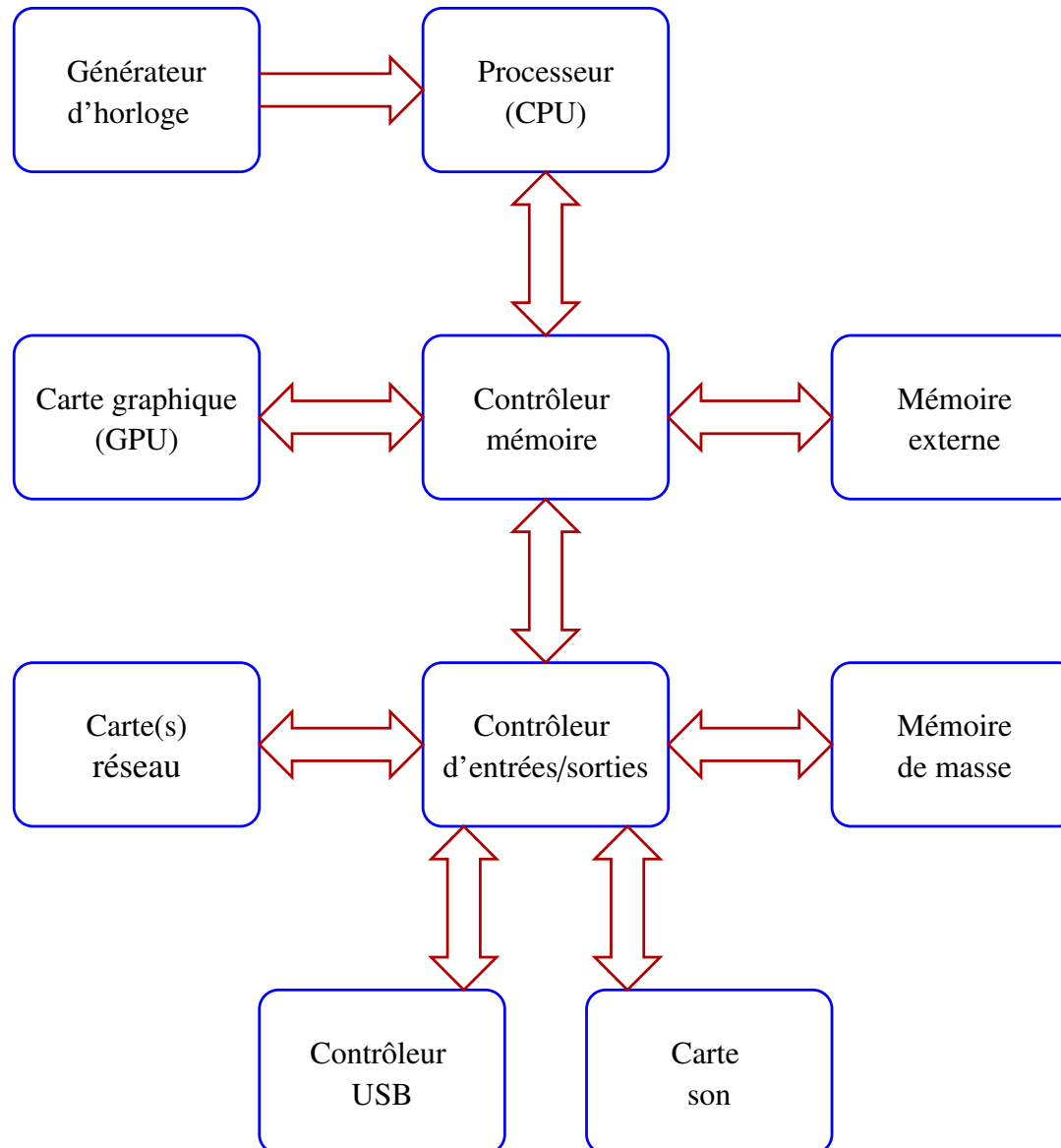
La structure d'un ordinateur

Les composants d'un ordinateur

Un ordinateur contient:

- un ou plusieurs **processeur(s)** (*Central Processing Unit, CPU*), exécutant les programmes.
- de la **mémoire** pour retenir les programmes et les données.
- un générateur d'**horloge** imposant le rythme d'exécution des instructions.
- un ensemble de **périphériques**: carte graphique (*Graphics Processing Unit, GPU*), carte(s) réseau, carte son, ...
- des **contrôleurs** gérant les flux de données.
- des **bus** de communication.

Exemple d'organisation



La mémoire vive

Le terme **mémoire** est très général, et désigne tous les composants capables de retenir de l'information.

La **mémoire vive** (*Random-Access Memory, RAM*) est un type de mémoire pour lequel

- la **consultation** et la **modification** de n'importe quelle partie de son contenu sont possibles de façon illimitée.
- Son contenu est préservé tant que l'ordinateur reste **sous tension**.
- le **taux de transfert** (nombre d'octets lus ou écrits par seconde) est élevé.
- la **latence** (délai avant la fin d'une opération) est faible.

Les utilisations de la mémoire vive

Dans un ordinateur, la mémoire vive est présente à plusieurs endroits:

- en tant que **mémoire externe**, pour retenir des données et des programmes.
- dans le processeur lui-même, en tant que **registres**.
- comme **mémoire cache** pour accélérer les accès du processeur à la mémoire externe.
- dans les **périphériques**.

Deux technologies sont principalement employées:

- Mémoire **statique** (*SRAM*): L'information est retenue par des boucles de *feedback*.
- Mémoire **dynamique** (*DRAM*): L'information est représentée par la charge de condensateurs.

La mémoire morte

La **mémoire morte** (*Read-Only Memory, ROM*) est un type de mémoire dont le contenu ne peut pas être modifié pendant son fonctionnement normal.

Elle est employée pour mémoriser les données et les programmes qui ne doivent **pas changer** au cours de la vie du système, par exemple:

- le **programme de démarrage** d'un ordinateur personnel.
- les polices de caractères d'une **console**.
- le logiciel d'un **système embarqué**.

Les technologies de mémoire morte

- Certains composants ont un contenu **fixé à la fabrication**.
- Les composants **OTP** (*One-Time Programmable*) sont programmables une seule fois.
- Les **EPROM** (*Erasable Programmable ROM*) et **EEPROM** (*Electrically Erasable PROM*) peuvent être **reprogrammés**, grâce à un mécanisme capable d'effacer leur contenu.
- La mémoire **Flash** est similaire à l'**EEPROM**, mais implémente un mécanisme d'effaçage plus flexible.

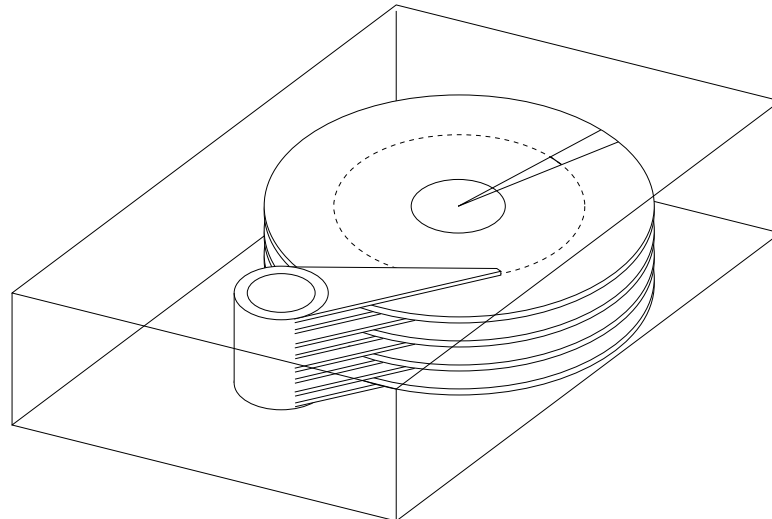
Dans les deux derniers cas, l'opération de reprogrammation est relativement **lente**, et ne peut être effectuée qu'un nombre limité de fois.

La mémoire de masse

La **mémoire de masse** sert à retenir les données et les programmes qui doivent être **préservés** lorsque l'ordinateur est éteint, et qui peuvent potentiellement être modifiés.

Les deux principales technologies actuellement utilisées sont:

- les **disques durs** (*Hard-Disk Drives, HDDs*).



- la **mémoire flash** (*Solid-State Drives, SSDs*).

L'adressage

La mémoire vive et la mémoire morte sont organisées de la façon suivante:

- Les données sont mémorisées dans des **cellules** de taille fixe.
- Chaque cellule est identifiée par son **adresse**.
- L'ensemble des adresses possibles forme l'**espace d'adressage**.

Exemple: Architecture x86-64 (PC):

- Chaque cellule contient **8 bits**.
- L'adressage s'effectue sur **64 bits**. L'espace d'adressage correspond donc à l'intervalle

$$[0, 2^{64} - 1].$$

La notion d'adresse est liée au concept de **pointeur** rencontré en programmation.

Exemple:

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

La cellule d'adresse 0x100 contient l'octet 0x12. On dit alors que 0x100 **pointe vers** 0x12.

Le stockage de données sur plus d'une cellule

Si une donnée doit être mémorisée sur plus d'une cellule, on la découpe en blocs placés dans des **cellules consécutives** de la mémoire.

Il y a deux façon de le faire. Les cellules d'adresse croissante énumèrent les blocs

- depuis le poids faible vers le poids fort: représentation **petit-boutiste** (*little-endian*).
- depuis le poids fort vers le poids faible: représentation **gros-boutiste** (*big-endian*).

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

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

Représentation petit-boutiste

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

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** (p.ex., 32, 64 ou 128 bits).

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 (p.ex., MIPS), interdisent les transferts de données non alignées. Pour d'autres (p.ex., x86-64), de tels transferts sont possibles, mais sont **inefficaces**.

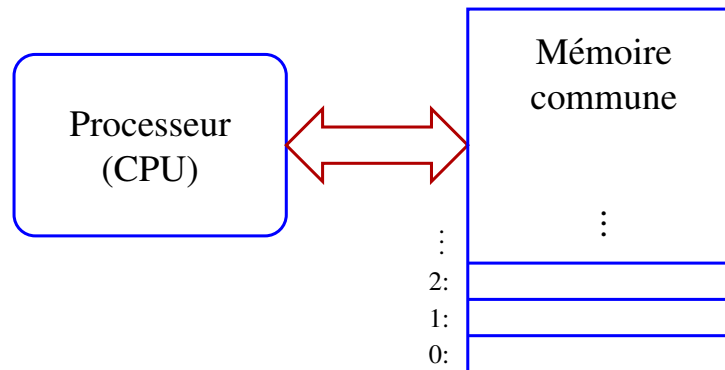
Le processeur

Le processeur est le composant de l'ordinateur responsable de l'**exécution des programmes**.

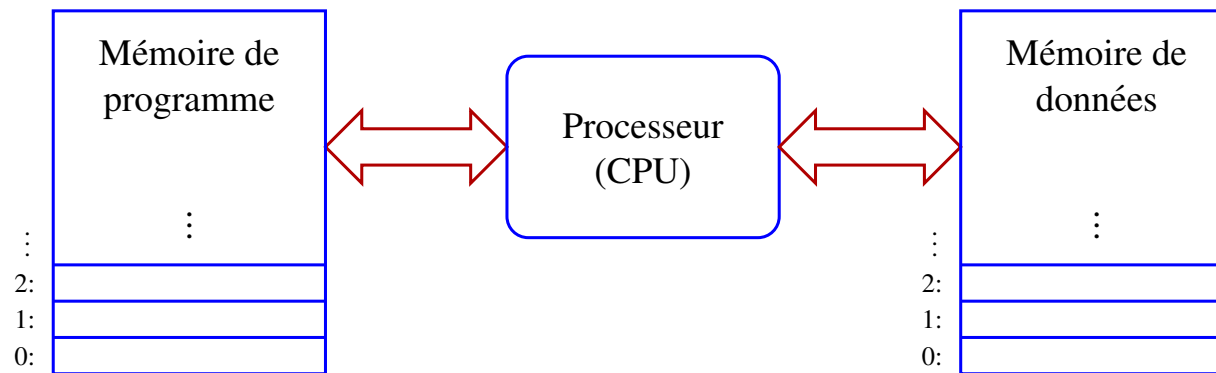
Les programmes que le processeur peut traiter doivent être exprimés en **code machine**.

Deux modèles d'architecture existent:

- Modèle **Von Neumann**: Les programmes et les données partagent le même espace d'adressage:

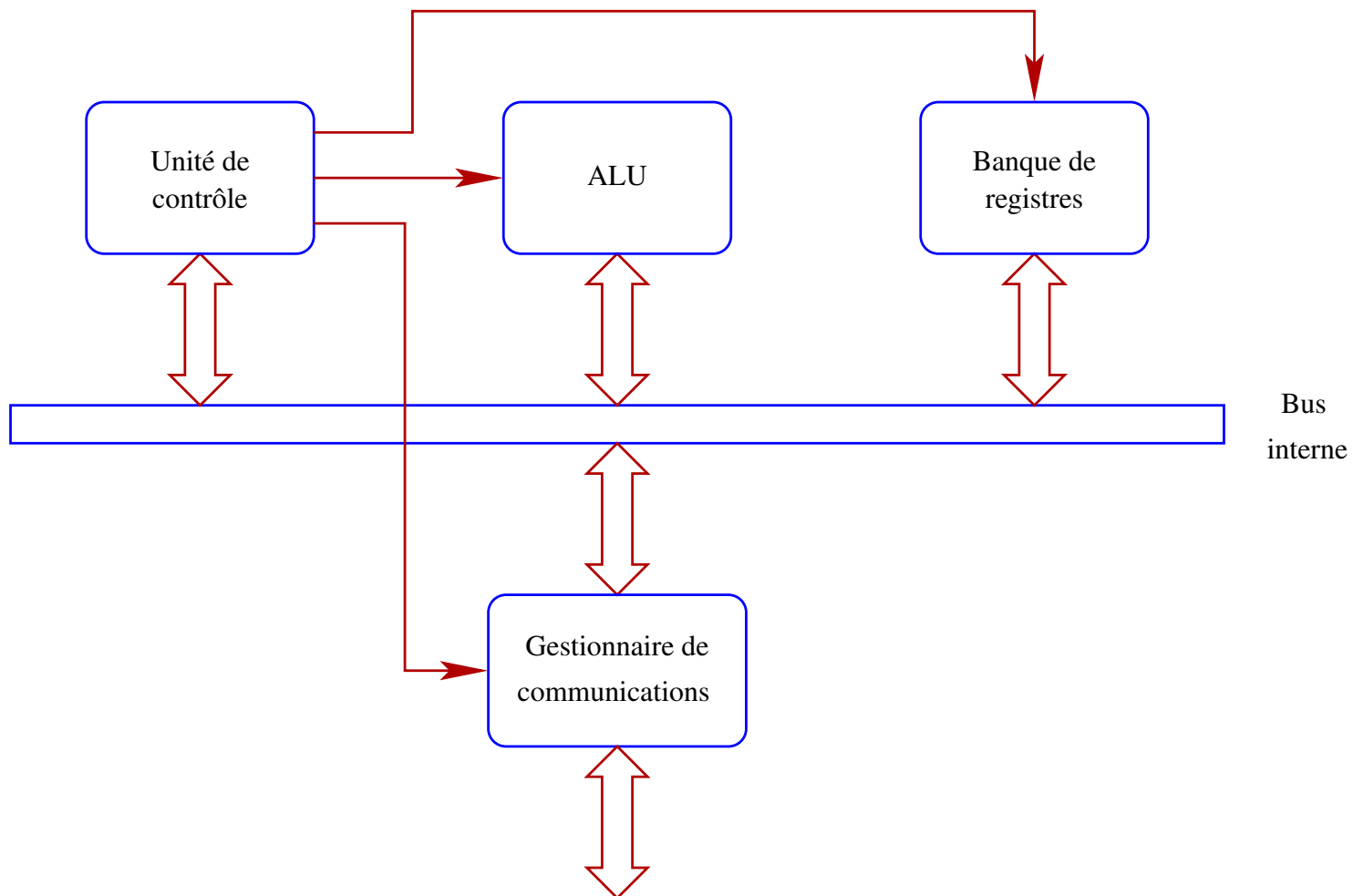


- Modèle **Harvard**: Les programmes et les données sont placés dans des mémoires séparées:



La structure d'un processeur

Organisation typique d'un processeur (à un niveau d'abstraction élevé):



- La **banque de registres** est une petite quantité de mémoire vive, utilisée comme espace de travail.

L'ensemble des registres disponibles dépend de l'architecture du processeur.

- L'**unité arithmétique et logique (Arithmetic Logic Unit, ALU)** est le composant chargé de traiter l'information.

Selon l'architecture, les opérations qu'il peut effectuer peuvent inclure

- les opérations arithmétiques sur les **nombres entiers**: addition, soustraction, multiplication, division,
- le traitement de **nombres réels** (*Floating-Point Unit, FPU*): addition, soustraction, multiplication, division, racine carrée, logarithmes, fonctions trigonométriques, . . .
- les opérations **logiques** (opérateurs booléens, décalages, manipulation de bits).

- Le **bus interne** est un canal de communication entre les composants du processeur.

Sa taille (habituellement 8, 16, 32 ou 64 bits) est une caractéristique importante de l'architecture.

- Le **gestionnaire de communications** relie le bus interne à l'interface extérieure du processeur.

Il est notamment responsable de gérer 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**.

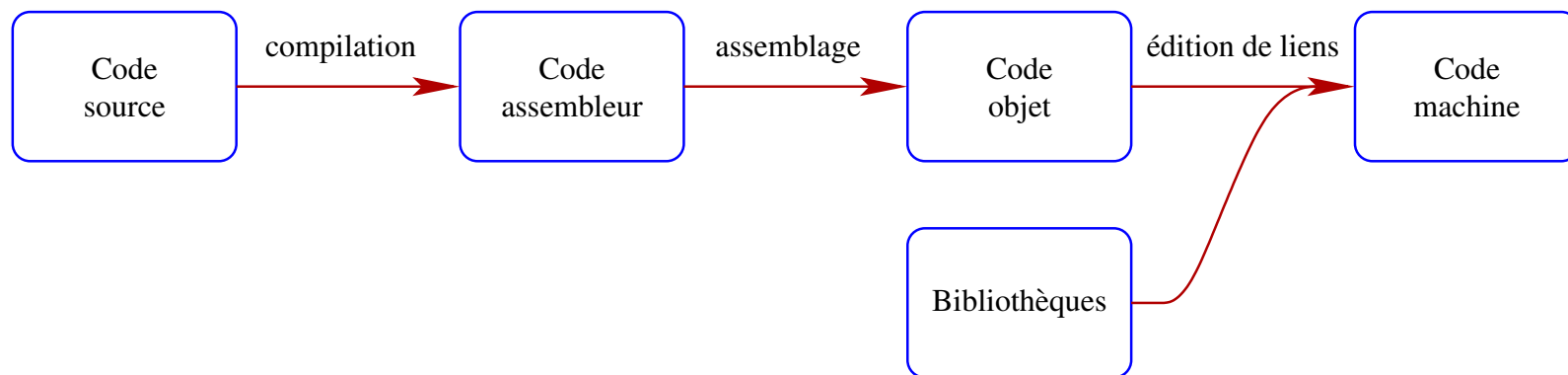
Le jeu d'instructions disponibles est défini par l'architecture.

Le code machine

Les programmes rédigés dans un **langage de programmation** tel que C, C++, Java ou Python ne peuvent pas être directement exécutés par le processeur; ils doivent être préalablement traduits en **code machine**.

Deux mécanismes sont possibles:

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



La 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 (p.ex., des sous-routines), ainsi qu'être fragmenté en plusieurs modules.

3. **Édition de liens**: Le code objet est combiné avec du code issu de **bibliothèques (librairies)** pour obtenir le **code machine** exécutable.

Les registres de contrôle

Pour exécuter les instructions, l'**unité de contrôle** du processeur gère deux registres:

- Le **registre d'instruction** (*Instruction Register, IR*) contient le code de l'instruction (**opcode**) en cours d'exécution.
- Le **compteur de programme** (*Program Counter, PC*, ou *Instruction Pointer, RIP* pour x86-64) contient l'adresse en mémoire de la prochaine instruction à exécuter.

L'exécution des instructions

L'unité de contrôle effectue les opérations suivantes, au rythme du **signal d'horloge**:

1. **Charger** dans IR l'opcode pointé par PC.
2. **Décoder** la valeur de IR.
3. **Exécuter** l'instruction correspondant en contrôlant les autres composants du processeur (banque de registres, ALU, ...).

Si l'instruction possède des **opérandes**, leur lecture et leur décodage font partie de cette étape.

Mettre **PC** à jour de façon à le faire pointer vers l'instruction suivante.

4. **Recommencer** à l'étape 1.

Illustration

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

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

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 (égal à 0xD8) est chargé et décodé.

Dans le cas présent, les opérandes signifient que l'addition porte sur deux registres appelés EAX et EBX, et que le résultat doit être écrit dans EAX.

4. RIP est **incrémenté** de deux unités, pour le faire pointer vers l'instruction suivante (en 0x1002).

5. L'opération d'addition est **exécutée**:

- (a) L'unité de contrôle demande à la **banque de registres** de transférer le contenu de EAX et EBX vers l'ALU via le bus.
- (b) L'**ALU** est pilotée 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**.

Remarques:

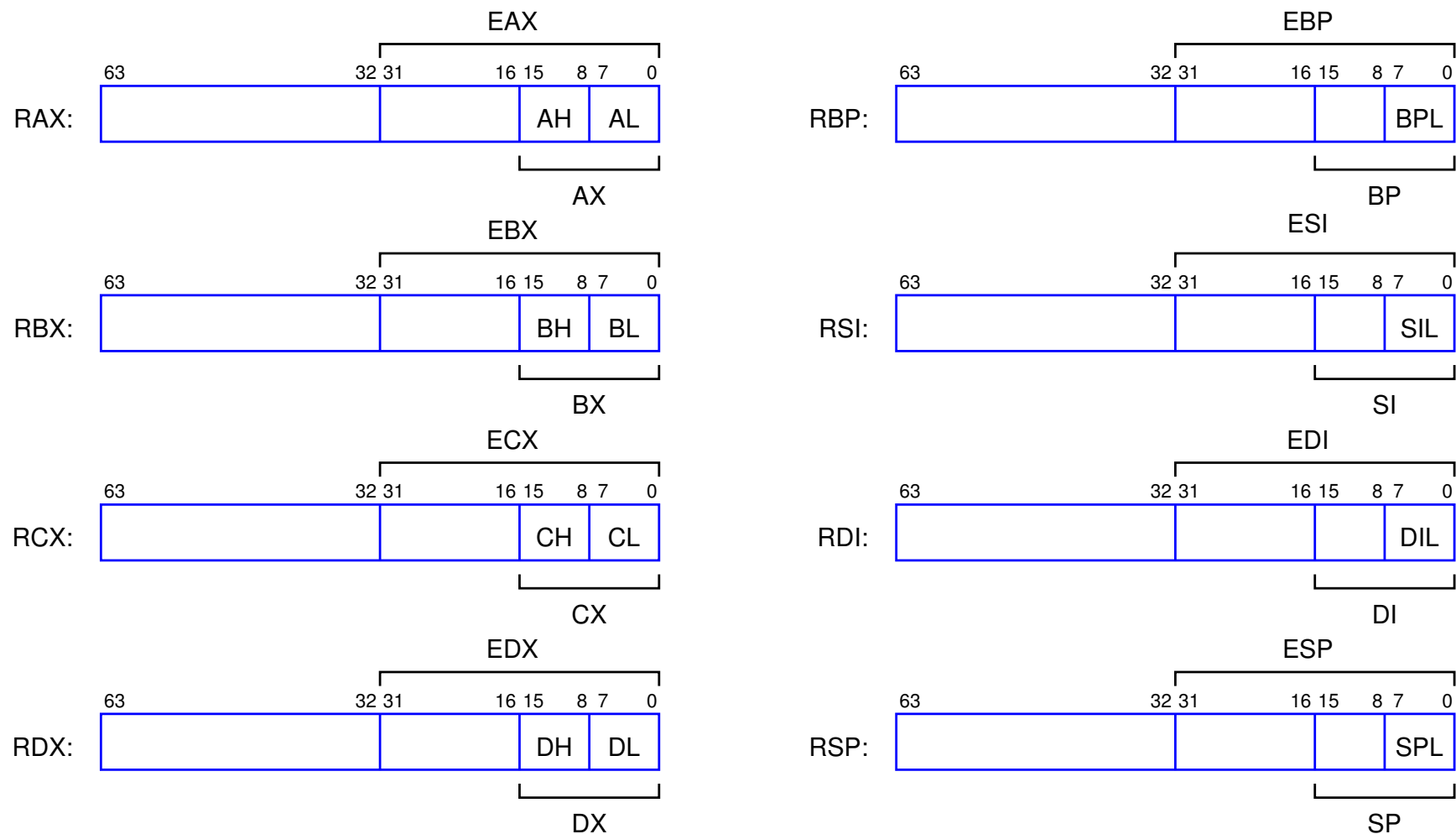
- Certaines instructions peuvent modifier explicitement le **pointeur de programme**, afin de le faire pointer ailleurs que vers l'instruction suivante (**saut**).
- Les processeurs modernes implémentent des mécanismes supplémentaires:
 - Réalisation **simultanée** de plusieurs étapes de chargement, décodage et exécution des instructions (*pipelining*).
 - **Gestionnaire de mémoire** (*Memory Management Unit, MMU*): Mise en œuvre de techniques de protection, de traduction d'adresses, de gestion de la mémoire cache, ...
 - Processeurs **multicœurs**: Plusieurs processeurs individuels (cœurs, *cores*) intégrés dans un même composant.
 - **Interruptions**: Mécanisme permettant de suspendre l'exécution du programme courant pour effectuer des **opérations urgentes**, et de le reprendre par la suite.
 - ...

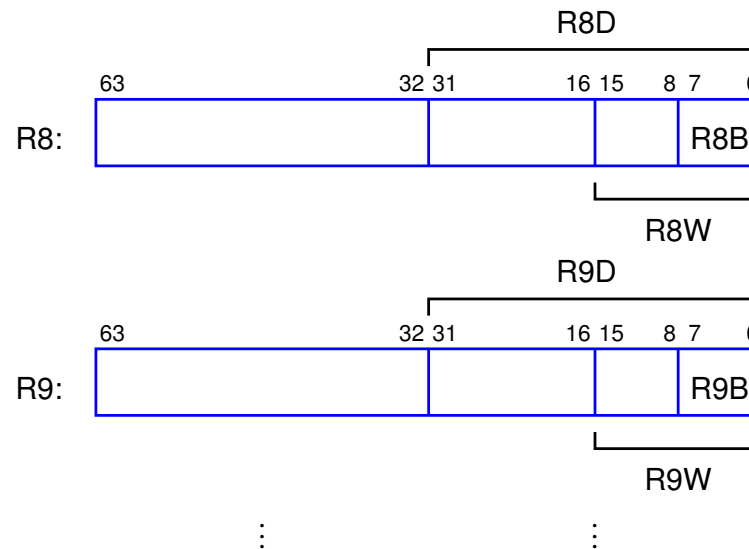
L'architecture x86-64

- Extension de l'architecture **x86** introduite en 1978 (IBM PC).
- Plusieurs **modes de fonctionnement**, notamment dans un but de compatibilité avec x86. Dans ce cours, nous nous limitons au **mode 64 bits**.
- Modèle mémoire **Von Neumann**. En mode 64 bits, l'espace d'adressage est
 $[0, 2^{64} - 1]$.
- 16 **registres généraux** de 64 bits: RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8, R9, R10, R11, R12, R13, R14, R15.
- Certains de ces registres possèdent des modalités d'utilisation particulières.

Les parties de registres

Il est possible de faire référence à une partie seulement du contenu des registres généraux:





Notes:

- Certaines restrictions existent sur les utilisations possibles de ces registres par les instructions.
- D'autres registres sont définis pour la manipulation des **nombre en virgule flottante**, pour exploiter certaines **instructions particulières**, et pour **configurer** le processeur.

Les drapeaux

Le registre RFLAGS contient des **drapeaux (flags)**, qui sont des bits d'information mis à jour par certaines instructions:

- **CF** (*Carry Flag*, bit 0): Indique qu'une opération arithmétique sur des nombres de n bits a produit un **report** à la position n .
- **ZF** (*Zero Flag*, bit 6): Indique qu'une opération a fourni un **résultat nul**.
- **SF** (*Sign Flag*, bit 7): Correspond au bit de poids fort du résultat d'une opération (**bit de signe** dans le cas d'une donnée signée).
- **OF** (*Overflow Flag*, bit 11): Indique un **dépassement arithmétique** pour des données signées.

Les modes d'adressage

En langage d'assemblage, une instruction est composée

- d'une **mnémonique**, qui est le nom conventionnel de l'instruction (p.ex., ADD).
- d'**opérandes**. Chaque opérande peut représenter une source, une destination, ou les deux.

L'architecture x86-64 définit plusieurs façons d'exprimer les opérandes (**modes d'adressage**).

Pour chaque instruction que nous allons étudier, les modes d'adressage permis seront spécifiés.

L'adressage registre

Cet adressage indique qu'un opérande est lu depuis un **registre** (source), ou qu'un résultat doit être placé dans un registre (destination).

Notation et exemple:

```
ADD RAX, RBX
```

Cette instruction additionne le contenu de RAX et de RBX, et place le résultat dans RAX.

L'adressage immédiat

Cet adressage, aussi appelé adressage **littéral**, définit un opérande **constant**.

Notation et exemple:

```
ADD RDI, 0x10
```

Cette instruction ajoute 16 à la valeur de RDI.

Note: L'adressage immédiat ne peut pas être employé pour des **destinations**!

L'adressage direct

Celui-ci indique qu'un opérande doit être lu depuis la **mémoire** (source), ou qu'un résultat doit être écrit en mémoire (destination), à une adresse fixée.

Syntaxe:

<taille> ptr [*<adresse>*],

où

- *<adresse>* est un **pointeur** vers l'emplacement de mémoire concerné.
- *<taille>* est un des mots-clés **qword** (64 bits), **dword** (32 bits), **word** (16 bits) ou **byte** (8 bits).

Exemple:

```
ADD dword ptr [0x1234], R8D
```

Cette instruction ajoute à l'entier de 32 bits situé à l'adresse 0x1234 le contenu de R8D.

Notes:

- L'architecture x86-64 est **petit-boutiste**.
- Il ne faut pas oublier d'**aligner** si nécessaire les données mémorisées sur plus d'une cellule.

L'adressage indirect

Comme l'adressage direct, il indique un accès à la mémoire. La différence est que l'adresse n'est plus constante, mais donnée par le contenu d'un registre.

Notation et exemple:

```
ADD AH, byte ptr [RBX]
```

Cette instruction ajoute à AH l'octet pointé par RBX.

Note: L'architecture x86-64 définit un modèle d'adressage sur 64 bits. L'adressage indirect ne peut donc faire intervenir que des registres de 64 bits.

L'adressage indirect indexé

Il s'agit d'une variante de l'adressage indirect, dans laquelle l'expression du pointeur peut faire intervenir un **index**, un **facteur** et un **déplacement**.

Syntaxe:

$$\langle \text{taille} \rangle \text{ ptr } [\langle \text{base} \rangle + \langle \text{facteur} \rangle * \langle \text{index} \rangle + \langle \text{déplacement} \rangle],$$

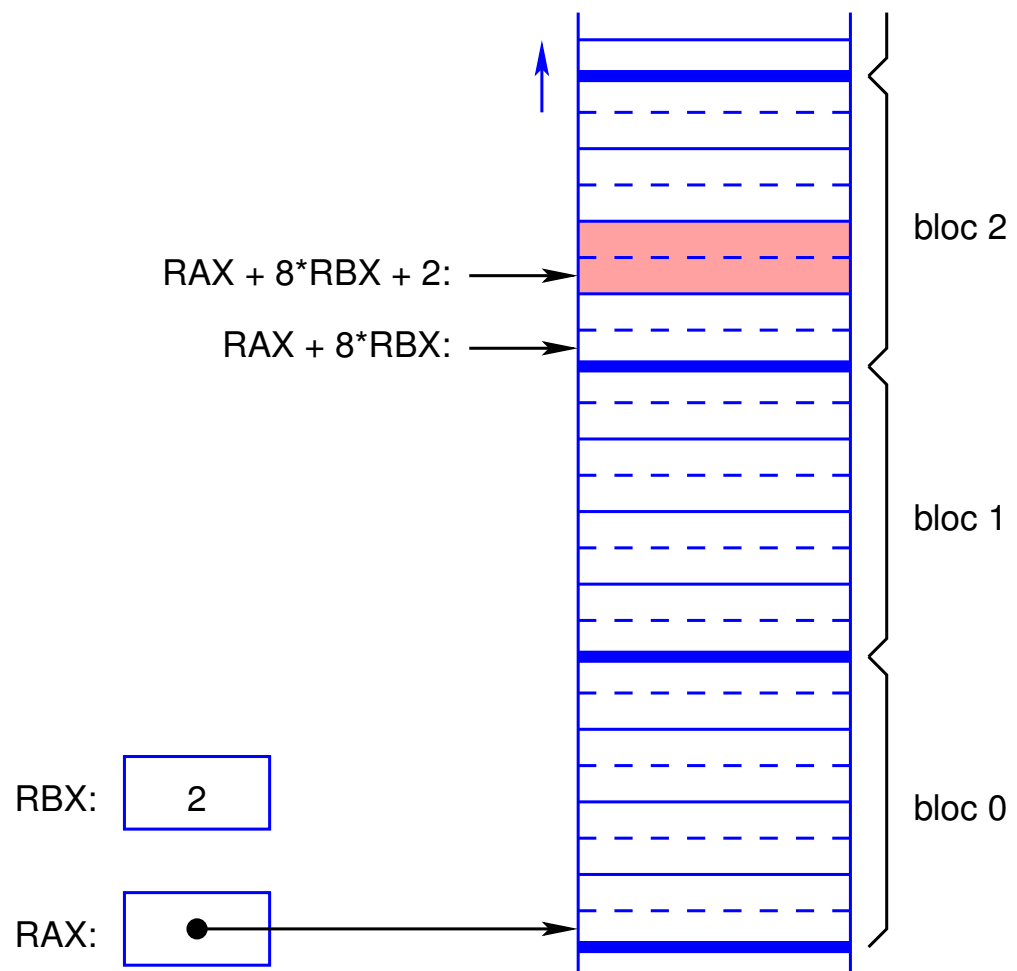
où

- $\langle \text{taille} \rangle$ est qword, dword, word ou byte
- $\langle \text{base} \rangle$ et $\langle \text{index} \rangle$ sont des **registres de 64 bits**.
- $\langle \text{facteur} \rangle$ vaut 1, 2, 4 ou 8.
- $\langle \text{déplacement} \rangle$ est une **constante** signée représentable sur 32 bits.

Note: Certains de ces éléments sont optionnels.

Exemple:

```
ADD DX, word ptr [RAX + 8*RBX + 2]
```



Les instructions x86-64

L'architecture x86-64 définit **plusieurs centaines** d'instructions. Dans ce cours, nous n'étudierons qu'un petit sous-ensemble d'entre elles.

Pour chaque instruction, nous préciserons

- les **modes d'adressage** qu'elle supporte, à l'aide des codes suivants:
 - *imm* pour immédiat.
 - *reg* pour registre.
 - *mem* pour direct, indirect, ou indirect indexé.
- les **drapeaux** affectés par son exécution.

Les instructions de manipulation des données

L'instruction MOV

Cette instruction **copie** des données, du second opérande vers le premier.

Exemples:

- L'instruction

```
MOV EBX, dword ptr [0x100]
```

lit quatre octets en mémoire depuis l'adresse 0x100, et les écrit dans le registre EBX.

- L'instruction

```
MOV byte ptr [RAX + RSI - 4], 0xFF
```

écrit l'octet 0xFF en mémoire, à l'endroit pointé par $RAX + RSI - 4$.

Modes d'adressage:

Op.1	Op.2
<i>reg</i>	<i>imm</i>
<i>mem</i>	<i>imm</i>
<i>reg</i>	<i>reg</i>
<i>reg</i>	<i>mem</i>
<i>mem</i>	<i>reg</i>

Note: Il n'est pas permis de combiner des accès à la mémoire pour les deux opérandes.

Drapeaux affectés: Aucun.

L'instruction XCHG

Cette instruction **échange** le contenu de ses deux opérandes.

Exemples:

- L'instruction

```
XCHG AL, AH
```

permuté les 8 bits de poids faible et de poids fort du registre AX.

- L'instruction

```
XCHG EAX, EAX
```

n'a aucun effet. L'instruction **NOP** (*No OPeration*) en est une abréviation.

Modes d'adressage:

Op.1	Op.2
<i>reg</i>	<i>reg</i>
<i>reg</i>	<i>mem</i>
<i>mem</i>	<i>reg</i>

Drapeaux affectés: Aucun.

Les instructions arithmétiques

L'instruction ADD

Nous avons déjà utilisé cette instruction dans des exemples. Son effet est d'**ajouter** son second opérande au premier.

Exemple:

L'instruction

```
ADD R10, -1
```

décrémente le contenu de R10 d'une unité.

Note: La même instruction peut être employée pour des nombres **non signés** ou bien représentés par **complément à deux**.

Modes d'adressage: Identiques à ceux de MOV.

Drapeaux affectés: CF, ZF, SF et OF.

L'instruction SUB

Cette instruction est similaire à ADD, mais **soustrait** son second opérande du premier.

Exemple:

L'instruction

```
SUB R10, 1
```

a le même effet que la précédente.

Modes d'adressages et drapeaux affectés: Identiques à ceux de ADD.

L'instruction CMP

Cette instruction implémente la **même opération** que SUB, mais ne modifie pas son premier opérande.

En d'autres termes, le seul effet de cette instruction est de mettre à jour les **drapeaux**.

Cela permet d'effectuer des **comparaisons** de valeurs, pouvant servir de base à des décisions dans les instructions suivantes.

Exemple:

L'instruction

```
CMP EAX, EBX
```

calcule la différence $\Delta = EAX - EBX$. On a alors:

- **ZF = 1** ssi $\Delta = 0$, c'est-à-dire **EAX = EBX**.
- **CF = 1** ssi $\Delta < 0$, c'est-à-dire **EAX < EBX** (si les nombres sont **non signés**).

Modes d'adressages et drapeaux affectés: Identiques à ceux de ADD.

Les instructions INC et DEC

Ces instructions **incrémentent** (INC) ou **décrémentent** (DEC) leur opérande, qui sert donc à la fois de source et de destination.

Exemple:

L'instruction

```
INC byte ptr [RBX]
```

ajoute 1 à l'octet pointé par RBX.

Modes d'adressage:

Op.1
<i>reg</i>
<i>mem</i>

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

L'instruction MUL

Cette instruction **multiplie** deux nombres non-signés de n bits, avec $n \in \{8, 16, 32, 64\}$. Le résultat est représenté sur $2n$ bits.

Modes d'adressage: L'instruction admet un seul opérande:

Op.1
<i>reg</i>
<i>mem</i>

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

- **8 bits:** L'opérande est multiplié par AL; le résultat est placé dans AX.
- **16 bits:** L'opérande est multiplié par AX; le résultat est placé dans DX:AX.
- **32 bits:** L'opérande est multiplié par EAX; le résultat est placé dans EDX:EAX.
- **64 bits:** L'opérande est multiplié par RAX; le résultat est placé dans RDX:RAX.

Exemple:

L'instruction

```
MUL dword ptr [0x1234]
```

multiplie l'entier non-signé de 32 bits pointé par 0x1234 par le contenu de EAX. Les 32 bits de poids fort du résultat sont écrits dans EDX, et les 32 bits de faible dans EAX.

Note: Contrairement à l'addition, le fait que les nombres sont représentés de façon **signée** ou **non signée** influence la multiplication.

Drapeaux affectés: CF et OF sont mis à 0 si le résultat de l'opération est **représentable sur n bits**, où n est la taille des opérandes, et à 1 sinon.

Les autres drapeaux sont modifiés de façon arbitraire.

L'instruction IMUL

Cette instruction est similaire à MUL, mais calcule le produit de deux nombres **signés**.

L'opération effectuée, les modes d'adressage et les drapeaux affectés sont identiques à ceux de MUL.

Note: Il existe **d'autres formes** de cette instruction (à deux et trois opérandes), que nous n'étudierons pas.

Les instructions logiques

Les instructions AND, OR et XOR

Ces instructions appliquent une **opération booléenne** bit par bit à leurs deux opérandes, et écrivent le résultat dans la première.

- **AND**: Le résultat est égal à 1 ssi les **deux bits** sont égaux à 1 (et logique).
- **OR**: Le résultat est égal à 1 ssi **au moins un des deux bits** est égal à 1 (ou inclusif).
- **XOR**: Le résultat est égal à 1 ssi **exactement un des deux bits** est égal à 1 (ou exclusif).

Ces instructions permettent de **forcer à 0** (AND), **forcer à 1** (OR) ou de **complémenter** (XOR) des bits à des positions données dans une valeur.

Exemples:

- L'instruction

```
AND byte ptr [0x100], 0xFC
```

force à 0 les deux bits de poids faible de l'octet situé à l'adresse 0x100.

- L'instruction

```
OR AL, 0xF0
```

force à 1 les quatre bits de poids fort du registre AL.

- L'instruction

```
XOR RBX, 0xFF00
```

complémente les bits 8 à 15 du registre RBX.

Modes d'adressage: Identiques à ceux de ADD.

Registres affectés:

- CF et OF sont mis à 0.
- ZF et SF sont mis à jour en fonction du résultat de l'opération.

L'instruction NOT

Cette instruction admet un seul opérande:

Op.1
<i>reg</i>
<i>mem</i>

Son effet est de **complémenter** tous les bits de cet opérande (c'est-à-dire, de la remplacer par son complément à un).

Exemple: Si le registre DX contient initialement 0x5A, alors l'instruction

NOT DX

lui attribuera la valeur 0xFFA5.

Note: Cette instruction est équivalente à

XOR DX, 0xFFFF

Drapeaux affectés: Aucun.

Les instructions de manipulation de la pile

Comme la plupart des autres architectures, les processeurs x86-64 gèrent une **pile**.

Une pile est une structure de données **LIFO (Last-In First-Out)**, définissant deux opérations:

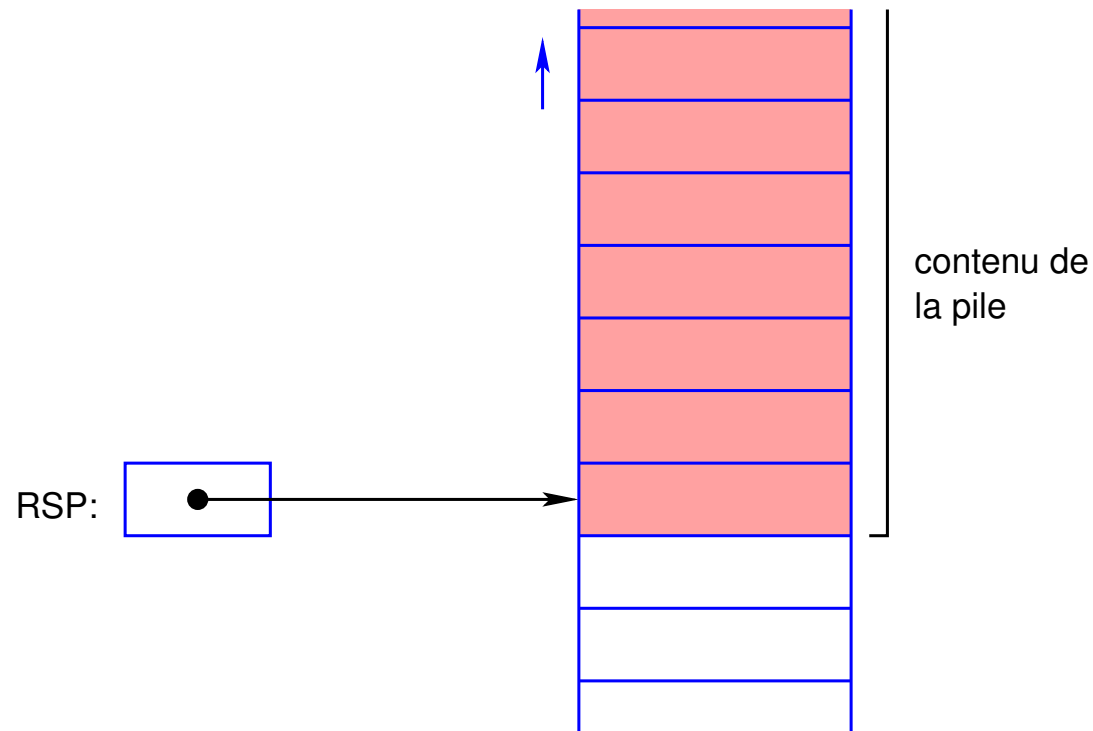
- **Empiler** une valeur (*push*) à son sommet.
- **Dépiler** une valeur (*pop*) depuis son sommet.

La pile sert notamment à

- mémoriser des **données temporaires**, comme les arguments, les variables locales et les points de retour des fonctions invoquées par un programme.
- **sauvegarder** le contenu de registres modifiés par une sous-routine.

La pile dans l'architecture x86-64

- Le contenu de la pile correspond à des **cellules consécutives** de la mémoire.
- La pile croît vers les **adresses décroissantes**.
- Le registre RSP pointe en permanence vers le **dernier octet empilé**.



L'instruction PUSH

Cette instruction **empile** une valeur de 64 bits, donnée par son opérande.

Modes d'adressage:

Op.1
<i>imm</i>
<i>reg</i>
<i>mem</i>

Note: L'adressage registre doit utiliser un registre de **64 bits**. Les adressages direct, indirect et indirect indexé doivent employer le mot-clé `qword`.

Opération réalisée:

1. **Décrémenter** RSP de 8 unités.
2. **Recopier** l'opérande à l'endroit pointé par RSP.

Drapeaux affectés: Aucun.

L'instruction POP

Cette instruction **dépile** une valeur de 64 bits, et l'écrit à l'endroit spécifié par son opérande.

Modes d'adressage:

Op.1
<i>reg</i>
<i>mem</i>

Opération réalisée:

1. **Lire 8 octets** depuis l'emplacement pointé par RSP, et les recopier à l'endroit représenté par l'opérande.
2. **Incrémenter** RSP de 8 unités.

Drapeaux affectés: Aucun.

Exemple:

Les instructions

```
PUSH R8  
PUSH R9  
POP R8  
POP R9
```

permutent le contenu de R8 et de R9 (à condition que RSP pointe vers une zone de la mémoire pouvant accueillir la pile).

Les instructions de contrôle

Ces instructions servent à modifier l'exécution séquentielle du programme, en **transférant le contrôle** (c'est-à-dire, en continuant l'exécution) à un endroit arbitraire de celui-ci.

L'instruction JMP

Cette instruction effectue un **saut inconditionnel** vers un emplacement de la mémoire de programme, donné par son opérande.

En d'autres termes, l'instruction charge cet opérande dans le **compteur de programme**.

Modes d'adressage:

Op.1
<i>imm</i>
<i>reg</i>
<i>mem</i>

Drapeaux affectés: Aucun.

Exemple:

L'instruction

```
JMP 0x1000
```

continue l'exécution du programme à l'adresse 0x1000.

Remarques:

- En pratique, la destination d'un saut est exprimée symboliquement, à l'aide d'une étiquette:

```
boucle:  NOP
          NOP
          NOP
          JMP boucle
```

Dans cet exemple, l'adressage est **immédiat**, la valeur de boucle étant calculée et substituée par le programme d'assemblage.

- L'instruction JMP peut employer un adressage **indirect ou indirect indexé**.

Exemple: L'instruction

```
JMP qword ptr [8*RBX + 0x1000]
```

1. lit une adresse de 64 bits depuis un tableau situé à l'adresse 0x1000, à la position spécifiée par RBX;
2. effectue un saut vers cette adresse.

Ce mécanisme permet notamment d'implémenter une **décision multiple**.

Les instructions de saut conditionnel

Ces instructions sont similaires à JMP, et possèdent les mêmes modalités d'utilisation et les mêmes modes d'adressage.

La différence est qu'elles n'effectuent un saut que si une **condition particulière** est satisfaite.

La condition peut porter sur l'**état d'un drapeau**:

Instruction	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

Exemple:

Les instructions

```
ADD R8, R9
JC report
```

calculent la somme de R8 et de R9, et continuent l'exécution du programme à l'étiquette `report` seulement si un report a été produit à la position 64.

La condition peut également porter sur le résultat d'une **comparaison** réalisée par `CMP`:

Instruction	Condition
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)
JAЕ	$op1 \geq op2$ (valeurs non signées)
JB	$op1 < op2$ (valeurs non signées)
JBE	$op1 \leq op2$ (valeurs non signées)

Exemple:

Les instructions

```
CMP EAX, 0xFFFF  
JA  dépassement
```

effectuent un saut vers dépassement seulement si EAX contient une valeur (non signée) supérieure à 0xFFFF.

Remarque: Certaines instructions sont **équivalentes** (par exemple, JE et JZ).

Drapeaux affectés: Aucun.

L'instruction LOOP

Cette instruction permet d'implémenter une **boucle**. Son opérande est défini comme celui de JMP, et peut employer les mêmes modes d'adressage.

Opérations réalisées:

1. **Décrémenter** RCX d'une unité.
2. Si la **nouvelle valeur de RCX** est non nulle, effectuer un saut à l'endroit spécifié par l'opérande.

Exemple:

Les instructions suivantes effectuent **256** itérations:

```
boucle:  MOV RCX, 0x100
         NOP
         NOP
         NOP
         LOOP boucle
```

Notes:

- Le registre employé comme **compteur de boucle** est nécessairement RCX.
- La valeur initiale de RCX ne correspond pas toujours au nombre d'itérations. En effet, pour $RCX = 0$, ce nombre est égal à 2^{64} et non 0!

Drapeaux affectés: Aucun.

Les instructions CALL et RET

Ces instructions permettent de programmer des **sous-routines**.

L'instruction **CALL** possède les mêmes modalités d'utilisation et les mêmes modes d'adressage que **JMP**.

Opérations réalisées:

1. Empiler la valeur courante de RIP, c'est-à-dire l'adresse où **l'exécution du programme doit reprendre** après la sous-routine.
2. Effectuer un **saut** vers l'endroit donné par l'opérande.

L'instruction **RET** ne prend pas d'argument.

Opérations réalisées:

1. **Dépiler** une valeur de 64 bits.
2. Effectuer un **saut** vers cette adresse.

Drapeaux affectés: Aucun.

Exemple: Définition d'une fonction `minswap`, invoquée depuis le reste du programme:

```
minswap:  CMP ECX, EDX
          JLE sortie
          XCHG ECX, EDX
sortie:   RET
          ...
          MOV ECX, dword ptr [0x100]
          MOV EDX, dword ptr [0x104]
          CALL minswap
          ...
          MOV ECX, dword ptr [0x108]
          MOV EDX, dword ptr [0x10C]
          CALL minswap
          ...
```

Chapitre 4

La programmation en assembleur

Introduction

Le langage d'assemblage n'est pas universel:

- Le modèle mémoire, les registres et le jeu d'instructions d'un processeur sont propres à son architecture.
- La syntaxe du langage peut dépendre des outils utilisés.
- La convention d'appel des fonctions diffère d'un système d'exploitation à un autre.

Environnement utilisé pour les exemples:

- Architecture x86-64.
- Système d'exploitation Linux 64 bits.
- Compilateur GCC.

Un premier programme

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

- La directive `.intel_syntax noprefix` indique la **variante syntaxique** du langage.
- La directive `.text` signale le début du **segment de code** (c'est-à-dire la partie du programme qui contient les **instructions**).

- Les directives

```
.global deep_thought  
.type deep_thought, @function
```

indiquent que l'étiquette `deep_thought` est **globale**, et représente une **fonction**.

- L'instruction `MOV EAX, 42` place la constante 42 dans le registre chargé de retenir la **valeur de retour** de la fonction.
- L'instruction `RET` termine la fonction.

Exemple de programme C de test:

```
#include <stdio.h>

extern int deep_thought(void);

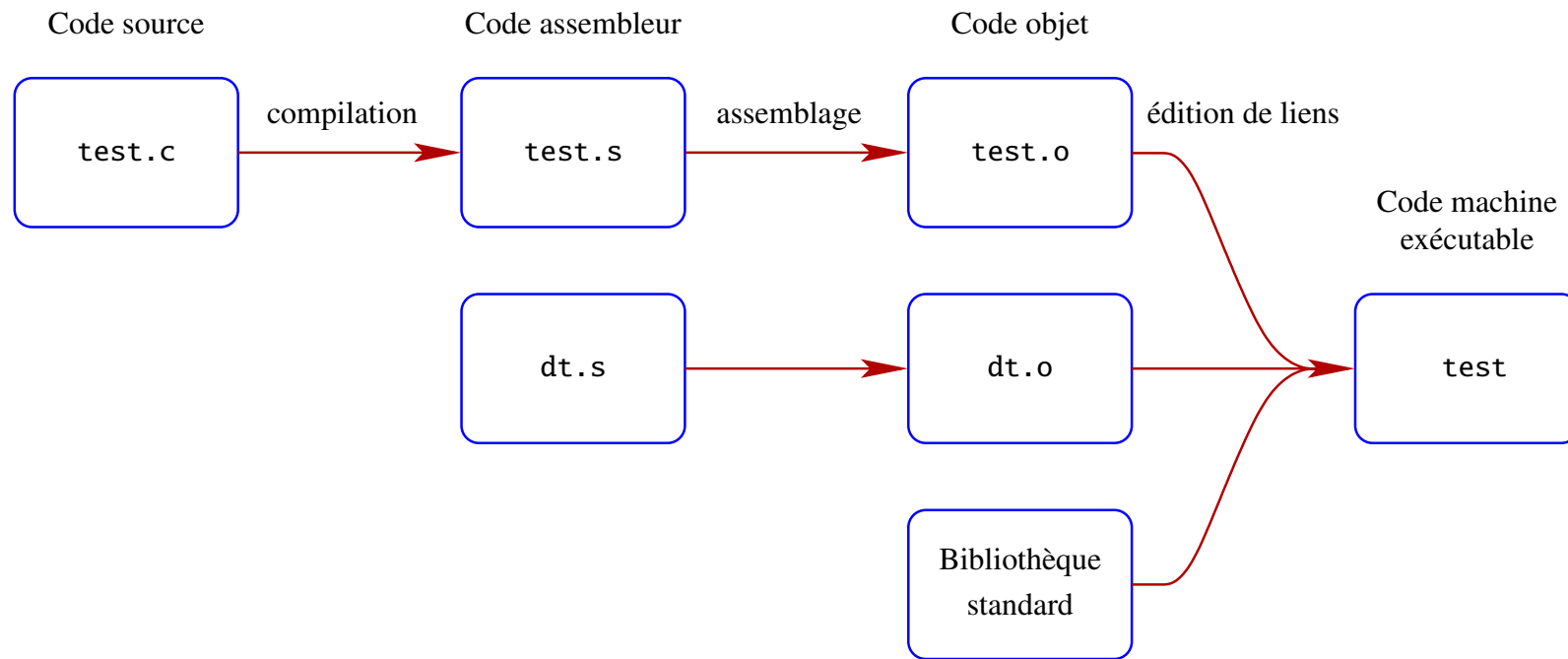
int main()
{
    printf("%d\n", deep_thought());
}
```

Commande de compilation:

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

où `dt.s` est le fichier source assembleur, et `test.c` le programme C de test.

Mécanisme de compilation:



Note: Les fichiers `test.s`, `test.o` et `dt.o` peuvent respectivement être générés par les commandes:

- `gcc -S -masm=intel test.c`
- `gcc -c test.c`
- `gcc -c dt.s`

Les étiquettes

En assembleur, une **étiquette** est une valeur représentée symboliquement. Deux formes d'étiquettes sont possibles:

- Une étiquette d'adresse se définit de la façon suivante:

<code><étiquette>: <instruction> <opérandes></code>

Une telle étiquette

- prend pour valeur **l'adresse** de l'instruction suivante. (Un mécanisme similaire existe pour la mémoire de données.)
- sera substituée en une **valeur numérique** par le programme d'assemblage.
- peut être utilisée **avant** sa définition.
- est soumise à certaines **restrictions** d'utilisation.

- Une étiquette peut également être définie par la directive

```
.equ <étiquette>, <valeur>
```

qui lui attribue la valeur *<valeur>*. Une telle étiquette ne peut être utilisée qu'**après** sa définition.

Exemple:

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

Le segment de données

Un programme assembleur peut également spécifier l'organisation et le contenu initial de la **mémoire de données**. Les directives suivantes sont disponibles:

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

Exemple: Emission de tickets pour une file d'attente:

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

Programme de test:

```
#include <stdio.h>

extern int ticket(void);

int main()
{
    for (;;)
        printf("%d\n", ticket());
}
```

Les étiquettes et l'adressage immédiat

Pour une raison technique, les **étiquettes d'adresse** ne peuvent pas toujours directement figurer dans un adressage immédiat.

Par exemple, si `x` est une telle étiquette, 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**, c'est-à-dire après la compilation du programme.

Cette situation n'est pas problématique pour les instructions de saut, ni pour l'adressage direct.

Par exemple, dans le programme

```
boucle:  NOP
         NOP
         JMP  boucle
```

l'opérande immédiat de l'instruction `JMP` sera encodé de façon **relative** au pointeur de programme dans le code machine, afin de rendre celui-ci **relocalisable**.

Il est cependant possible d'obtenir l'adresse absolue d'une étiquette d'adresse en mentionnant le préfixe **offset flat:** On écrira donc

```
MOV RAX, offset flat:x
```

au lieu de

```
MOV RAX, x # Invalide!
```

Exemple: Fonction générant un tableau contenant le carré de tous les nombres entiers non signés encodables sur 16 bits:

```
                .intel_syntax noprefix
                .data
tableau:        .fill 0x10000, 4, 0
                .text
                .global squares
                .type squares, @function
squares:       MOV    RDI, 0
boucle:        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
```

Programme de test:

```
#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]);
}
```


La convention d'appel d'une fonction

En addition au mécanisme de sauvegarde de l'**adresse de retour** par CALL, et de récupération de cette adresse par RET, il est nécessaire de spécifier un protocole pour

- transmettre des **arguments** à une fonction appelée,
- récupérer la **valeur de retour** de cette fonction, et
- permettre à cette fonction d'allouer des **données temporaires** (par exemple, pour ses variables locales).

Un tel protocole porte le nom de **convention d'appel**. Dans ce cours, nous allons étudier la convention employée par les systèmes UNIX (Linux, macOS, ...).

Note: Nous nous limiterons à des arguments et à une valeur de retour de type **entier** ou **pointeur**.

Principes:

- Les **six premiers arguments** de l'appel (s'ils existent) sont fournis dans les registres RDI, RSI, RDX, RCX, R8 et R9, dans cet ordre.
- Les **arguments suivants** sont empilés (avant l'adresse de retour), dans l'ordre inverse de leur position.
- La **valeur de retour** est placée dans le registre RAX.
- La fonction appelée doit préserver les registres 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).

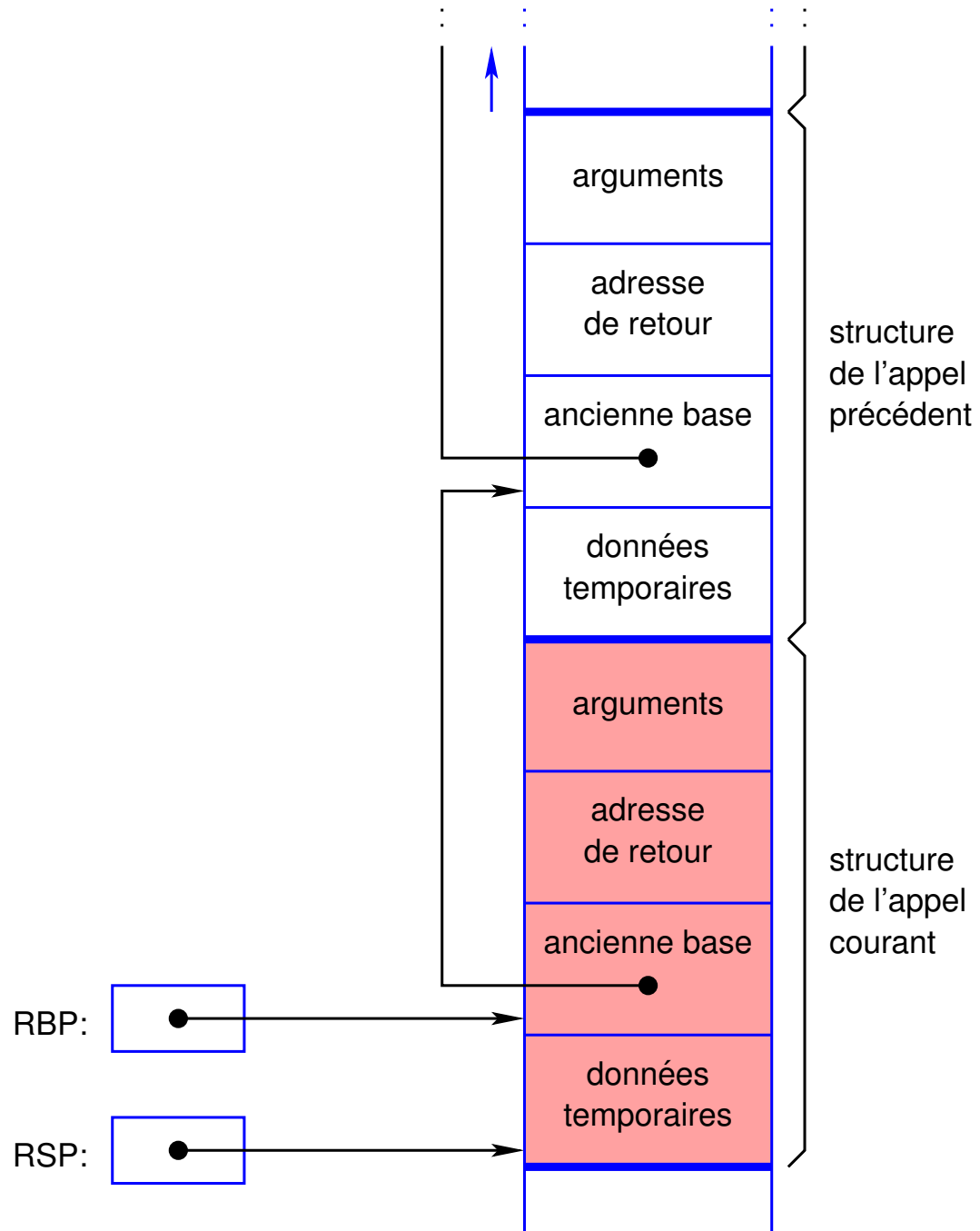
La structure de pile

La **structure de pile** (*stack frame*) 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 partir du registre RBP, qui pointe vers sa **base**.

Elle est composée des éléments suivants, dans l'ordre où ils sont empilés:

- les **arguments** de la fonction appelée, à partir du septième, 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.



Note: Les éléments de cette structure sont facilement accessibles par un adressage indirect indexé basé sur RBP. Par exemple, les **arguments** de la fonction correspondent à

```
qword ptr [RBP + 16], qword ptr [RBP + 24], ...
```

et les **données temporaires** à

```
qword ptr [RBP - 8], qword ptr [RBP - 16], ...
```

(en supposant des données de 64 bits).

Exemple 1: Calcul récursif d'une factorielle

Problème: traduire en assembleur le programme C suivant, calculant la **factorielle** d'un nombre entier:

```
unsigned long factorielle(unsigned n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorielle(n - 1);
}
```

Solution:

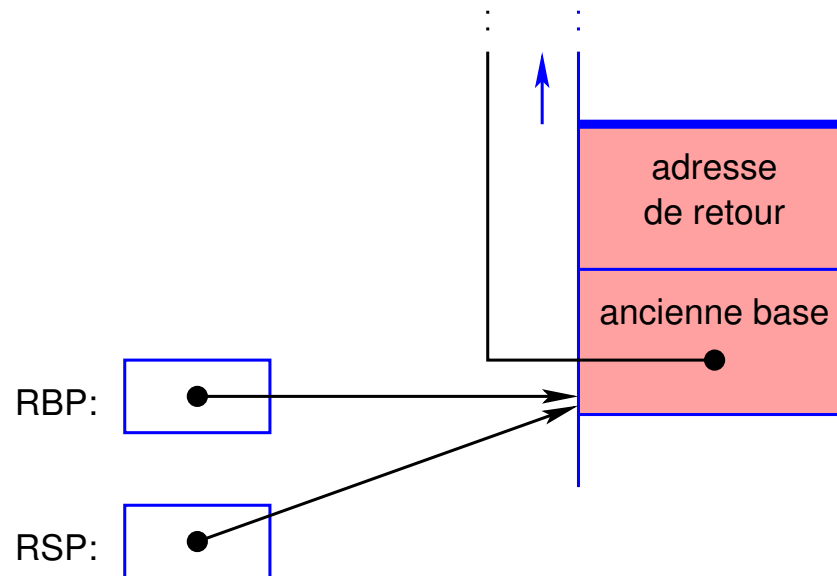
```
                                .intel_syntax noprefix
                                .text
                                .global factorielle
                                .type factorielle, @function
factorielle:                   PUSH RBP
                                MOV  RBP, RSP
                                CMP  EDI, 1
                                JBE  retour_un
                                PUSH RDI
                                DEC  RDI
                                SUB  RSP, 8
                                CALL factorielle
                                ADD  RSP, 8
                                POP  RDI
                                MUL  RDI
                                JMP  retour
retour_un:                     MOV  RAX, 1
retour:                         POP  RBP
                                RET
```

Explications:

Les instructions

```
PUSH RBP
MOV  RBP, RSP
```

créent la structure de pile, en positionnant correctement RBP et RSP:



Ensuite, les instructions

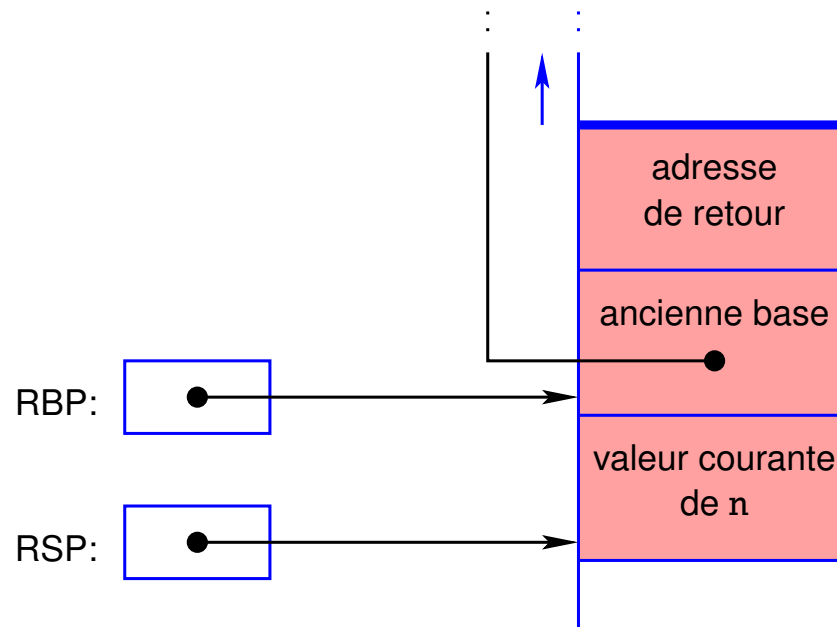
```
retour_un:    CMP    EDI, 1
              JBE    retour_un
              ...
              MOV    RAX, 1
              POP    RBP
              RET
```

traitent le **cas de base** de la récursion: Si l'argument n est tel que $n \leq 1$, alors la fonction retourne 1.

Remarque: L'instruction `POP RBP` supprime la structure de pile courante.

Si $n > 1$, il faut alors appeler `factorielle` avec l'argument $n - 1$.

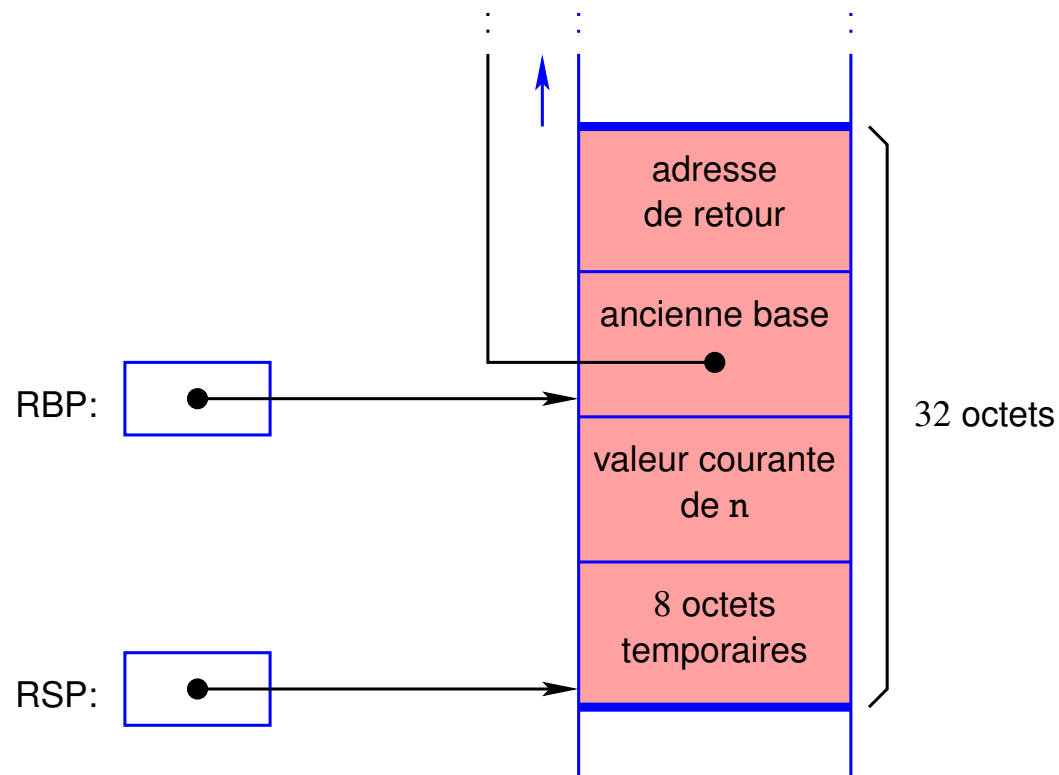
Il est cependant important de **préserver la valeur de n** , car elle intervient dans la suite du calcul. L'instruction `PUSH RDI` sauvegarde cette valeur dans un emplacement temporaire alloué sur la pile:



L'instruction `DEC RDI` calcule l'argument $n - 1$ de l'appel à factorielle.

Cet appel n'est **pas autorisé** dans la situation courante, car la structure de pile possède une taille qui n'est pas un multiple de 16.

Pour remédier à ce problème l'instruction `SUB RSP, 8` alloue 8 octets supplémentaires sur la pile:



Les instructions suivantes

```
CALL    factorielle
ADD     RSP, 8
POP     RDI
```

appellent `factorielle`, libèrent les données temporaires, et récupèrent la valeur de `n` dans `RDI`.

La **valeur de retour** de `factorielle(n - 1)` est quant à elle disponible dans `RAX`.

L'instruction `MUL RDI` calcule le produit de `RAX` et de `RDI`. Les 64 bits de poids faible du résultat sont placés dans `RAX`, qui contiendra la valeur de retour de la fonction. Cette dernière peut donc se terminer.

Programme de test:

```
#include <stdio.h>

extern unsigned long factorielle(unsigned);

int main()
{
    unsigned i;

    for (i = 0; i < 20; i++)
        printf("%u: %lu\n", i, factorielle(i));
}
```

Exemple 2: Hello, world

Pour obtenir un **programme autonome**, il suffit d'implémenter la fonction `main`, qui en sera le **point d'entrée**.

Illustration:

```
                .intel_syntax noprefix
                .data
msg:            .asciz "Hello, world!\n"
                .text
                .global main
                .type main, @function
main:          PUSH  RBP
                MOV   RBP, RSP
                MOV   RDI, offset flat:msg
                CALL  printf
                MOV   EAX, 0
                POP   RBP
                RET
```

Notes:

- L'étiquette `msg` représente un **pointeur** vers la chaîne de caractères utilisée comme argument de la fonction `printf`.

Il est donc nécessaire d'utiliser le préfixe **offset flat:** pour obtenir une représentation absolue de ce pointeur.

- L'instruction `MOV EAX, 0` attribue une **valeur de retour nulle** à la fonction `main`, ce qui signale une exécution sans erreur.
- Si ce code source est placé dans un fichier `hw.s`, il peut être compilé grâce à la commande

```
gcc -o hw hw.s
```

Exemple 3: Conversion en minuscules

Ce programme convertit, dans les **arguments** qui lui sont fournis en ligne de commande, les lettres majuscules en minuscules, et affiche ces arguments sur des lignes séparées:

```
                .intel_syntax noprefix
                .text
                .global main
                .type main, @function
main:           PUSH RBP
                MOV  RBP, RSP
boucle:        DEC  RDI
                JZ   fin
                ADD  RSI, 8
                PUSH RDI
                PUSH RSI
                MOV  RDI, qword ptr[RSI]
                CALL conversion
                MOV  RAX, qword ptr[RBP - 16]
                MOV  RDI, qword ptr[RAX]
                CALL puts
                POP  RSI
                POP  RDI
                JMP  boucle
```



```

fin:          MOV    EAX, 0
              POP    RBP
              RET

conversion:   PUSH   RBP
              MOV    RBP, RSP
boucle2:      MOV    AL, byte ptr[RDI]
              CMP    AL, 0
              JE     fin
              INC    RDI
              CMP    AL, 'A'
              JB     boucle2
              CMP    AL, 'Z'
              JA     boucle2
              ADD    AL, 0x20
              MOV    byte ptr[RDI - 1], AL
              JMP    boucle2

```

Explications:

La fonction `main` reçoit deux valeurs:

- le **nombre d'arguments** `argc` fournis au programme lors de son exécution, incluant son nom (comme premier argument).
- un pointeur `argv` vers un **tableau** dont chaque élément est un pointeur vers une chaîne de caractères représentant un argument.

Selon la convention d'appel, `argc` est reçu dans RDI, et `argv` dans RSI.

La fonction entre dans une **boucle** visant à traiter séparément chaque argument. A chaque itération, RDI contient le nombre d'arguments encore à traiter, et RSI pointe vers l'entrée du tableau `argv` associée à l'argument courant.

La fonction `conversion` (définie dans la suite du programme) accepte comme argument un pointeur vers une chaîne de caractères, dans laquelle elle convertit les majuscules en minuscules.

Avant d'appeler cette fonction, les valeurs de `RDI` et de `RSI` sont **sauvegardées sur la pile**, car elles interviennent dans la suite du programme.

Ensuite, les instructions

```
MOV    RAX, qword ptr[RBP - 16]
MOV    RDI, qword ptr[RAX]
```

récupèrent dans `RAX` la valeur sauvegardée de `RSI`, et placent dans `RDI` la valeur extraite de la case correspondante du tableau. Cette valeur pointe vers la chaîne de caractères qui **vient d'être convertie**. Un appel à la fonction `puts` de la bibliothèque standard C affiche alors cette chaîne.

L'implémentation de la fonction `conversion` est directe. Les constantes `'A'` et `'Z'` apparaissant dans le code assembleur seront remplacées par le code ASCII de ces symboles par le programme d'assemblage.