Introduction à l'informatique

pour les mathématiques, la physique et les sciences computationnelles

Yann Thorimbert

Chapitre 6
Conception et exécution de programmes

Yann Thorimbert

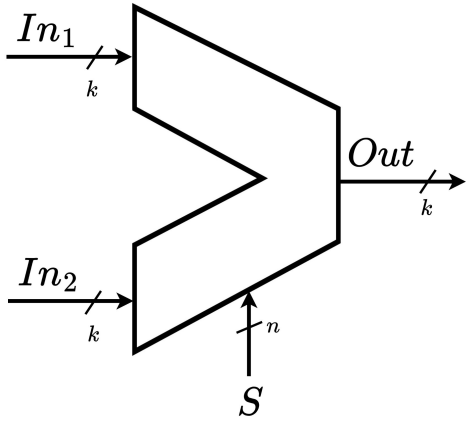
Chapitres du cours

1. Origines des ordinateurs et des réseaux informatiques
2. Codage des nombres
3. Codage des médias
4. Circuits logiques
5. Architecture des ordinateurs
6. **Conception et exécution de programmes** ←
7. Algorithmique, programmation et structures de données

Fonctionnement du CPU et formulation d'une instruction

- Pour que le processeur effectue une opération liée à une instruction, il faut lui indiquer :
  - La nature de l'opération (S) que l'UAL doit effectuer
  - Où trouver les données d'input ($\texttt{In}_1$ et $\texttt{In}_2$)
  - Où ranger la donnée du résultat (`Out`)

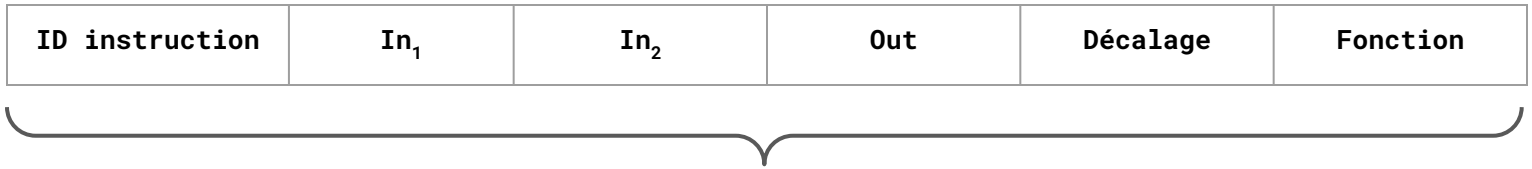
- Tout ceci doit être transmis sous forme d'états binaires à la machine, *via* un **langage machine**.



Décodage d’une instruction par l’unité de contrôle

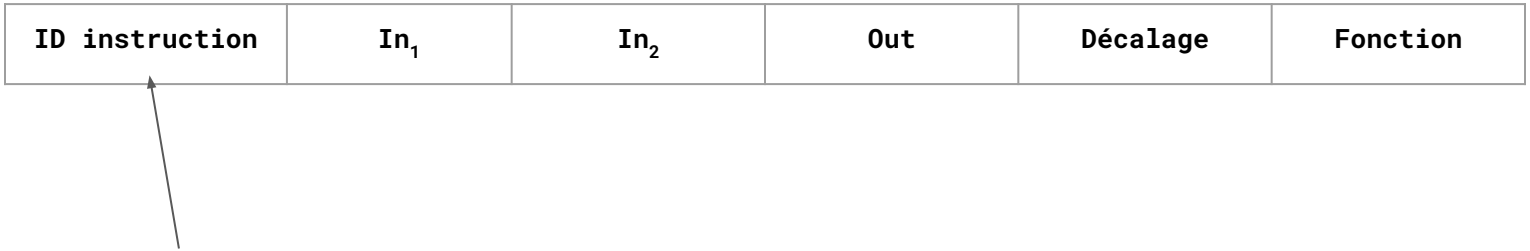
- Tout ceci doit être transmis (codé) sous forme d’états binaires à la machine, *via* un **langage machine**.

- Une telle instruction est **décodée** par l’unité de contrôle (étape *decode*) :



Décodage d’une instruction par l’unité de contrôle

Exemple pour les opérations arithmétiques et logiques (le format peut varier en fonction du type d’instruction) :



Type d’instruction (**Opcode**)
Exemples :

- Opération arithmétique et logique (`000000`)
- Chargement de données (`100011`)

(Diapositive hors champ)

Décodage d'une instruction par l'unité de contrôle

Exemple pour les opérations arithmétiques et logiques (le format peut varier en fonction du type d'instruction) :

Type d'instruction (**Opcode**)
Exemples :

- Opération arithmétique et logique (`000000`)
- Chargement de données (`100011`)

Fonction
Exemples :

- Add (`100000`)
- Sub (`100010`)
- Or (`100101`)

(Diapositive hors champ)

Décodage d'une instruction par l'unité de contrôle

Exemple pour les opérations arithmétiques et logiques (le format peut varier en fonction du type d'instruction) :

Type d'instruction (**Opcode**)
Exemples :
- Opération arithmétique et logique (`000000`)
- Chargement de données (`100011`)

Arguments (registres)
Exemples :
- `10011` (registre 19)

Fonction
Exemples :
- Add (`100000`)
- Sub (`100010`)
- Or (`100101`)

(Diapositive hors champ)

Décodage d'une instruction par l'unité de contrôle

Exemple pour les opérations arithmétiques et logiques (le format peut varier en fonction du type d'instruction) :

Type d'instruction (**Opcode**)
Exemples :
- Opération arithmétique et logique (`000000`)
- Chargement de données (`100011`)

Arguments (registres)
Exemples :
- `10011` (registre 19)

Output (registre)
Exemples :
- `10000` (registre 16)

Fonction
Exemples :
- Add (`100000`)
- Sub (`100010`)
- Or (`100101`)

(Diapositive hors champ)

Décodage d'une instruction par l'unité de contrôle

Exemple pour les opérations arithmétiques et logiques (le format peut varier en fonction du type d'instruction) :

Type d'instruction (**Opcode**)
Exemples :
- Opération arithmétique et logique (`000000`)
- Chargement de données (`100011`)

Arguments (registres)
Exemples :
- `10011` (registre 19)

Output (registre)
Exemples :
- `10000` (registre 16)

Décalages
(cf. chap 2)

Fonction
Exemples :
- Add (`100000`)
- Sub (`100010`)
- Or (`100101`)

(Diapositive hors champ)

Langage machine

- Tout ceci doit être transmis sous forme d'états binaires à la machine, *via* un **langage machine**.

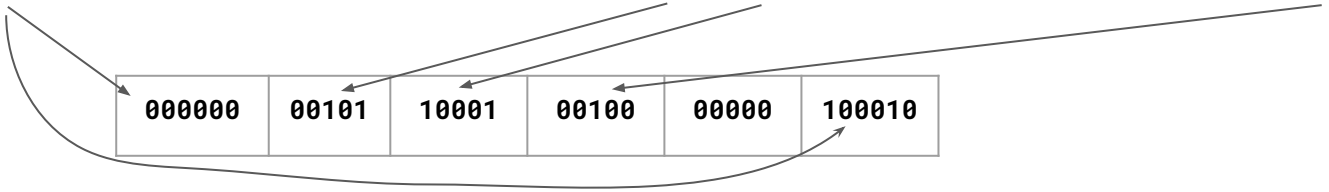
Soustraction *des données des registres* ***5*** *et* ***17*** *à ranger dans le registre* ***4***

`000000`	`00101`	`10001`	`00100`	`00000`	`100010`

Langage machine

- Tout ceci doit être transmis sous forme d'états binaires à la machine, *via* un **langage machine**.

Soustraction *des données des registres* ***5*** *et* ***17*** *à ranger dans le registre* ***4***



- Pas commode pour la programmation par un être humain.
  ⇒ Nécessaire de définir des alias plus pratiques pour l'humain.

De langage machine à langage assembleur

Soustraction** des données des registres **5** et **17** à ranger dans le registre **4

Machine

000000	00101	10001	00100	00000	100010

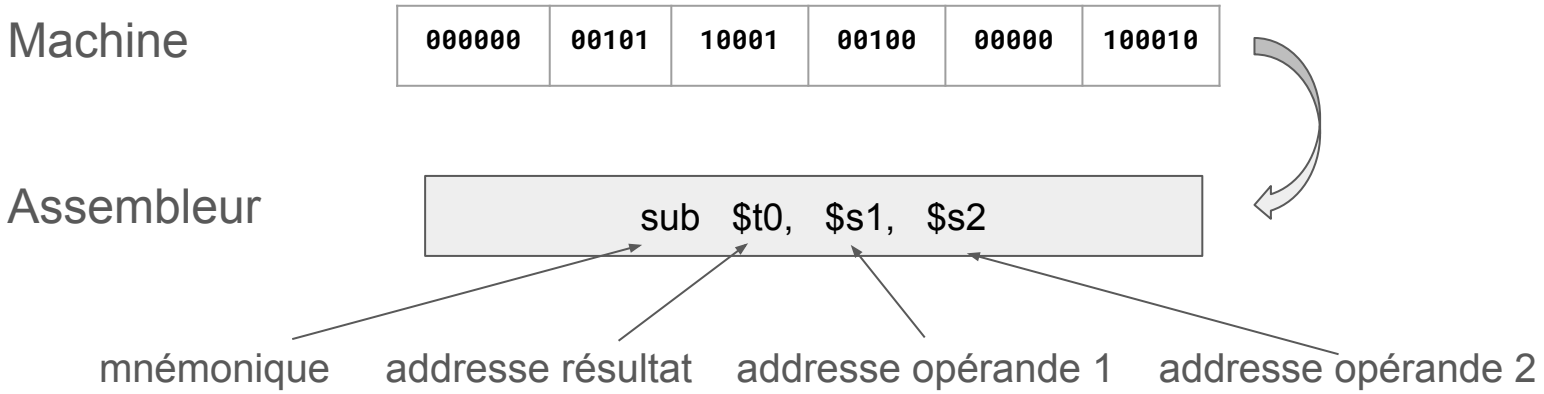
Assembleur

```
sub   $t0,   $s1,   $s2
```

- Assembleur : version lisible par l'humain du langage machine.
- Apporte déjà quelques **abstractions**.

De langage machine à langage assembleur

Soustraction** des données des registres **5** et **17** à ranger dans le registre **4



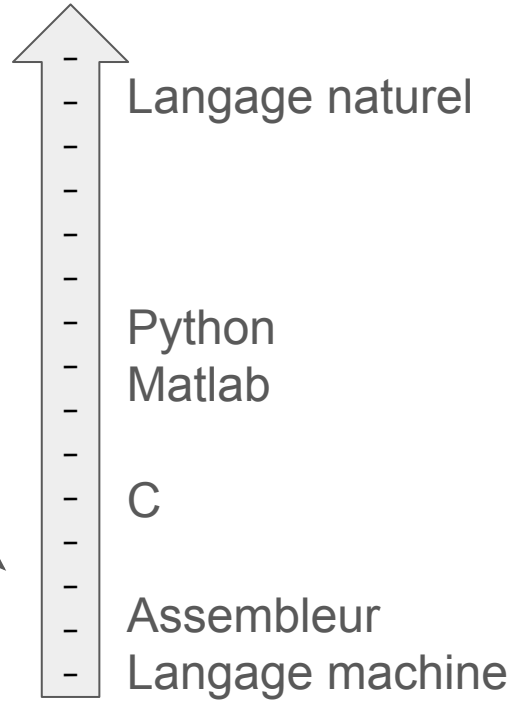
(Diapositive hors champ)

Assembleur

- Tout comme le langage machine, est **propre à une architecture donnée** !
- Comment écrire un programme fonctionnant sur des processeurs ne codant pas les instructions de la même façon ? Voire sur des processeurs ne possédant pas le même jeu d'instructions ?
- Comment formuler des instructions sans se soucier de l'endroit précis où sont stockées les valeurs en mémoire ?
- Solution : utiliser un **langage** qui **abstrait** la machine.

Niveau d’abstraction des langages

- "Haut niveau" se réfère à un langage éloigné de la machine relativement à un autre.
- "Bas niveau" se réfère à un langage proche de la machine relativement à un autre.
- Exemple : C est un langage de haut niveau comparé à de l’assembleur, mais de bas niveau comparé à Python.



Niveau d’abstraction des langages

Exemple informel pour une soustraction

Type de langage	**Exemples de langage**	**Instruction de soustraction**
Naturel	Français	"Stocker dans t0 la différence entre s1 et s2."
"Haut niveau"	Python	`t0 = s1 - s2`
"Bas niveau"	C	`t0 = s1 - s2;`
Assembleur	MIPS	`sub $t0, $s1, $s2`
Machine	MIPS	`00000010001100100100000000100010`

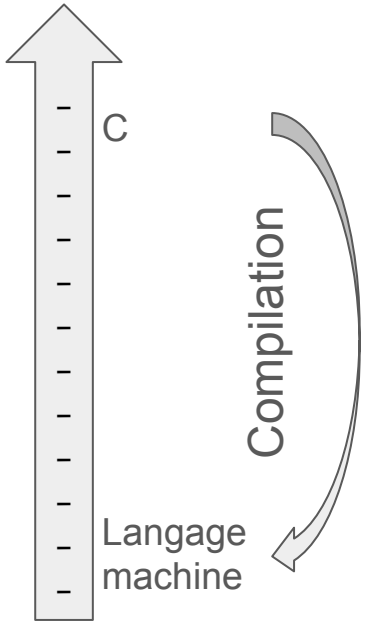
Attention : une instruction de haut niveau ne correspond pas forcément à une seule instruction de bas niveau.

Compilation

- En fin de compte, une machine concrète doit traiter l'information.
  ⇒ On n'échappe pas à une formulation du programme en langage machine, mais ceci peut être automatisé grâce au processus de **compilation**.

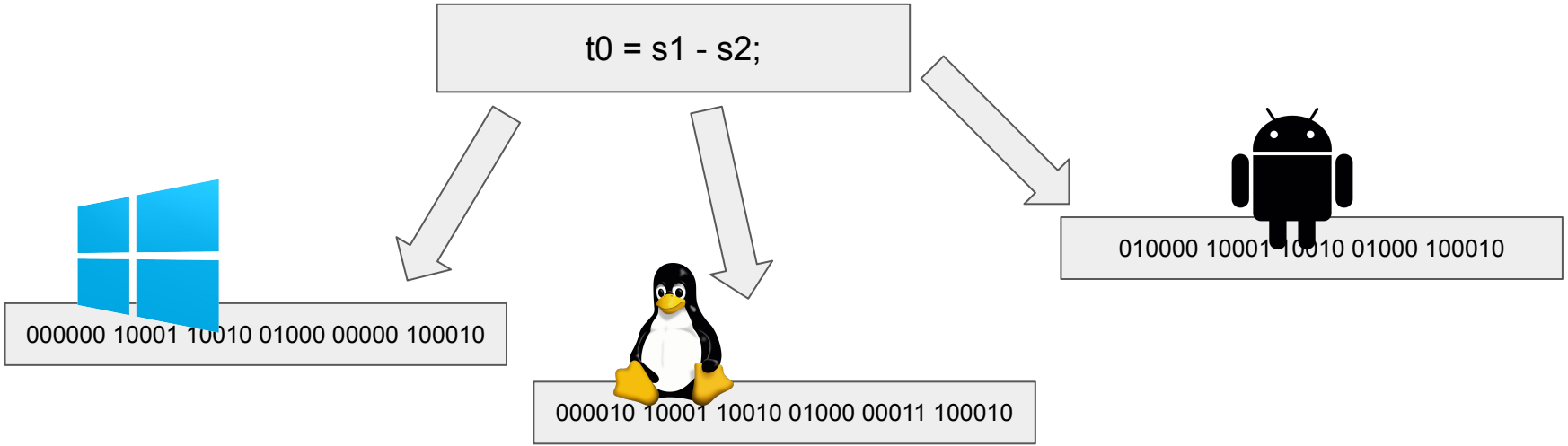
- Compilation : traduction d'un code de haut niveau vers du code machine.

- Il faut compiler le code pour chaque **architecture cible**.



Compilation

Il faut compiler le code pour chaque **architecture cible** (type de processeur + OS)



Avantages de la compilation

- Permet des **abstractions** et de la **concision** par rapport à l'assembleur. Exemples : boucles, variables sans gestion des adresses, …
    - NB : on doit parfois gérer les allocations (réservation) de la mémoire, mais pas les adresses exactes.
- Permet de détecter les **incohérences** avant l'exécution du code.
- Compilateurs modernes permettent des **optimisations** importantes.
- Code **portable** sans trop d'efforts si un compilateur existe pour la machine cible.

Note sur les optimisations

- Il a existé une époque où les compilateurs étaient moins performants. Jusque vers 2000, pas rarissime que des programmes soient quasi-intégralement écrits en assembleur pour des raisons de performance. Exemple de RollerCoaster Tycoon (Chris Sawyer)
- De nos jours, compilateurs performants, beaucoup de mémoire disponible, CPU + GPU, …

Image https://pbs.twimg.com/

Interprétation

- Couche d’abstraction supplémentaire : un logiciel spécial (l’**interpréteur**) reçoit en input le code des instructions à exécuter. Exemple : Python.

- L’interpréteur est lui-même un programme pour lequel on a défini que faire en fonction du texte (code source) reçu en input !

Langage machine

```
1001010100000011101010111110010101000000111
0101011111001010100000011101010111110010101
1000000111010101111100101010000001110101011
1110010101000000111010101111100101010000001
1101010111110010101000000111010101111100101
1010000001110101011111001010100000011101010
1111100101010000001110101011111001010100000
0111010101111100101010000001110101011111001
1010100000011101010111110010101000000111010
101111
```

Code en C

```
int main(int argc, char* argv[]) {

    if (argc != 2) {
        return 1;
    }

    const char* const filename = argv[1];
.
.
.
```

Compilé
(1 seule fois)

Input

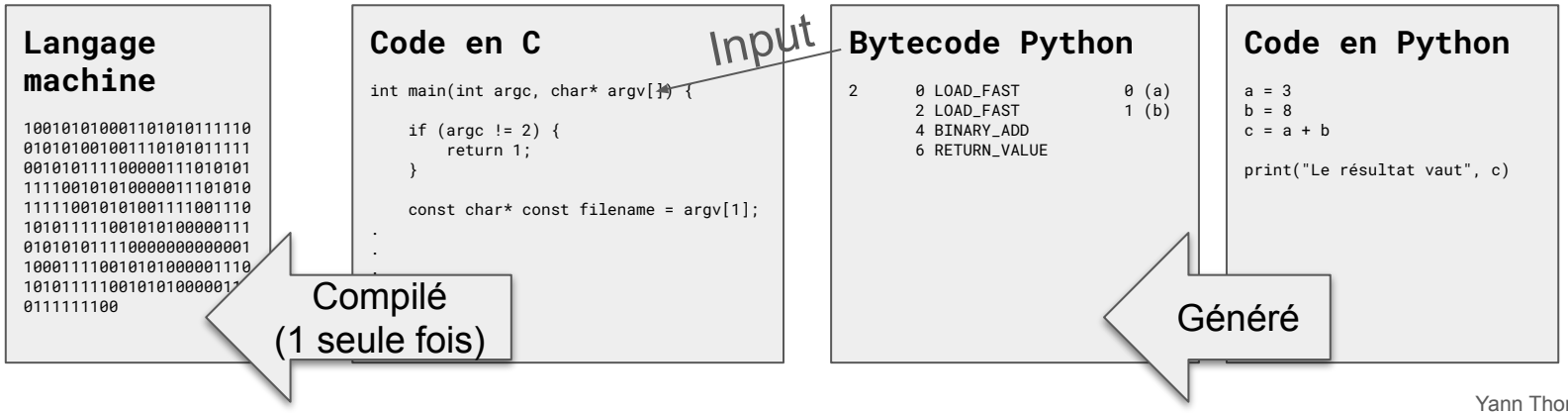
Code en Python

```
a = 3
b = 8
c = a + b

print("Le résultat vaut", c)
```

Interprétation | **Bytecode**

- En général, on passe par une forme intermédiaire appelée Bytecode (générée automatiquement). Exemple : fichiers .pyc en Python.
- Toujours indépendant de la plateforme.
- Permet des optimisations automatiques.



Interprétation | **Langages semi-interprétés**

- Problème : langages interprétés sont lents dans certains cas.

- Une solution : interprétation partielle.
  Exemples : Matlab, C#, Java, …

- Compilation Just-In-Time :
  - Parties critiques transformées si possible en langage machine plutôt qu'interprétées.
  - Tant que la compilation des parties critiques n'est pas effectuée, interprétation quand même.

Interprétation | **Avantages et inconvénients**

- Avantages :
  - Portabilité aisée en général.
  - Gestion de la mémoire facilitée ("ramasse-miettes").

- Discutable :
  - Facilité de débogage.
  - Syntaxe attrayante.
  - Rigueur moindre (par exemple typage dynamique).

- Désavantages :
  - Lenteur à l’exécution (pas toujours un problème).

Note sur la compilation et l'interprétation

- En général, on associe à chaque langage une façon **standard** d'exécuter son code source. Exemple : compilation pour C, interprétation pour Python.

- C'est un raccourci, car un langage spécifie une collection de règles pour **coder des instructions**, non pas une façon de les faire **exécuter** par la machine en fin de compte.

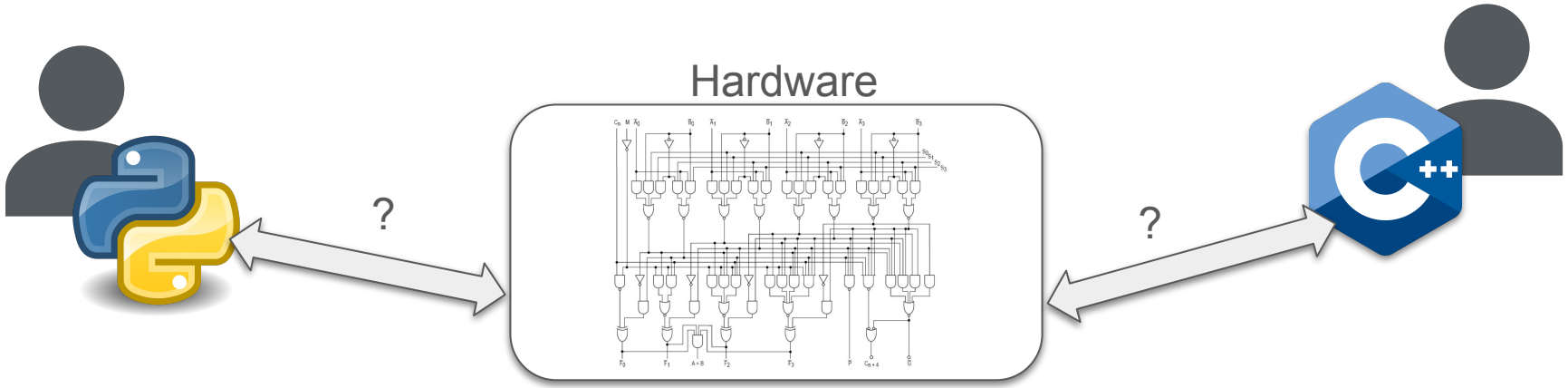
- On peut imaginer un interpréteur C, ou un compilateur Python (ceci existe en réalité).

Exemples de modes d’exécution standard

Exécution standard	**Exemples de langage**
Compilé	C, C++, Go, Rust, Haskell, Lisp, Fortran, Pascal
Hybride	Java, C#, Matlab, Julia, Erlang, Lua
Interprété	Python, JavaScript, Ruby, Perl, PHP, Octave, Mathematica, Maple

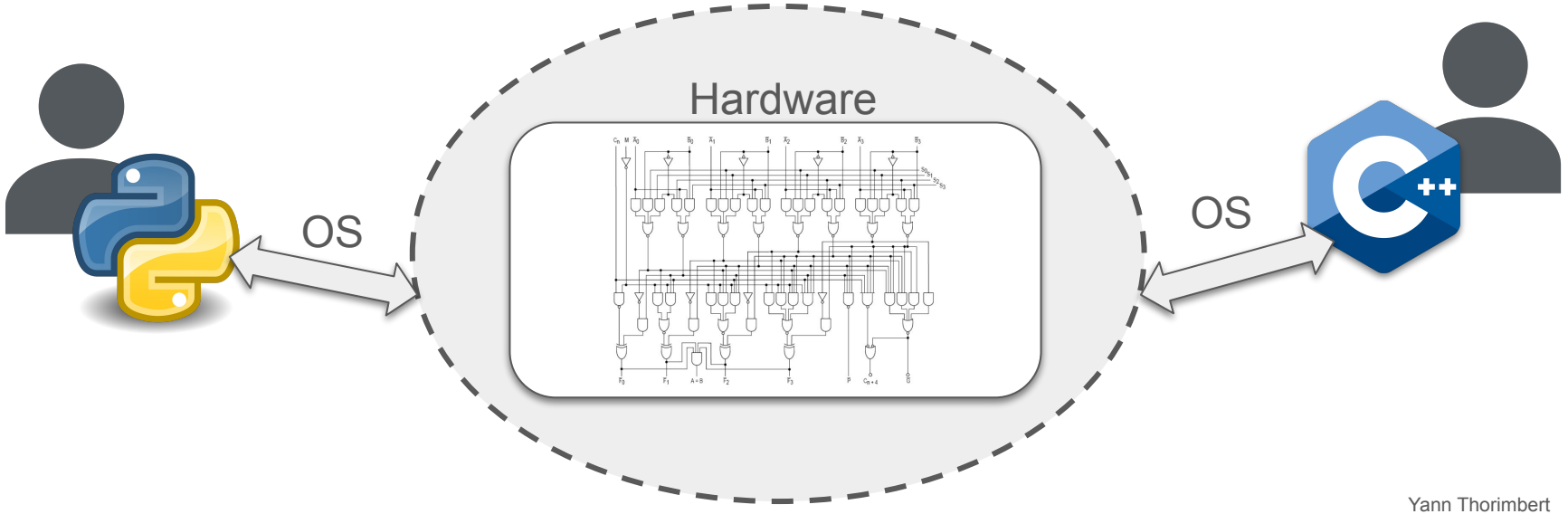
Fin du cours sur le fonctionnement des ordinateurs ?

Certaines abstractions et intermédiaires manquent. Dont l'un d'une grande importance au sein des machines modernes.



Fin du cours sur le fonctionnement des ordinateurs ?

Certaines abstractions et intermédiaires manquent. Dont l'un d'une grande importance au sein des machines modernes.
Le **système d'exploitation** (OS, operating system) fait l'intermédiaire software-hardware.



Origine historique des systèmes d’exploitation

Origine historique des systèmes d’exploitation

- L’insertion des cartes perforées est fastidieuse.

- Solution : faire des groupes ("Batchs") puis laisser la machine gérer le traitement de chaque carte.
  ⇒ Besoin d’un programme spécial pour la gestion des batchs, en plus du "vrai" programme.

Origine historique des systèmes d’exploitation

- Gestion de la lecture/écriture des cartes.
- Code pour interagir avec les autres périphériques.
- Inactivité des machines durant les accès aux périphériques.
- Cohabitation de plusieurs utilisateurs.

Origine historique des systèmes d'exploitation

- Gestion de la lecture/écriture des cartes.
- Code pour interagir avec les autres périphériques.
  ⇒ **Pilotes de périphériques.**

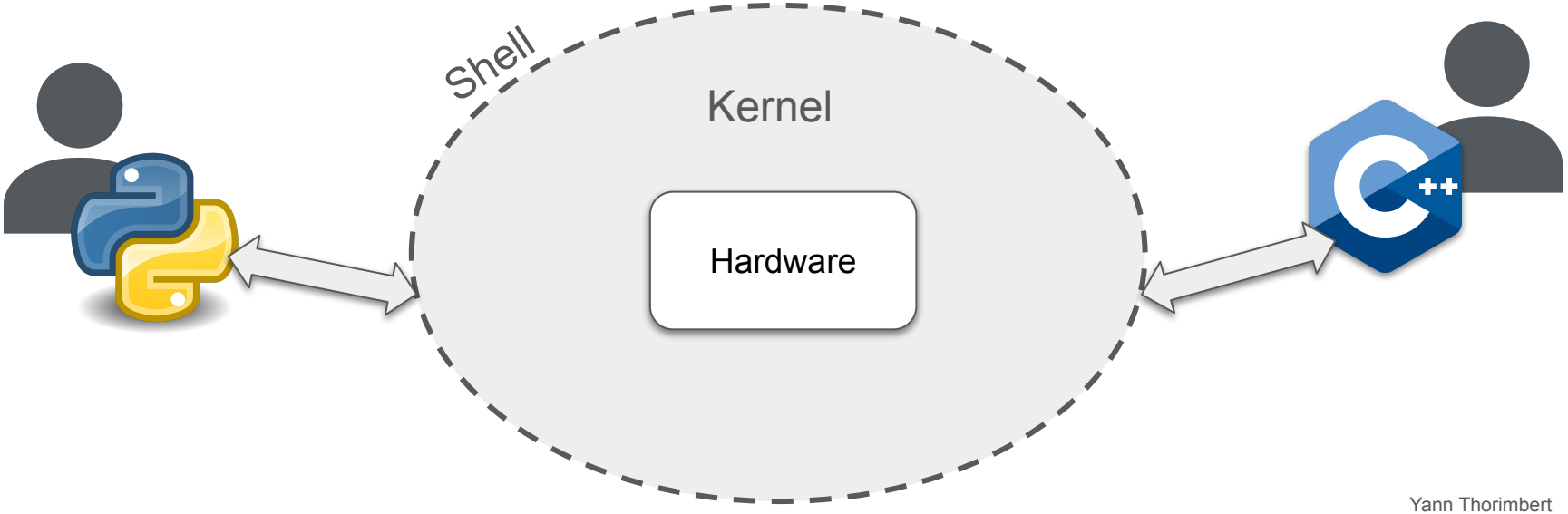
- Inactivité des machines durant les accès aux périphériques.
- Cohabitation de plusieurs utilisateurs.
  ⇒ **Multitasking et gestion des utilisateurs**.

- Puis gestion des fichiers, sécurité, …

OS

Noyau de l'OS

- Les composants essentiels de l'OS forment son **noyau** (*kernel*).
- On interagit avec la machine via l'interface offerte par l'OS (*shell*).
- On accède au noyau via des **appels système**.



Noyau de l’OS

- Les composants essentiels de l’OS forment son **noyau** (*kernel*).
- On interagit avec la machine via l’interface offerte par l’OS (*shell*).
- On accède au noyau via des **appels système** :
  - Lecture/écriture de données
  - Allocation de mémoire
  - …
- Les appels système sont typiquement des programmes écrit en C.

OS et logiciels applicatifs

- Les OS modernes viennent avec des logiciels qui ne sont pas strictement essentiels :
  - Editeurs de texte
  - Visionneuses d'images/vidéos
  - Explorateurs de fichiers
  - …
- Ces logiciels ne sont pas l'OS !

Exemples d’OS

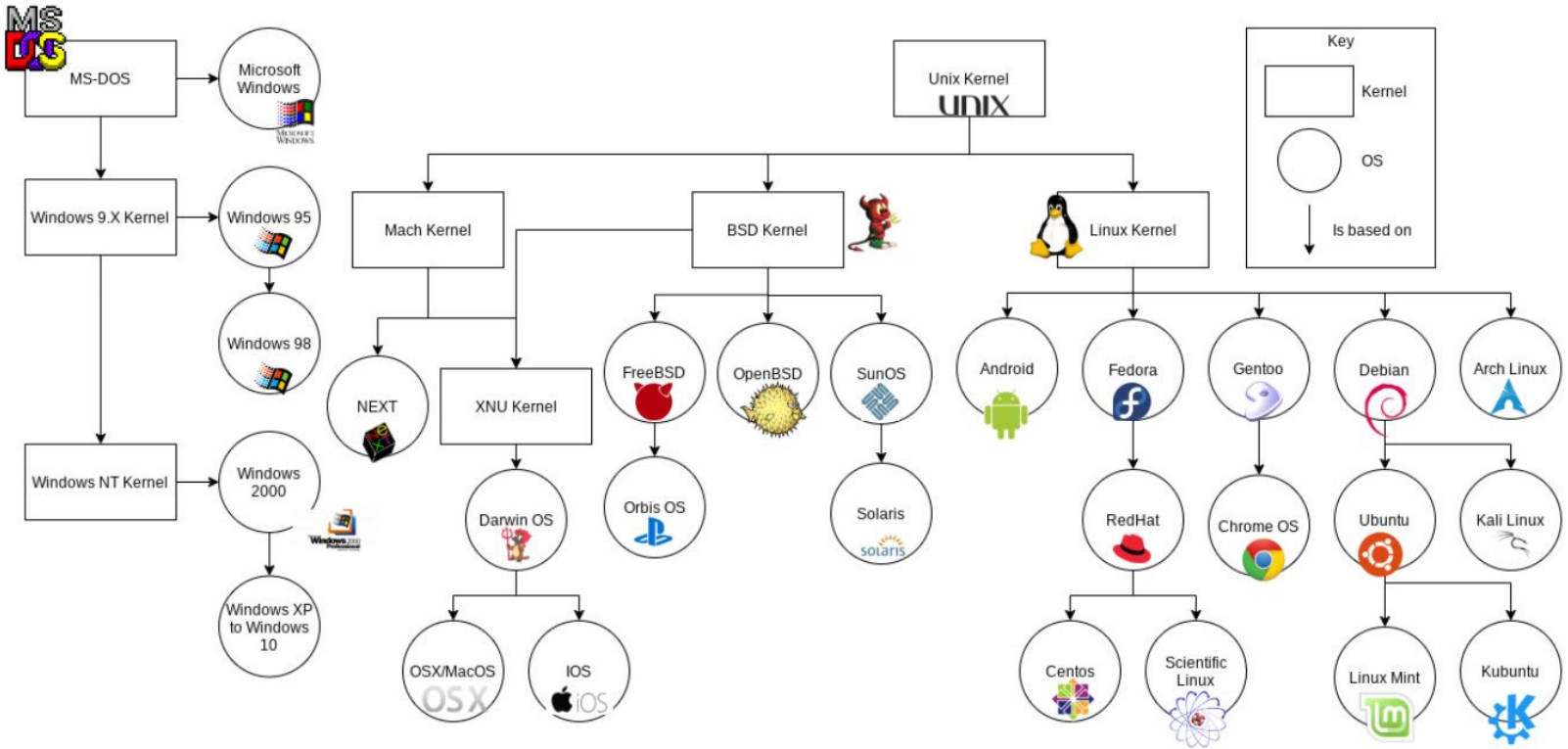
- Windows
- MacOS
- Android
- iOS
- Ubuntu
- …

Famille d’OS

- Unix (années 1970) → MacOS, iOS.
- Linux et BSD (années 1980) "Versions" libres et open-source d’Unix.
  → Ubuntu
  → Android
- Windows : descend de MS-DOS, branche à part.

(Diapositive hors champ)

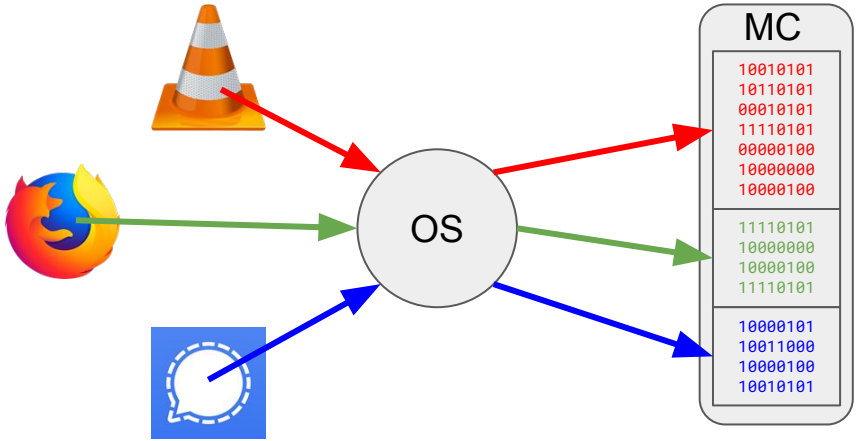
Familles d'OS



(Diapositive hors champ)

Gestion de la mémoire

- l'OS **alloue** une partie de la mémoire à chaque programme en cours d'exécution.



- Chaque programme en cours d'exécution peut faire appel à l'OS pour que l'OS lui alloue plus ou moins de mémoire si besoin.

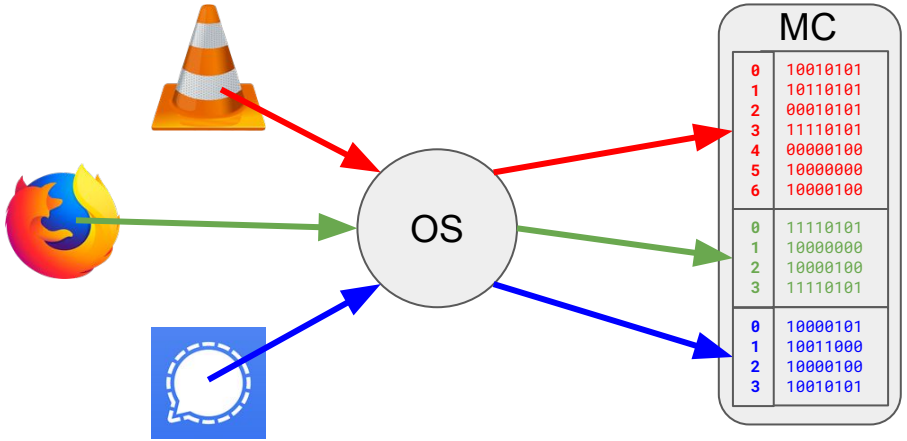
Programme vs Processus

- Programme : suite d’instructions.
- Processus : programme **en cours d’exécution** sur la machine.
- Processus caractérisé par son **état** : toutes les données nécessaires à reprendre son fonctionnement tel quel (valeurs des registres, instructions, données en cours de manipulation).
- Un processus peut contenir plusieurs fils d'exécutions parallèles (threads). Sous Linux, le terme "tâche" désigne indifféremment processus et thread.

(Diapositive hors champ)

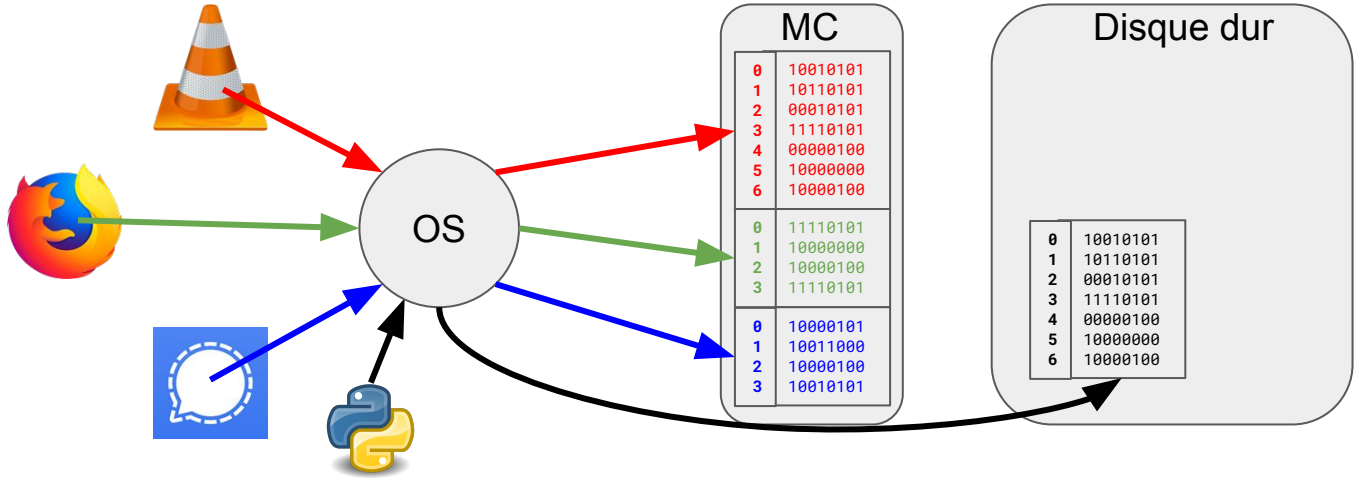
Gestion de la mémoire | **Mémoire virtuelle**

- Chaque processus voit les adresses disponibles à partir d'un certain "zéro".



Gestion de la mémoire | **Mémoire virtuelle**

- Chaque processus voit les adresses disponibles à partir d'un certain "zéro".



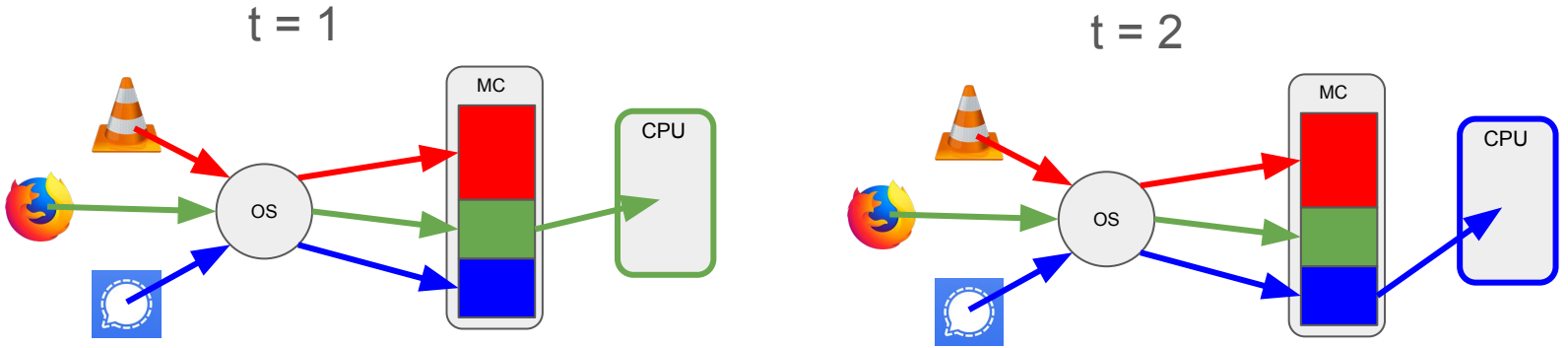
- La mémoire virtuelle permet également de déplacer l'état d'un processus sur une mémoire auxiliaire.

Un sujet technique

- La façon de gérer la mémoire est arbitraire.
- Certaines stratégies se sont imposées et sont devenue des normes, dont l'exploration sort du cadre de ce cours.
- Certains concepts importants (pile, tas) vous seront dispensés dans la suite de votre cursus, si besoin.

Ordonnancement des processus

- L'OS décide également quel processus est traité par le CPU via l'**ordonnanceur**.



- Les processus s'exécutent à tour de rôle, mais à une fréquence telle que cela crée l'illusion qu'ils s'exécutent en parallèle !

Aparté sur le parallélisme

- Nous n’avons considéré que le cas d’un processeur unique, lui-même contenant une seule UAL.

- Un processeur peut posséder plusieurs UAL… et une machine peut posséder plusieurs processeurs. Dans ce cas, ce n’est pas strictement l’architecture de von Neumann qui est impliquée.

- Par ailleurs, la façon dont les instructions sont formulées peut spécifier explicitement un **traitement en parallèle** de plusieurs données.
  ⇒ D’une certaine importance en programmation scientifique.

Aparté sur le parallélisme

- Calcul parallèle : permet de traiter différentes informations au même instant.

- Au "même" instant est à entendre pour une échelle donnée.
  - Si deux programmes avancent par "petits bouts" successifs sur le même processeur, l'un après l'autre et rapidement, ils semblent avancer en parallèle pour l'humain.

  - Il est également possible de réaliser un véritable traitement en parallèle si l'on dispose de plusieurs unités de calcul.

(Diapositive hors champ)

Aparté sur le parallélisme

- Processeur multicoeur : plusieurs UAL **sur la même puce**, se partageant généralement beaucoup de ressources (mémoire).
- Multiprocessing : plusieurs **processeurs distincts** se partageant ou pas la mémoire centrale. Commun en calcul scientifique.
- Multitasking : traitement de **plusieurs processus** en même temps (sur une ou plusieurs unités de calcul). Automatisé par l'OS.
- Multithreading : traitement de **plusieurs parties d'un même programme** en même temps (sur une ou plusieurs unités de calcul). Doit avoir été pensé.

(Diapositive hors champ)

Performance d’un ordinateur

Plusieurs paramètres impactent le temps d’exécution d’un **programme donné** :

- Fréquence d’horloge du / des CPU.
- Nombre de fils d’exécution en parallèle et efficacité de leur collaboration.
- Latence des accès à la mémoire.
- Optimisations faites par le compilateur / interpréteur.
- Partage des ressources avec d’autres programmes.

Mesure de la performance d’un ordinateur

- Programme composé d’une suite de n instructions.
- T : temps d’exécution
- f : fréquence d’horloge
- Il se trouve que $T \neq n \,/\, f$

Mesure de la performance d'un ordinateur

- Programme composé d'une suite de n instructions.
- T : temps d'exécution
- f : fréquence d'horloge
- Il se trouve que $T \neq n \,/\, f$

- Raison :
  - Bien des instructions mettent plus d'un cycle à être exécutées…
  - … d'autres moins (en moyenne et grâce à une forme de parallélisme) !
  - ⇒ On définit c comme le **nombre moyen d'instruction par cycle** d'horloge, tel que :

$$T = c \cdot n \,/\, f$$

Mesure de la performance d'un ordinateur

- Nombre moyen d'instructions par cycle : $T = c \cdot n \,/\, f$

- Nombre d'opérations en virgule flottante par seconde : FLOPS
  (*Floating Point Operations Per Second*)
  Dépend du type d'opération, et toutes les instructions ne consistent pas en des calculs.

- En pratique : utilisation de benchmarks (opérations matricielles, approximation de zéros, …)

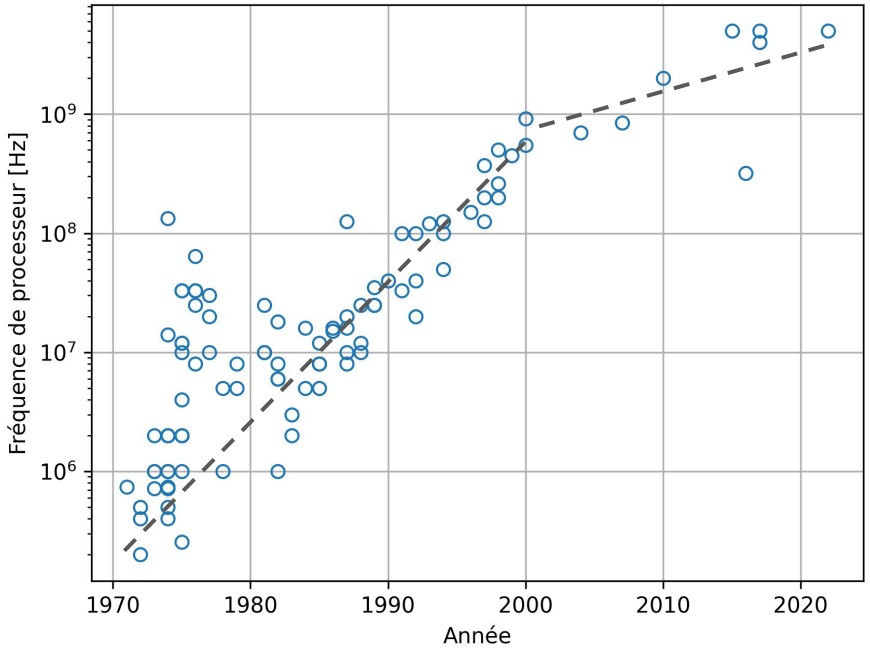
Conjecture de Moore

- Observation des années soixante : le **nombre de transistors** au sein d'un microprocesseur d'un **prix** donné double tous les deux ans.

- Souvent nommée "Loi de Moore" et souvent appliquée à d’autres grandeurs que le nombre de transistors.

- Cas de la fréquence d’horloge : limites techniques (et physiques ?) atteintes ! Depuis les années 2000, les processeurs restent cadencés à quelques GHz.

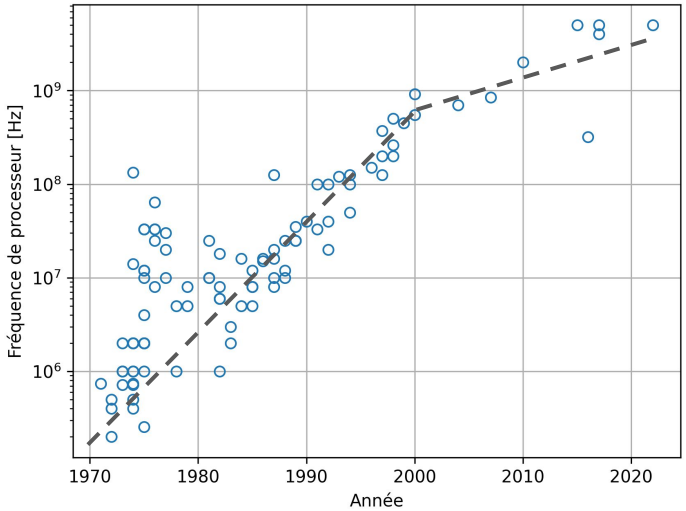
Évolution de la fréquence d’horloge



Évolution de la fréquence d’horloge

Parades : amélioration du reste !

- Accès mémoire
- Parallélisme (CPU multicoeurs, GPUs)
- Algorithmes



- Quid de la conjecture de Moore quant à la densité de transistors ?

Évolution de la densité de transistors par puce d'un prix fixe