1. L'architecture de Von Neuman

L'architecture de tous les ordinateurs actuels est conforme à un schéma qui a assez peu évolué depuis les premiers ordinateurs électronique à tubes à vide de 1945 (Colossus et ENIAC).

Ce modèle est dit de von Neumann du nom de son inventeur, **John von Neumann** mathématicien et physicien américano-hongrois (1903-1957).

Les principales structures

Dans l'architecture de von Neumann, un ordinateur est constitué de quatre parties distinctes :

- Le **CPU** : **C**entral **P**rocessing **U**nit (unité centrale de traitement) appelé aussi **processeur** ;
- La **mémoire** où sont stockés les données et les programmes qui se divise en :

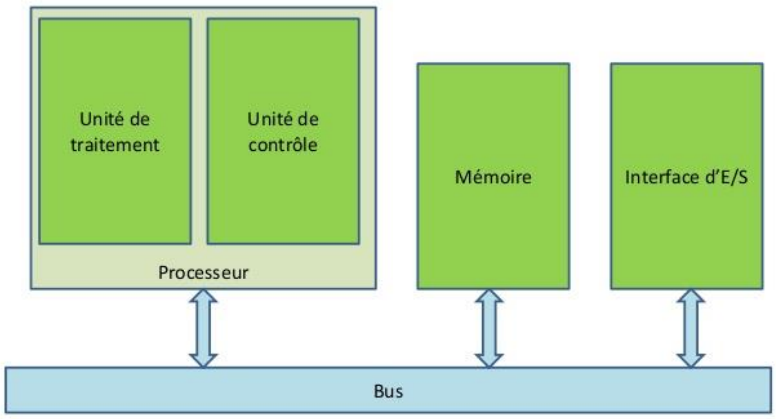
La mémoire volatile (**RAM** ou mémoire vive) très rapide d'accès qui gère les programmes et données en cours de fonctionnement. Elle perd ses données lorsqu'on coupe son alimentation électrique.

La mémoire permanente (**ROM** ou mémoire morte) plus qui gère les programmes et données de base de la machine. Elle conserve ses données même lorsqu'on coupe son alimentation électrique.

- Des **entrées-sorties** (E/S ou I/O input/output) pour échanger avec l'extérieur.
- Des **bus** qui sont des fils conduisant des impulsions électriques et qui relient les différents composants ;

Modèle de Von Neumann

schéma



Les sous-structures

Les échanges entre la mémoire et les registres du CPU se font via des bus selon une chronologie organisée par **l'horloge** et suivant la nature des échanges : données ou adresses.

Un **programme** est enregistré dans la mémoire.

L'adresse (un entier) de l'instruction en cours de traitement est stockée dans une mémoire interne au processeur, le **cp** ou **pc** : registre compteur de programme

La valeur de cette instruction (un entier) est stockée dans une autre mémoire interne, le **ri** : registre d'instruction.

Le processeur (CPU) comporte une **unité arithmétique et logique** (**UAL** ou encore **ALU**) permettant de réaliser des opérations arithmétiques sur des entiers et des opérations logiques en interprétant les impulsions électriques.

Il comprend également des **registres** permettant de stocker de façon très temporaire des opérandes ou résultats intermédiaires de calcul. La plupart des PC actuels ont des registres de tailles 64 bits.

2. Le rôle de l'horloge

Les traitements réalisés par l'ordinateur sont faits sur des représentations binaires des données et des traitements à réaliser, et ce en calculant des fonctions logiques.

Cadence d'un processeur

Le CPU dispose d'une horloge qui cadence l'accomplissement des instructions et dont l'unité est appelée cycle. La fréquence s'exprime en Gigahertz (GHz), elle signifie le nombre d'opérations que fait le processeur en une seconde.

3GHz : 3 milliards d'opération à la seconde. En clair, elle influe sur la vitesse de fonctionnement du processeur.

En 2022, les processeurs tournent entre 1,5 et 3 GHz. Certains atteignent 3.6 GHz mais la course à la fréquence à pris fin depuis 2005 environ car au-delà d'un certain cap, la chaleur dégagée est trop importante et perturbe leur fonctionnement.

Cycle d'instructions

Dans un processeur, 5 étapes sont nécessaires pour traiter une instruction. Chacune de ces 5 étapes est exécuté lors d'un cycle.

- **IF** (Instruction Fetch) : charge l'instruction à exécuter dans le pipeline (fetch = aller chercher).
- **ID** : décoder l'instruction ;
- **EX** : exécuter l'opération dans l'UAL ;
- **MEM** : accéder à la mémoire en lecture ou en écriture ;
- **WB** (Write Back) : écrire le résultat dans un registre.

Pour gagner du temps, le processeur n'exécute pas les instruction de façon séquentielle mais il exécute simultanément plusieurs instructions qui sont à des étapes différentes de leur traitement.

C'est le **pipeline d'instruction**.

Grâce au pipeline, le traitement des instructions nécessite au maximum les cinq étapes précédentes. Dans la mesure où l'ordre de ces étapes est invariable (LI, DI, EX, MEM et WB), il est possible de créer dans le processeur un certain nombre de circuits spécialisés pour chacune de ces phases.

Exemple :

Si on veut exécuter 3 instructions et qu'on exécute une des 5 étapes de l'instruction à chaque cycle, il faudra donc 3 x 5 = 15 cycles pour les 3 instructions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instr1	IF	ID	EX	MEM	WB										
Instr2						IF	ID	EX	MEM	WB					
Instr3											IF	ID	EX	MEM	WB

En utilisant le pipeline, notre processeur peut alors contenir plusieurs instructions, chacune à une étape différente.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instr1	IF	ID	EX	MEM	WB										
Instr2		IF	ID	EX	MEM	WB									
Instr3			IF	ID	EX	MEM	WB								

Et donc en 7 cycles, les 3 instructions seront exécutées.

C'est un peu comme sur une chaine de production automobile où on paralléliserait chaque opération élémentaire (carrosserie, moteur, peinture…)

3. Le langage machine

Introduction

Sur Python, le module DIS permet de désassembler le code pour avoir un aperçu du langage machine associé :

```
import dis
dis.dis('x = 1; x = x+2')
```

```
  1           0 LOAD_CONST               0 (1)
              2 STORE_NAME               0 (x)
              4 LOAD_NAME                0 (x)
              6 LOAD_CONST               1 (2)
              8 BINARY_ADD
             10 STORE_NAME               0 (x)
             12 LOAD_CONST               2 (None)
             14 RETURN_VALUE
```

Ce code est du **bytecode** (ou **opcode**) c'est-à-dire du code intermédiaire qui représente le langage machine, pour voir les instructions binaires transmises à la machine on peut utiliser les fonctions **opmap** et **opname** qui donnent le code décimal des instructions à effectuer :

```
dis.opmap['LOAD_CONST']
100                                    Soit 01100100 en binaire


dis.opname[23]
'BINARY_ADD'
```

L'assembleur

Un programme écrit dans un langage de haut niveau (comme Python) éloigné du **langage machine** (dit de bas niveau) dépend le moins possible du processeur et du système d'exploitation.

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient par exemple :

```
01ebe8140637274737475666662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
```

C'est une suite d'instructions lisible uniquement par la machine.

On peut traduire la partie `01ebe814`, de façon plus lisible : **`add $t7, $t3 , $sp`**

C'est ce qu'on appelle **l'assembleur**. L'assembleur est donc une représentation du langage machine lisible par l'humain.

Il y a autant d'assembleurs que de type de processeurs différents.

Les traitements réalisés par l'ordinateur sont faits sur des représentations binaires des données et des traitements à réaliser, et ce en calculant des fonctions logiques.

Exemple d'assembleur

En assembleur, étudions la suite d'instructions suivante :

```
MOV R0, #1
STR R0, x
LDR R0, x
MOV R1, #2
ADD R0, R0, R1
STR R0, x
```

Ce code est comparable au code figurant à la page précédente.

Dans ce code, les instructions sont exécutées dans l'ordre d'écriture, l'une après l'autre.

Jeu d'instructions

Chaque machine peut interpréter ce code binaire en instructions à effectuer, c'est ce qu'on appelle le **jeu d'instructions**.

Chaque instruction, correspond une opération à effectuer qui peut être :

- Un **transfert** entre les registres et la mémoire par exemple (chargement, sauvegarde),
- Un **calcul** arithmétique ou logique (addition, comparaison, …)
- Un **branchement** : un saut vers une autre instruction en fonction du résultat de l'opération précédente (conditionnel) ou non (inconditionnel).

Mnémonique	Signification
`LDR Rd, <Memory ref>`	Charge dans le registre `d`, la valeur stockée dans la mémoire à l'adresse `<Memory ref>`.
`STR Rd, <Memory ref>`	Range dans la mémoire à l'adresse `<Memory ref>`, la valeur stockée dans le registre `d`.
`ADD Rd, Rn, <operand2>`	Ajoute la valeur spécifiée par `<operand2>` de la valeur contenue dans le registre `n` et range le résultat dans le registre `d`.
`SUB Rd, Rn, <operand2>`	Soustrait la valeur spécifiée par `<operand2>` de la valeur contenue dans le registre `n` et range le résultat dans le registre `d`.
`MOV Rd, <operand2>`	Copie la valeur spécifiée par `<operand2>` dans le registre `d`.
`CMP Rd, <operand2>`	Compare la valeur contenue dans le registre `d` avec la valeur spécifiée par `<operand2>`.
`B <label>`	Saute à l'instruction à la position du `<label>` dans le programme.
`B<condition> <label>`	Saute à l'instruction à la position du `<label>` dans le programme, si le résultat de la dernière comparaison correspond au critère spécifié par `<condition>`. Ce critère peut-être : `EQ` : égal à `NE` : non égal à `GT` : plus grand que `LT` : plus petit que
`AND Rd, Rn, <operand2>`	Opère un ET logique entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`.
`ORR Rd, Rn, <operand2>`	Opère un OU logique entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`.
`EOR Rd, Rn, <operand2>`	Opère un XOR logique (ou exclusif) entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`.
`MVN Rd, <operand2>`	Opère un NON logique de la valeur spécifiée par `<operand2>` et range le résultat dans le registre `d`.
`LSL Rd, Rn, <operand2>`	Fait subir à la valeur contenue dans le registre `n`, un décalage vers la gauche d'un nombre de bits spécifié par `<operand2>` et range le résultat dans le registre `d`.
`LSR Rd, Rn, <operand2>`	Fait subir à la valeur contenue dans le registre `n`, un décalage vers la droite d'un nombre de bits spécifié par `<operand2>` et range le résultat dans le registre `d`.
`HALT`	Arrête l'exécution du programme.