

Thème : Architectures matérielles et systèmes d'exploitation.

1 Histoire des ordinateurs

1.1 Machines mécaniques



Histoire 1

Les premières machines à calculer mécaniques apparaissent au XVII^{ème} siècle avec la *Pascaline* de **Blaise Pascal** capable d'effectuer addition et soustraction en 1642 et la première machine capable d'effectuer les quatre opérations, réalisée par Leibniz en 1694.

Au XIX^{ème} siècle, **Charles Babbage** conçoit les plans d'une machine analytique. Elle ne fut jamais réalisée mais elle comportait une mémoire, une unité de calcul et une unité de contrôle, comme dans les ordinateurs modernes, ainsi que des périphériques de sortie (clavier et imprimante). **Ada Lovelace** compose les premiers programmes pour la machine analytique, elle a compris qu'une telle machine est universelle et peut exécuter n'importe quel programme de calcul.

1.2 Fondements théoriques de l'informatique



Histoire 2

Dans un article fondateur de 1936 *On computable numbers, with an application to the entscheidungsproblem*, **Alan Turing**, définit précisément la notion de calcul et la relie à l'exécution d'un algorithme par une machine imaginaire qui servira de modèle aux ordinateurs modernes.

En même temps qu'il fonde l'informatique comme science, il en pose les limites avec l'exemple de fonctions non calculables. Il résout le *problème de l'arrêt* en démontrant qu'on ne saurait construire un programme général capable de prouver la terminaison de n'importe quel programme.

En 1937, **Claude Shannon** démontre comment il est possible d'effectuer des calculs à l'aide de l'électricité avec des relais électromagnétiques en prolongeant les travaux du logicien Georges Boole. Il explique comment construire un additionneur à quatre chiffres binaires qu'il désigne pour la première fois sous le terme de bit pour *binary digit*.

1.3 Machines à programmes externes



Histoire 3

La seconde guerre mondiale accélère la réalisation de machines pour des calculs de balistique ou pour déchiffrer des codes secrets.

En Allemagne, Konrad Zuse réalise en 1941, le Z1, première machine entièrement automatique lisant son programme sur une carte perforée. Aux États-Unis, Howard Aiken conçoit le Mark I. Ces premières machines électromécaniques sont colossales, et occupent des pièces entières.

En 1945, Mauchly et Eckert conçoivent avec l'ENIAC une première machine utilisant des tubes à vide.

1.4 L'ordinateur, une machine universelle à programme enregistré



Histoire 4

Les premiers ordinateurs apparaissent aux États-Unis et en Angleterre, juste après-guerre, ils sont réalisés selon l'architecture décrite par John Von Neumann dans son rapport sur la construction de l'EDVAC.

Un ordinateur est une machine programmable, capable d'exécuter tous les programmes calculables sur une machine de Turing et dont les programmes et les données sont enregistrés dans la même mémoire.



Exercice 1

Visionner le cours d'architecture disponible sur la plateforme Lumni entre les temps 7'40'' et 12'.

<https://www.lumni.fr/video/une-histoire-de-l-architecture-des-ordinateurs>

1. Comment étaient transmis les programmes avec l'ENIAC?
2. Reproduire le schéma de l'architecture de Von Neumann présenté dans la vidéo.
3. Quel était le composant fondamental des premiers ordinateurs (ENIAC, EDVAC)? Quels étaient ses inconvénients? Quel composant l'a remplacé dans les années 1950 et est encore utilisé dans les ordinateurs modernes?

1.5 Miniaturisation et démocratisation



Histoire 5

Dans les années 1950, les firmes DEC, BULL et surtout IBM développent les premiers ordinateurs commerciaux et les progrès technologiques s'enchaînent.

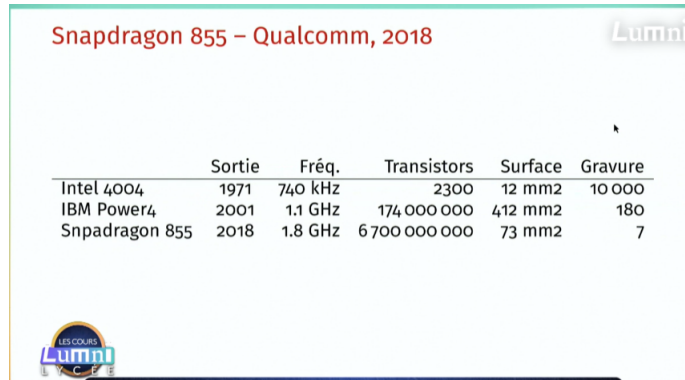
Le **transistor**, réalisé en matériau **semi-conducteur** (germanium puis silicium), inventé en 1947, remplace progressivement les **tubes à vide**. Au début des années 1960, on réalise les premiers **circuits intégrés**, en gravant tous les transistors d'un circuit dans une même plaque de silicium. Depuis la fin des années 1960, la densité de transistors par unité de surface des circuits intégrés, double environ tous les 18 mois, selon la feuille de route des industriels établie en loi empirique sous le nom de *loi de Moore*, du nom d'un des fondateurs d'Intel. La miniaturisation est le facteur principal d'amélioration des performances et de la démocratisation des ordinateurs. En effet si la lumière parcourt 30 cm en une nanoseconde dans le vide, la vitesse de propagation des électrons dans un semi-conducteur est plus lente de l'ordre de 1 cm par nanoseconde, la taille physique des circuits est donc importante.

En 1971, apparaît le premier **microprocesseur**, l'*Intel 4004* qui marque les débuts de la micro-informatique : une unité de traitement complète (unité de commande et UAL) est gravée sur un circuit intégré. Avec l'essor du réseau Internet et de ses applications comme le Web et l'explosion des télécommunications mobiles, les objets se transforment en ordinateurs : smartphones, objets connectés ... et de plus en plus de composants (mémoire, interface d'entrée/sortie) sont intégrés aux puces : on parle de **System On Chip**.

Exercice 2

Visionner le cours d'architecture disponible sur la plateforme Lumni entre les temps 17' et 22'.
<https://www.lumni.fr/video/une-histoire-de-l-architecture-des-ordinateurs>

1. Commenter le tableau ci-dessous.
2. Qu'appelle-t-on System On a Chip?



	Sortie	Fréq.	Transistors	Surface	Gravure
Intel 4004	1971	740 kHz	2300	12 mm2	10 000
IBM Power4	2001	1.1 GHz	174 000 000	412 mm2	180
Snapdragon 855	2018	1.8 GHz	6 700 000 000	73 mm2	7

Histoire 6 Amélioration des performances

Source : « Architecture et technologie des ordinateurs » d'Emmanuel Lazard

Entre le microprocesseur *Intel 4004* de 1971 et un *core i9* de 2017, le rapport de performance est de quelques millions.

L'amélioration des performances est due principalement aux progrès dans la finesse de gravure des circuits intégrés passée de 10 µm pour un transistor en 1971 à 10 nm en 2017. Avec la miniaturisation, on a pu :

- ☞ augmenter la fréquence de fonctionnement des circuits avec cependant un mur de dissipation thermique atteint vers 2004, la fréquence plafonnant désormais à 3 ou 4 GHz;
- ☞ améliorer l'architecture interne des unités de commande et d'exécution avec des évolutions pour les architectures monoprocesseur (jeux d'instructions **CISC** ou **RISC**, pipelines, multiplication des unités fonctionnelles) puis le développement des architectures multiprocesseur et du **parallélisme** ce qui présente de nouveaux défis en terme de programmation.

CPU	Horloge	Exécution	Nombre de coeurs
4004	740 kHz	10 cycles/instruction	1
core i9	4 GHz	10 instructions/cycle/coeur	8
gain	× 5000	× 100	× 8

2 Architecture de Von Neumann

2.1 Un processeur pour calculer et une mémoire pour stocker programme et données



Cours-Essentiel 1

Un ordinateur est une machine **programmable, automatique et universelle** :

- **programmable** : la séquence d'opérations exécutée par un ordinateur peut être entièrement spécifiée dans le texte d'un programme;
- **automatique** : un ordinateur peut exécuter un programme sans intervention extérieure (câblage ...);
- **universelle** : un ordinateur peut exécuter tout programme calculable (selon la théorie de Turing) avec le jeu d'instructions câblé dans son processeur.

En 1945, **John von Neumann**, mathématicien hongrois exilé aux États-Unis, publie un rapport sur la réalisation du calculateur EDVAC où il propose une architecture permettant d'implémenter une machine universelle, décrite par **Alan Turing** dans son article fondateur de 1936 sur le problème de l'arrêt.


L'**architecture de Von Neumann** va servir de modèle pour la plupart des ordinateurs de 1945 jusqu'à nos jours, elle se compose de quatre parties distinctes :

☞ L'**Unité Centrale de Traitement** (*Central Processing Unit* en anglais) ou **Processeur** est constituée de deux sous-unités :

- L'**Unité de Commande** charge la prochaine instruction dont l'adresse mémoire se trouve dans un registre appelé **Compteur de Programme** (PC en anglais) ou **Compteur ordinal**, la décode avec le **décodeur** et commande l'exécution par l'ALU avec le **séquenceur**. L'instruction en cours d'exécution est chargée dans le **Registre d'Instruction**. L'**Unité de Commande** peut aussi effectuer une opération de branchement, un saut dans le programme, en modifiant le **Compteur de Programme**, qui par défaut est incrémenté de 1 lors de chaque instruction.
- L'**Unité Arithmétique et Logique** (ALU en anglais) qui réalise des opérations arithmétiques (addition, multiplication ...), logiques (et, ou ...), de comparaisons ou de déplacement de mémoire (copie de ou vers la mémoire). L'ALU stocke les données dans des mémoires d'accès très rapide appelées **registres**. Les opérations sont réalisées par des circuits logiques constituant le **jeu d'instructions** du processeur.

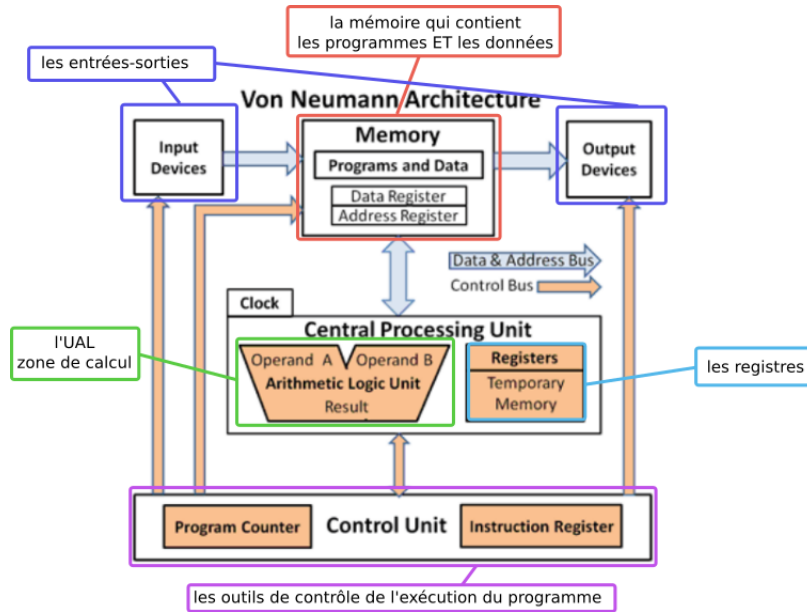
☞ La **mémoire** où sont stockés les **données** et les **programmes**.

☞ Des **bus** qui sont des fils reliant le **CPU** et la **mémoire** et permettant les échanges de données et d'adresses.

Les adresses, les données et les commandes circulent par les bus.  **Un bus ne peut être utilisé que par deux composants (émetteur/récepteur) à la fois!**

☞ Des dispositifs d'**entrées/sorties** permettant d'échanger avec l'extérieur (lecture ou écriture de données).

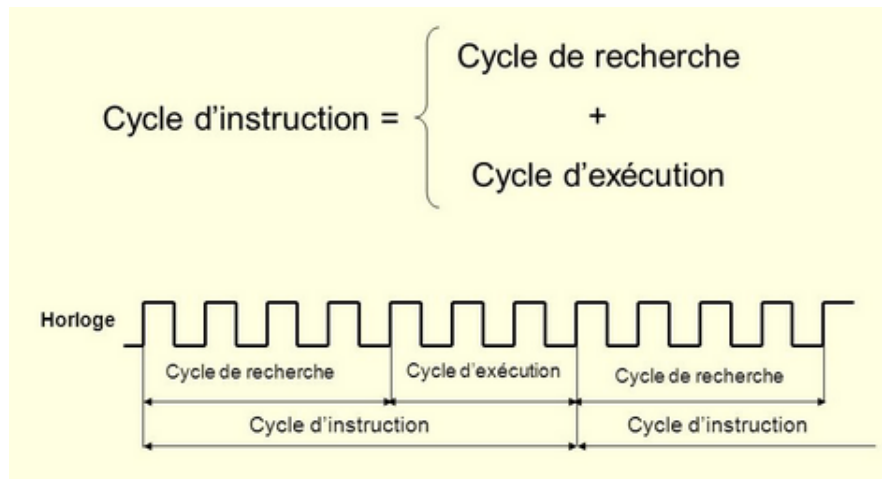
Dans le modèle de Von Neumann, le processeur exécute une instruction à la fois, de façon séquentielle.



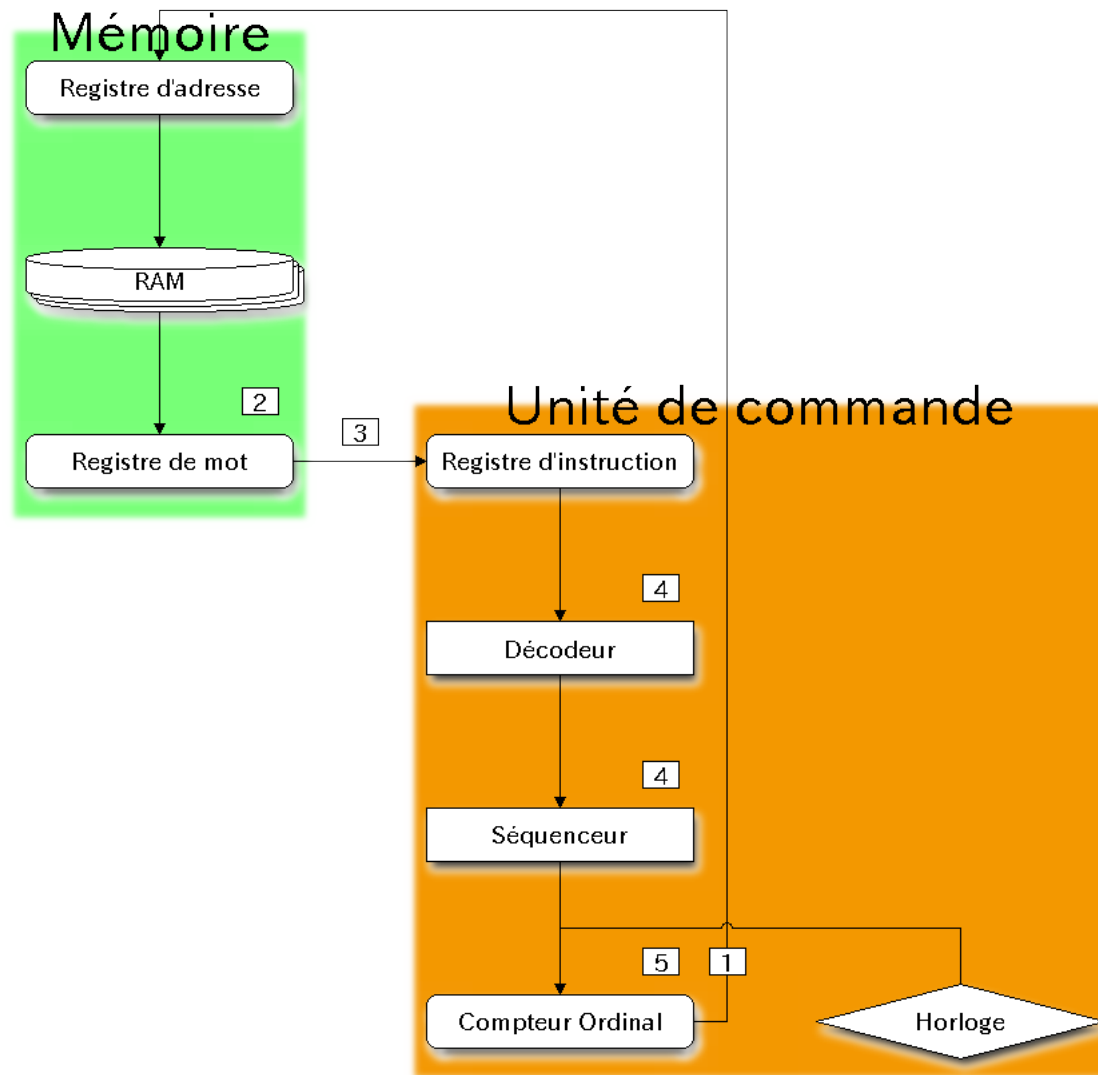
Source : Gilles Lassus

📌 Cours-Complément 1 Cycle d'instruction

Le cycle d'une instruction est constitué d'un cycle de recherche puis d'un cycle d'exécution qui mettent en jeu des échanges entre la **mémoire** et l'**Unité Centrale de Traitement**, synchronisés par des signaux périodiques émis par une horloge à quartz. Chaque cycle peut couvrir plusieurs cycles d'horloge



Source : Architecture et technologie des ordinateurs d'Emmanuel Lazard

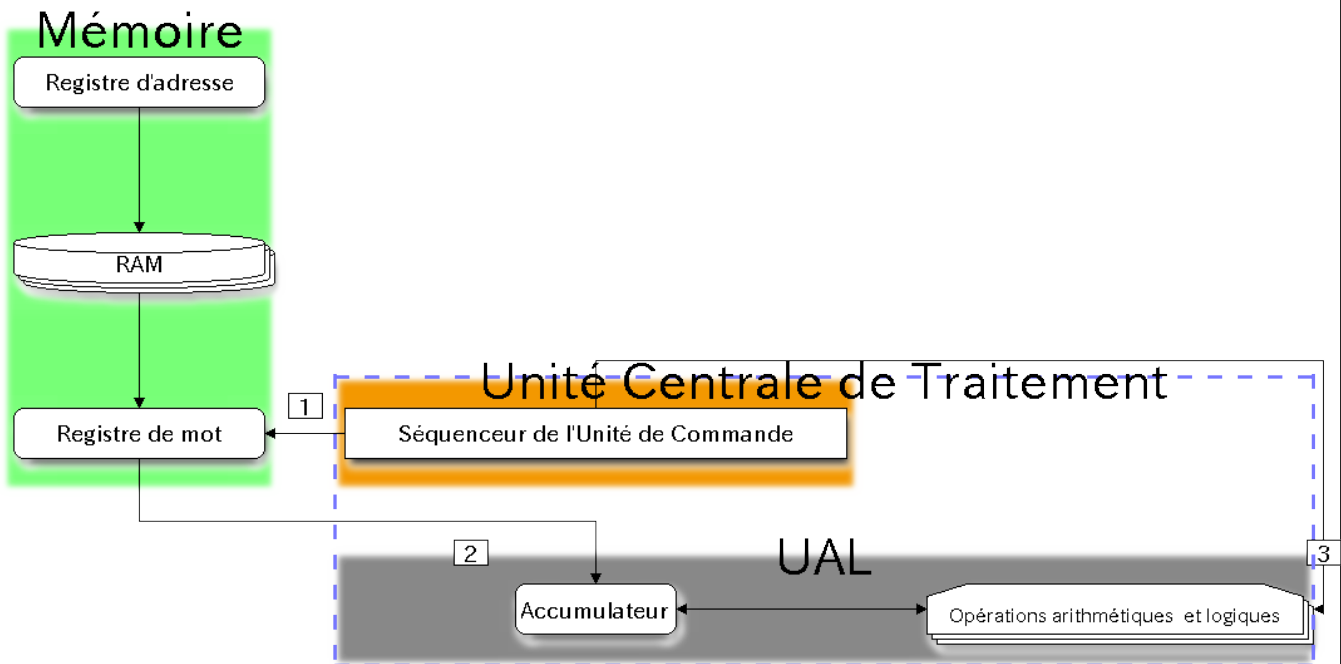


Cycle de recherche

Source : Architecture et technologie des ordinateurs d'Emmanuel Lazard

Le cycle de recherche d'instruction s'exécute d'abord avec pour acteurs la **mémoire** et l'**Unité Centrale de Commande** :

1. l'adresse de la nouvelle instruction est transférée du **Compteur Ordinal** vers le **Registre d'Adresse** de la mémoire;
2. l'**Unité de commande** envoie un signal de lecture de l'instruction et de son transfert dans le **Registre de Mot** de la mémoire qui est un tampon entre la mémoire et l'unité de traitement;
3. l'instruction constituée du *code d'opération* et de *l'adresse de l'opérande* est transférée dans le **Registre d'Instruction**;
4. par lecture du **code d'opération**, le **décodeur** détermine le type d'opération et le transmet au **séquenceur**, tandis que *l'adresse de l'opérande* est envoyée vers le **Registre d'Adresse**;
5. le **Compteur Ordinal** est incrémenté pour l'instruction suivante.



Cycle d'exécution

Source : Architecture et technologie des ordinateurs d'Emmanuel Lazard

Ensuite le *cycle d'exécution* fait intervenir en plus l'**Unité Arithmétique et Logique (UAL)** :

1. le **séquenceur** commande la lecture de l'opérande à l'adresse stockée dans le **Registre d'Adresse** et la valeur est copiée dans le **Registre de Mot** ;
2. le contenu du **Registre de Mot** est transféré vers le registre **Accumulateur** de l'**UAL** ;
3. l'opération est réalisée par **UAL** sous le contrôle du **séquenceur**

Exercice 3 QCM type E3C

1. Dans l'architecture générale de Von Neumann, la partie qui a pour rôle d'effectuer les opérations de base est :

- Réponse A :** l'unité de contrôle
- Réponse B :** la mémoire
- Réponse C :** l'unité arithmétique et logique
- Réponse D :** les dispositifs d'entrée-sortie

2. Parmi tous les registres internes que possède une architecture mono-processeur, il en existe un appelé compteur ordinal *program counter*.

Quel est le rôle de ce registre ?

- Réponse A :** il contient l'adresse mémoire de la prochaine instruction à exécuter
- Réponse B :** il contient le nombre d'instructions contenues dans le programme

Réponse C : il contient l'adresse mémoire de l'opérande à récupérer

Réponse D : il contient le nombre d'opérandes utilisés

2.2 Hiérarchie des mémoires



Cours-Essentiel 2

On distingue plusieurs types de mémoires suivant leur **persistance**, leur **capacité**, leur **rapidité d'accès**. Tous les types de mémoire utilisent une unité élémentaire d'information qui peut prendre deux états 0 ou 1 qu'on appelle **binary digit** ou **bit**. La plus petite unité adressable en mémoire comporte traditionnellement 8 bits soit un **octet**.

De nos jours les mémoires sont réalisées en matériaux supraconducteurs à partir de transistors : il faut deux portes logiques NOR pour réaliser une cellule mémoire de type *bistable*, chacune nécessitant deux transistors.

☞ Une **mémoire vive** ou **volatile** nécessite une alimentation électrique, elle est non persistante mais d'un accès rapide.

- Un **registre** est une mémoire de très petite capacité mais d'accès très rapide car située directement dans l'**Unité Centrale de Traitement**.

- La **mémoire centrale** contient les **instructions** et les **données**. Tout programme que l'on souhaite exécuter doit être chargé en mémoire centrale ou **Random Access Memory** qui est une mémoire de grande capacité.

Les bits disponibles sont regroupés par blocs en **mots mémoires** de même taille, correspondant à un ou plusieurs octets. C'est aussi la largeur du **Registre de mot** : 32 bits ou 64 bits en général de nos jours.



Chaque mot mémoire est adressable directement à partir de son adresse, on parle de **mémoire à accès direct**.

- Pour améliorer les performances, une **mémoire cache**, placée entre le processeur et la **mémoire centrale** permet d'accéder plus rapidement aux instructions et données en cours d'utilisation qui ont une plus grande probabilité d'être réutilisées.

- **Mémoire cache** et **registres** sont des *mémoires statiques (SRAM)* par opposition à la **mémoire centrale**, qui est *dynamique (DRAM)* : chaque cellule mémoire est constituée d'un transistor couplé à un condensateur qu'il faut recharger périodiquement (toutes les millisecondes).

La mémoire dynamique est plus simple à fabriquer et permet une plus grande intégration : le coût par bit est moins élevé mais le rafraîchissement en fait une mémoire plus lente.



Le temps d'accès à la mémoire centrale est entre 5 et 50 fois supérieur à celui d'un registre, cette différence de vitesse entre le processeur et la mémoire est le **goulot d'étranglement de l'architecture de Von Neumann**.

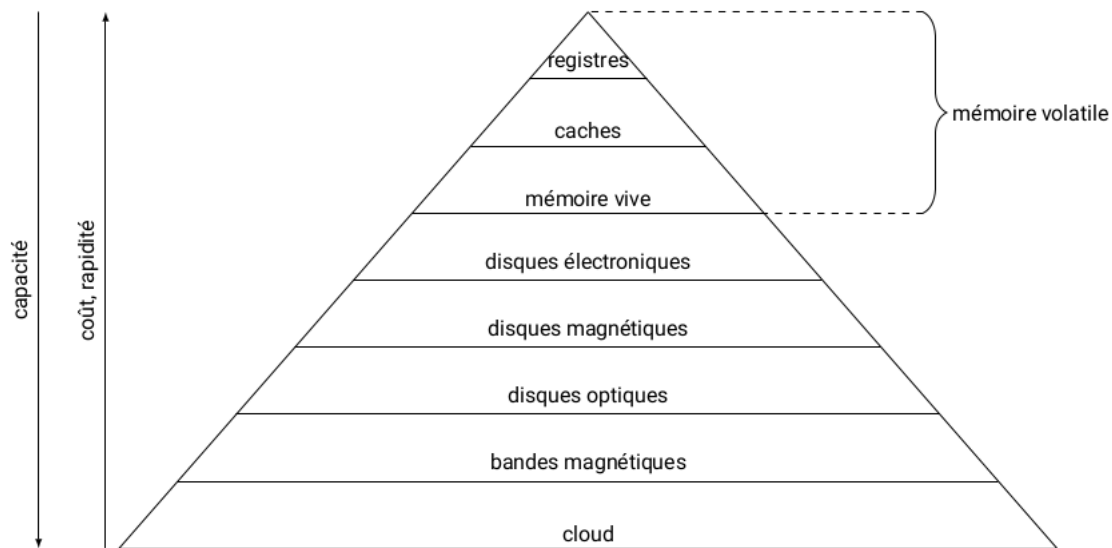
☞ Une **mémoire persistante** ou **non volatile** permet de stocker des données sans alimentation électrique, avec d'autres procédés physiques (disques magnétiques) :

- Les disques durs magnétiques ou SSD, permettent de stocker données et programmes, dont le système d'exploitation, chef d'orchestre de tous les programmes. Leur capacité est presque 500 fois supérieure à celle de la mémoire centrale mais avec un temps d'accès inversement proportionnel. Le **système d'exploitation** peut étendre virtuellement la mémoire centrale en utilisant les capacités des mémoires persistantes (**swapping**)

- La carte mère d'un ordinateur contient des données nécessaires au démarrage dans la **mémoire morte** ou **Read Only Memory**. Cette mémoire n'est pas modifiable.

mémoire	temps d'accès	débit	capacité
registre	1 ns		≈ Kio
mémoire cache	2 – 3 ns		≈ Mio
RAM	5 – 60 ns	1 – 20 Gio/s	≈ Gio
disque dur	3 – 20 ms	10 – 320 Mio/s	≈ Tio

Lorsqu'on se rapproche du processeur, le coût de la mémoire et sa rapidité d'accès augmentent tandis que sa capacité diminue.



Exercice 4

Pour mesurer une quantité d'information en informatique, on utilise l'octet représentant 8 bits.

Les préfixes multiplicateurs les plus utilisés sont définis en base 10 : 1 kilooctet (Ko) = 10^3 octets, 1 mégaoctet (Mo) = 10^6 octets, 1 gigaoctet (Go) = 10^9 octets.

Il existe aussi des préfixes multiplicateurs en base 2 : 1 kibioctet (Kio) = 2^{10} = 1 024 octets, 1 mébioctet (Mio) = 2^{20} ≈ 1,049 mégaoctet, 1 gibioctet (Gio) = 2^{30} ≈ 1,074 gigaoctet.

1. Quelle taille de mémoire centrale en octets peut-on adresser avec un processeur d'architecture 64 bits?
2. Un disque dur a une capacité de 500 Go. Exprimer cette capacité en Gio.
3. Sachant qu'une minute de musique au format mp3 occupe un espace de 1 Mo, combien d'heures de musique peut-on stocker sur un baladeur de 18 Go.
4. Avec un débit de 80 Mbits/s, combien de temps faut-il pour télécharger un fichier de 1,8 Go?

3 Langage machine et assembleur

3.1 Du texte d'un langage de programmation aux instructions du langage machine

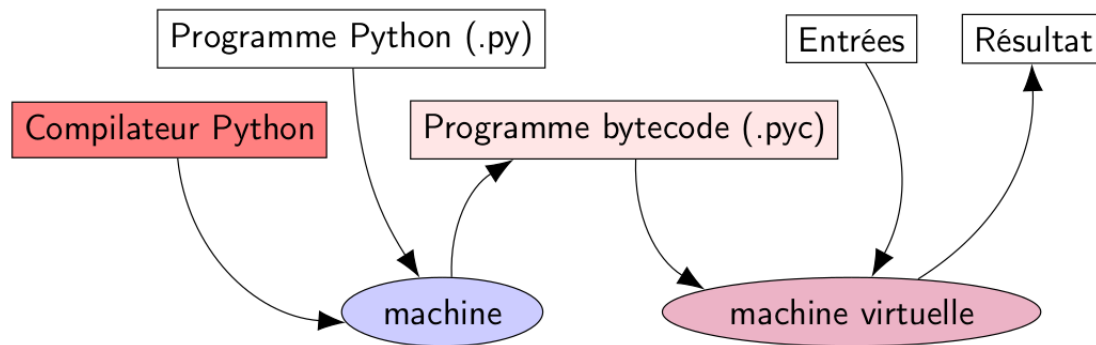
Cours-Essentiel 3

Les programmes stockés dans la mémoire centrale de l'ordinateur sont constitués d'instructions de bas niveau, exécutables directement par les circuits logiques du processeur. Le **jeu d'instructions** du microprocesseur est restreint. L'unité de contrôle décode la série de bits composant chaque instruction : les premiers bits forment un code (**opcode**) qui déclenche l'activation des circuits nécessaires dans l'ALU et les bits suivants portent les opérandes.

Un programme nommé **compilateur** permet de transformer le texte d'un programme en langage de haut niveau (comme Python ou C) en une série d'instructions en langage machine.

Python est un langage permettant une exécution en direct dans un interpréteur sans production de fichier binaire comme en C. L'opération de compilation se passe différemment : lors de l'exécution d'un code Python, le texte du programme est d'abord compilé en **bytecode** qui est ensuite exécuté par une machine virtuelle (l'interpréteur). Python peut ainsi être exécuté sur n'importe quel processeur, alors que pour un langage compilé comme C, il faut un compilateur adapté au processeur pour produire l'exécutable binaire à partir d'un programme source.

Un **langage d'assembleur** est un intermédiaire lisible par un humain, entre le langage machine et un langage de haut niveau. Il traduit les instructions du langage machine par des **symboles** ou **mnémoniques**. Il est possible d'écrire un programme en assembleur (dépendant du processeur) et un programme nommé également assembleur, le traduira directement en langage machine.



source : Judicael Courant

Exercice 5

On peut récupérer tous les fichiers nécessaires pour cet exercice dans l'archive compilation.zip.

On donne ci-dessous le texte d'un programme `exemple.c` écrit en langage C.

```

1 #include "stdlib.h"
2
3 int main()
4 {
5     int a = 5;
6     a = a + 3;
7     return 0;
8 }
  
```

1. Récupérer le fichier source du programme `exemple.c`.
2. Compiler le programme pour produire le fichier binaire `exemple` avec la commande `gcc exemple.c -o exemple`.

Rendre exécutable le binaire avec la commande `chmod +x exemple` puis l'exécuter avec `exemple` et constater avec `echo $?` que le programme s'est exécuté sans erreur (sortie 0).

```
fred@portable:~/sandbox$ gcc exemple.c -o exemple
fred@portable:~/sandbox$ chmod +x exemple
fred@portable:~/sandbox$ ./exemple
fred@portable:~/sandbox$ echo $?
0
```

3. Si on édite le fichier `exemple` avec un éditeur de textes, obtient-on un affichage lisible?
4. Afficher le contenu du fichier binaire `exemple` avec un éditeur hexadécimal en ligne comme <https://hexed.it/>.
5. Récupérer dans un fichier texte le code d'assembleur généré à partir du fichier source `exemple.c` avec la commande :

```
fred@portable:~/sandbox$ gcc exemple.c -o - -S > assembleur.txt
```

6. On donne ci-dessous le texte du programme en assembleur x86.

`%eax` désigne les quatre premiers octets du registre accumulateur, recevant les résultats des calculs, tandis que `%rsp` et `%rbp` sont des registres pointant respectivement vers la base et le sommet de la pile, une zone de la mémoire centrale dédiée au programme. De plus `-4(%rbp)` désigne une adresse mémoire située 4 octets en-dessous de la base de la pile.

À partir du guide fourni sur <http://www.lsv.fr/~goubault/CoursProgrammation/Doc/minic007.html>, expliquer les instructions des lignes 13, 14 et 15.

```
1  .file  "exemple.c"
2  .text
3  .globl main
4  .type  main, @function
5  main:
6  .LFB2:
7  .cfi_startproc
8  pushq  %rbp
9  .cfi_def_cfa_offset 16
10 .cfi_offset 6, -16
11 movq   %rsp, %rbp
12 .cfi_def_cfa_register 6
13 movl  $5, -4(%rbp)
14 addl  $3, -4(%rbp)
15 movl  $0, %eax
16 popq  %rbp
17 .cfi_def_cfa 7, 8
18 ret
19 .cfi_endproc
20 .LFE2:
21 .size  main, .-main
22 .ident "GCC:(Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
23 .section .note.GNU-stack,"",@progbits
```

3.2 Premiers programmes en assembleur avec le simulateur Aqua

 **Méthode**

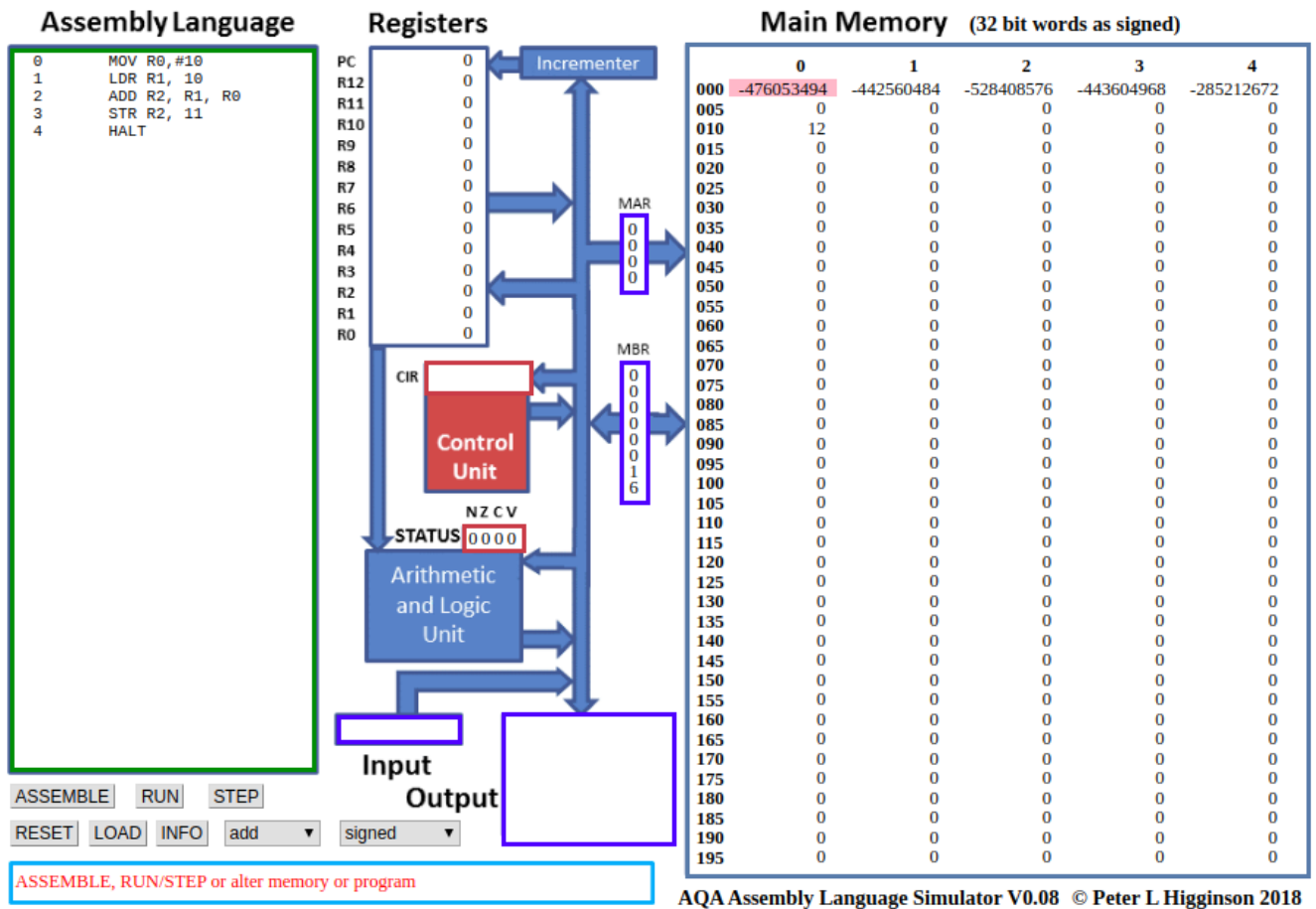
Nous allons utiliser un simulateur d'architecture de Von Neumann, réalisé par Peter Higginson pour préparer des étudiants anglais à leur examen de Computer Science. Il se nomme AQUA et on peut l'exécuter en ligne sur <http://www.peterhigginson.co.uk/AQA/>.

Quelques principes de base :

- On ne peut pas définir de variables. Les données manipulées sont soit stockées à un endroit précis en mémoire soit dans un des registres R0 à R12.
- Il n'existe pas de structure de contrôle conditionnelle comme le "if ... then ... else" ou les boucles "while", "for". Pour les implémenter, on utilise des instructions de saut inconditionnel ou conditionnel en fonction du résultat de la comparaison précédente. Les points de chute de saut sont repérés par des étiquettes placées dans le programme.
- Pour calculer avec une donnée en mémoire, il faut d'abord la transférer dans un registre.

L'interface se divise verticalement en trois zones :

- À gauche, l'éditeur de programme en assembleur. On remplit le formulaire et on le soumet avec `submit`, puis on assemble le programme en mémoire avec `assemble` et on l'exécute avec `run`. Plusieurs vitesses d'exécution sont disponibles.
- Au centre, le **processeur**, avec les treize registres de données de R0 à R12, le **Compteur de Programme PC**, l' **Unité de Contrôle** avec son **Registre d'Instruction CIR** et l'**ALU** avec ses quatre drapeaux de test (Z pour zéro, N pour négatif, C pour carry, retenue et V pour overflow). Les bus reliant les différents composants du processeur et la mémoire sont en bleu. Les registres MAR et MBR servent à transférer des données entre la mémoire et les registres : MAR contient l'adresse (en décimal) où l'on veut lire ou écrire et MBR la valeur lue ou à écrire (en hexadécimal).
- À droite, la mémoire divisée en mots de largeur 32 bits et dont les adresses commencent à 0. Dans `options` on peut choisir le format d'affichage (décimal signé ou non, binaire, hexadécimal).



Le jeu d'instructions est précisé dans la documentation <http://peterhigginson.co.uk/AQA/info.html>.
Voici quelques exemples d'instructions d'opérations arithmétiques et de transfert de mémoire :

Instruction	Traduction
LDR R1, 78	Charge dans le registre R1 la valeur stockée en mémoire à l'adresse 78
STR R1, 123	Stocke le contenu du registre R1 à l'adresse 123 en mémoire
LDR R1, [R2]	Charge dans le registre R1 la valeur stockée en mémoire à l'adresse contenue dans le registre R2
ADD R1, R0, #128	Additionne le nombre 128 (une valeur immédiate est identifiée grâce au symbole #) et la valeur stockée dans le registre R0, place le résultat dans le registre R1
SUB R1, R0, #128	Soustrait le nombre 128 de la valeur stockée dans le registre R0, place le résultat dans le registre R1
SUB R0, R1, R2	Soustrait la valeur stockée dans le registre R2 de la valeur stockée dans le registre R1, place le résultat dans le registre R0
MOV R1, #23	Place le nombre 23 dans le registre R1
MOV R0, R3	Place la valeur stockée dans le registre R3 dans le registre R0
HALT	Symbole de fin de programme, indispensable pour que le programme se termine sans erreur

Exercice 6

- Ouvrir le simulateur AQUA depuis le lien sur le bureau ou le Web : <http://www.peterhigginson.co.uk/AQA/>.
- Saisir le programme ci-dessous dans la fenêtre d'édition puis le soumettre avec submit.

```
1 MOV R0, #10
2 LDR R1, 10
3 ADD R2, R1, R0
4 STR R2, 11
5 HALT
```

3. Assembler le programme avec `assemble` et modifier le mot mémoire d'adresse 10 en lui donnant la valeur 12. Sélectionner ensuite l'affichage binaire.
4. A quoi correspond le mot de 32 bits contenu en mémoire à l'adresse 0 : 11100011 10100000 00000000 00001010?
5. Repérer les mots mémoires de 32 bits stockant le programme et le mot mémoire stockant la donnée 12.
6. Sélectionner l'affichage `Unsigned`. Exécuter le programme pas à pas (`step`) en vitesse lente (`options` puis `def slow`).

Décrire l'enchaînement d'opérations élémentaires lors de l'exécution des instructions de transfert de mémoire `MOV R0, #10` puis `LDR R1, 10`. Observer l'évolution des registres PC (Compteur de programme), CIR (Registre d'instructions), MAR (adresse d'écriture/lecture en mémoire) et MBR (donnée à lire/écrire). Pour quelle(s) instruction(s), l'ALU est-elle sollicitée?

Exercice 7

On considère le programme en assembleur ci-dessous.
Les commentaires sont précédés des caractères `//`.

1. Décrire la modification de l'état de la mémoire (registre et mémoire centrale) provoquée par la séquence d'instruction d'*initialisation* des lignes 2 à 6.
2. Décrire la modification de l'état de la mémoire (registre et mémoire centrale) provoquée par la séquence d'instruction de *itération 1* des lignes 8 à 11.
3. Où sont stockées dans la mémoire centrale les quatre valeurs calculées par ce programme? Il s'agit des premières valeurs d'une suite célèbre, laquelle? Quelle structure algorithmique serait nécessaire pour calculer les termes suivants sans copier-coller?
4. Rajouter les calculs de deux termes supplémentaires de la suite, par copier-coller des lignes 23 à 26, puis exécuter le programme dans le simulateur. Observer l'état de la mémoire, expliquer l'erreur signalée par l'Unité de Contrôle et corriger le programme.

```
1 //initialisation
2 MOV R0, #25
3 MOV R1, #1
4 STR R1, [R0]
5 ADD R0, R0, #1
6 MOV R2, #1
7 //itération 1
8 STR R2, [R0]
9 ADD R2, R2, R1
10 LDR R1, [R0]
11 ADD R0, R0, #1
12 //itération 2
```

```

13 STR R2, [R0]
14 ADD R2, R2, R1
15 LDR R1, [R0]
16 ADD R0, R0, #1
17 //itération 3
18 STR R2, [R0]
19 ADD R2, R2, R1
20 LDR R1, [R0]
21 ADD R0, R0, #1
22 //itération 4
23 STR R2, [R0]
24 ADD R2, R2, R1
25 LDR R1, [R0]
26 ADD R0, R0, #1
27 //fin
28 HALT
    
```

Exercice 8

On considère le programme Python ci-dessous

```

a = 42      #valeur 42 à l'adresse 20 en mémoire centrale
b = 69      #valeur 69 à l'adresse 21 en mémoire centrale
a = a + b   #changement de valeur à l'adresse 20
b = a - b   #changement de valeur à l'adresse 21
a = a - b   #changement de valeur à l'adresse 20
    
```

1. Déterminer le contenu des variables a et b à la fin de l'exécution de ce programme Python.
2. Traduire ce programme en assembleur et le tester dans le simulateur.



En assembleur, les identifiants de variables sont remplacés par des adresses en mémoire centrale et les opérations arithmétiques ne sont effectuées que sur des registres, il faut donc d'abord transférer les opérandes de la mémoire centrale vers des registres.

3.3 Programmer une instruction conditionnelle en assembleur

Méthode

Dans le programme assembleur ci-dessous, on introduit de nouveaux symboles :

- 🔊 INP R1, 2 est une **instruction d'entrée**, qui lit un entier saisi dans le champ Input et le charge dans le registre R1.
- 🔊 OUT R1, 4 est une **instruction de sortie**, qui affiche le contenu du registre R1 dans le champ Output.
- 🔊 else: et fin: sont des **étiquettes** qui jouent le rôle de repères / points de chute, dans les instructions de branchement / saut. Une étiquette est un mot suivi du symbole colonne :
- 🔊 CMP R0, #0 est une instruction de comparaison qui compare le contenu du registre R0 au nombre 0. Elle est suivie d'une instruction de **branchement (ou saut) conditionnel** BLT else: le programme se poursuit soit à partir de l'étiquette else si R0 est plus petit que 0, sinon avec l'instruction de la ligne suivante (comportement par défaut).

☞ B fin est une instruction de **branchement / saut inconditionnel** : le programme se poursuit à partir de l'étiquette fin, le flux normal (passage à la ligne suivante) est interrompu.

Pour bien comprendre, le fonctionnement des instructions de branchement, exécuter le programme dans le simulateur en mode pas à pas, avec une vitesse lente au niveau des instructions BLT else et B fin. Effectuer un test avec une valeur positive 4 et l'autre avec une valeur négative -4.

Noter que le **Compteur de Programme PC** est incrémenté par défaut de 1 pour chaque instruction mais qu'il peut être de plus modifié par une instruction de branchement.

```
1 //Lecture d'un entier dans Input et chargement dans le registre R0
2     INP R0, 2
3 //Comparaison du registre R0 avec le nombre 0
4     CMP R0, #0
5 //Branchement conditionnel sur l'étiquette else si R0 négatif
6     BLT else
7     MOV R1, R0
8 //Branchement inconditionnel sur l'étiquette fin
9     B fin
10 //étiquette else
11 else:
12     MOV R2, #0
13     SUB R1, R2, R0
14 //étiquette fin
15 fin:
16 //affichage du registre R1 dans Output
17     OUT R1, 4
18     HALT
```

Exercice 9

On considère le programme Python ci-dessous

```
a = int(input()) #entier lu stocké dans le registre R0
b = int(input()) #entier lu stocké dans le registre R1
if a > b:
    m = a
else:
    m = b
#le maximum m de a et b est stocké dans le registre R2
#et en mémoire centrale à l'adresse 20
print(m)
```

Traduire ce programme en assembleur puis le tester dans le simulateur.

3.4 Programmer une boucle en assembleur

Méthode

Dans le simulateur AQUA, sélectionner puis exécuter le programme ascii en mode pas à pas. Observer l'évolution du **Compteur de Programme PC** lors de chaque exécution du branchement conditionnel BLT LOOP.

```
1 //initialise le registre R2 avec le nombre 32
2     MOV R2,#32
3 LOOP:
4     //affiche dans Output le caractère dont le code ascii est contenu dans R2
```



```
5      OUT R2,7
6      //incrémente R2
7      ADD R2,R2,#1
8      //compare R2 avec 127
9      CMP R2,#127
10     //si R2 < 127 branchement conditionnel sur l'étiquette loop
11     BLT LOOP
12     //sinon le programme se poursuit
13     MOV R2,#10
14     //affichage du caractère de code ascii 10 qui est un saut de ligne
15     OUT R2,7
16     HALT
```

Ce programme permet d'afficher tous les caractères dont le code ascii est compris entre 32 et 126, par ordre croissant du code. C'est une implémentation de boucle while en assembleur, une traduction en Python pourrait être :

```
code_ascii = 32
while code_ascii < 127:
    print(chr(code_ascii), end='')
    code_ascii = code_ascii + 1
print()
```

Exercice 10

1. Modifier le programme ascii pour qu'il affiche tous les caractères dont le code ascii est compris entre 126 et 32 dans l'ordre décroissant du code.
2. Modifier le programme de l'exercice 7, pour qu'il stocke en mémoire à partir de l'adresse 20, les 30 premiers termes de cette suite célèbre.
3. Traduire en assembleur le programme Python ci-dessous. On peut utiliser uniquement des registres.

```
s = 0
k = 1
while k <= 100:
    s = s + k
    k = k + 1
print(s)
```

4. Traduire en assembleur le programme Python ci-dessous. On peut utiliser uniquement des registres.



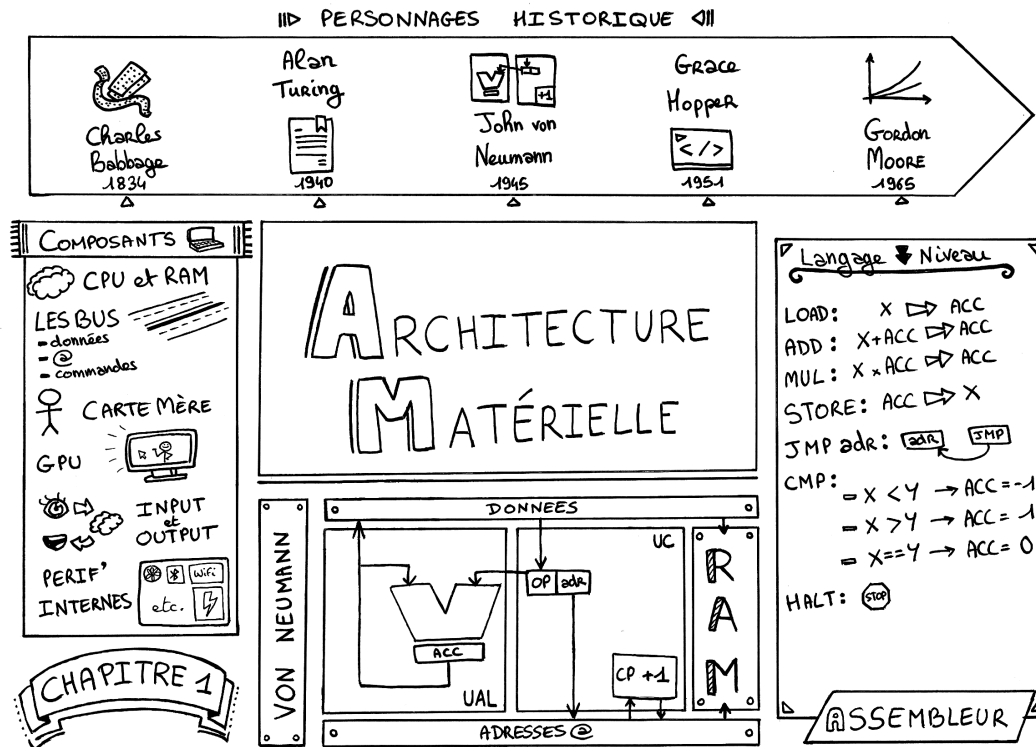
Le langage d'assembleur du simulateur AQUA ne dispose pas d'instruction pour multiplier deux nombres.

```
a = int(input())
b = int(input())
c = a * b
print(c)
```

5. Traduire en assembleur le programme Python ci-dessous. On peut utiliser uniquement des registres.

```
code_ascii = 32
while code_ascii < 127:
    i = 0
    while i < 10:
```

```
#affichage du caractère sans saut de ligne
print(chr(code_ascii), end='')
code_ascii = code_ascii + 1
print() #saut de ligne
```



Sketchnote réalisée par prof2NSI licence <http://snt-nsi.fr/licences>

Table des matières

1 Histoire des ordinateurs	1
1.1 Machines mécaniques	1
1.2 Fondements théoriques de l'informatique	1
1.3 Machines à programmes externes	1
1.4 L'ordinateur, une machine universelle à programme enregistré	2
1.5 Miniaturisation et démocratisation	2
2 Architecture de Von Neumann	3
2.1 Un processeur pour calculer et une mémoire pour stocker programme et données	3
2.2 Hiérarchie des mémoires	8
3 Langage machine et assembleur	9
3.1 Du texte d'un langage de programmation aux instructions du langage machine	9
3.2 Premiers programmes en assembleur avec le simulateur Aqua	11
3.3 Programmer une instruction conditionnelle en assembleur	15
3.4 Programmer une boucle en assembleur	16