

# Architectures Matérielles et Systèmes d'Exploitation

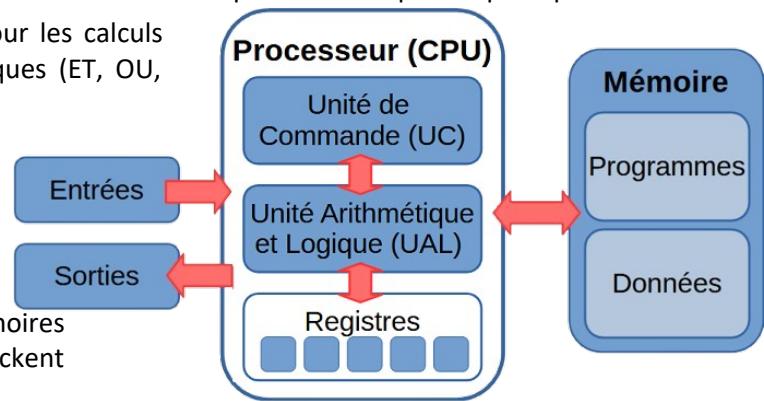
## Assembleur

### L'architecture von Neumann

On a vu dans une activité précédente le modèle d'architecture de von Neumann qui décompose un ordinateur en quatre parties distinctes :

1. **Le processeur ou CPU** : C'est le "cerveau" de l'ordinateur. Il se compose de deux parties principales :

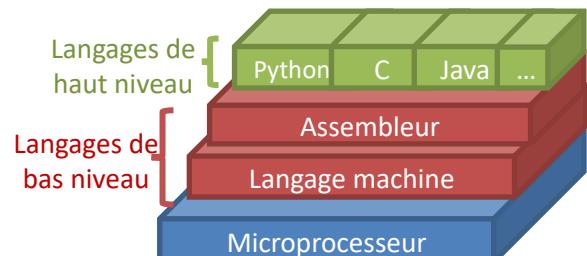
- **L'unité arithmétique et logique (UAL)** pour les calculs arithmétiques (+, -, ×, ÷) et opérations logiques (ET, OU, NON, comparaisons) ;
- **l'unité de commande (UC)** coordonne et synchronise toutes les opérations, décode les instructions et gère le séquencement. Elle contient une horloge système qui rythme l'exécution ;
- ainsi que des **registres**, petites mémoires ultra-rapides intégrées au processeur qui stockent temporairement les données.



2. la **mémoire** où sont stockés ensembles les données et les programmes.
3. des **bus** qui sont des fils conduisant des impulsions électriques et qui relient les différents composants.
4. des **périphériques d'entrées-sorties (E/S ou I/O pour Input/Output)** pour échanger avec l'extérieur.

### Langages de bas niveau/haut niveau

Le langage Python, comme de nombreux autres langages informatiques connus comme C ou Java, sont des **langages dit « haut niveau »**. Ils permettent une programmation **rapide** et plus **simple**. Ils sont aussi **portables**, c'est à dire qu'on peut coder un programme en Python et utiliser le code source sur différents ordinateurs sans ce soucier du type de processeur. Le processeur de l'ordinateur ne sait pas exécuter directement ces langages de haut niveau.



Le langage natif du microprocesseur est le **langage machine**, c'est le seul langage qu'il puisse traiter. Il est composé d'**instructions et de données codées en binaire**. Le langage **assembleur** associe des noms aux mots binaires, afin qu'il soit lisible par un humain. Le langage machine et l'assembleur sont des **langages dit de « bas niveau »** constitués d'instructions très élémentaires.

Un langage d'assemblleur est propre à chaque type de processeur, mais ils se ressemblent beaucoup.

- Chaque instruction effectue une opération de base et il faut de nombreuses instructions pour réaliser une ligne de code dans un langage de haut niveau.
- On ne peut pas créer de variables. Les données sont stockées soit en mémoire, soit dans des registres.
- Il n'y a pas de structures comme **if**, **while** ou **for**.
- Les instructions s'exécutent les unes après les autres. On peut placer des repères (étiquettes) pour faire des sauts dans le programme.
- Dans les processeurs de type RISC, comme pour l'assemblleur que l'on utilisera ici, les calculs se font uniquement avec les registres.
- Si les valeurs sont en mémoire, il faut d'abord les charger dans des registres avant de faire le calcul.

```
MOV R0, #0
loop:
    ADD R0, R0, #1
    CMP R0, #9
    BLT loop
```

Boucle « for » écrite en assembleur

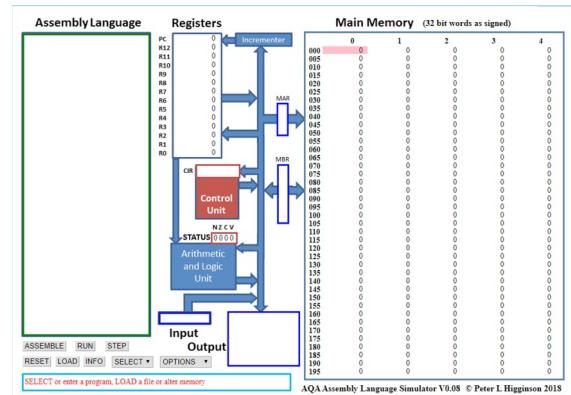
# Simulateur AQA

Cette activité est réalisée sur un simulateur d'une architecture de von Neumann développé par Peter L Higginson.

- Ouvrir le site Web du simulateur <http://www.peterhigginson.co.uk/AQA/> dans un navigateur.

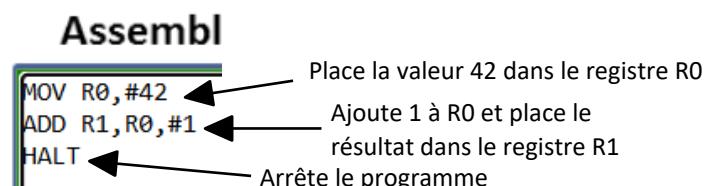
On retrouve l'architecture de von Neumann avec :

- À droite, on trouve la **RAM (Main Memory)**.
- Au centre, le processeur avec l'**unité de contrôle (Control Unit)**, l'**unité arithmétique et logique (Arithmetic and Logic Unit)** et des **registres**.
- À gauche, on trouve la zone de programmation en assembleur (*Assembly Language*).
- Les **entrées (Input)** et **sorties (Output)**.
- Des bus, en bleu, qui relient les différents composants



## Un premier programme

- Saisissez dans la partie gauche (*Assembly Language*) ce programme **en langage assembleur** contenant trois instructions.
- Cliquez sur le bouton « Submit ». Les nombres suivants apparaissent dans la mémoire :



Main Memory (32 bit words :)

	0	1	2
000	-476053462	-494923775	-285212672
005	0	0	0
010	0	0	0

Les 3 lignes du programme ont été converties **en langage machine**, la première instruction ("MOV R0,#42") est stockée à l'adresse mémoire 0 ; la seconde ("ADD R1,R0,#1") en 1; et la troisième ("HALT") en 2.

- Changez l'affichage en binaire avec le bouton "OPTION" → "binary" :

On voit maintenant le **langage machine écrit en binaire** dans la mémoire. Par exemple, l'instruction assembleur

Main Memory (32 bit words as binary)

	0	1	2
000	11100011 10100000 00000000 00101010	11100010 10000000 00010000 00000001	11101111 0
005	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000	00000000 0
010	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000	00000000 0

'MOV R0,#42` s'écrit en langage machine `11100011 10100000 00000000 00101010` (les 8 derniers bits à droite sont l'écriture de 42 en binaire). C'est la même instruction en langage machine et en assembleur, **l'assembleur est juste la traduction dans une écriture lisible par un humain du langage machine écrit en binaire**.

- Repassiez l'affichage en base 10 ("OPTION" → "signed"), cela sera plus lisible par la suite.
- Cliquez une première fois sur « STEP » pour exécuter le programme pas à pas. Observez l'animation montrant le processeur qui prend la première instruction à l'adresse 0 de la mémoire (MOV R0,#42) et l'exécute dans l'unité de contrôle. La valeur 42 est bien placée dans le registre R0.
- Cliquez une seconde fois sur « STEP ». Le processeur prend la seconde instruction à l'adresse 1 de la mémoire (ADD R1,R0,#1) pour l'exécuter. Cette fois, l'unité de contrôle fait appel à l'unité arithmétique et logique pour réaliser l'addition, puis place le résultat dans le registre R1.
- Cliquez une dernière fois sur STEP ». Le processeur prend la troisième instruction à l'adresse 2 de la mémoire (HALT) et l'exécute, le programme s'arrête.
- Avant de relancer le programme, cliquez sur « RESET » pour remettre les registres à 0.
- Relancez le programme, cette fois en cliquant sur « RUN » et observez le **compteur ordinal** dans le registre PC (pour *program counter*) qui indique l'adresse mémoire de l'instruction en cours d'exécution.

## Entrées- sorties

L'instruction « INP R0,2 » permet à l'utilisateur d'entrer une valeur (*Input*) et de la placer dans le registre R0.

11. Modifiez le programme précédent en remplaçant la première instruction « MOV R0,#42 » par « INP R0,2 » puis cliquez sur « Submit » puis « RUN » pour exécuter le programme. Testez votre programme en saisissant la valeur 42 dans la zone d'entrée.

12. Écrire un nouveau programme qui demande deux nombres et ajoute leur somme dans le registre R2.

```
INP R0,2  
INP R1,2  
ADD R2,R0,R1  
HALT
```

L'instruction « OUT R2,4 » permet d'afficher la valeur du registre R2 en sortie (*Output*).

13. Modifiez le programme précédent pour qu'il affiche la somme calculée, R2, dans la zone de sortie.

## Branchements et labels

Par défaut, le processeur exécute les instructions les unes après les autres dans l'ordre en incrémentant de 1 l'adresse mémoire de la prochaine instruction à exécuter dans le compteur ordinal. Mais dans certains cas, par exemple pour réaliser une boucle ou une instruction conditionnelle, ce n'est pas l'instruction suivante qui doit être immédiatement exécutée, il faut alors faire un **branchement** ou **saut**.

Voilà un exemple de programme qui contient un label « fin: » et une instruction de branchement « B fin ».

```
INP R0,2  
B fin          // sauter à fin directement  
ADD R1,R0,#1    // cette instruction n'est jamais executée  
fin:  
HALT
```

Noter que les commentaires en assembleur AQA s'écrivent après « // »

## Branchements conditionnels

On peut rajouter une condition à l'instruction de branchement « B » après avoir comparer des valeurs, par exemple celles des registres R0 et R1 avec l'instruction « CMP R0,R1 » :

- « BEQ label » pour un branchement vers label si les deux registres ont des valeurs égales,
- « BNE label » pour un branchement vers label si leurs valeurs sont différentes,
- « BLT label » si une valeur est inférieure à la seconde,
- « BGT label » si une valeur est supérieure à la seconde.

Si la condition de branchement est vraie, le programme saute au label indiqué, sinon l'instruction de branchement est ignorée et l'instruction suivante est exécutée.

Voici un programme qui permet de stocker dans le registre R2 la plus grande valeur des registres R0 et R1 :

```
INP R0,2  
INP R1,2  
CMP R0,R1          // compare R0 à R1  
BLT r1_plusgrand   // si R0 < R1 aller à r1_plusgrand  
r1_pluspetit :  
    OUT R0,4        // affiche R0  
    HALT  
r1_plusgrand:  
    OUT R1,4        // affiche R1  
    HALT
```

14. Copiez ce programme et testez le en saisissant quelques valeurs dans la zone *Input*.

## Boucles

Le principe d'une boucle consiste à faire un branchement en arrière tant qu'une condition est remplie. Voilà le code d'une boucle qui affiche les chiffres entre 0 et 10 (exclus) :

```
MOV R0,#0      // indice de boucle
loop:
    OUT R0,4    // affiche R0
    ADD R0,R0,#1 // incrémenté R0 de 1
    CMP R0,#10   // compare R0 à 10
    BLT loop    // si R0 < 10 retourner à loop
    HALT
```

15. Écrivez un programme qui affiche la somme des nombres entiers entre 0 et 10 (exclus).

```
MOV R0,#0
MOV R1,#1      // somme
loop:
    ADD R1,R1,R0 // ajoute R0 à R1
    ADD R0,R0,#1 // incrémenté R0 de 1
    CMP R0,#10   // compare R0 à 10
    BLT loop    // si R0 < 10 sauter à loop
    OUT R1,4
    HALT
```

Aide : le programme doit afficher 55.

16. Écrivez un programme qui demande deux nombres (entiers positifs) et affiche leur produit. Aide : il n'y a pas de multiplication en assembleur AQA, il faut se débrouiller avec les additions !

```
INP R0,2
INP R1,2
MOV R2,#0      // indice de boucle
MOV R3,#0      // produit
loop:
    ADD R3,R3,R0 // ajoute R0 à R3
    ADD R2,R2,#1 // incrémenté R2 de 1
    CMP R2,R1    // compare R2 à R1
    BLT loop    // si R2 < R1 sauter à loop
    OUT R3,4
    HALT
```

## Écrire et lire dans la mémoire

Aujourd'hui, les vitesses de microprocesseurs sont incroyablement rapides par rapport aux temps d'accès à la mémoire, il est donc important de travailler au maximum sur les données stockées dans les registres, car le temps d'accéder à des données en mémoire des centaines d'instructions ont le temps d'être exécutées. Néanmoins toutes les données ne peuvent être stockées dans des registres et il faut pouvoir les lire et les écrire en mémoire.

L'instruction « STR R0,100 » permet d'écrire la valeur du registre R0 à l'adresse mémoire 100.

17. Modifiez le programme précédent pour qu'il écrive les deux nombres entrés l'utilisateur et leur produit aux adresses mémoires 100, 101 et 102.

```
...
STR R0,100
STR R1,101
STR R3,102
```

Cette fois-ci, pour relancer le programme, il faut cliquer sur « RESET » pour remettre les registres à 0, mais aussi « ASSEMBLE » pour réinitialiser la mémoire.

On peut aussi utiliser la valeur d'un registre comme adresse mémoire. Dans ce cas là, on écrit :

```
MOV R0,#42      // valeur à écrire en mémoire
MOV R1,#100     // adresse mémoire
STR R0,[R1]     // écrit la valeur de R0 à l'adresse mémoire R1
```

18. Écrivez un programme qui écrit tous les nombres entre 0 et 10 (exclus) aux adresses mémoire allant de 100 à 109.

```
MOV R0,#0      // indice de boucle
MOV R1,#100    // adresse mémoire
loop:
  STR R0,[R1]  // écrit la valeur de R0 à l'adresse mémoire R1
  ADD R0,R0,#1 // incrémente R0 de 1
  ADD R1,R1,#1 // incrémente R1 de 1
  CMP R0,#10   // compare R0 à 10
  BLT loop    // si R0 < 10 sauter à loop
HALT
```

De la même façon qu'on a écrit en mémoire, on a souvent besoin de lire une valeur depuis la mémoire.

L'instruction « LDR R1, 100 » permet de placer dans R1 la valeur de la mémoire à l'adresse 100.

On remarque que **les données et les programmes sont stockés ensemble dans la mémoire**. C'est le propre du modèle d'architecture von Neumann. Il ne faut pas écraser des données ou des programmes involontairement !

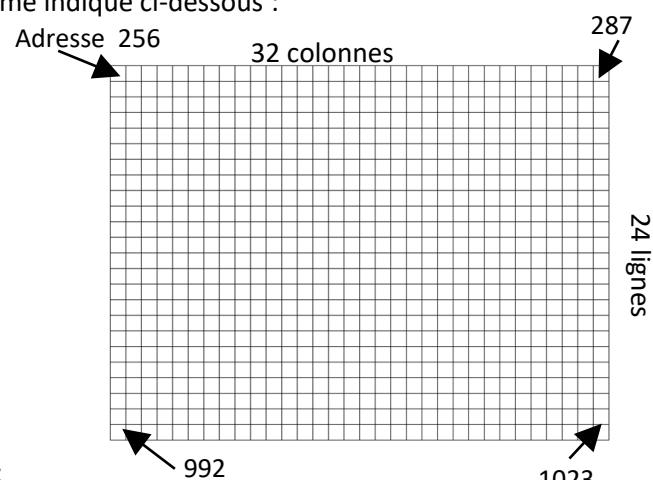
## Mémoire vidéo adressable

Le simulateur fournit une « mémoire vidéo adressable », qui s'exécute à partir de l'emplacement mémoire 256 (0x100 en hexadécimal) pour le coin supérieur gauche comme indiqué ci-dessous :

19. Testez les instructions suivantes et observez l'affichage dans la zone de sortie (*Output*) :

```
MOV R0,#0
STR R0,256
HALT
```

On peut aussi spécifier d'autres couleurs en utilisant le format RGB (*Red-Green-Blue*) en hexadécimal ; par exemple, 0x0000FF donne une couleur bleue.

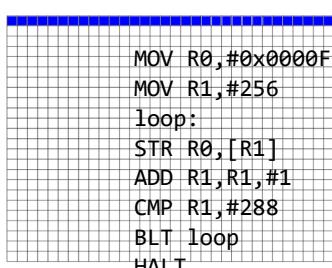


20. Modifiez la valeur de R0 pour afficher le point en bleu :

```
MOV R0,#0x0000FF // bleu
```

21. Écrivez les programmes qui affichent les figures suivantes :

a) Ligne bleue horizontale



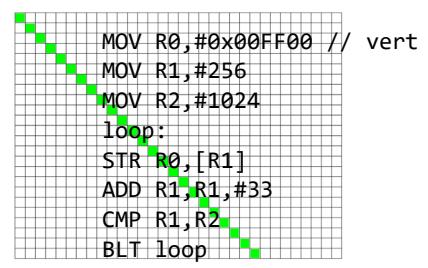
```
MOV R0,#0x0000FF // bleu
MOV R1,#256
loop:
  STR R0,[R1]
  ADD R1,R1,#1
  CMP R1,#288
  BLT loop
HALT
```

b) Ligne rouge verticale



```
MOV R0,#0xFF0000 // rouge
MOV R1,#256
loop:
  STR R0,[R1]
  ADD R1,R1,#32
  CMP R1,#992
  BLT loop
HALT
```

c) Ligne verte diagonale



```
MOV R0,#0x00FF00 // vert
MOV R1,#256
MOV R2,#1024
loop:
  STR R0,[R1]
  ADD R1,R1,#33
  CMP R1,R2
  BLT loop
HALT
```

22. Challenge : Écrivez un programme qui déplace un point depuis le haut à gauche (adresse 256) jusqu'en bas à droite (adresse 1023) en parcourant les lignes de l'écran.

```
1  MOV R0,#0x000000 // noir
2  MOV R1,#256
3  MOV R3,#257
4  MOV R12,#1024
loop:
5  STR R1,[R2]
6  STR R0,[R3]
7  ADD R2,R2,#1
8  ADD R3,R3,#1
9  CMP R3,R12
10 BLT loop
HALT
```

## Résumé des instructions assebleur AQA

Chaque famille de processeurs utilise un jeu d'instructions différent. Les instructions utilisées ici sont propres au simulateur développé par Peter L Higginson. Un autre processeur utilisera d'autres instructions, même si on retrouve souvent les mêmes. Le simulateur s'inspire de l'assemblleur que l'on trouve sur les microprocesseurs ARM, de type RISC.

Les microprocesseurs actuels se répartissent en deux grandes familles qui se distinguent par la conception de leurs jeux d'instructions:

- Les processeurs **CISC** (*Complex Instruction Set Computer*) ont beaucoup d'instructions complexes. Une seule instruction peut faire plusieurs actions à la fois, comme lire une valeur, faire un calcul et enregistrer le résultat. Les instructions les plus compliquées prennent plusieurs cycles d'horloge (entre 2 et 10).
- Les processeurs **RISC** (*Reduced Instruction Set Computer*) ont moins d'instructions et elles sont plus simples. Chaque instruction fait une seule action. Elles ont toutes la même taille et s'exécutent généralement en un seul cycle d'horloge.

Les processeurs Intel ont une architecture CISC, ceux d'Apple sont en RISC. On retrouve aussi des processeurs de type RISC dans différents SoC<sup>1</sup>, des objets connectés ou embarqués comme les cartes Arduino, des cartes d'ordinateurs SBC<sup>2</sup>, etc.

LDR Rd,<memory ref>	Charge la valeur stockée à l'emplacement mémoire spécifié par <memory ref> dans le registre Rd.
STR Rd,<memory ref>	Enregistre la valeur qui se trouve dans le registre Rd à l'emplacement mémoire spécifié par <memory ref>.
ADD Rd,Rn,<operand2>	Ajoute la valeur spécifiée dans <operand2> à la valeur du registre Rn et stocke le résultat dans le registre Rd.
SUB Rd,Rn,<operand2>	Soustrait la valeur spécifiée dans <operand2> de la valeur dans le registre Rn et stockez le résultat dans le registre Rd.
MOV Rd,<operand2>	Copie la valeur spécifiée dans <operand2> dans le registre Rd.
CMP Rn,<operand2>	Comparez la valeur stockée dans le registre Rn avec la valeur spécifiée dans <operand2>.
B <label>	Branchement vers l'instruction à la position <label> dans le programme.
B<condition> <label>	Branchement conditionnel à l'instruction à la position <label> dans le programme si la dernière comparaison satisfait aux critères spécifiés par la <condition>. Les valeurs possibles pour B <condition> et leur signification sont : <ul style="list-style-type: none"><li>• BEQ : égal à,</li><li>• BNE : différent de,</li><li>• BGT : supérieur à,</li><li>• BLT : inférieur à.</li></ul>
AND Rd,Rn,<operand2>	Effectue une opération ET logique au niveau des bits entre la valeur du registre Rn et la valeur spécifiée par <operand2> et stocke le résultat dans le registre Rd.
ORR Rd,Rn,<operand2>	Effectue une opération OU logique au niveau des bits entre la valeur du registre Rn et la valeur spécifiée par <operand2> et stocke le résultat dans le registre Rd.
INP Rd,2	Entrée d'une valeur dans le registre Rd.
OUT Rd,4	Sortie de la valeur du registre Rd.
HALT	Arrête l'exécution du programme.
<operand2>	peut être #nnn ou Rm pour utiliser soit une constante, soit le contenu du registre Rm.
Les registres vont de R0 à R12.	

Source: peterhigginson.co.uk/AQA/info.html

1 « System On a Chip » pour « système sur une puce » est un système complet embarqué sur une seule puce.

2 Single Board Computer