

# Architecture des ordinateurs

**Samuele Giraudo**

Université Paris-Est Marne-la-Vallée

LIGM, bureau 4B055

`samuele.giraudo@u-pem.fr`

`http://igm.univ-mlv.fr/~giraudo/`

**Informatique** : contraction d'« **information** » et d'« **automatique** »  
↪ Traitement automatique de l'information.

**Ordinateur** : machine automatique de traitement de l'information obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques.

On utilise des ordinateurs pour

- 1 **accélérer** les calculs ;
- 2 **traiter** de gros volumes de données.

# Point de vue adopté

*« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes. »*

— M. R. Fellows, I. Parberry

L'informatique est un donc vaste domaine qui comprend, entre autres,

- l'algorithmique ;
- la combinatoire ;
- la calculabilité ;
- la cryptographie ;
- l'intelligence artificielle ;
- le traitement d'images ;
- les réseaux ;
- l'étude des machines **l'étude des machines.**

Ce cours s'inscrit dans ce dernier point.

# Objectifs du module et bibliographie

Trois axes principaux :

- 1 introduction au **codage des données** ;
- 2 connaissance des principes de **fonctionnement d'un ordinateur** ;
- 3 bases de la **programmation en langage machine**.

Bibliographie (minimale) :

- P. Carter, *PC Assembly Language*, 2006,  
<http://www.drpaulcarter.com/pcasm/> ;
- J. L. Hennessy, D. A. Patterson, *Architectures des ordinateurs : une approche quantitative*, 2003.

## 1 Histoire

- Chronologies
- Mécanismes
- Lampes
- Transistors

## 2 Représentation

- Bits
- Entiers
- Réels

- Caractères

## 3 Programmation

- Assembleur
- Bases
- Sauts
- Fonctions

## 4 Optimisations

- Pipelines
- Mémoires

# Domaines de progression

Les progrès en la matière du développement des ordinateurs sont la conséquence d'avancées de trois sortes :

- 1 les **découvertes théoriques** (mathématiques, informatique théorique) ;
- 2 les **avancées technologiques** (physique, électronique) ;
- 3 les **réalisations concrètes d'ordinateurs** (ingénierie).

# Les avancées théoriques majeures

- 350 : Aristote fonda les bases de la **logique** ;
- 1703 : G. W. Leibniz inventa le **système binaire** ;
- 1854 : G. Boole introduisit l'**algèbre de Boole** ;
- 1936 : A. M. Turing introduisit la **machine de Turing** ;
- 1938 : C. Shannon expliqua comment **utiliser l'algèbre de Boole** dans des circuits ;
- 1945 : J. von Neumann définit l'architecture des ordinateurs modernes : la **machine de von Neumann** ;
- 1948 : C. Shannon introduisit la **théorie de l'information**.

# Les avancées technologiques majeures

1904 : J. A. Fleming inventa la **diode à vide** ;

1948 : J. Bardeen, W. Shockley et W. Brattain inventèrent le **transistor** ;

1958 : J. Kilby construisit le premier **circuit intégré** ;

1971 : la société Intel conçut le premier **microprocesseur**, l'INTEL 4004.

# Les réalisations majeures

Antiquité : utilisation et conception d'**abaques** ;

1623 : W. Schickard conçut les plans de la première machine à calculer, l'**horloge calculante** ;

1642 : B. Pascal construisit la **Pascaline** ;

1801 : J. M. Jacquard inventa le premier **métier à tisser programmable** ;

1834 : C. Babbage proposa les plans de la **machine analytique** ;

1887 : H. Hollerith construisit une **machine à cartes perforées** ;

1949 : M. V. Wilkes créa un ordinateur à architecture de von Neumann, l'**EDSAC**.

# Les générations d'ordinateurs

Le dernier siècle peut se découper en **générations**, suivant le matériel utilisé. Des quatre principales avancées technologiques découlent les quatre générations suivantes.

1<sup>re</sup> **génération** : de 1936 à 1956, emploi de **tubes à vide**.

2<sup>e</sup> **génération** : de 1956 à 1963, emploi de **transistors**.

3<sup>e</sup> **génération** : de 1963 à 1971, emploi de **circuits intégrés**.

4<sup>e</sup> **génération** : de 1971 à 2016 (et plus), emploi de **microprocesseurs**.

Parcourons maintenant l'évolution des machines en les rangeant en trois catégories : les machines à mécanismes, les machines à lampes et les machines à transistors.

# Le temps des machines à mécanismes

Cette période s'étend de l'antiquité et se termine dans les années 1930.

Souvent d'initiative isolées, les machines construites à cette époque possédaient néanmoins souvent quelques points communs :

- utilisation d'**engrenages** et de **courroies** ;
- nécessité d'une **source physique de mouvement** pour fonctionner ;
- machines construites pour un **objectif fixé a priori** ;
- espace de **stockage très faible** ou inexistant.

Ces machines faisaient appel à des connaissances théoriques assez rudimentaires, toute proportion gardée vis-à-vis de leur époque d'introduction.

# Les abaques

**Abaque** : instrument mécanique permettant de réaliser des calculs.

Exemples :



Cailloux



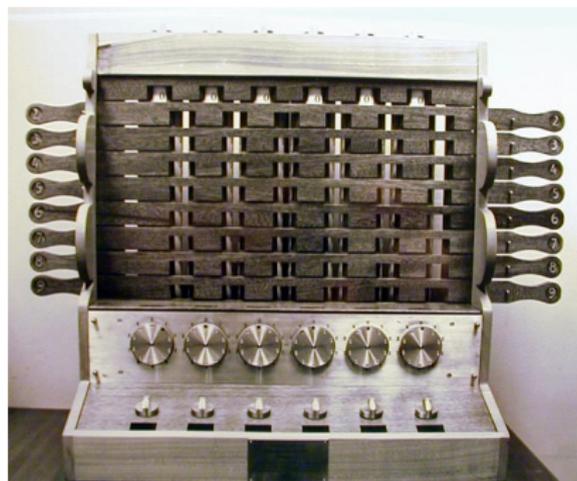
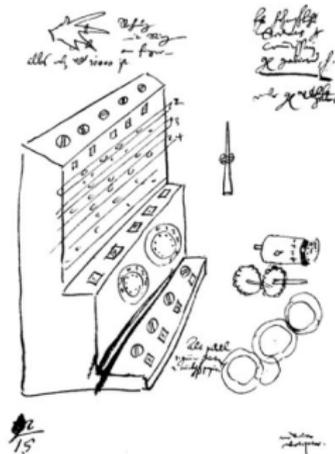
Bouliers



Bâtons de Napier  
(J. Napier, 1617)

# L'horloge calculante

En 1623, Schickard écrit à Kepler pour lui présenter les plans de son « horloge calculante ».



Elle permettait de faire des calculs arithmétiques, utilisait des roues dentées et gérait le report de retenue. Elle ne fut construite qu'en 1960.

# La Pascaline

En 1642, Pascal conçut la « **Pascaline** ».



Elle permettait de réaliser des additions et soustractions de nombres décimaux jusqu'à six chiffres.

En 1671, Leibniz améliora la Pascaline de sorte à gérer multiplications et divisions.



Ces machines permettent de faire des calculs mais leur **tâche** est **fixée dès leur construction** : la notion de programme n'a pas encore été découverte.

# Le métier à tisser programmable

En 1801, Jacquard proposa le premier métier à tisser programmable, le « **Métier Jacquard** ».

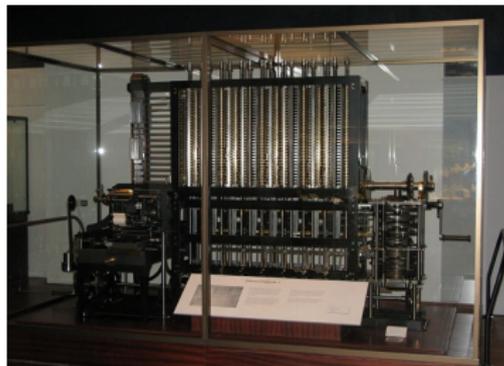
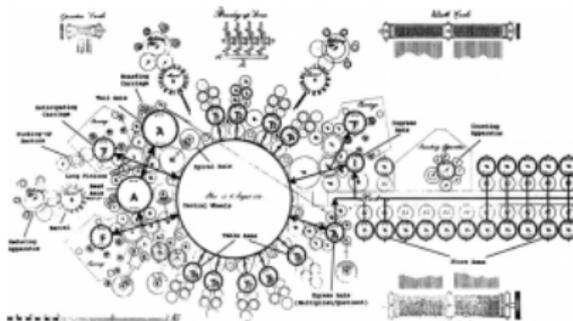


Il était piloté par des **cartes perforées**. Celles-ci dirigeaient le comportement de la machine pour tisser des motifs voulus.

Ces cartes jouaient le rôle de **programme** : une même machine pouvait exécuter des tâches différentes sans avoir à être matériellement modifiée.

# La machine analytique

En 1834, Babbage proposa les plans d'une machine, la **machine analytique**.



Il ne put cependant pas la construire. Elle ne le fut finalement que dans les années 1990.

# La machine analytique — quelques caractéristiques

La machine analytique est étonnamment moderne pour avoir été imaginée si tôt dans l'histoire.

Elle comprend, de manière séparée et bien organisée,

- un lecteur de cartes perforées pour les instructions du programme ;
- un lecteur de cartes perforées pour les données passées en entrée ;
- une unité de calcul (le « moulin ») ;
- une mémoire (le « magasin »).

# La machine à cartes perforées

En 1887, Hollerith construit une **machine à carte perforées** pour faciliter le recensement de la population des États-Unis.



	L	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9			
LA																																								
CA																																								
LS																																								
CS																																								
LS																																								
KA																																								
RA																																								
GC																																								
AV																																								
EA																																								

3994

Les cartes perforées servaient à décrire les caractéristiques des individus en y perçant des trous.

La machine pouvait ainsi dénombrer les individus selon plusieurs critères.

# Le temps des machines à lampes

Cette période débuta dans les années 1930 avec l'introduction de la machine (théorique) de Turing et se termina dans les années 1950.

D'un **point de vue matériel**, elle était basée sur

- les relais ;
- les tubes à vide ;
- les mémoires à tores de ferrite ;
- les bandes magnétiques.

D'un **point de vue théorique**, elle avait pour fondements le système binaire, la machine de Turing et l'architecture de von Neumann (pour la plupart).

Les machines de cette ère se programmaient en

- langage machine pour les toutes premières ;
- assembleur.

# La diode à vide

En 1904, Fleming inventa la **diode à vide**.



Elle s'utilise principalement comme **interrupteur** ou comme **amplificateur** de signal électrique.

Ceci mena à l'invention des **tubes à vide**, composants électriques utilisés dans la conception des télévisions, postes de radio et les premières machines électriques.



Ils sont encore utilisés aujourd'hui dans des domaines très précis : fours à micro-ondes, amplificateurs audio et radars entre autres.

# Le système binaire, la logique et l'électronique

En 1703, Leibniz s'intéressa à la représentation des nombres en **binaire** et à leurs opérations.

Les bases de la logique étaient connues au temps d'Aristote mais il fallut attendre 1854 pour que Boole formalise la notion de **calcul booléen**.

Celui-ci est formé de deux valeurs, le **faux** (codé par le 0 binaire) et le **vrai** (codé par le 1 binaire) et par diverses opérations :

- le *ou* logique, noté  $\vee$  ;
- le *ou exclusif*, noté  $\oplus$  ;
- le *et* logique, noté  $\wedge$  ;
- la *négation* logique, notée  $\neg$ .

Ces opérations donnèrent lieu aux **portes logiques** dans les machines et constituent les composants de base des unités chargées de réaliser des calculs arithmétique ou logiques.



*ou*



*et*



*ou excl.*



*nég.*

# La machine de Turing — calculabilité

La **machine de Turing** est un concept mathématique qui fut découvert par Turing en 1936.

Son but est de fournir une abstraction des mécanismes de calcul.

Elle pose la définition de ce qui est **calculable** :

« tout ce qui est effectivement *calculable*  
est **calculable** par une machine de Turing ».

Des deux occurrences du mot « calculable »,

- 1 la 1<sup>re</sup> signifie ce que l'on peut calculer de manière intuitive (ce qu'un être humain peut calculer par un raisonnement) ;
- 2 la 2<sup>e</sup> signifie ce que la machine de Turing peut produire comme résultat à l'issue de l'exécution d'un programme.

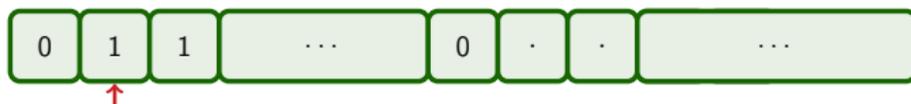
# La machine de Turing — fonctionnement et complétude

Une machine de Turing travaille sur un ruban (infini) contenant des cases qui peuvent être soit vides ( $\cdot$ ), soit contenir la valeur 1, soit contenir la valeur 0.



Le ruban est la **mémoire** de la machine.

Une tête de lecture/écriture vient produire un résultat sur le ruban en modifiant le contenu de la case pointée et en se déplaçant d'un pas à gauche ou à droite.



Cette tête de lecture est pilotée par un **programme**.

Une machine est **Turing-complète** si elle peut calculer tout ce que peut calculer une machine de Turing.

# L'architecture de von Neumann — caractéristiques

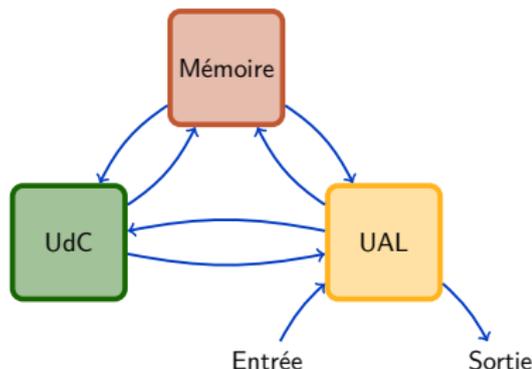
L'**architecture de von Neumann** est un modèle organisationnel d'ordinateurs décrit par von Neumann en 1945.

Quelques caractéristiques importantes :

- une machine de von Neumann possède diverses parties bien distinctes et dédiées à des tâches précises (mémoire, unité de calcul et unité de contrôle) ;
- le programme à exécuter se situe dans la mémoire interne de la machine (un programme est une donnée comme une autre). Cette caractéristique se nomme **machine à programme enregistré** ;
- elle est pourvue d'entrées et de sorties qui permettent la saisie et la lecture de données par un humain ou bien par une autre machine.

Encore aujourd'hui la très grande majorité des ordinateurs sont des machines de von Neumann.

# L'architecture de von Neumann — organisation



- La **mémoire** contient à la fois des données et le programme en cours d'exécution.
- L'**UdC**, Unité de Contrôle, permet de traiter les instructions contenues dans le programme. Elle est chargée du **séquençage** des opérations.
- L'**UAL**, Unité Arithmétique et Logique, permet de réaliser des instructions élémentaires : opérations arithmétiques (+, -, ×, /, ...), opérations logiques ( $\vee$ ,  $\wedge$ ,  $\oplus$ ,  $\neg$ , ...), opérations de comparaison (=,  $\neq$ ,  $\leq$ , <, ...).
- L'**entrée** permet de recevoir des informations.
- La **sortie** permet d'envoyer des informations.

# L'ABC

L'**Atanasoff-Berry Computer** (ABC) fut le premier ordinateur numérique électronique. Il fut construit en 1937 par Atanasoff et Berry.



Il représentait les données en binaire et il adoptait une **séparation entre mémoire et unité de calcul**.

Il a été construit pour la résolution de systèmes d'équations linéaires (il pouvait manipuler des systèmes à vingt-neuf équations).

# L'ASCC

L'**Automatic Sequence Controlled Calculator** (ASCC), également appelé Harvard Mark I, fut construit par IBM en 1944. Il fut le premier ordinateur a **exécution totalement automatique**.

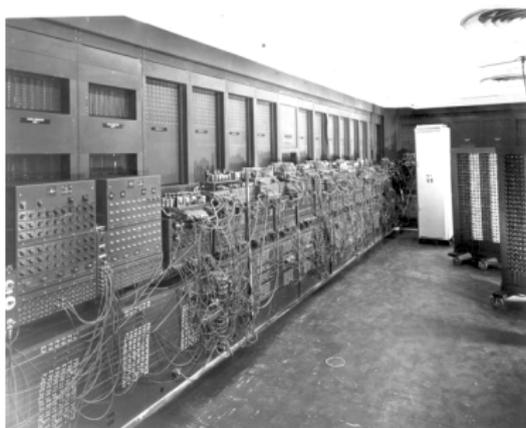


Il pouvait réaliser une multiplication de nombres de vingt-trois chiffres décimaux en six secondes, une division en quinze secondes et des calculs trigonométriques en une minute.

Il ne vérifie pas l'architecture de von Neumann car il fonctionne à cartes perforées (le programme n'est pas une donnée).

# L'ENIAC

L'**Electronic Numerical Integrator Analyser and Computer** (ENIAC) fut achevé en 1946. C'est le premier ordinateur électronique **Turing-complet**.

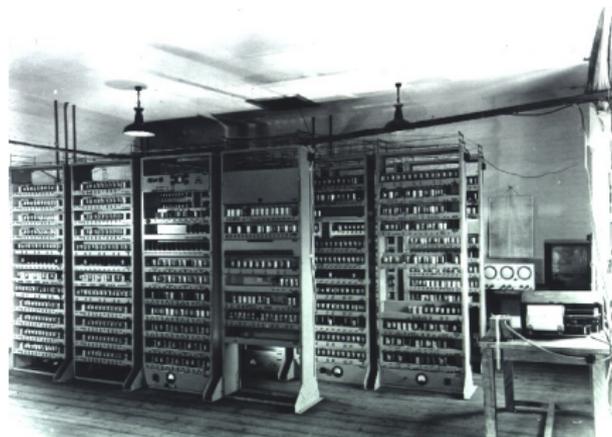


Il pouvait réaliser des multiplications de nombres de dix chiffres décimaux en trois millièmes de seconde.

Il contient 17468 tubes à vide et sa masse est de trente tonnes.

# L'EDSAC

L'**Electronic Delay Storage Automatic Calculator** (EDSAC), descendant de l'ENIAC, fut construit en 1949. C'est une **machine de von Neumann**.



Sa mémoire utilisable est organisée en 1024 régions de 17 bits. Une instruction est représentée par un code 5 bits, suivi d'un bit de séparation, de 10 bits d'adresse et d'un bit de contrôle.

# Le temps des machines à transistors

Cette période débuta dans les années 1950 et se prolonge encore aujourd'hui (en 2016).

D'un **point de vue matériel**, elle se base sur

- les transistors ;
- les circuits intégrés ;
- les microprocesseurs ;
- les mémoires SRAM et flash.

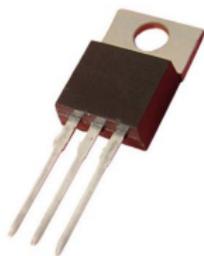
D'un **point de vue théorique**, il y a assez peu de différences par rapport à l'ère précédente. Les machines sont de von Neumann et Turing-complètes. Les principales avancées théoriques sont de nature algorithmique où de nombreuses découvertes ont été réalisées.

L'apparition des premiers langages de programmation est un signe annonciateur de cette nouvelle ère :

- le FORTRAN (Formula Translator) en 1957 ;
- le COBOL (Common Business Oriented Language) en 1959.

# Le transistor

En 1948, Bardeen, Shockley et Brattain inventèrent le **transistor**.



C'est une **version améliorée du tube à vide** :

- élément moins volumineux et plus solide ;
- fonctionne sur de faibles tensions ;
- pas de préchauffage requis.

Ils peuvent être miniaturisés au point de pouvoir être assemblés en très grand nombre (plusieurs milliards) dans un très petit volume.

# L'IBM 1401

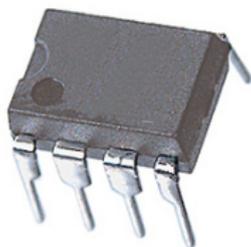
L'**IBM 1401** fut fabriqué de 1959 à 1965. Il fut l'une des machines à transistor les plus vendues de son époque.



Il pouvait réaliser 193000 additions de nombres de huit chiffres décimaux par seconde et disposait d'une mémoire d'environ 8 Kio.

# Le circuit intégré

En 1958, Kilby inventa le **circuit intégré**.



C'est intuitivement un composant obtenu en connectant d'une certaine manière des transistors entre eux.

Son rôle est donc de regrouper dans un espace très réduit un très grand nombre de composants électroniques (transistors, portes logiques, *etc.*).

# L'IBM 360

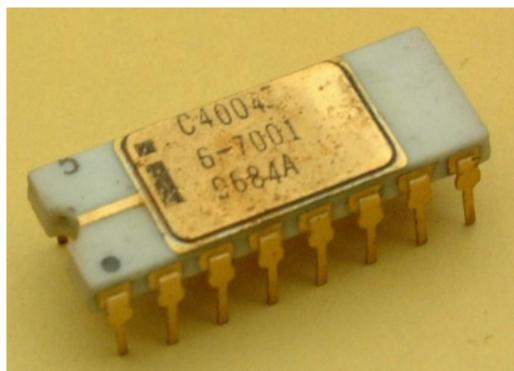
L'IBM 360 fut commercialisé dès 1966 et fut l'une des premières machines à circuits intégrés les plus célèbres de son époque.



Il pouvait accueillir jusqu'à 8 Mio de mémoire.

# Le microprocesseur

Le premier **microprocesseur** fut conçu en 1971. Il s'agit de l'INTEL 4004.



Il contenait 2300 transistors et sa fréquence était de 740 KHz.

Il fournissait une puissance équivalente à l'ENIAC.

# Comparaison de quelques réalisations

Machine	Date	Fonctionnement	Base	T.-complet
Mach. analytique	1834	Mécanique	Décimal	Oui (!)
ABC	1937	Électronique	Binaire	Non
ASCC	1944	Électromécanique	Décimal	Non
ENIAC	1946	Électronique	Décimal	Oui
IBM 360	1966	Électronique	Binaire	Oui

# Vers les ordinateurs d'aujourd'hui — matériel

Depuis l'invention du microprocesseur, les machines ont subi de nombreuses **évolutions matérielles** :

- **miniaturisation** de plus en plus poussée. Apparition des ordinateurs portables dans les années 1980, des PDA dans les années 1990, des smartphones et des tablettes dans les années 2010 ;
- meilleure **gestion de l'énergie**. La consommation énergétique d'un composant est une problématique importante. Les batteries offrent des capacités (mesurées en ampères-heure A.h) de plus en plus grandes et les composants électroniques consomment de moins en moins ;
- plus grande **puissance de calcul** ;
- augmentation des quantités de **mémoire**. Ceci concerne les mémoires de stockage ainsi que les mémoires de travail ;
- meilleure **fiabilité**.

# Vers les ordinateurs d'aujourd'hui — théorie

En parallèle, de nouvelles **connaissances théoriques** viennent se joindre à ces avancées :

- nouveaux **algorithmes**. De nombreuses découvertes théoriques améliorent, p.ex., l'efficacité des algorithmes de recherche de données, des calculs de chemins dans les graphes et des calculs algébriques pour la cryptographie ;
- nouveaux **langages de programmation** et paradigmes de programmation. Les langages deviennent de plus en plus abstraits (éloignés des considérations matérielles), ce qui permet de programmer des choses de plus en plus complexes et des projets de plus en plus conséquents ;
- apparition du **calcul parallèle** ;
- apparition des **réseaux** ;
- apparition des **machines virtuelles**.

Pour qu'un ordinateur puisse traiter des données, il est nécessaire de les **représenter** de sorte qu'il puisse les « comprendre » et les manipuler.

Il est facile de représenter électroniquement deux états :

- **l'état** 0, incarné par l'absence de courant électrique ;
- **l'état** 1, incarné par la présence de courant électrique.

On appelle **bit** l'unité de base d'information, à valeur dans l'ensemble  $\{0, 1\}$ , symbolisant l'état 0 ou l'état 1.

# Suites de bits

Dans une suite de bits, on distingue deux bits particuliers :



Chaque bit d'une suite de  $n$  bits est implicitement indicé de 0 pour le bit de poids faible à  $n - 1$  pour le bit de poids fort :



Un bit d'indice plus grand (resp. petit) qu'un autre est dit de « poids plus fort » (resp. « poids plus faible »).

Dans un ordinateur, les informations sont représentées par des **suites finies de bits**. On parle de **représentation sur  $n$  bits** lorsque l'on fixe le nombre de bits  $n$  pour représenter une donnée.

# Représenter des informations

Pour qu'une suite de bits représente de l'information, il faut savoir comment l'**interpréter**. Une interprétation s'appelle un **codage**.

On placera, si possible pour chaque suite de symboles, le nom de son codage en indice.

Les principales informations que l'on souhaite représenter sont

- les entiers (positifs ou négatifs) ;
- les nombres réels ;
- les caractères ;
- les textes ;
- les instructions (d'un programme).

Pour chacun de ces points, il existe beaucoup de codages différents. Leur raison d'être est que, selon la situation, un codage peut s'avérer meilleur qu'un autre (plus simple, plus économique, plus efficace).

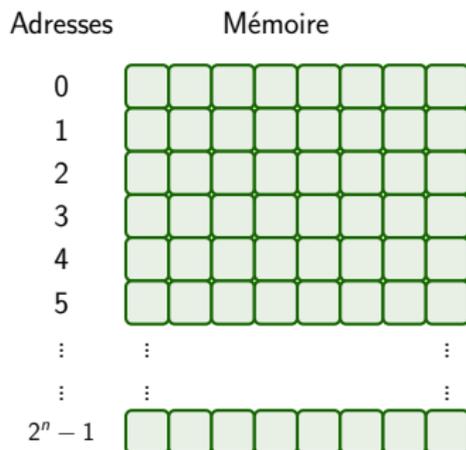
# Types de données fondamentaux

Type	Taille
bit	1 bit
octet (byte)	8 bits
mot (word)	16 bits
double mot (dword)	32 bits
quadruple mot (qword)	64 bits
<b>kibioctet (Kio)</b>	$2^{10} = 1024$ octets
<b>mébioctet (Mio)</b>	$2^{20} = 1048576$ octets
<b>gibioctet (Gio)</b>	$2^{30} = 1073741824$ octets
<b>kilooctet (Ko)</b>	$10^3 = 1000$ octets
<b>mégaoctet (Mo)</b>	$10^6 = 1000000$ octets
<b>gigaoctet (Go)</b>	$10^9 = 1000000000$ octets

On utilisera de préférence les unités Kio, Mio et Gio à la place de Ko, Mo et Go.

# Mémoire d'une machine

La **mémoire** d'un ordinateur est un tableau dont chaque case contient un octet.



Chaque octet de la mémoire est repéré par son **adresse**. C'est sa position dans le tableau.

Sur un **système  $n$  bits**, l'adressage va de 0 à  $2^n - 1$  au maximum.

La structure octet par octet de la mémoire donne lieu au problème suivant : comment organiser en mémoire une suite de bits  $u$  constituée de plus de huit bits ?

Il existe pour cela deux conventions : le **petit boutisme** (*little-endian*) et le **grand boutisme** (*big-endian*). Les architectures qui nous intéressent sont en **petit boutisme**.

L'organisation se fait en trois étapes :

- 1 si  $u$  n'est pas constitué d'un nombre de bits multiple de huit, on ajoute des 0 à sa **gauche** de sorte qu'il le devienne. On appelle  $u'$  cette nouvelle suite ;
- 2 on découpe  $u'$  en  $k$  morceaux de huit bits

$$u' = u'_{k-1} \dots u'_1 u'_0 ;$$

- 3 en **petit** (resp. **grand**) boutisme, les morceaux sont placés, des petites aux grandes adresses, de  $u'_0$  à  $u'_{k-1}$  (resp.  $u'_{k-1}$  à  $u'_0$ ) dans la mémoire.

# Boutisme

P.ex., pour

$$u := 011110000000101110111,$$

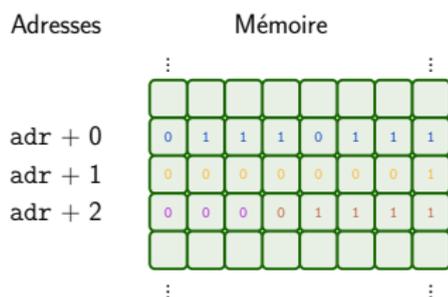
on a

$$u' = 000011110000000101110111$$

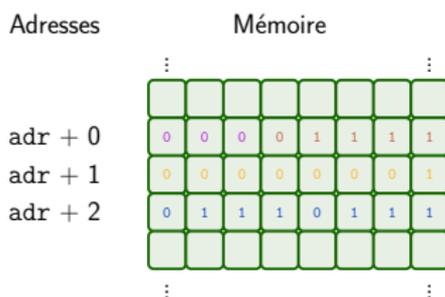
et

$$u'_2 = 00001111, \quad u'_1 = 00000001, \quad u'_0 = 01110111.$$

Selon les deux conventions, le placement de  $u$  à l'adresse  $\text{adr}$  en mémoire donne lieu à



en petit boutisme



en grand boutisme

# Codage unaire

Le moyen le plus simple de représenter un entier positif  $n$  consiste à le coder par une suite de  $n$  bits à 1. Ceci est le **codage unaire** de  $n$ .

P.ex., l'entier  $(7)_{\text{dix}}$  est codé par la suite  $(1111111)_{\text{un}}$ .

Avec ce codage, les **opérations arithmétiques** sont **très simples** :

- addition : concaténation des codages. P.ex.,

$$(7)_{\text{dix}} + (9)_{\text{dix}} = (1111111)_{\text{un}} + (111111111)_{\text{un}} = (111111111111111)_{\text{un}} ;$$

- produit de  $u$  et  $v$  : substitution de  $v$  à chaque 1 de  $u$ . P.ex.,

$$(3)_{\text{dix}} \times (5)_{\text{dix}} = (111)_{\text{un}} \times (11111)_{\text{un}} = (11111 11111 1111)_{\text{un}}.$$

**Avantages** : opérations arithmétiques faciles sur les entiers positifs.

**Inconvénients** : espace mémoire en  $\Theta(n)$  pour représenter l'entier  $n$  ; possibilité de ne représenter que des entiers positifs.

# Représentation Binary Coded Decimal (BCD)

Le **codage BCD** consiste à représenter un entier positif

$$(c_{n-1} \dots c_1 c_0)_{\text{dix}}$$

exprimé en base dix par la suite de  $4n$  bits

$$(c'_{n-1} \dots c'_1 c'_0)_{\text{bcd}}$$

où chaque  $c'_i$  est le code BCD de  $c_i$ . Celui-ci code chaque chiffre décimal par une suite de quatre bits au moyen de la table suivante :

$(0)_{\text{dix}} \rightarrow (0000)_{\text{bcd}}$	$(5)_{\text{dix}} \rightarrow (0101)_{\text{bcd}}$
$(1)_{\text{dix}} \rightarrow (0001)_{\text{bcd}}$	$(6)_{\text{dix}} \rightarrow (0110)_{\text{bcd}}$
$(2)_{\text{dix}} \rightarrow (0010)_{\text{bcd}}$	$(7)_{\text{dix}} \rightarrow (0111)_{\text{bcd}}$
$(3)_{\text{dix}} \rightarrow (0011)_{\text{bcd}}$	$(8)_{\text{dix}} \rightarrow (1000)_{\text{bcd}}$
$(4)_{\text{dix}} \rightarrow (0100)_{\text{bcd}}$	$(9)_{\text{dix}} \rightarrow (1001)_{\text{bcd}}$

P.ex.,  $(201336)_{\text{dix}} = (0010\ 0000\ 0001\ 0011\ 0011\ 0110)_{\text{bcd}}$ .

# Représentation Binary Coded Decimal (BCD)

**Avantages** : représentation très simple des entiers et naturelle car très proche de la base dix.

**Inconvénients** : gaspillage de place mémoire (six suites de 4 bits ne sont jamais utilisées), les opérations arithmétiques ne sont ni faciles ni efficaces.

Ce codage est très peu utilisé dans les ordinateurs. Il est utilisé dans certains appareils électroniques qui affichent et manipulent des valeurs numériques (calculatrices, réveils, *etc.*).

# Changement de base — algorithme

Soient  $b \geq 2$  un entier et  $x$  un entier positif. Pour écrire  $x$  en base  $b$ , on procède comme suit :

1 Si  $x = 0$  : renvoyer la liste  $[0]$

2 Sinon :

1  $L \leftarrow []$

2 Tant que  $x \neq 0$  :

1  $L \leftarrow (x \% b) \cdot L$

2  $x \leftarrow x / b$

3 Renvoyer  $L$ .

Ici,  $[]$  est la liste vide,  $\cdot$  désigne la concaténation des listes et  $\%$  est l'opérateur modulo.

# Changement de base — exemple

Avec  $x := (294)_{\text{dix}}$  et  $b := 3$ , on a

$x$	$x \% 3$	$x / 3$	$L$
$(294)_{\text{dix}}$	$(0)_{\text{dix}}$	$(98)_{\text{dix}}$	$[0]$
$(98)_{\text{dix}}$	$(2)_{\text{dix}}$	$(32)_{\text{dix}}$	$[2, 0]$
$(32)_{\text{dix}}$	$(2)_{\text{dix}}$	$(10)_{\text{dix}}$	$[2, 2, 0]$
$(10)_{\text{dix}}$	$(1)_{\text{dix}}$	$(3)_{\text{dix}}$	$[1, 2, 2, 0]$
$(3)_{\text{dix}}$	$(0)_{\text{dix}}$	$(1)_{\text{dix}}$	$[0, 1, 2, 2, 0]$
$(1)_{\text{dix}}$	$(1)_{\text{dix}}$	$(0)_{\text{dix}}$	$[1, 0, 1, 2, 2, 0]$
$(0)_{\text{dix}}$	—	—	$[1, 0, 1, 2, 2, 0]$

Ainsi,  $(294)_{\text{dix}} = (101220)_{\text{trois}}$ .

# Représentation binaire des entiers positifs

Le **codage binaire** consiste à représenter un entier positif en base deux.

En machine, si l'écriture d'un entier est plus longue que huit bits, il est d'usage de le représenter par tranches de huit bits.

Par ailleurs, si son écriture n'est pas de longueur multiple de huit, on complète généralement à gauche par des zéros de sorte qu'elle le soit.

P.ex.,  $(35)_{\text{dix}} = (100011)_{\text{deux}}$  est représenté par l'octet

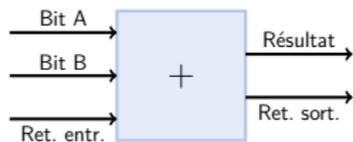
0 0 1 0 0 0 1 1

et  $(21329)_{\text{dix}} = (101001101010001)_{\text{deux}}$  est représenté par la suite de deux octets

0 1 0 1 0 0 1 1 0 1 0 1 0 0 1 0

# L'addition d'entiers — l'additionneur simple

L'**additionneur simple** est un opérateur qui prend en entrée **deux bits** et une **retenue d'entrée** et qui renvoie deux informations : le **résultat** de l'addition et la **retenue de sortie**.



Il fonctionne selon la table

	Bit A	Bit B	Ret. entr.	Résultat	Ret. sort.
0 →	0	0	0	0	0
1 →	0	0	1	1	0
1 →	0	1	0	1	0
2 →	0	1	1	0	1
1 →	1	0	0	1	0
2 →	1	0	1	0	1
2 →	1	1	0	0	1
3 →	1	1	1	1	1

# L'addition d'entiers — l'algorithme usuel

Un additionneur  $n$  bits fonctionne de la même manière que l'**algorithme d'addition usuel**.

Il effectue l'addition chiffre par chiffre, en partant de la droite et en reportant les retenues de proche en proche.

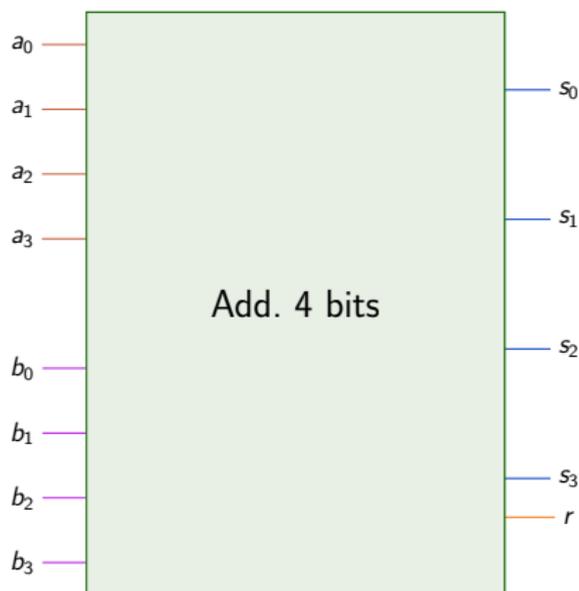
P.ex., l'addition binaire de  $(159)_{\text{dix}}$  et de  $(78)_{\text{dix}}$  donne  $(237)_{\text{dix}}$  :

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

La retenue de sortie est 0.

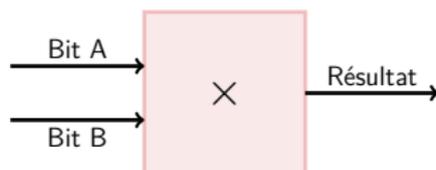
# L'addition d'entiers — l'additionneur 4 bits

On construit un additionneur 4 bits en combinant quatre additionneurs simples :



# La multiplication d'entiers — le multiplicateur simple

Le **multiplicateur simple** est un opérateur qui prend **deux bits** en entrée et qui renvoie un **résultat**.



Il fonctionne selon la table

Bit A	Bit B	Résultat
0	0	0
0	1	0
1	0	0
1	1	1

**Note** : ce multiplicateur simple binaire n'a pas de retenue de sortie, contrairement au multiplicateur simple décimal.



# Bit de signe, codage de taille fixée et plage

Pour l'instant, nous avons vu uniquement la représentation d'entiers positifs.

On peut représenter des entiers négatifs et positifs à l'aide d'un **bit de signe**. C'est le **bit de poids** fort qui renseigne le signe de l'entier

- s'il est égal à 1, l'entier codé est négatif  ;
- s'il est égal à 0, l'entier codé est positif .

De plus, à partir de maintenant, on précisera toujours le **nombre  $n$  de bits** sur lequel on code les entiers.

Si un entier requiert moins de  $n$  bits pour être codé, on complétera son codage avec des 0 ou des 1 à gauche en fonction de son signe et de la représentation employée.

À l'inverse, si un entier  $x$  demande strictement plus de  $n$  bits pour être codé, on dira que «  $x$  ne peut pas être codé sur  $n$  bits ».

La **plage d'un codage** désigne l'ensemble des valeurs qu'il est possible de représenter sur  $n$  bits.

# Représentation en magnitude signée

Le codage en **magnitude signée** permet de représenter un entier en le codant en binaire et en lui adjoignant un **bit de signe**.

P.ex., en se plaçant sur 8 bits, on a

$$(45)_{\text{dix}} = (00101101)_{\text{ms}} \quad \text{et} \quad (-45)_{\text{dix}} = (10101101)_{\text{ms}}.$$

**Avantage** : calcul facile de l'opposé d'un entier.

**Inconvénients** : l'addition de deux entiers ne peut pas se faire en suivant l'algorithme classique d'addition.

Il y a de plus deux encodages différents pour zéro (qui est pos. et nég.) :

$$(0)_{\text{dix}} = (00 \dots 0)_{\text{ms}} = (10 \dots 0)_{\text{ms}}.$$

**Plage** (sur  $n$  bits) :

plus petit entier :  $(11 \dots 1)_{\text{ms}} = -(2^{n-1} - 1)$  ;

plus grand entier :  $(01 \dots 1)_{\text{ms}} = 2^{n-1} - 1$ .

# Représentation en complément à un

Le **complément à un** d'une suite de bits

$$u_{n-1} \dots u_1 u_0$$

est la suite

$$\overline{u_{n-1}} \dots \overline{u_1} \overline{u_0},$$

où  $\overline{0} := 1$  et  $\overline{1} := 0$ .

Le codage en **complément à un** consiste à coder un entier négatif par le complément à un de la représentation binaire de sa valeur absolue.

Le codage d'un entier positif est son codage binaire habituel.

Le bit de poids fort doit être un bit de signe : lorsqu'il est égal à 1, l'entier représenté est négatif ; il est positif dans le cas contraire.

**Remarque** : sur  $n$  bits, le complément à un revient à représenter un entier négatif  $x$  par la représentation binaire de  $(2^n - 1) - |x|$ .

# Représentation en complément à un

P.ex., sur 8 bits, on a

$$(98)_{\text{dix}} = (01100010)_{\text{c1}} \quad \text{et} \quad (-98)_{\text{dix}} = (10011101)_{\text{c1}}.$$

**Avantage** : calcul facile de l'opposé d'un entier.

**Inconvénient** : l'addition de deux entiers ne peut pas se faire en suivant l'algorithme classique d'addition.

Il y a de plus deux encodages différents pour zéro (qui est pos. et nég.) :

$$(0)_{\text{dix}} = (00 \dots 0)_{\text{c1}} = (11 \dots 1)_{\text{c1}}.$$

**Plage** (sur  $n$  bits) :

plus petit entier :  $(10 \dots 0)_{\text{c1}} = -(2^{n-1} - 1)$  ;

plus grand entier :  $(01 \dots 1)_{\text{c1}} = 2^{n-1} - 1$ .

# Représentation avec biais

On fixe un entier  $B \geq 0$  appelé **biais**.

Soit  $x$  un entier (positif ou non) que l'on souhaite représenter.

Posons  $x' := x + B$ .

Si  $x' \geq 0$ , alors le codage de  $x$  **avec un biais de  $B$**  est la représentation binaire de  $x'$  sur  $n$  bits. Sinon,  $x$  n'est pas représentable (sur  $n$  bits et avec le biais  $B$  donné).

P.ex., en se plaçant sur 8 bits, avec un biais de  $B := 95$ , on a

- $(30)_{\text{dix}} = (01111101)_{\text{biais}=95}$ .

En effet,  $30 + 95 = 125$  et  $(01111101)_{\text{deux}} = (125)_{\text{dix}}$ .

- $(-30)_{\text{dix}} = (01000001)_{\text{biais}=95}$ .

En effet,  $-30 + 95 = 65$  et  $(01000001)_{\text{deux}} = (65)_{\text{dix}}$ .

- l'entier  $(-98)_{\text{dix}}$  n'est pas représentable car  $-98 + 95 = -3$  est négatif.

# Représentation avec biais

**Note** : ce codage n'est pas compatible avec le bit de signe. En effet, le bit de poids fort d'un entier codé en représentation avec biais ne renseigne pas sur le signe de l'entier.

**Avantages** : permet de représenter des intervalles quelconques (mais pas trop larges) de  $\mathbb{Z}$ .

**Inconvénient** : l'addition de deux entiers ne peut pas se faire en suivant l'algorithme classique d'addition.

**Plage** (sur  $n$  bits) :

plus petit entier :  $(0 \dots 0)_{\text{biais}=B} = -B$  ;

plus grand entier :  $(1 \dots 1)_{\text{biais}=B} = (2^n - 1) - B$ .

# Représentation en complément à deux

On se place sur  $n$  bits.

Le **complément à deux** (puissance  $n$ ) d'une suite de bits  $u$  se calcule en

- 1 complémentant  $u$  à un pour obtenir  $u'$  ;
- 2 incrémentant  $u'$  (addition de  $u'$  et  $0\dots 01$ ) pour obtenir  $u''$ .

P.ex., sur 8 bits, avec  $u := 01101000$ , on obtient

- 1  $u' = 10010111$  ;
- 2  $u'' = 10011000$ .

Ainsi, le complément à deux de la suite de bits  $01101000$  est la suite de bits  $10011000$ .

**Remarque** : l'opération qui consiste à complémenter à deux une suite de bits est involutive (le complément à deux du complément à deux d'une suite de bits  $u$  est  $u$ ).

# Représentation en complément à deux — codage/décodage

Le codage en **complément à deux** consiste à coder un entier  $k$  de la manière suivante.

- Si  $k \geq 0$ , le codage de  $k$  est simplement son codage binaire habituel.
- Sinon,  $k < 0$ . Le codage de  $k$  est le complément à deux du codage binaire de la valeur absolue de  $k$ .

Pour **décoder** une suite de bits  $u$  représentant un entier codé en complément à deux, on procède de la manière suivante.

- Si le bit de poids fort de  $u$  est 0, on obtient la valeur codée par  $u$  en interprétant directement sa valeur en binaire.
- Sinon, le bit de poids fort de  $u$  est 1. On commence par considérer le complément à deux  $u''$  de  $u$ . La valeur codée par  $u$  est obtenue en interprétant la valeur de  $u''$  en binaire et en considérant son opposé.

## Représentation en complément à deux — remarques

Le codage en complément à deux fait que le **bit de poids fort** est un **bit de signe** : lorsque le bit de poids fort est 1, l'entier représenté est négatif. Il est positif dans le cas contraire.

Une valeur  $k$  est **codable sur  $n$  bits** en complément à deux si et seulement si la suite de bits obtenue par le codage de  $k$  en complément à deux sur  $n$  bits possède un **bit de signe cohérent** avec le signe de  $k$ .

Sur  $n$  bits, le complément à deux revient à représenter un entier strictement négatif  $x$  par  $2^n - |x|$ .

# Représentation en complément à deux — exemples

**Représentation de  $(98)_{\text{dix}}$  sur 8 bits.** On a  $(98)_{\text{dix}} = (01100010)_{\text{deux}}$ .  
Cohérence avec le bit de signe. Ainsi,  $(98)_{\text{dix}} = (01100010)_{\text{c2}}$ .

**Représentation de  $(-98)_{\text{dix}}$  sur 8 bits.** Le complément à deux de  $01100010$  est  $10011110$ . Cohérence avec le bit de signe. Ainsi,  $(-98)_{\text{dix}} = (10011110)_{\text{c2}}$ .

**Représentation de  $(130)_{\text{dix}}$  sur 8 bits.** On a  $(130)_{\text{dix}} = (10000010)_{\text{deux}}$ .  
Incohérence avec le bit de signe : il est à 1 alors que l'entier est positif. Cet entier n'est pas représentable sur 8 bits.

**Représentation de  $(-150)_{\text{dix}}$  sur 8 bits.** On a  $(150)_{\text{dix}} = (10010110)_{\text{deux}}$ . Le complément à deux de cette suite est  $01101010$ . Incohérence avec le bit de signe : il est à 0 alors que l'entier est négatif. Cet entier n'est pas représentable sur 8 bits.

**Représentation de  $(-150)_{\text{dix}}$  sur 10 bits.** On a  $(150)_{\text{dix}} = (0010010110)_{\text{deux}}$ . Le complément à deux de cette suite est  $1101101010$ . Cohérence avec le bit de signe. Ainsi,  $(-150)_{\text{dix}} = (1101101010)_{\text{c2}}$ .

## Représentation en complément à deux — exemples

**Décodage de  $(00110101)_{c2}$  sur 8 bits.** Le bit de poids fort est 0. Ainsi, l'entier codé est positif et  $(00110101)_{c2} = (00110101)_{\text{deux}}$ . On a donc  $(00110101)_{c2} = (53)_{\text{dix}}$ .

**Décodage de  $(10110101)_{c2}$  sur 8 bits.** Le bit de poids fort est 1. Ainsi, l'entier codé est négatif. On doit considérer le complément à deux de la suite 10110101 qui est 01001011. On a maintenant  $(01001011)_{\text{deux}} = (75)_{\text{dix}}$ . Ainsi,  $(10110101)_{c2} = (-75)_{\text{dix}}$ .

**Décodage de  $(11111111)_{c2}$  sur 8 bits.** Le bit de poids fort est 1. Ainsi, l'entier codé est négatif. On doit considérer le complément à deux de la suite 11111111 qui est 00000001. On a maintenant  $(00000001)_{\text{deux}} = (1)_{\text{dix}}$ . Ainsi,  $(11111111)_{c2} = (-1)_{\text{dix}}$ .

# Représentation en complément à deux

**Avantage** : l'addition de deux entiers se calcule selon l'algorithme classique d'addition.

**Inconvénient** : représentation des entiers négatifs légèrement plus complexe que par les autres codages.

**Plage** (sur  $n$  bits) :

plus petit entier :  $(10 \dots 0)_{c2} = -2^{n-1}$  ;

plus grand entier :  $(01 \dots 1)_{c2} = 2^{n-1} - 1$ .

Dans les ordinateurs d'aujourd'hui, les entiers sont habituellement encodés selon cette représentation.

# Représentation en complément à deux — calcul rapide

Il existe une **méthode plus rapide** pour calculer le complément à deux d'une suite de bits  $u$  que de la complémenter à un et de l'incrémenter.

Elle consiste en les deux étapes suivantes :

- 1 repérer le bit à 1 de poids le plus faible de  $u$ ;
- 2 complémenter à un les bits de  $u$  qui sont de poids strictement plus forts que ce bit.

P.ex., l'application de cette méthode sur la suite 01101000 donne :

$u$	0	1	1	0	1	0	0	0	
bit à 1 à droite	0	1	1	0	1	0	0	0	
cpl. à 1 bits à gauche	1	0	0	1	1	0	0	0	cpl. à 2 de $u$

# Dépassement de capacité

En représentation en complément à deux sur  $n$  bits, l'addition de deux entiers  $x$  et  $y$  peut produire un entier qui sort de la plage représentable. On appelle ceci un **dépassement de capacité** (*overflow*).

**Fait 1.** Si  $x$  est négatif et  $y$  est positif,  $x + y$  appartient toujours à la plage représentable.

**Fait 2.** Il ne peut y avoir dépassement de capacité que si  $x$  et  $y$  sont de **même signe**.

**Règle :** il y a dépassement de capacité si et seulement si le **bit de poids fort** de  $x + y$  est **différent** du bit de poids fort de  $x$  (et donc de  $y$ ).

P.ex., sur 4 bits, l'addition

$$\begin{array}{rcccc} & 0 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

engendre un dépassement de capacité.

# Retenue de sortie

En représentation en complément à deux sur  $n$  bits, on dit que l'addition de deux entiers  $x$  et  $y$  produit une **retenue de sortie** si l'addition des bits de poids forts de  $x$  et de  $y$  renvoie une retenue de sortie à 1.

P.ex., sur 4 bits, l'addition

$$\begin{array}{rcccc} & & 1 & 1 & 0 & 0 \\ + & & 0 & 1 & 0 & 1 \\ \hline (1) & 0 & 0 & 0 & 0 & 1 \end{array}$$

engendre une retenue de sortie.

**Attention** : ce n'est pas parce qu'une addition provoque une retenue de sortie qu'il y a dépassement de capacité.

# Dépassement de capacité et retenue de sortie

Le dépassement de capacité (**D**) et la retenue de sortie (**R**) sont deux choses disjointes. En effet, il peut exister les quatre cas de figure suivants (exemples sur 4 bits) :

■ non (**D**) et non (**R**) :

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \\ \hline (0) \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \end{array}$$

■ non (**D**) et (**R**) :

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{0} \\ \hline (1) \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{0} \end{array}$$

■ (**D**) et non (**R**) :

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{1} \phantom{1} \phantom{1} \\ \hline (0) \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{1} \end{array}$$

■ (**D**) et (**R**) :

$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{1} \phantom{0} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{1} \phantom{0} \phantom{0} \phantom{0} \\ \hline (1) \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{0} \end{array}$$

# Résumé des représentations des entiers

## Représentation en magnitude signée.

**Plage** : de  $-2^{n-1} + 1$  à  $2^{n-1} - 1$ .

**Avantage** : codage simple ; calcul facile de l'opposé.

**Inconvénients** : addition non naturelle ; deux encodages pour zéro.

## Représentation en complément à un.

**Plage** : de  $-2^{n-1} + 1$  à  $2^{n-1} - 1$ .

**Avantage** : codage simple ; calcul facile de l'opposé.

**Inconvénients** : addition non naturelle ; deux encodages pour zéro.

## Représentation avec biais (de $B \geq 0$ ).

**Plage** : de  $-B$  à  $(2^n - 1) - B$ .

**Avantage** : possibilité d'encoder un intervalle arbitraire.

**Inconvénient** : addition non naturelle.

## Représentation en complément à deux.

**Plage** : de  $-2^{n-1}$  à  $2^{n-1} - 1$ .

**Avantage** : addition naturelle.

**Inconvénient** : encodage moins intuitif.

# L'hexadécimal

Un entier codé en **hexadécimal** est un entier écrit en base seize.

Ce codage utilise seize symboles : les chiffres 0, 1, ..., 9 et les lettres A, B, C, D, E, F.

Pour exprimer une **suite de bits**  $u$  en hexadécimal, on remplace, en partant de la droite, chaque groupe de quatre bits de  $u$  par un chiffre hexadécimal au moyen de la table suivante :

$(0000)_{\text{deux}} \rightarrow (0)_{\text{hex}}$	$(0100)_{\text{deux}} \rightarrow (4)_{\text{hex}}$	$(1000)_{\text{deux}} \rightarrow (8)_{\text{hex}}$	$(1100)_{\text{deux}} \rightarrow (C)_{\text{hex}}$
$(0001)_{\text{deux}} \rightarrow (1)_{\text{hex}}$	$(0101)_{\text{deux}} \rightarrow (5)_{\text{hex}}$	$(1001)_{\text{deux}} \rightarrow (9)_{\text{hex}}$	$(1101)_{\text{deux}} \rightarrow (D)_{\text{hex}}$
$(0010)_{\text{deux}} \rightarrow (2)_{\text{hex}}$	$(0110)_{\text{deux}} \rightarrow (6)_{\text{hex}}$	$(1010)_{\text{deux}} \rightarrow (A)_{\text{hex}}$	$(1110)_{\text{deux}} \rightarrow (E)_{\text{hex}}$
$(0011)_{\text{deux}} \rightarrow (3)_{\text{hex}}$	$(0111)_{\text{deux}} \rightarrow (7)_{\text{hex}}$	$(1011)_{\text{deux}} \rightarrow (B)_{\text{hex}}$	$(1111)_{\text{deux}} \rightarrow (F)_{\text{hex}}$

P.ex.,  $(10\ 1111\ 0101\ 1011\ 0011\ 1110)_{\text{deux}} = (2F5B3E)_{\text{hex}}$ .

La conversion dans l'autre sens se réalise en appliquant la même idée.

On utilise ce codage pour sa **concision** : un octet est codé par seulement deux chiffres hexadécimaux.

Nombres entiers vs nombres réels :

- il y a une infinité de nombre entiers mais il y a un **nombre fini** d'entiers dans tout intervalle  $[[a, b]]$  où  $a \leq b \in \mathbb{Z}$ ;
- il y a également une infinité de nombre réels mais il y a cette fois un **nombre infini** de réels dans tout intervalle  $[\alpha, \beta]$  où  $\alpha < \beta \in \mathbb{R}$ .

**Conséquence** : il n'est possible de représenter qu'un sous-ensemble de nombres réels d'un intervalle donné.

Les nombres représentables sont appelés **nombre flottants**. Ce sont des approximations des nombres réels.

# Représentation à virgule fixe

Le codage à **virgule fixe** consiste à représenter un nombre à virgule en deux parties :

- 1 sa **troncature**, sur  $n$  bits ;
- 2 sa **partie décimale**, sur  $m$  bits ;

où les entiers  $n$  et  $m$  sont fixés.

La troncature est codée en utilisant la représentation en complément à deux.

Chaque bit de la partie décimale correspond à l'inverse d'une puissance de deux.

P.ex., avec  $n = 4$  et  $m = 5$ ,

$$\begin{aligned}(0110.01100)_{\text{vf}} &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= (6.375)_{\text{dix}}\end{aligned}$$

# Représentation à virgule fixe

Soient  $b \geq 2$  un entier et  $0 < x < 1$  un nombre rationnel. Pour écrire  $x$  en **base  $b$** , on procède comme suit :

- 1  $L \leftarrow []$
- 2 Tant que  $x \neq 0$  :
  - 1  $x \leftarrow x \times b$
  - 2  $L \leftarrow L \cdot \lfloor x \rfloor$
  - 3  $x \leftarrow x - \lfloor x \rfloor$
- 3 Renvoyer  $L$ .

Ici,  $[]$  est la liste vide,  $\cdot$  désigne la concaténation des listes et  $\lfloor - \rfloor$  est l'opérateur partie entière inférieure.

# Représentation à virgule fixe

P.ex., avec  $x := (0.375)_{\text{dix}}$  et  $b := 2$ , on a

$x$	$x \times 2$	$x - \lfloor x \rfloor$	$L$
$(0.375)_{\text{dix}}$	$(0.75)_{\text{dix}}$	$(0.75)_{\text{dix}}$	$[0]$
$(0.75)_{\text{dix}}$	$(1.5)_{\text{dix}}$	$(0.5)_{\text{dix}}$	$[0, 1]$
$(0.5)_{\text{dix}}$	$(1.0)_{\text{dix}}$	$(0.0)_{\text{dix}}$	$[0, 1, 1]$
$(0)_{\text{dix}}$	-	-	$[0, 1, 1]$

Ainsi,  $(0.375)_{\text{dix}} = (0.011)_{\text{vf}}$ .

# Limites de la représentation à virgule fixe

Appliquons l'« algorithme » précédent sur les entrées  $x := (0.1)_{\text{dix}}$  et  $b := 2$ . On a

$x$	$x \times 2$	$x - \lfloor x \rfloor$	$L$
$(0.1)_{\text{dix}}$	$(0.2)_{\text{dix}}$	$(0.2)_{\text{dix}}$	$[0]$
$(0.2)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$[0, 0]$
$(0.4)_{\text{dix}}$	$(0.8)_{\text{dix}}$	$(0.8)_{\text{dix}}$	$[0, 0, 0]$
$(0.8)_{\text{dix}}$	$(1.6)_{\text{dix}}$	$(0.6)_{\text{dix}}$	$[0, 0, 0, 1]$
$(0.6)_{\text{dix}}$	$(1.2)_{\text{dix}}$	$(0.2)_{\text{dix}}$	$[0, 0, 0, 1, 1]$
$(0.2)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$(0.4)_{\text{dix}}$	$[0, 0, 0, 1, 1, 0]$
$\dots$	$\dots$	$\dots$	$\dots$

Ainsi,

$$(0.1)_{\text{dix}} = (0.000110\ 0110\ 0110\dots)_{\text{vf}}.$$

Ce rationnel n'admet pas d'écriture finie en représentation à virgule fixe.

# Représentation IEEE 754

Le codage **IEEE 754** consiste à représenter un nombre à virgule  $x$  par trois données :

- 1 un **bit de signe**  $s$  ;
- 2 un **exposant**  $e$  ;
- 3 une **mantisse**  $m$ .



Principaux formats (tailles en bits) :

Nom	Taille totale	Bit de signe	Exposant	Mantisse
half	16	1	5	10
single	32	1	8	23
double	64	1	11	52
quad	128	1	15	112

# Représentation IEEE 754

Le **bit de signe**  $s$  renseigne sur le signe : si  $s = 0$ , le nombre codé est positif, sinon il est négatif.

L'**exposant**  $e$  code un entier en représentation avec biais avec un biais donné par la la table suivante.

Nom	Biais $B$
half	15
single	127
double	1023
quad	16383

On note  $\text{vale}(e)$  la valeur ainsi codée.

La **mantisse**  $m$  code la partie décimale d'un nombre de la forme  $(1.m)_{\text{vf}}$ .

On note  $\text{valm}(m)$  la valeur ainsi codée.

Le nombre à virgule est codé par  $s$ ,  $e$  et  $m$  si l'on a

$$x = (-1)^s \times \text{valm}(m) \times 2^{\text{vale}(e)}.$$

# Représentation IEEE 754

Codons le nombre  $x := (-23.375)_{\text{dix}}$  en single.

1 Comme  $x$  est négatif,  $s := 1$ .

2 On écrit  $|x|$  en représentation à virgule fixe. On obtient

$$|x| = (10111.011)_{\text{vf}}$$

et donc,

$$|x| = \text{valm}(0111011) \times 2^4.$$

3 On en déduit  $\text{vale}(e) = 4$  et ainsi,

$$e = (10000011)_{\text{biais}=127}.$$

4 On en déduit par ailleurs que

$$m = 011101100000000000000000.$$

Finalement,

$$x = (1\ 10000011\ 011101100000000000000000)_{\text{IEEE 754 single}}.$$

# Représentation IEEE 754

Codons le nombre  $x := (0.15625)_{\text{dix}}$  en single.

1 Comme  $x$  est positif,  $s := 0$ .

2 On écrit  $|x|$  en représentation à virgule fixe. On obtient

$$|x| = (0.00101)_{\text{vf}}$$

et donc,

$$|x| = \text{valm}(01) \times 2^{-3}.$$

3 On en déduit  $\text{vale}(e) = -3$  et ainsi,

$$e = (01111100)_{\text{biais}=127}.$$

4 On en déduit par ailleurs que

$$m = 010000000000000000000000.$$

Finalement,

$$x = (0\ 01111100\ 010000000000000000000000)_{\text{IEEE 754 single}}.$$

Il existe plusieurs **représentations spéciales** pour coder certains éléments.

Valeur	<i>s</i>	<i>e</i>	<i>m</i>
Zéro	0	0...0	0...0
+Infini	0	1...1	0...0
-Infini	1	1...1	0...0
NaN	0	1...1	010...0

« NaN » signifie « Not a Number ». C'est un code qui permet de représenter une valeur mal définie provenant d'une opération qui n'a pas de sens (p.ex.,  $\frac{0}{0}$ ,  $\infty - \infty$  ou  $0 \times \infty$ ).

# Le code ASCII

ASCII est l'acronyme de **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. Ce codage des caractères fut introduit dans les années 1960.

Un caractère occupe **un octet** dont le bit de poids fort vaut 0.

La correspondance octet (en héxa.) / caractère est donnée par la table

	0x	1x	2x	3x	4x	5x	6x	7x
x0	NUL	DLE	esp.	0	@	P	'	p
x1	SOH	DC1	!	1	A	Q	a	q
x2	STX	DC2	"	2	B	R	b	r
x3	ETX	DC3	#	3	C	S	c	s
x4	EOT	DC4	\$	4	D	T	d	t
x5	ENQ	NAK	%	5	E	U	e	u
x6	ACK	SYN	&	6	F	V	f	v
x7	BEL	ETB	'	7	G	W	g	w
x8	BS	CAN	(	8	H	X	h	x
x9	HT	EM	)	9	I	Y	i	y
xA	LF	SUB	*	:	J	Z	j	z
xB	VT	ESC	+	;	K	[	k	{
xC	FF	FS	,	<	L	\	l	
xD	CR	GS	-	=	M	]	m	}
xE	SO	RS	.	>	N	^	n	~
xF	SI	US	/	?	O	_	o	DEL

# Le code ISO 8859-1

Ce codage est également appelé **Latin-1** ou **Europe occidentale** et fut introduit en 1986.

Un caractère occupe **un octet**. La valeur du bit de poids fort n'est plus fixée.

À la différence du code ASCII, ce codage permet en plus de représenter, entre autres, des lettres accentuées.

Ce codage des caractères est aujourd'hui (2016) de moins en moins utilisé.

# Le code Unicode

Le codage **Unicode** est une version encore étendue du code ASCII qui fut introduite en 1991.

Chaque caractère est représenté sur **deux octets**.

Il permet ainsi de représenter une large variété de caractères : caractères latins (accentués ou non), grecs, cyrilliques, *etc.*

Il existe des extensions où plus d'octets encore par caractère sont utilisés.

**Problème** : un texte encodé en Unicode prend plus de place qu'en ASCII.

# Le code UTF-8

Le codage **UTF-8** apporte une réponse satisfaisante au problème précédent.

Voici comment un caractère  $c$  se représente selon le codage UTF-8 :

- si  $c$  est un caractère qui peut être représenté par le codage ASCII, alors  $c$  est représenté par son code ASCII ;
- sinon,  $c$  peut être représenté par le codage Unicode. Il est codé par **trois octets**



où



est la suite des deux octets du code Unicode de  $c$ .

Lorsqu'un texte contient principalement des caractères ASCII, son codage est en général moins coûteux en place que le codage Unicode.

De plus, il est **rétro-compatible** avec le codage ASCII.

Un **texte** est une suite de caractères.

Chacun des codages de caractères vus précédemment produit un codage de texte. En effet, un texte est représenté par la suite de bits obtenue en remplaçant chacun de ses caractères par son codage.

Un codage de texte est un **code** si toute suite de bits se décode en au plus un texte (il n'y a pas d'ambiguïté sur l'interprétation d'une suite de bits).

**Exercice** : vérifier que les codages ASCII, Latin-1, Unicode et UTF-8 sont bien des codes.

Un langage de programmation **bas niveau** est un langage qui dépend fortement de la structure matérielle de la machine.

Cette caractéristique offre des avantages et des inconvénients divers.

P.ex., elle permet

- d'avoir un bon contrôle des **ressources matérielles** ;
- d'avoir un contrôle très fin sur la **mémoire** ;
- souvent, de produire du code dont l'exécution est très **rapide**.

En revanche, elle ne permet pas

- d'utiliser des techniques de programmation abstraites ;
- de programmer rapidement et facilement.

# Langage machine

Le **langage machine** est un langage compris directement par un processeur donné en vue d'une exécution.

C'est un **langage binaire** : ses seules lettres sont les bits 0 et 1.

Chaque modèle de processeur possède son propre langage machine. Étant donnés deux modèles de processeurs  $x_1$  et  $x_2$ , on dit que  $x_1$  est **compatible** avec  $x_2$  si toute instruction formulée pour  $x_2$  peut être comprise et exécutée par  $x_1$ .

Dans la plupart des langages machine, une instruction commence par un **opcode**, une suite de bits qui porte la nature de l'instruction. Celui-ci est suivi des suites de bits codant les **opérandes** de l'instruction.

P.ex., la suite

01101010 00010101

est une instruction dont le opcode est 01101010 et l'opérande est 00010101. Elle ordonne de placer la valeur  $(21)_{\text{dix}}$  en tête de la pile.

# Langages d'assemblage

Un **langage d'assemblage** (ou assembleur) est un langage qui se trouve à mi-distance entre le programmeur et la machine en terme de facilité d'accès.

En effet, d'un côté, la machine peut convertir presque immédiatement un programme en assembleur vers du langage machine. De l'autre, l'assembleur est un langage assez facile à manipuler pour un humain.

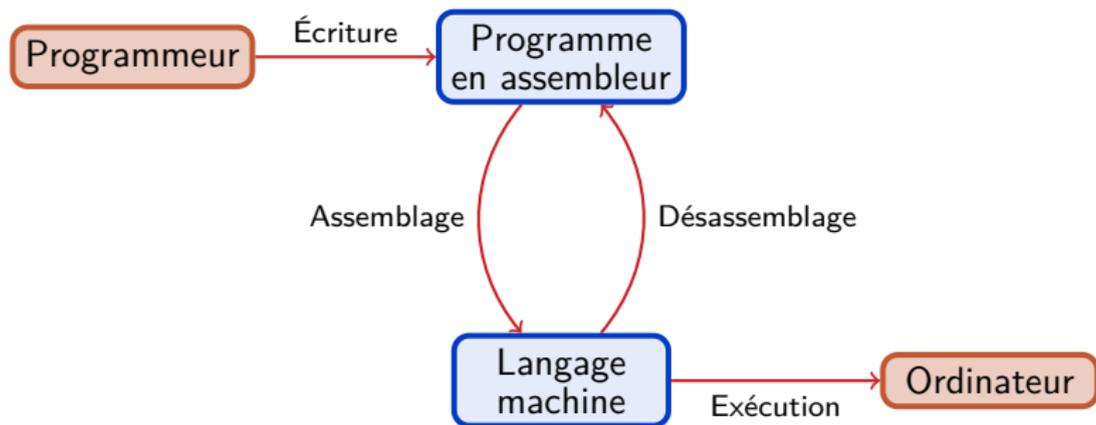
Les opcodes sont codés via des **mnémoniques**, mots-clés bien plus manipulables pour le programmeur que les suites binaires associées.

Du fait qu'un langage d'assemblage est spécifiquement dédié à un processeur donné, il existe presque autant de langages d'assemblage qu'il y a de modèles de processeurs.

# Langages d'assemblage et assembleurs

L'**assemblage** est l'action d'un programme nommé **assembleur** qui consiste à traduire un programme en assembleur vers du langage machine.

Le **désassemblage**, réalisé par un **désassembleur**, est l'opération inverse. Elle permet de retrouver, à partir d'un programme en langage machine, un programme en assembleur qui lui correspond.



# L'assembleur NASM

Pour des raisons pédagogiques, nous choisissons de travailler sur une architecture **x86** en 32 bits. C'est une architecture dont le modèle de processeur est compatible avec le modèle INTEL 8086.

Nous utiliserons l'**assembleur NASM** (Netwide Assembler).

Pour programmer, il faudra disposer :

- 1 d'un ordinateur (moderne) ;
- 2 d'un système LINUX en 32 bits (réel ou virtuel) ou 64 bits ;
- 3 d'un éditeur de textes ;
- 4 du programme `nasm` (assembleur) ;
- 5 du programme `ld` (lieur) ou `gcc` ;
- 6 du programme `gdb` (débugueur).

Un **programme assembleur** est un fichier texte d'extension `.asm`.

Il est constitué de plusieurs parties dont le rôle est

- 1 d'invoquer des **directives** ;
- 2 de définir des **données initialisées** ;
- 3 de réserver de la mémoire pour des **données non initialisées** ;
- 4 de contenir une suite **instructions**.

Nous allons étudier chacune de ces parties.

Avant cela, nous avons besoin de nous familiariser avec trois ingrédients de base dans la programmation assembleur : les **valeurs**, les **registres** et la **mémoire**.

# Exprimer des valeurs

Il y a plusieurs manières d'exprimer des valeurs en assembleur.

On peut exprimer des **entiers** (représentés par leurs suites de bits obtenues en repr. complément à deux) :

- directement en base dix, p.ex., 0, 10020, -91 ;
- en hexadécimal, avec le préfixe 0x, p.ex., 0xA109C, -0x98 ;
- en binaire, avec le préfixe 0b, p.ex., 0b001, 0b11101.

On peut exprimer des **caractères** (en repr. ASCII) :

- directement, p.ex., 'a', '9' ;
- par leur code ASCII, p.ex., 10, 120.

On peut exprimer des **chaînes de caractères** (comme suites de carac.) :

- directement, p.ex., 'abbaa', 0 ;
- caractère par caractère, p.ex., 'a', 'a', 46, 36, 0.

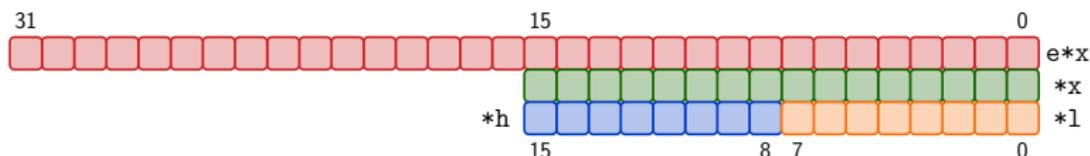
Le code ASCII du marqueur de fin de chaîne est 0 (à ne pas oublier).

# Registres et sous-registres

Un **registre** est un emplacement de 32 bits.

On peut le considérer comme une **variable globale**.

Il y a quatre **registres de travail** : `eax`, `ebx`, `ecx` et `edx`. Ils sont subdivisés en **sous-registres** selon le schéma suivant :



P.ex., `bh` désigne le 2<sup>e</sup> octet de `ebx` et `cx` désigne les deux 1<sup>ers</sup> octets de `ecx`.

Il est possible d'écrire/lire dans chacun de ces (sous-)registres.

**Attention** : toute modification d'un sous-registre entraîne une modification du registre tout entier (et réciproquement).

# Registres de diverses sortes

Il existe d'autres registres. Parmi ceux-ci, il y a

- le **pointeur d'instruction** `eip`, contenant l'adresse de la prochaine instruction à exécuter ;
- le **pointeur de tête de pile** `esp`, contenant l'adresse de la tête de la pile ;
- le **pointeur de pile** `ebp`, utilisé pour contenir l'adresse d'une donnée de la pile dans les fonctions ;
- le **registre de drapeaux** `flags`, utilisé pour contenir des informations sur le résultat d'une opération qui vient d'être réalisée.

**Attention** : ce ne sont pas des registres de travail, leur rôle est fixé. Même s'il est possible pour certains d'y écrire / lire explicitement, il faut essayer, pour la plupart, de le faire le moins possible.



# L'opération `mov` — respect des tailles

Il est important, pour que l'instruction `mov REG, VAL` soit correcte, que la taille en octets de la valeur `VAL` soit la même que celle du (sous-)registre `REG`.

P.ex., les instructions suivantes **ne sont pas correctes** :

- `mov al, 0xA5A5`

Le `ss`-registre `al` occupe 1 octet alors que la valeur `0xA5A5` en occupe 2.

- `mov ax, eax`

Le `ss`-registre `ax` occupe 2 octets alors que la valeur contenue dans `eax` en occupe 4.

- `mov eax, ax`

Le `ss`-registre `eax` occupe 4 octets alors que la valeur contenue dans `ax` en occupe que 2.

En revanche, les instructions suivantes **sont correctes** :

- `mov al, 0xA5`

- `mov ax, 0xF`

Le `ss`-registre `ax` occupe 2 octets alors que la valeur `0xF` n'en occupe que 1. Néanmoins, cette valeur est étendue sur 2 octets sans perte d'information en `0x000F`.

# Opérations sur les registres — arithmétique

## Opérations **arithmétiques** :

- `add REG, VAL`

incrmente le (sous-)registre REG de la valeur VAL ;

- `sub REG, VAL`

décrémente le (sous-)registre REG de la valeur VAL ;

- `mul VAL`

multiplie la valeur contenue dans `eax` et `VAL` et place le résultat dans `edx:eax` (c.-à-d. la suite de 64 bits dont les bits de `edx` sont ceux de poids forts et ceux de `eax` ceux de poids faibles) ;

- `div VAL`

place le quotient de la division de `edx:eax` par la valeur `VAL` dans `eax` et le reste dans `edx`.

P.ex.,

```
mov eax, 20           ; eax = 20
add eax, 51           ; eax = 71
add eax, eax          ; eax = 142
```

# Opérations sur les registres — logique

Opérations **logiques** :

- not REG

place dans le (sous-)registre REG la valeur obtenue en réalisant le *non* bit à bit de sa valeur ;

- and REG, VAL

place dans le (sous-)registre REG la valeur du *et* logique bit à bit entre les suites contenues dans REG et VAL ;

- or REG, VAL

place dans le (sous-)registre REG la valeur du *ou* logique bit à bit entre les suites contenues dans REG et VAL ;

- xor REG, VAL

place dans le (sous-)registre REG la valeur du *ou exclusif* logique bit à bit entre les suites contenues dans REG et VAL.

# Opérations sur les registres — bit à bit

Opérations **bit à bit** ;

- `shl REG, NB`

décale les bits du (sous-)registre `REG` à gauche de `NB` places et complète à droite par des 0 ;

- `shr REG, NB`

décale les bits du (sous-)registre `REG` à droite de `NB` places et complète à gauche par des 0 ;

- `rol REG, NB`

réalise une rotation des bits du (sous-)registre `REG` à gauche de `NB` places ;

- `ror REG, NB`

réalise une rotation des bits du (sous-)registre `REG` à droite de `NB` places.

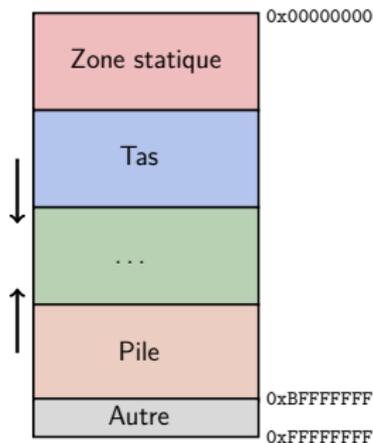
# Mémoire

Les registres n'offrent pas assez de mémoire pour construire des programmes élaborés. C'est pour cette raison que l'on utilise la **mémoire**.

La mémoire est segmentée en plusieurs parties :

- la **zone statique** qui contient le code et les données statiques ;
- le **tas**, de taille variable au fil de l'exécution ;
- la **pile**, de taille variable au fil de l'exécution.

Il y a d'autres zones (non repr. ici).



En **mode protégé**, chaque programme en exécution possède son propre environnement de mémoire. Les adresses y sont relatives et non absolues.

De cette manière, un programme en exécution ne peut empiéter sur la mémoire d'un autre.

La lecture / écriture en mémoire suit la convention *little-endian*.

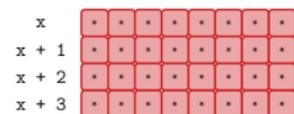
# Lecture en mémoire

## L'instruction

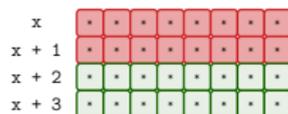
```
mov REG, [ADR]
```

place dans le (sous-)registre REG un, deux ou quatre octets en fonction de la taille de REG, **lus** à partir de l'adresse ADR **dans la mémoire**.

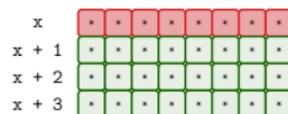
En supposant que  $x$  soit une adresse accessible en mémoire, les parties rouges sont celles qui sont lues et placées dans le (sous-)registre opérande de l'instruction :



```
mov eax, [x]
```



```
mov ax, [x]
```



```
mov ah, [x]
```

ou

```
mov al, [x]
```



# Écriture en mémoire

L'instruction

`mov DT [ADR], VAL`

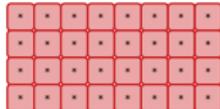
**écrit dans la mémoire** à partir de l'adresse ADR la valeur VAL.

Le champ DT est un descripteur de taille qui permet de préciser la taille de VAL en octet selon la table suivante :

Descripteur de taille	Taille (en octets)
byte	1
word	2
dword	4

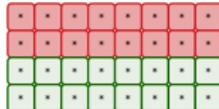
En supposant que  $x$  soit une adresse accessible en mémoire et  $val$  une valeur, les parties rouges de la mémoire sont celles qui sont modifiées :

$x$   
 $x + 1$   
 $x + 2$   
 $x + 3$



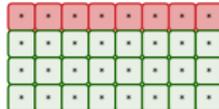
`mov dword [x], val`

$x$   
 $x + 1$   
 $x + 2$   
 $x + 3$



`mov word [x], val`

$x$   
 $x + 1$   
 $x + 2$   
 $x + 3$



`mov byte [x], val`

## Écriture en mémoire — tailles

Le champ `val` peut être un (sous-)registre. Dans ce cas, il n'est pas nécessaire de préciser le descripteur de taille (le nombre d'octets écrits en mémoire dépend de la taille du (sous-)registre).

P.ex., les instructions suivantes **ne sont pas correctes** (ici, `x` est une adresse accessible en mémoire) :

- `mov byte [x], eax`

Le registre `eax` occupe 4 octets, ce qui est contradictoire avec le descripteur `byte` (1 octet).

- `mov word [x], bl`

Le sous-registre `bl` occupe 1 octet, ce qui est contradictoire avec le descripteur `word` (2 octets).

- `mov byte [x], 0b010010001`

La donnée à écrire tient sur au moins 2 octets (9 bits), ce qui est contradictoire avec le descripteur `byte` (1 octet).

- `mov [x], -125`

La taille de la donnée à écrire n'est pas connue.

# Écriture en mémoire — tailles

En revanche, les instructions suivantes **sont correctes** :

- `mov [x], eax`

Le registre `eax` occupe implicitement 4 octets.

- `mov dword [x], eax`

Ceci est correct, bien que pléonastique.

- `mov word [x], 0b010010001`

La donnée à écrire est vue sur 16 bits, en ajoutant des 0 à gauche.

- `mov dword [x], 0b010010001`

La donnée à écrire est vue sur 32 bits, en ajoutant des 0 à gauche.

- `mov word [x], -125`

La donnée à écrire est vue sur 2 octets, en ajoutant des 1 à gauche car elle est négative.

# Écriture en mémoire — exemple

Observons l'effet des instructions avec la mémoire dans l'état suivant :

x	1	0	0	0	0	0	0	1
x + 1	1	1	1	1	1	1	1	1
x + 2	1	1	0	0	0	0	1	1
x + 3	1	0	1	0	1	0	1	0

```
mov eax, 0xAA11BB50
mov [x], ah
mov [x], ax
mov [x], eax
mov byte [x + 1], 0
mov dword [x], 0x5
```

x	1	0	0	0	0	0	0	1
x+1	1	1	1	1	1	1	1	1
x+2	1	1	0	0	0	0	1	1
x+3	1	0	1	0	1	0	1	0

x	1	0	1	1	1	0	1	1
x+1	1	1	1	1	1	1	1	1
x+2	1	1	0	0	0	0	1	1
x+3	1	0	1	0	1	0	1	0

x	0	1	0	1	0	0	0	0
x+1	1	0	1	1	1	0	1	1
x+2	1	1	0	0	0	0	1	1
x+3	1	0	1	0	1	0	1	0

x	0	1	0	1	0	0	0	0
x+1	1	0	1	1	1	0	1	1
x+2	0	0	0	1	0	0	0	1
x+3	1	0	1	0	1	0	1	0

x	0	0	0	0	0	1	0	1
x+1	0	0	0	0	0	0	0	0
x+2	0	0	0	0	0	0	0	0
x+3	0	0	0	0	0	0	0	0

# Section de données initialisées

La **section .data** est la partie (facultative) du programme qui regroupe des définitions de données pointées par des adresses.

Elle commence par `section .data`.

On **définit une donnée** par

ID: DT VAL

où ID est un identificateur (appelé **étiquette**), VAL une valeur et DT un descripteur de taille parmi les suivants :

Descripteur de taille	Taille (en octets)
db	1
dw	2
dd	4
dq	8

Ceci place en mémoire à l'adresse ID la valeur VAL, dont la taille est spécifiée par DT.

La valeur de l'adresse ID est attribuée par le système.

# Section de données initialisées — exemples

Quelques exemples de définitions de données initialisées :

- `entier: dw 55`

Créé à l'adresse `entier` un entier sur 2 octets, initialisé à  $(55)_{\text{dix}}$ .

- `x: dd 0xFFE05`

Créé à l'adresse `x` un entier sur 4 octets, initialisé à  $(000FFE05)_{\text{hex}}$ .

- `y: db 0b11001100`

Créé à l'adresse `y` un entier sur 1 octet initialisé à  $(11001100)_{\text{deux}}$ .

- `c: db 'a'`

Créé à l'adresse `c` un entier sur 1 octet dont la valeur est le code ASCII du caractère `'a'`.

- `chaine: db 'Test', 0`

Créé à partir de l'adresse `chaine` une suite de 5 octets contenant successivement les codes ASCII des lettres `'T'`, `'e'`, `'s'`, `'t'` et du marqueur de fin de chaîne.

De plus, à l'adresse `chaine + 2` figure le code ASCII du caractère `'s'`.

## Section de données initialisées — définitions multiples

On peut **définir plusieurs données** de manière concise par

```
ID: times NB DT VAL
```

où ID est un identificateur, VAL une valeur, DT un descripteur de taille et NB une valeur positive.

Ceci place en mémoire, à partir de l'adresse ID, NB occurrences de la valeur VAL, dont la taille est spécifiée par DT.

P.ex.,

- `suite: times 85 dd 5`

Créé à partir de l'adresse `suite` une suite de  $85 \times 4$  octets, où chaque double mot est initialisé à la valeur  $(5)_{\text{dix}}$ .

L'adresse du 1<sup>er</sup> double mot est `suite`.

L'adresse du 7<sup>e</sup> double mot est `suite + (6 * 4)`.

- `chaine: times 9 db 'a'`

Créé à partir de l'adresse `chaine` une suite de 9 octets tous initialisés par le code ASCII du caractère 'a'.

L'adresse du 3<sup>e</sup> octet est `chaine + 2`.

# Section de données non initialisées

La **section .bss** est la partie (facultative) du programme qui regroupe des déclarations de données non initialisées pointées par des adresses.

Elle commence par `section .bss`.

On **déclare une donnée non initialisée** par

ID: DT NB

où ID est un identificateur, NB une valeur positive et DT un descripteur de taille parmi les suivants :

Descripteur de taille	Taille (en octets)
resb	1
resw	2
resd	4
resq	8

Ceci réserve une zone de mémoire commençant à l'adresse ID et pouvant accueillir NB données dont la taille est spécifiée par DT.

## Section de données non initialisées — exemples

P.ex., l'instruction

```
x: resw 120
```

réserve, à partir de l'adresse  $x$ , une suite de  $120 \times 2$  octets non initialisés.

L'adresse de la  $i^{\text{e}}$  donnée à partir de  $x$  est  $x + ((i - 1) * 2)$ .

Pour écrire la valeur  $(0xEF01)_{\text{hex}}$  en 4<sup>e</sup> position, on utilise l'instruction  
`mov word [x + (3 * 2)], 0xEF01`

Pour lire la valeur située en 7<sup>e</sup> position à partir de  $x$ , on utilise les instructions

```
mov eax, 0
```

```
mov ax, [x + (6 * 2)]
```

**Attention** : il ne faut jamais rien supposer sur la valeur initiale d'une donnée non initialisée.

## Section d'instructions

La **section .text** est la partie du programme qui regroupe les instructions. Elle commence par `section .text`.

Pour définir le point d'entrée du programme, il faut définir une **étiquette de code** et faire en sorte de la rendre visible depuis l'extérieur.

Pour cela, on écrit

```
section .text
global main
    main:
    INSTR
```

où `INSTR` dénote la suite des instructions du programme. Ici, `main` est une étiquette et sa valeur est l'adresse de la 1<sup>re</sup> instruction constituant `INSTR`.

La ligne `global main` sert à rendre l'étiquette `main` visible pour l'édition des liens.

# Interruptions — généralités

L'exécution d'un programme se fait instruction par instruction. Dès qu'une instruction est traitée, le processeur s'occupe de la suivante.

Cependant, certaines instructions ont besoin d'**interrompre l'exécution** pour être menées à bien. Parmi celles-ci, nous avons p.ex.,

- l'écriture de texte sur la sortie standard ;
- la lecture d'une donnée sur la sortie standard ;
- l'écriture d'une donnée sur le disque ;
- la gestion de la souris ;
- la communication via le réseau ;
- la sollicitation de l'unité graphique ou sonore.

Dans ce but, il existe des instruction particulières appelées **interruptions**.

# Interruptions — instruction

L'instruction

```
int 0x80
```

permet d'**appeler une interruption** dont le traitement est délégué au système (Linux).

La tâche à réaliser est spécifiée par un code lu depuis le registre `eax`. Voici les principaux :

Code	Rôle
1	Arrêt et fin de l'exécution
3	Lecture sur l'entrée standard
4	Écriture sur la sortie standard

Les autres registres de travail `ebx`, `ecx` et `edx` jouent le rôle d'arguments à la tâche en question.

**Attention** : le traitement d'une interruption peut modifier le contenu des registres. Il faut sauvegarder leur valeur dans la mémoire si besoin est.

# Interruptions — exemples

Pour **stopper** l'exécution d'un programme, on utilise

```
mov ebx, 0  
mov eax, 1  
int 0x80
```

Le registre ebx contient la valeur de retour de l'exécution.

Pour **afficher un caractère** sur la sortie standard, on utilise

```
mov ebx, 1  
mov ecx, x  
mov edx, 1  
mov eax, 4  
int 0x80
```

La valeur de ebx spécifie que l'on écrit sur la sortie standard. Le registre ecx contient l'adresse x du caractère à afficher et la valeur de edx signifie qu'il y a un unique caractère à afficher.

# Interruptions — exemples

Pour **lire un caractère** sur l'entrée standard, on utilise

```
mov ebx, 1
mov ecx, x
mov edx, 1
mov eax, 3
int 0x80
```

La valeur de `ebx` spécifie que l'on lit sur la sortie standard. Le registre `ecx` contient l'adresse `x` à laquelle le code ASCII du caractère lu sera enregistré et la valeur de `edx` signifie qu'il y a un unique caractère à lire.

Il est bien entendu possible, pour les interruptions commandant l'écriture et l'affichage de caractères, de placer d'autres valeurs dans `edx` pour pouvoir écrire/lire plus de caractères.

# Directives

Une **directive** est un élément d'un programme qui n'est pas traduit en langage machine mais qui sert à informer l'assembleur, entre autre, de

- la définition d'une constante ;
- l'inclusion d'un fichier.

Pour **définir une constante**, on se sert de

```
%define NOM VAL
```

Ceci fait en sorte que, dans le programme, le symbole NOM est remplacé par le symbole VAL.

Pour **inclure un fichier** (assembleur `.asm` ou en-tête `.inc`), on se sert de

```
%include CHEM
```

Ceci fait en sorte que le fichier de chemin relatif CHEM soit inclus dans le programme. Il est ainsi possible d'utiliser son code dans le programme appelant.

# Assemblage

Pour **assembler** un programme PRGM.asm, on utilise la commande

```
nasm -f elf32 PRGM.asm
```

Ceci créé un **fichier objet** nommé PRGM.o.

On obtient un exécutable par l'édition des liens, en utilisant la commande

```
ld -o PRGM -e main PRGM.o
```

Ceci créé un **exécutable** nommé PRGM.

L'option `-e main` spécifie que le point d'entrée du programme est l'instruction à l'adresse `main`.

**Astuce** : sur un système 64 bits, on ajoute pour l'édition des liens l'option `-melf_i386`, ce qui donne donc la commande

```
ld -o PRGM -melf_i386 -e main PRGM.o.
```

# Exemple complet de programme

```
; Def. de donnees
section .data
chaine_1:
    db 'Caractere? ',0
chaine_2:
    db 'Suivant : ',0

; Decl. de donnees
section .bss
car: resb 1

; Instructions
section .text
global main
main:

; Aff. chaine_1
    mov ebx, 1
    mov ecx, chaine_1
    mov edx, 13
    mov eax, 4
    int 0x80

; Lect. car.
    mov ebx, 1
    mov ecx, car
    mov edx, 1
    mov eax, 3
    int 0x80

; Incr. car.
    mov eax, [car]
    add eax, 1
    mov [car], al

; Aff. chaine_2
    mov ebx, 1
    mov ecx, chaine_2
    mov edx, 11
    mov eax, 4
    int 0x80

; Aff. car.
    mov ebx, 1
    mov ecx, car
    mov edx, 1
    mov eax, 4
    int 0x80

; Sortie
    mov ebx, 0
    mov eax, 1
    int 0x80
```

Ce programme lit un caractère sur l'entrée standard et affiche le caractère suivant dans la table ASCII.

# Étiquettes d'instruction

Les **instructions** d'un programme sont des données comme des autres. Elles ont donc une **adresse**.

Tout comme pour les données, il est possible de disposer des **étiquettes** dans un programme, dont les valeurs sont des adresses d'instructions.

Ceci se fait par

```
ETIQ: INSTR
```

où ETIQ est le nom de l'étiquette et INSTR une instruction.

```
mov eax, 0
instr_2: mov ebx, 1
add eax, ebx
```

P.ex., ici l'étiquette `instr_2` pointe vers l'instruction `mov ebx, 1`.

**Remarque** : nous avons déjà rencontré l'étiquette `main`. Il s'agit d'une étiquette d'instruction. Sa valeur est l'adresse de la 1<sup>re</sup> instruction du programme.

# Pointeur d'instruction et exécution

À chaque instant de l'exécution d'un programme, le registre `eip`, appelé **pointeur d'instruction**, contient l'**adresse de la prochaine instruction** à exécuter.

L'exécution d'un programme s'organise en l'algorithme suivant :

- 1 Répéter, tant que l'exécution n'est pas interrompue :
  - 1 charger l'instruction / d'adresse `eip` ;
  - 2 mettre à jour `eip` ;
  - 3 traiter l'instruction `I`.

Par défaut, après le traitement d'une instruction (en tout cas de celles que nous avons vues pour le moment), `eip` est mis à jour de sorte à contenir l'adresse de l'instruction suivante en mémoire.

Il est impossible d'intervenir directement sur la valeur de `eip`.

# Sauts inconditionnels

Ainsi, par défaut, l'exécution d'un programme se fait instruction par instruction, dans l'ordre dans lequel elles sont écrites.

Néanmoins, il est possible de rompre cette ligne d'exécution en réalisant des **sauts**. Ils consistent, étant donné un point de départ, à poursuivre l'exécution du programme vers un point cible.

Pour cela, on se sert de l'instruction

```
jmp ETIQ
```

où ETIQ est une étiquette d'instruction. Cette instruction **saute** à l'endroit du code pointé par ETIQ.

Elle agit en **modifiant** de manière adéquate **le pointeur d'instruction** `eip`.

# Sauts inconditionnels

```
mov ebx, 0xFF
jmp endroit
mov ebx, 0
endroit:
    mov eax, 1
```

L'instruction `mov ebx, 0` n'est pas exécutée puisque le `jmp endroit` qui la précède fait en sorte que l'exécution passe à l'étiquette `endroit`.

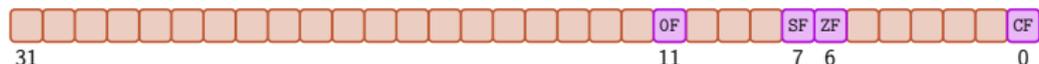
```
mov eax, 0
debut:
    add eax, 1
    jmp debut
```

L'exécution de ces instructions provoque une boucle infinie. Le saut inconditionnel vers l'étiquette `debut` précédente provoque la divergence.

# Le registre flags

À tout moment de l'exécution d'un programme, le **registre de drapeaux flags** contient des informations sur la dernière instruction exécutée.

Comme son nom l'indique, il fonctionne comme un drapeau : chacun de ses bits code une information du type oui (bit à 1) / non (bit à 0).



Voici certaines des informations qu'il contient. Le bit

- CF, « *Carry Flag* » vaut 1 si l'instruction produit une retenue de sortie et 0 sinon ;
- ZF, « *Zero Flag* » vaut 1 si l'instruction produit un résultat nul et 0 sinon ;
- SF, « *Sign Flag* » vaut 1 si l'instruction produit un résultat négatif et 0 sinon ;
- OF, « *Overflow Flag* » vaut 1 si l'instruction produit un dépassement de capacité et 0 sinon.

# Instruction de comparaison

L'**instruction de comparaison `cmp`** s'utilise par

```
cmp VAL_1, VAL_2
```

et permet de **comparer** les valeurs `VAL_1` et `VAL_2` en mettant à jour le registre `flags`.

Plus précisément, cette instruction calcule la différence `VAL_1 - VAL_2` et modifie `flags` de la manière suivante :

- si `VAL_1 - VAL_2 = 0`, alors `ZF` est positionné à 1 ;
- si `VAL_1 - VAL_2 > 0`, alors `ZF` est positionné à 0 et `CF` est positionné à 0 ;
- si `VAL_1 - VAL_2 < 0`, alors `ZF` est positionné à 0 et `CF` est positionné à 1.

On peut préciser la taille des valeurs à comparer à l'aide d'un descripteur de taille (`dbyte`, `word`, `dword`) si besoin est.

```
mov ebx, 5  
cmp dword 21, ebx
```

P.ex., cette comparaison fait que `ZF` et `CF` sont positionnés à 0.

# Sauts conditionnels

Un **saut conditionnel** est un saut qui n'est réalisé que si une condition impliquant le registre `flags` est vérifiée ; si celle-ci n'est pas vérifiée, l'exécution se poursuit en l'instruction qui suit le saut conditionnel.

Pour ce faire, on adopte le schéma

```
cmp VAL_1, VAL_2
SAUT ETIQ
```

`VAL_1` et `VAL_2` sont des valeurs, `ETIQ` est une étiquette d'instruction et `SAUT` est une instruction de saut conditionnel.

Il y a plusieurs instructions de saut conditionnel. Elles diffèrent sur la condition qui provoque le saut :

Instruction	Saute si
<code>je</code>	$VAL\_1 = VAL\_2$
<code>jne</code>	$VAL\_1 \neq VAL\_2$
<code>j1</code>	$VAL\_1 < VAL\_2$
<code>jle</code>	$VAL\_1 \leq VAL\_2$
<code>jg</code>	$VAL\_1 > VAL\_2$
<code>jge</code>	$VAL\_1 \geq VAL\_2$

# Sauts conditionnels

```
cmp eax, ebx
jl inferieur
jmp fin
inferieur:
    mov eax, ebx
fin:
```

Ceci saute à inferieur si la valeur de eax est strict. inf. à celle de ebx.

Ceci fait en sorte que eax vaille  $\max(\text{eax}, \text{ebx})$ .

```
mov ecx, 15
debut:
    cmp ecx, 0
    je fin
    sub ecx, 1
    jmp debut
fin:
```

Ceci est une boucle. Tant que la valeur de ecx est diff. de 0, ecx est décrémenté et un tour de boucle est réalisé.

Quinze tours de boucle sont effectués avant de rejoindre l'étiquette fin.

# Simulation du if

L'équivalent du pseudo-code

```
Si a = b
    BLOC
FinSi

est

cmp eax, ebx
je then
jmp end_if
then:
    BLOC
end_if:
```

L'équivalent du pseudo-code

```
Si a = b
    BLOC_1
Sinon
    BLOC_2
FinSi

est

cmp eax, ebx
jne else
    BLOC_1
    jmp end_if
else:
    BLOC_2
end_if:
```

# Simulation du while et du do while

L'équivalent du pseudo-code

```
TantQue a = b  
    BLOC  
FinTantQue
```

est

```
while:
```

```
    cmp eax, ebx  
    jne end_while  
    BLOC  
    jmp while
```

```
end_while:
```

L'équivalent du pseudo-code

```
Faire
```

```
    BLOC
```

```
TantQue a = b
```

est

```
do:
```

```
    BLOC
```

```
    cmp eax, ebx  
    je do
```

# Simulation du for

L'équivalent du pseudo-code

```
Pour a = 1 à b
    BLOC
FinPour
```

est

```
mov eax, 1
for:
    cmp eax, ebx
    jg end_for
    BLOC
    add eax, 1
    jmp for
end_for:
```

On peut simuler ce pseudo-code de manière plus compacte grâce à l'instruction

```
loop ETIQ
```

Celle-ci saute vers l'étiquette d'instruction ETIQ si ecx est non nul et décrémente ce dernier.

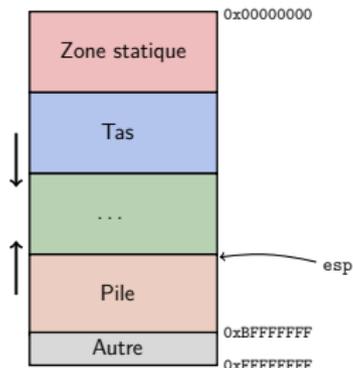
On obtient la suite d'instructions suivante :

```
mov ecx, ebx
boucle:
    BLOC
    loop boucle
```

# Pile

La **pile** est une zone de la mémoire dans laquelle on peut empiler et dépiler des données.

La zone qu'elle occupe en mémoire est de taille variable mais elle se situe toujours avant l'adresse 0xBFFFFFFF.



La pile est de type LIFO : les données sont dépilées de la plus récente à la plus ancienne.

On place et on lit dans la pile uniquement des **doublets mots**.

Le registre esp contient l'adresse de la **tête de pile**.

On utilise le registre ebp pour sauvegarder une position dans la pile (lorsque esp est susceptible de changer).

On dispose de deux opérations pour manipuler la pile :

- 1 **empiler** une valeur ;
- 2 **dépiler** une valeur.

Pour **empiler** une valeur VAL à la pile, on utilise

`push VAL`

Ceci décrémente esp de 4 et écrit à l'adresse esp la valeur VAL.

Pour **dépiler** vers le registre REG la valeur située en tête de pile, on utilise

`pop REG`

Ceci recopie les 4 octets à partir de l'adresse esp vers REG et incrémente esp de 4.

**Attention** : l'ajout d'éléments dans la pile fait décroître la valeur de esp et la suppression d'éléments fait croître sa valeur, ce qui est peut-être contre-intuitif.

Observons l'effet des instructions avec la pile dans l'état suivant :

Adresses	Pile	
1000	0x01010101	esp = 1008
1004	0x20202020	
1008	0xAA55AA55	
1012	0xFFFFFFFF	
1016	0x00000000	

push 0x3  
pop eax  
pop ebx  
push eax

0x01010101	esp = 1004	0x01010101	esp = 1008
0x00000003		0x00000003	eax = 0x3
0xAA55AA55		0xAA55AA55	
0xFFFFFFFF		0xFFFFFFFF	
0x00000000		0x00000000	

0x01010101	esp = 1012	0x01010101	esp = 1008
0x00000003	ebx = 0xA...5	0x00000003	
0xAA55AA55		0x00000003	
0xFFFFFFFF		0xFFFFFFFF	
0x00000000		0x00000000	

# Instruction call

On souhaite maintenant établir un mécanisme pour pouvoir **écrire des fonctions et les appeler**.

L'un des ingrédients pour cela est l'instruction

```
call ETIQ
```

Elle permet de sauter à l'étiquette d'instruction ETIQ.

La différence avec l'instruction `jmp ETIQ` réside dans le fait que `call ETIQ` **empile**, avant le saut, l'**adresse de l'instruction qui la suit** dans le programme.

Ainsi, les deux suites d'instructions suivantes sont équivalentes :

```
call cible  
suite:  
...
```

```
push suite  
jmp cible  
suite:  
...
```

# Instruction ret

L'intérêt d'enregistrer l'adresse de l'instruction qui suit un `call` ETIQ repose sur le fait que l'exécution peut **revenir** à cette instruction.

Ceci est offert par l'instruction (sans opérande)

`ret`

Elle fonctionne en dépilant la donnée en tête de pile et en sautant à l'adresse spécifiée par cette valeur.

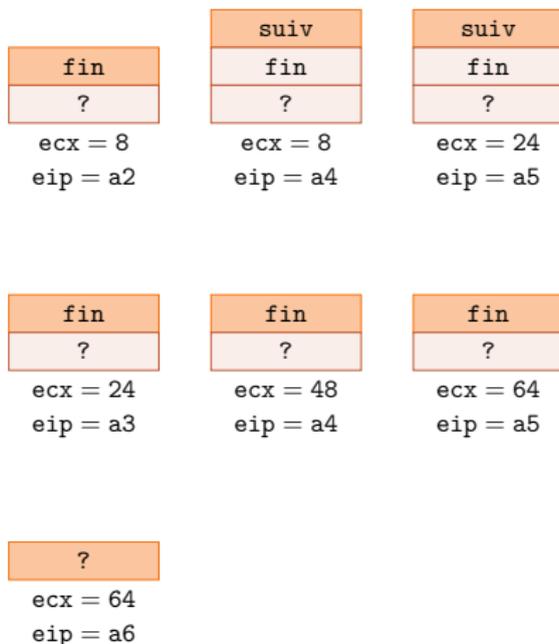
Ainsi, les deux suites d'instructions suivantes sont équivalentes (excepté pour la valeur de `eax` qui est modifiée dans la seconde) :

<code>call cible</code>	<code>push suite</code>
<code>suite:</code>	<code>jmp cible</code>
<code>...</code>	<code>suite:</code>
<code>cible:</code>	<code>...</code>
<code>...</code>	<code>cible:</code>
<code>ret</code>	<code>...</code>
	<code>pop eax</code>
	<code>jmp eax</code>

# Exemple d'utilisation call / ret

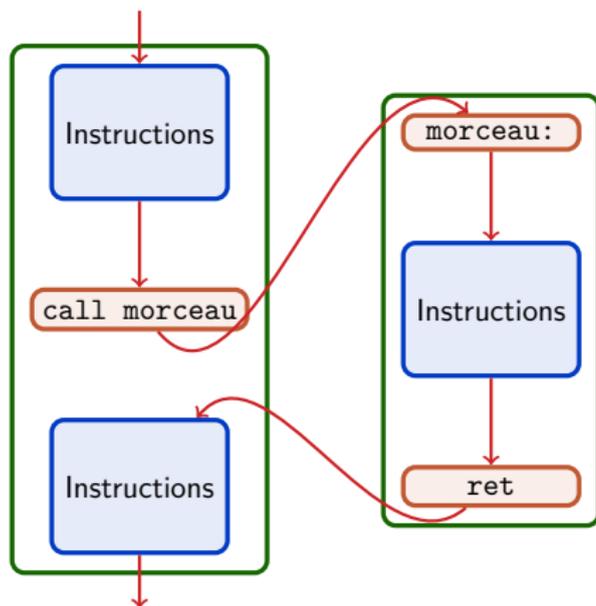
Considérons la suite d'instructions suivante et observons l'état de la pile et de l'exécution.

```
→ a1 | mov ecx, 8
→ a2 | call loin
→ a3 | suiv: add ecx, 24
→ a4 | loin: add ecx, 16
→ a5 | ret
      | ...
→ a6 | fin:
```



## Exemple d'utilisation call / ret

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple call / ret :



**Attention** : le retour à l'endroit du code attendu par l'instruction ret n'est correct que si l'état de la pile à l'étiquette morceau est le même que celui juste avant le ret.

# Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions, dites **conventions d'appel du C**. Ceci consiste en le respect des points suivants :

- 1 les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
- 2 le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
- 3 les **valeurs des registres de travail** `ebx`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel d'une fonction ;
- 4 la **pile** doit être dans le **même état** avant l'appel et après l'appel d'une fonction. Ceci signifie que l'état des pointeurs `esp` et `ebp` sont conservés et que le contenu de la pile qui suit l'adresse `esp` est également conservé.

**Note** : le point 3 n'est pas obligatoire à suivre.

# Fonctions — écriture et appel

L'**écriture** d'une fonction suit le squelette

**NOM\_FCT:**

```
push ebp
mov ebp, esp
INSTR
pop ebp
ret
```

Ici, **NOM\_FCT** est une étiquette d'instruction qui fait d'office de nom pour la fonction. De plus, **INSTR** est un bloc d'instructions.

Il est primordial que **INSTR** **conserve l'état de la pile.**

L'**appel** d'une fonction se fait par

```
push ARG_N
...
push ARG_1

call NOM_FCT

add esp, 4 * N
```

Ici, **NOM\_FCT** est le nom de la fonction à appeler. Elle admet **N** arguments qui sont **empilés du dernier au premier.**

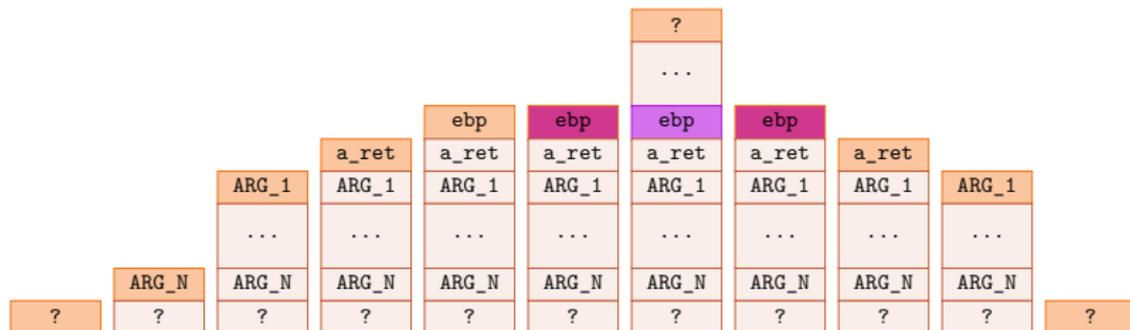
Après l'appel, on incrémente **esp** pour dépiler d'un coup les **N** arguments de la fonction.

# Fonctions — appel et état de la pile

Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

→		→	<b>NOM_FCT:</b>
→	push ARG_N	→	push ebp
	...	→	mov ebp, esp
→	push ARG_1	→	INSTR
→	call NOM_FCT	→	pop ebp
→	ADD esp, 4 * N	→	ret

Légende : en orange, la zone de la pile pointée par esp, en violet, la zone de la pile pointée par ebp, a\_ret est l'adresse de l'instr. qui suit le call.



## Fonctions — préservation de l'état de la pile

L'état de la **pile** doit être **préservé** par le bloc d'instructions INSTR.

Cela signifie que l'état de la pile juste avant d'exécuter INSTR et son état juste après son exécution sont les mêmes. En d'autres termes,

- esp doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que esp ne doivent pas être modifiées.

C'est bien le cas si

- INSTR contient autant de push que de pop ;
- à tout instant, il y a au moins autant de push que de pop qui ont été exécutés.

Il faut bien respecter ces deux conditions dans la pratique.

## Fonctions — rôle de ebp

La **valeur** de **esp** est susceptible de **changer** dans INSTR. C'est pour cela que l'on **sauvegarde** sa valeur, à l'entrée de la fonction, **dans ebp**.

Le registre ebp sert ainsi, dans INSTR, à **accéder** aux **arguments**. En effet, l'adresse du 1<sup>er</sup> argument est  $ebp + 8$ , celle du 2<sup>e</sup> est  $ebp + 12$  et plus généralement, celle du  $i^e$  argument est

$$ebp + 4 * (i + 1)$$

On **sauvegarde** et on **restaure** tout de même, par un `push ebp` et `pop ebp` l'état de **ebp** à l'entrée et à la sortie de la fonction.

Ce même mécanisme peut être utilisé pour sauvegarder/restaurer l'état des registres de travail `eax` (sauf si la fonction renvoie une valeur), `ebx`, `ecx` et `edx`.

# Fonctions — exemple 1

Voici un exemple de fonction :

```
; Fonction qui renvoie la somme de deux  
; entiers  
; Arguments :  
; (1) une valeur entière signée  
; sur 4 octets  
; (2) une valeur entière signée  
; sur 4 octets  
; Renvoi : la somme des deux  
; arguments  
somme:  
    push ebp  
    mov ebp, esp  
  
    mov eax, [ebp + 8]  
    add eax, [ebp + 12]  
  
    pop ebp  
    ret
```

Pour calculer dans `eax` la somme de  $(43)_{\text{dix}}$  et  $(1996)_{\text{dix}}$ , on utilise

```
push 1996  
push 43  
call somme  
add esp, 8
```

**Rappel 1** : on empile les arguments dans l'ordre inverse de ce que la fonction attend.

**Rappel 2** : on ajoute 8 à `esp` après l'appel pour dépiler d'un seul coup les deux arguments ( $8 = 2 \times 4$ ).

## Fonctions — exemple 2

```
; Fonction qui affiche un  
; caractere  
; Arguments :  
    (1) valeur du  
    caractere a afficher.  
; Renvoi : rien.
```

```
print_char:
```

```
; Debut
```

```
    push ebp  
    mov ebp, esp
```

```
; Sauv. des reg.
```

```
    push eax  
    push ebx  
    push ecx  
    push edx
```

```
; Affichage
```

```
    mov ebx, 1  
    mov ecx, ebp  
    add ecx, 8  
    mov edx, 1  
    mov eax, 4  
    int 0x80
```

```
; Rest. des reg.
```

```
    pop edx  
    pop ecx  
    pop ebx  
    pop eax
```

```
; Fin
```

```
    pop ebp  
    ret
```

## Fonctions — exemple 2 bis

```
; Fonction qui affiche un  
; caractere  
; Arguments :  
; (1) adresse du  
; caractere a afficher.  
; Renvoi : rien.
```

```
print_char_2:
```

```
; Debut
```

```
    push ebp  
    mov ebp, esp
```

```
; Sauv. des reg.
```

```
    push eax  
    push ebx  
    push ecx  
    push edx
```

```
; Affichage
```

```
    mov ebx, 1  
    mov ecx, [ebp + 8]  
    mov edx, 1  
    mov eax, 4  
    int 0x80
```

```
; Rest. des reg.
```

```
    pop edx  
    pop ecx  
    pop ebx  
    pop eax
```

```
; Fin
```

```
    pop ebp  
    ret
```

## Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1<sup>re</sup> prend son argument **par valeur**, tandis que la 2<sup>e</sup> prend son argument **par adresse**.

Ainsi, pour afficher p.ex. le caractère 'W', on appelle `print_char` par

```
push 'W'  
call print_char  
add esp, 4
```

La fonction `print_char_2` s'appelle par

```
push c1  
call print_char_2  
add esp, 4
```

où `c1` est l'adresse d'un caractère en mémoire. Celle-ci a été définie par exemple par `c1 : db 'W'` dans la section de données.

## Fonctions — exemple 3

```
; Fonction qui affiche une  
; chaine de caracteres  
; Arguments :  
; (1) adresse de la  
; chaine de caracteres.  
; Renvoi : nbr carac. aff.
```

```
print_string:
```

```
; Debut  
    push ebp  
    mov ebp, esp  
  
; Sauv. des reg.  
    push ebx  
  
; Adresse debut chaine  
    mov ebx, [ebp + 8]  
  
; Init. compteur  
    mov eax, 0
```

```
; Boucle d'affichage
```

```
    boucle:  
        cmp byte [ebx], 0  
        je fin_boucle  
        push dword [ebx]  
        call print_char  
        add esp, 4  
        add ebx, 1  
        add eax, 1  
        jmp boucle  
    fin_boucle:
```

```
; Rest. des reg.
```

```
    pop ebx
```

```
; Fin
```

```
    pop ebp  
    ret
```

## Fonctions — exemple 4

On souhaite écrire une fonction pour calculer **récurivement** le  $n^e$  nombre triangulaire  $\text{triangle}(n)$  défini par

$$\text{triangle}(n) := \begin{cases} 0 & \text{si } n = 0, \\ n + \text{triangle}(n - 1) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; nombres triangulaires.
; Arguments :
; (1) entier positif
; Renvoi : le nombre
; triangulaire de l'arg.
triangle:
    ; Debut
    push ebp
    mov ebp, esp

    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Prepa. appel rec.
    mov ecx, ebx
    sub ecx, 1

    ; Appel rec.
    push ecx
    call triangle
    add esp, 4

    ; Calcul res.
    add eax, ebx

    jmp fin

cas_terminal:
    mov eax, 0

fin:
; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

; Fin
    pop ebp
    ret
```

## Fonctions — exemple 5

On souhaite écrire une fonction pour calculer **récurivement** la factorielle  $\text{fact}(n)$  de tout entier positif  $n$ . On rappelle que  $\text{fact}(n)$  est défini par

$$\text{fact}(n) := \begin{cases} 1 & \text{si } n = 0, \\ n \times \text{fact}(n - 1) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; la factorielle
; Arguments :
; (1) entier positif
; Renvoi : la factorielle
; de l'argument
fact:
    ; Debut
    push ebp
    mov ebp, esp

    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Prepa. appel rec.
    mov ecx, ebx
    sub ecx, 1

    ; Appel rec.
    push ecx
    call fact
    add esp, 4

    ; Calcul res.
    mul ebx

    jmp fin

cas_terminal:
    mov eax, 1

fin:
    ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

    ; Fin
    pop ebp
    ret
```

## Fonctions — exemple 6

On souhaite écrire une fonction pour calculer **récurivement** le  $n^{\text{e}}$  nombre de Fibonacci  $\text{fibonacci}(n)$ . On rappelle que  $\text{fibonacci}(n)$  est défini par

$$\text{fibonacci}(n) := \begin{cases} n & \text{si } n \leq 1, \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & \text{sinon.} \end{cases}$$

```
; Fonction de calcul de
; nombres de Fibonacci
; Arguments :
; (1) entier positif
; Renvoi : le nombre de
; Fibonacci de l'argument
fibonacci:
    ; Debut
    push ebp
    mov ebp, esp
    ; Sauv. des reg.
    push ebx
    push ecx
    push edx

    ; Sauvegarde de l'arg.
    mov ebx, [ebp + 8]

    cmp ebx, 1
    jle cas_terminaux

    ; Appel rec. 1
    push ecx
    call fibonacci
    add esp, 4

    ; Appel rec. 2
    push ecx
    call fibonacci
    add esp, 4

    ; Calcul res.
    pop ebx
    add eax, ebx

    jmp fin

cas_terminaux:
    mov eax, ebx

fin:
    ; Rest. des reg.
    pop edx
    pop ecx
    pop ebx

    pop ebp
    ret
```

# Étiquettes locales

Dans un programme complet, il peut être difficile de gérer de nombreuses étiquettes de code.

Il existe pour cette raison la notion d'**étiquettes locales**, dont la syntaxe de définition et de référence est

`.ETIQ:`

P.ex.,

```
etiq_globale:
    .action:
        INSTR_1
        jmp .action
```

```
autre_etiq_globale:
    .action:
        INSTR_2
        jmp .action
```

déclare plusieurs étiquettes de code, dont deux du même nom et locales, `.action`.

Le 1<sup>er</sup> `jmp` saute à l'instruction correspondant au 1<sup>er</sup> `.action`, tandis que le 2<sup>e</sup> `jmp` saute à l'instruction correspondant au 2<sup>e</sup> `action`.

Le noms absolus (`etiq_globale.action` et `autre_etiq_globale.action`) de ces étiquettes permettent de faire référence à la bonne si besoin est.

# Programmation modulaire

Il est possible de réaliser des projets en assembleur sur plusieurs fichiers, découpés en modules.

Un **module** est constitué

- d'un **fichier source** d'extension `.asm` ;
- d'un **fichier d'en-tête** d'extension `.inc`.

Seul le module qui contient la fonction principale `main` ne dispose pas de fichier d'en-tête.

Pour compiler un projet sur plusieurs fichiers, on se sert des commandes

```
nasm -f elf32 Main.asm
nasm -f elf32 M1.asm
...
nasm -f elf32 Mk.asm
ld -o Exec -melf_i386 -e main Main.o M1.o ... Mk.o
```

# Programmation modulaire

Un module (non principal) contient une collection de fonctions destinées à **être utilisées depuis l'extérieur**.

On autorise une étiquette d'instruction ETIQ à être visible depuis l'extérieur en ajoutant la ligne

```
global ETIQ
```

juste avant la définition de l'étiquette.

De plus, on renseigne dans le fichier d'en-tête l'existence de la fonction par

```
extern ETIQ
```

Il est d'usage de documenter à cet endroit la fonction.

# Programmation modulaire

Pour bénéficier des fonctions définies dans un module M dans un fichier F.asm, on invoque, au tout début de F.asm, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de M peuvent être appelées dans F.asm.

Voici p.ex., un module ES et son utilisation dans Main.asm :

<i>; ES.inc</i>	<i>; ES.asm</i>	<i>; Main.asm</i>
<i>; Documentation...</i>	...	%include "ES.inc"
extern print_char	global print_char	...
	print_char:	call print_string
<i>; Documentation...</i>	...	...
extern print_string	global print_string	
	print_string:	
	...	

# Étapes d'exécution d'une instruction

Pour des raisons pédagogiques, on se place sur une architecture simplifiée où toute instruction est exécutée selon le schéma suivant :

- 1 (**IF, Instruction Fetch**) **chargement** de l'instruction et mise à jour du pointeur d'instruction `eip` de sorte qu'il contienne l'adresse de la prochaine instruction à exécuter ;
- 2 (**ID, Instruction Decode**) **identification** de l'instruction. Les arguments éventuels de l'instruction sont placés dans l'unité arithmétique et logique.
- 3 (**EX, Execute**) **exécution** de l'instruction par l'unité arithmétique et logique.
- 4 (**WB, Write Back**) **écriture** éventuelle dans les registres ou dans la mémoire.

# Étapes d'exécution d'une instruction — exemple 1

Par exemple, l'instruction

```
add eax, [adr]
```

où `adr` est une adresse, est traitée de la manière suivante :

- 1 **(IF)** l'instruction `add` est **chargée** et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction ;
- 2 **(ID)** le système **repère** qu'il s'agit de l'instruction `add` et il lit les 4 octets situés à partir de l'adresse `adr` dans la mémoire, ainsi que la valeur du registre `eax` ;
- 3 **(EX)** l'unité arithmétique et logique **effectue** l'addition entre les deux valeurs chargées ;
- 4 **(WB)** le résultat ainsi calculé est **écrit** dans `eax`.

# Étapes d'exécution d'une instruction — exemple 2

Par exemple, l'instruction

```
jmp adr
```

où `adr` est une adresse d'instruction, est traitée de la manière suivante :

- 1 **(IF)** l'instruction `jmp` est **chargée** et `eip` est incrémenté afin qu'il pointe vers la prochaine instruction (il ne pointe donc pas encore forcément sur l'instruction d'adresse `adr` comme souhaité) ;
- 2 **(ID)** le système **repère** qu'il s'agit de l'instruction `jmp` et il lit la valeur de l'adresse `adr` ;
- 3 **(EX)** l'adresse d'instruction cible est **calculée** à partir de la valeur de `adr` ;
- 4 **(WB)** l'adresse d'instruction cible est **écrite** dans `eip`.

# Étapes d'exécution d'une instruction — horloge

L'**horloge du processeur** permet de rythmer ces étapes.

L'exécution de chacune de ces étapes demande au moins un **cycle d'horloge**.

Certaines opérations demandent plusieurs cycles d'horloge pour être traitées. La division (instruction `div`), par exemple, demande plusieurs dizaines de cycles d'horloge pour réaliser l'étape **EX**.

La **vitesse d'horloge** d'un processeur est exprimée en hertz (Hz).

Un processeur dont la vitesse d'horloge est de 1 Hz évolue à 1 cycle d'horloge par seconde.

**Problématique** : comment optimiser l'exécution des instructions ?

# Le travail à la chaîne

Considérons une usine qui produit des pots de confiture. L'instruction que suit l'usine est d'assembler, en boucle, des pots de confiture. Cette tâche se divise en quatre sous-tâches :

- 1 placer un pot vide sur le tapis roulant ;
- 2 remplir le pot de confiture ;
- 3 visser le couvercle ;
- 4 coller l'étiquette.

La création séquentielle de trois pots de confiture s'organise de la manière suivante :

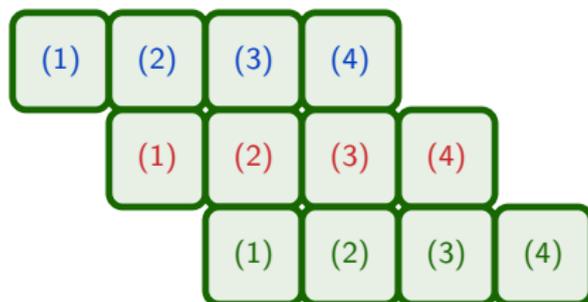


et nécessite  $3 \times 4 = 12$  étapes.

# Le travail à la chaîne efficace

**Observation** : au lieu de réaliser ces assemblages séquentiellement, lorsqu'un pot est sujet à une sous-tâche ( $i$ ),  $1 \leq i \leq 4$ , on peut appliquer à un autre pot une autre sous-tâche ( $j$ ),  $1 \leq j \neq i \leq 4$ .

Cette organisation se schématise en



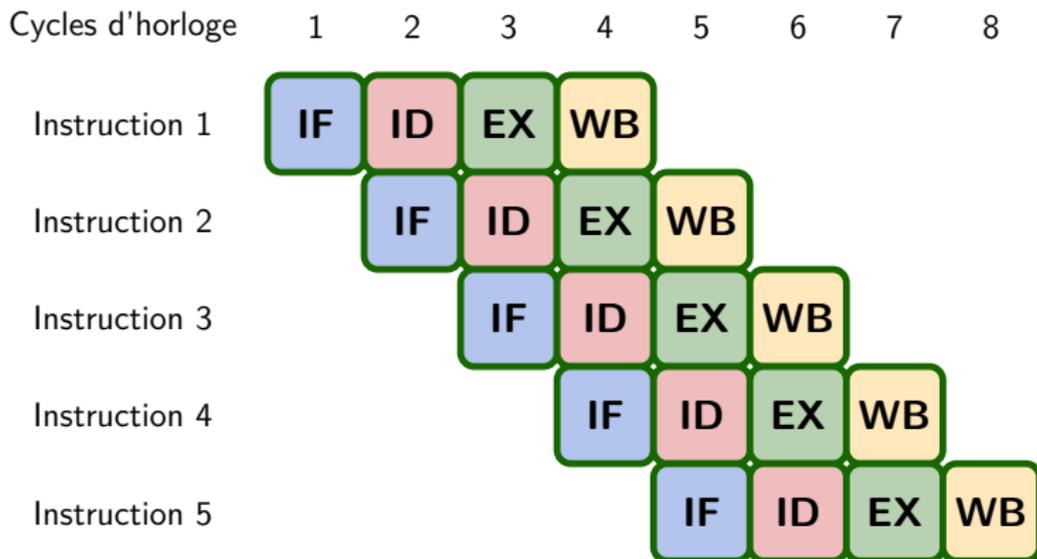
et nécessite  $4 + 1 + 1 = 6$  étapes.

À chaque instant, toute sous-tâche est sollicitée au plus une fois.

Plusieurs sous-tâches sont exécutées **en même temps**.

# Pipeline — schéma

Le **pipeline** (chaîne de traitement) permet l'organisation suivante :



Il permet (dans cet exemple) d'exécuter 5 instructions en 8 cycles d'horloge au lieu de 20.

# Pipeline — informations

Le pipeline étudié ici est une variante simplifiée du *Classic RISC pipeline* introduit par D. Patterson.

Chaque étape est prise en charge par un **étage** du pipeline. Il dispose ici de quatre étages.

Les processeurs modernes disposent de pipelines ayant bien plus d'étages :

Processeur	Nombre d'étages du pipeline
Intel Pentium 4 Prescott	31
Intel Pentium 4	20
Intel Core i7	14
AMD Athlon	12

Étant donné que l'utilisation d'un pipeline induit un entrelacement dans l'exécution des instructions, des problèmes peuvent survenir. Ceux-ci sont appelés **aléas** et peuvent être de trois sortes :

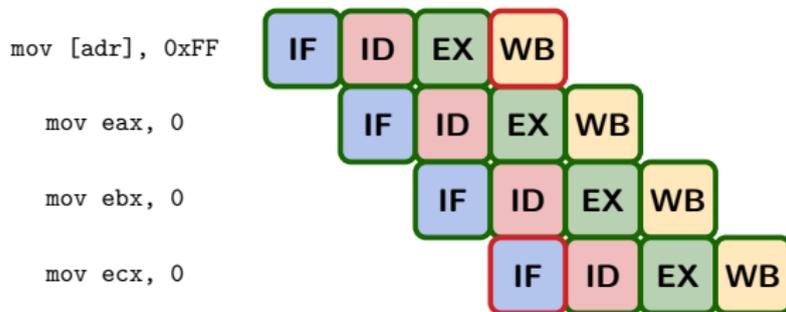
- 1 **aléas structurels** (conflit d'utilisation d'une ressource matérielle) ;
- 2 **aléas de donnée** (dépendance d'un résultat d'une précédente instruction) ;
- 3 **aléas de contrôle** (saut vers un autre endroit du code).

Une **solution générale** pour faire face aux aléas est de suspendre l'exécution de l'instruction qui pose problème jusqu'à ce que le problème se résolve. Cette mise en pause crée des « **bulles** » dans le pipeline.

# Aléas structurels

Un **aléa structurel** survient lorsque deux étapes d'exécution d'instructions souhaitent accéder au même moment à une même ressource.

Considérons p.ex. un pipeline dans la configuration suivante :

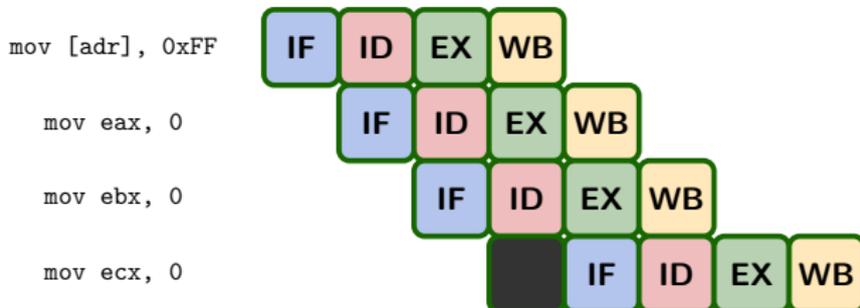


**Problème** : le **WB** de la 1<sup>re</sup> instruction tente d'écrire en mémoire, tandis que le **IF** de la 4<sup>e</sup> instruction cherche à charger la 4<sup>e</sup> instruction. Toutes deux sollicitent au même moment le bus reliant l'unité arithmétique et logique et la mémoire.

# Aléas structurels

**Solution 1.** : temporiser la 4<sup>e</sup> instruction en la précédant d'une bulle.

On obtient :



On continue d'appliquer ce raisonnement (en ajoutant des bulles) jusqu'à ce qu'il n'y ait plus d'aléa structurel.

**Solution 2.** : diviser la connection entre unité arithmétique et logique et la mémoire en deux bus.

L'un sera utilisé pour accéder aux données et l'autre, au programme.

Ceci ne résout tout de même pas l'aléa **ID** vs **WB** car

- d'une part, **ID** lit des données en mémoire (qui sont les arguments de l'instruction) ;
- d'autre part, **WB** écrit des données en mémoire (dans le cas où l'instruction le demande).

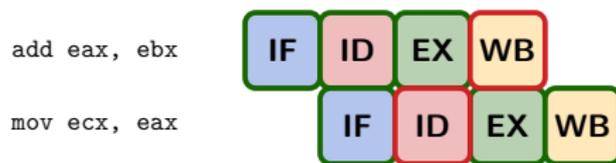
Il y a donc ici une sollicitation simultanée du bus d'accès aux données.

On peut, pour améliorer cette solution, imaginer des architectures avec **plusieurs bus** connectant l'unité arithmétique et logique et les données.

# Aléas de donnée

Un **aléa de donnée** survient lorsqu'une instruction  $I_1$  a besoin d'un résultat calculé par une instruction  $I_0$  précédente mais  $I_0$  n'a pas encore produit un résultat exploitable.

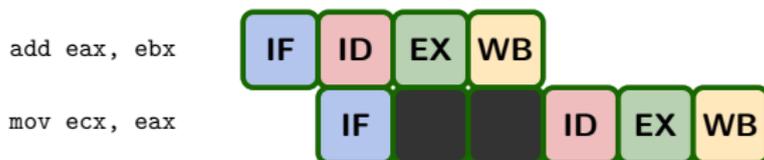
Considérons un pipeline dans la configuration suivante :



**Problème** : le **ID** de la 2<sup>e</sup> instruction charge son argument `eax`. Cependant, ce chargement s'effectue avant que le **WB** de la 1<sup>re</sup> instruction n'ait été réalisé. C'est donc une mauvaise (la précédente) valeur de `eax` qui est chargée comme argument dans la 2<sup>e</sup> instruction.

**Solution** : temporiser les trois dernières étapes de la 2<sup>e</sup> instruction en les précédant de deux bulles.

On obtient :



La seconde bulle est nécessaire pour éviter l'aléa structurel **ID vs WB**.

On continue d'appliquer ce raisonnement (en ajoutant des bulles) jusqu'à ce qu'il n'y ait plus d'aléa de donnée ou structurel.

# Aléas de donnée

Changer, lorsque cela est sémantiquement possible, l'ordre des instructions d'un programme peut avoir une conséquence sur le **nombre de bulles** créées dans le pipeline (et donc sur la **vitesse d'exécution**).

Considérons p.ex. les instructions en C suivantes :

```
a = a + b;  
c = c + d;
```

Il existe au moins deux manière de les traduire en assembleur :

```
mov eax, [adr_a]           mov eax, [adr_a]  
mov ebx, [adr_b]           mov ebx, [adr_b]  
add eax, ebx               mov ecx, [adr_c]  
mov [adr_a], eax           mov edx, [adr_d]  
mov ecx, [adr_c]           add eax, ebx  
mov edx, [adr_d]           add ecx, edx  
add ecx, edx               mov [adr_a], eax  
mov [adr_c], ecx           mov [adr_c], ecx
```

# Aléas de donnée

La première version est une traduction directe du programme en C. Elle provoque cependant de nombreux aléas de donnée à cause des instructions `mov` et `add` trop proches et opérant sur des mêmes données.

La seconde version utilise l'idée suivante.

On sépare deux instructions qui se partagent la même donnée par d'autres instructions qui leur sont indépendantes. Ceci permet de diminuer le nombre de bulles dans le pipeline.

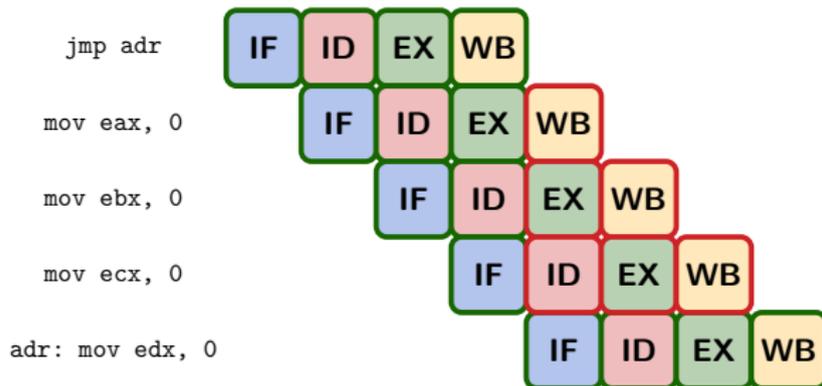
Un **aléa de contrôle** survient systématiquement lorsqu'une instruction de saut est exécutée.

Si  $I$  est une instruction de saut, il est possible qu'une instruction  $I'$  qui suit  $I$  dans le pipeline ne doive pas être exécutée. Il faut donc éviter l'étape **EX** de  $I'$  (parce qu'elle modifie la mémoire).

L'adresse cible d'une instruction de saut est calculée lors de l'étape **EX**. Ensuite, lors de l'étape **WB**, le registre `eip` est mis à jour.

# Aléas de contrôle

Considérons un pipeline dans la configuration suivante :

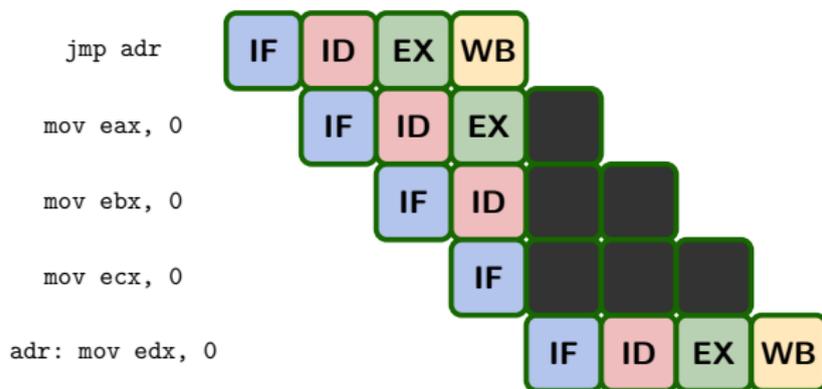


**Problème** : la 1<sup>re</sup> instruction, qui est un saut, ordonne le fait qu'il faut interrompre le plus tôt possible l'exécution des trois instructions suivantes (situées entre la source et la cible du saut).

Ces trois instructions sont chargées inutilement dans le pipeline.

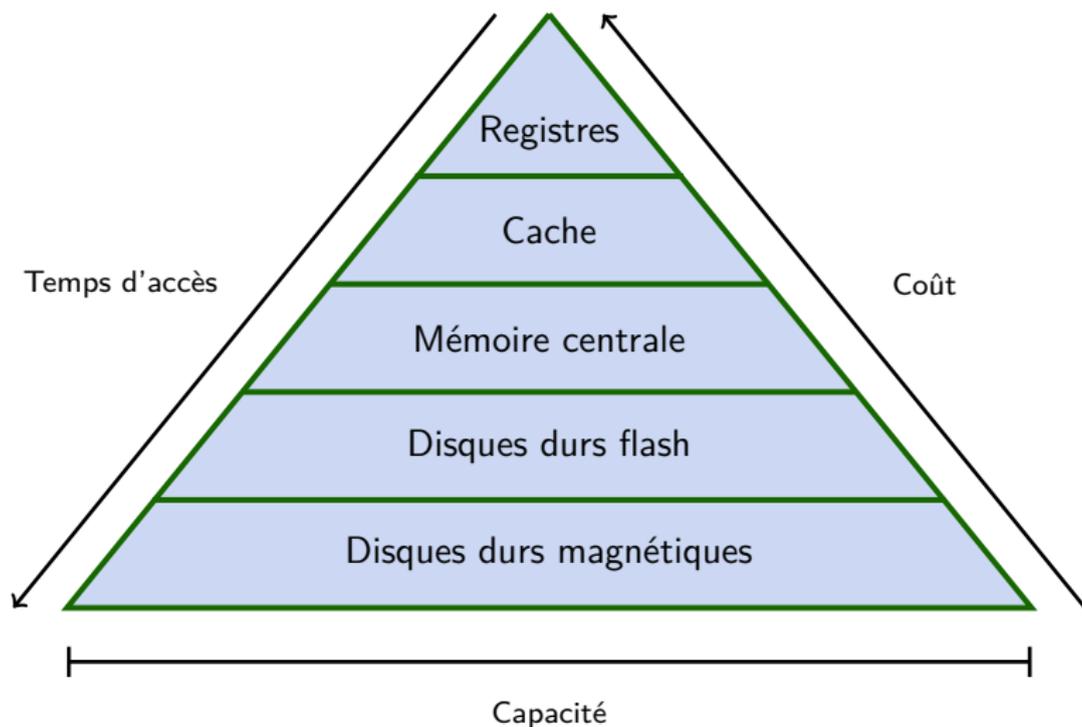
**Solution** : ne pas exécuter les étapes des instructions sautées après avoir exécuté le **WB** de l'instruction de saut.

On obtient :



Ce faisant, trois cycles d'horloge ont été perdus.

# Les trois dimensions de la mémoire



Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la **volatilité** (présence obligatoire ou non de courant électrique pour conserver les données mémorisées) ;
- le nombre de **réécritures** possibles ;
- le **débit de lecture** ;
- le **débit d'écriture**.

Voici les caractéristiques de quelques mémoires :

- **registre** : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- **mémoire morte** (ROM, **R**ead **O**nly **M**emory) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;
- **mémoire vive** (RAM, **R**andom **A**ccess **M**emory) : volatile, réécriture possible, débit de lecture/écriture de l'ordre de 8 Gio/s, temps d'accès de l'ordre de 10 ns ;
- **mémoire flash** : non volatile, réécriture possible (de l'ordre de  $10^5$  fois), débit de lecture/écriture de l'ordre de 500 Mio/s, temps d'accès de l'ordre de 0.1 ms ;
- **mémoire de masse magnétique** : non volatile, réécriture possible, débit de lecture/écriture de l'ordre de 100 Mio/s, temps d'accès de l'ordre de 10 ms.

# Principes de localité

**Problématique** : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

**Localité temporelle** : si une zone de la mémoire a été considérée à un instant  $t$  donné, elle a une forte chance d'être reconsidérée à un instant  $t'$  proche de  $t$ .

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

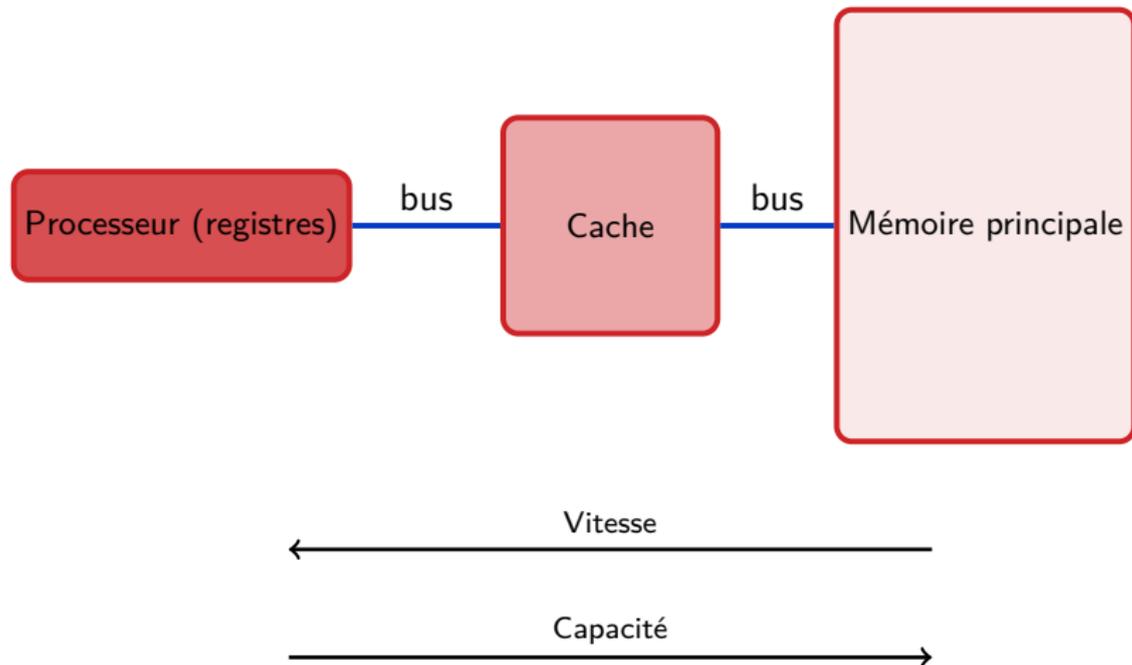
**Localité spatiale** : si une zone de la mémoire à une adresse  $x$  donnée a été considérée, les zones de la mémoire d'adresses  $x'$  avec  $x'$  proches de  $x$  ont une forte chance d'être considérées.

La localité spatiale s'observe dans la manipulation de **tableaux** ou encore de la **pile** : les données sont organisées de manière contiguë en mémoire.

Ces deux principes impliquent le fait qu'à un instant donné, un programme **n'accède qu'à une petite partie de son espace d'adressage**.

# Organisation de la mémoire

La mémoire est organisée comme suit :



# La mémoire cache dans l'organisation de la mémoire

La **mémoire cache** est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs **couches** : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

Il fonctionne de la manière suivante :

- 1 le processeur demande à lire une donnée en mémoire ;
- 2 la mémoire cache, couche par couche, est interrogée :
  - 1 si elle contient la donnée, elle la communique au processeur ;
  - 2 sinon, la mémoire principale est interrogée. La mémoire principale envoie la donnée vers la mémoire cache qui l'enregistre (pour optimiser une utilisation ultérieure) et la transmet au processeur.

# Organisation du cache

La mémoire cache est organisée en **lignes**. Chaque ligne est en général constituée de 32 octets.



- V est un **bit de validité** : il informe si la ligne est utilisée.
- Indicateur permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.
- Mot\_1, Mot\_2, Mot\_3 et Mot\_4 contiennent des **données**.

La ligne est la plus petite donnée qui peut circuler entre la mémoire cache et la mémoire principale.

Le **mot** est la plus petite donnée qui peut circuler entre le processeur et la mémoire cache. Celui-ci est en général composé de 4 octets.

# Stratégies de gestion de la mémoire cache

**Invariant important** : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la **propriété d'inclusion**.

Ceci implique que lorsqu'une donnée qui figure dans la mémoire cache est modifiée, il faut également modifier l'instance de la donnée située en mémoire principale.

Il existe deux stratégies pour cela :

- 1 **l'écriture simultanée** : lorsqu'une ligne du cache est modifiée, la mémoire principale est immédiatement mise à jour. Cette méthode est lente.
- 2 La **recopie** : lorsqu'une ligne du cache est modifiée, on active un drapeau qui la signale comme telle et la mémoire principale n'est mise à jour que lorsque nécessaire (juste avant de modifier à nouveau la ligne du cache en question).

# Stratégies d'écriture dans la mémoire cache

Il existe plusieurs stratégies d'écriture dans la mémoire cache, plus ou moins complexes et plus ou moins rapides :

- l'organisation à **correspondance directe** : à toute donnée est associée une position dans la mémoire cache (par un calcul modulaire).

Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, ce dernier est écrasé.

- L'organisation **totalemtent associative** : une donnée peut se retrouver à une place quelconque dans la mémoire cache.

Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, une position aléatoire est générée pour tenter de placer la nouvelle donnée.