Compilation

Le langage dans lequel l'humain programme (ex. Python) n'est pas le langage du processeur. Il faut traduire : **compilation** / **interprétation**. Le langage du processeur est bien sûr en binaire. On utilise en général un langage intermédiaire, quasiment le langage du processeur, mais avec des mots et des symboles. C'est l'**assembleur**.

Exemple de traduction

Prenons un exemple simple de code en Python (ci-dessous) et voyons comment il peut être transformé en assembleur (à droite)

```
1 x = input()
2 if x > 3 and x <= 10:
3     y = 3*(x + 2)
4 else:
5     y = 0
6 print(y)
```

Le code assembleur est plus long. Par exemple, le calcul `y = 3*(x+2)` de la ligne 3 est traduit par les lignes 7 à 10 en assembleur.

```
0      INP @x
1      MOV @x
2      CMP #3
3      BLE L0
4      MOV @x
5      CMP #10
6      BGT L0
7      MOV @x
8      ADD #2
9      MUL #3
10     STR @y
11     JMP L1
12 L0  MOV #0
13     STR @y
14 L1  OUT @y
15     HALT
```

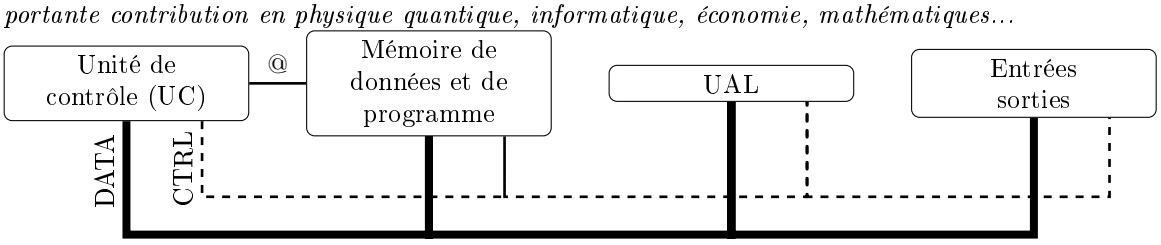
Des langages assembleurs différents

Il n'y a pas d'assembleur universel. Il y a **des** langages assembleurs qui dépendent des modèles de processeur. Des innovations technologiques dans la structure d'un processeur pourront entraîner des modifications du langage. Bien sûr, les grandes entreprises comme Intel, Microchip, Motorola ou AMD peuvent imposer des standards.

Le langage étudié dans ce cours n'existe pas. C'est une version un peu simplifiée à but pédagogique. Il est assez complexe néanmoins pour vous permettre de comprendre.

Architecture Von Neumann

John Von Neumann *(1903–1957) est un scientifique Américano-Hongrois ayant apporté d'importante contribution en physique quantique, informatique, économie, mathématiques...*



C'est la structure de tous les processeurs (il existe aussi une architecture Harvard, plus rare). Cette structure comporte une **unité de contrôle** qui contrôle tous les signaux permettant d'exécuter les instructions l'une après l'autre ; une **mémoire** qui contiendra à la fois le programme à exécuter et les données ; l'**Unité arithmétique et logique** qui fait les calculs ; les entrées sorties pour dialoguer avec l'extérieur.

Tous les organes sont reliés par le bus DATA sur lequel circulent les données ; CTRL est l'ensemble des signaux par lesquels l'UC contrôle les autres organes ; le bus d'adresse permet à UC de sélectionner la case mémoire voulue.

Fonctionnement des microprocesseurs

- UC répète toujours les mêmes opération pour réaliser un **cycle instruction**.
- Pour chaque ligne du code assembleur, on exécute un cycle instruction. Les lignes sont exécutées dans l'ordre.
- Les instructions de **saut** (`BLE`, `BGT`, `JMP`) font sauter à une autre ligne, seul cas où les lignes ne se lisent pas dans l'ordre.
- Une grande partie du travail consiste à faire des calculs élémentaires (`ADD #2`) grâce à l'**Unité Arithmétique et Logique** (UAL).
- Autrement, on déplace des données : de l'UAL vers la mémoire (`STR @y`) ou de l'entrée vers la mémoire (`INP @x`) ou de la mémoire vers l'UAL (`MOV @x`) ou de la mémoire vers la sortie (`OUT @y`)

Binaire et mémoire

Le code assembleur est traduit en binaire ce qui suppose des règles de codage. Comment code-t-on `MOV` ? ou `ADD` ?

Il faut décider de l'emplacement des variables. Par ex. puisque le programme précédent occupera les adresses mémoire `0` à `15`, on pourra placer `@x` à l'adresse 16 et `@y` à l'adresse 17.

Je rappelle que dans l'architecture Von Neumann, les données et le programme sont dans la même mémoire.

À la ligne 3, on peut sauter à l'étiquette `L0`. Cette étiquette correspond à la ligne 12 donc on traduira `BLE L0` par `BLE 12`.

On peut alors traduire chaque ligne par un code binaire unique. Pour le processeur qui nous intéresse, chaque case mémoire contient contient un mot 16 bits. Alors chaque ligne de code sera traduite de façon univoque en un mot de 16 bits.

Le programme fait 16 lignes, avec deux variables `x` et `y` initialisées à `0`. On obtient alors un code binaire de 18 lignes (à droite).

```
1011011000010000
1010011000010000
1010000100000011
0101101000001100
1010011000010000
1010000100001010
0100111000001100
1010011000010000
1000000100000010
1000100100000011
1011101000010001
0100001000001110
1010010100000000
1011101000010001
1011001000010001
0000000000000000
0000000000000000
0000000000000000
```

Dans ce document, j'ai donné le principe général. L'autre document indique tout le détail du codage des instructions avec ce processeur fictif. Vous pouvez alors interpréter chaque ligne de ce code binaire pour retrouver l'assembleur donné plus haut.

Compiler / Interpréter

On comprend qu'il faut traduire le code Python (ou autre) en un code binaire. Dans les cas compilé ou interprété, cette traduction ne se fait pas au même moment.

Cas interprété

- Le programme interprète – par exemple python.exe – doit être présent sur la machine qui exécute le programme.
- Le concepteur du programme n'a pas à se soucier du type de machine qui exécutera le programme du moment que celle ci possède une version de l'interprète.
- Le concepteur du programme est obligé de fournir le code du programme. On est open source.
- La traduction se faisant à la volée, elle est moins optimale

Cas compilé

- Le programme compilé fonctionne seul. On n'a besoin de rien d'autre pour l'exécuter.
- Le programme a été compilé pour un type de machine. Un programme compilé pour un PC 64 bits ne fonctionnera pas sur un PC 32 bits. Il faut produire diverses versions.
- Le concepteur n'a pas à fournir son code. Il donne seulement l'exécutable (version compilé) qui est quasiment illisible (puisque dans le langage de la machine)
- Le compilateur peut produire un exécutable mieux optimisé.

Autrefois, pour produire un code ultra rapide et efficace, il fallait programmer en Assembleur. Aujourd'hui, les compilateurs sont tellement optimisés que c'est devenu inutile.