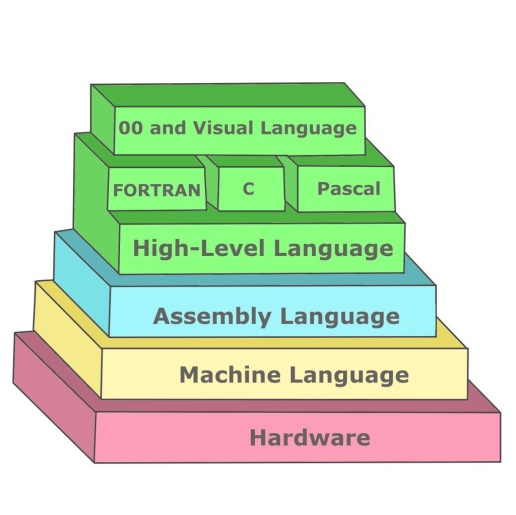
|  |  |  |
| --- | --- | --- |
| Numérique et Sciences Informatiques | | |
| 2h | **Langage machine et assembleur** |  |
| Objectif : Dérouler l'exécution d'une séquence d'instructions en langage machine. | | |
| **Matériel :** PC + simulateur RISC 16 bits | | |

**Un processeur effectue des actions à partir d’instructions simples en binaire, qui peuvent être directement traduites dans un langage assembleur.**

**Différence entre langage de haut et de bas niveau**

  
Un programme informatique écrit dans un langage de **haut niveau** (proche du langage **humain** mais éloigné du langage machine) dépend le moins possible du **type** de **processeur** (Intel core i5, AMD ryzen, Atmel 328P ...) etde **système d'exploitation (**Windows 10, ios 13, android 9 ...) utilisés.

Le programme de haut niveau est donc transportable sur n'importe quelle type de machine.

**Langage de haut niveau** : Python, C, Java, PHP ...

**Langage de bas niveau** : langage assembleur et langage machine de chaque type de processeur.

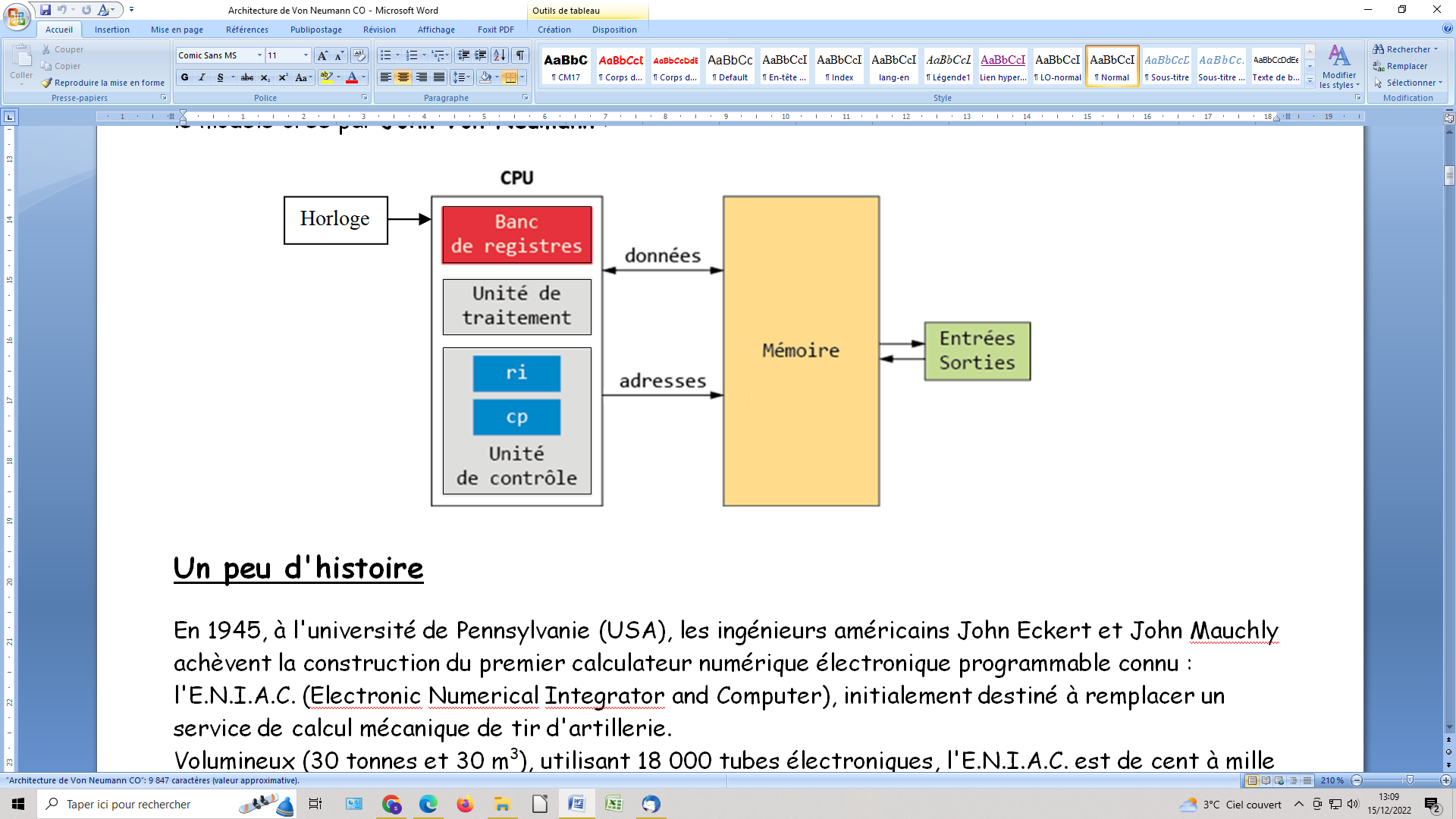
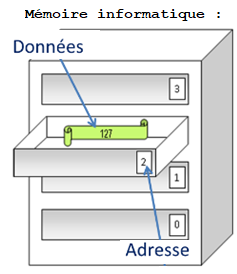
**Langage machine** : codes binaires (=instructions) spécifiques à un processeur.

**Langage assembleur** : c'est le langage machine où les codes sont remplacés par des instructions élémentaires.

Un **compilateur (**ou un **interpréteur**) permet de passer du langage de **haut niveau** a son équivalent en **langage machine** (code binaire).

Chaque **instruction** d'un **langage assembleur** contient un code correspondant à l'**opération** à effectuer et aux **opérandes** (données qui entrent dans une opération).

Architecture de **John Von Neumann** :



**Une opération peut être :**

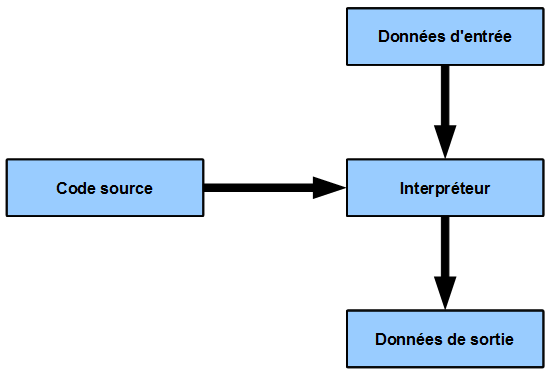
* Un **transfert** entre les registres (cases mémoires du processeur) et la mémoire.
* Un **calcul** arithmétique (addition, comparaison ...) ou logique (AND, OR ...).
* Un **branchement** via des sauts vers une certaine adresse selon le résultat de l'opération précédente (branchement conditionnel) ou non (inconditionnel).

Les **langages assembleurs** permettent de :

* Gérer jusqu'au moindre octet de mémoire et de toujours savoir quel code est exécuté par le microprocesseur à un instant donné.
* Créer des programmes de plus faible volume que s'ils avaient été écrits en langages de haut niveau.
* Etre plus rapides que les langages de haut niveau.

**Langage de haut niveau compilé et langage de haut niveau interprété**

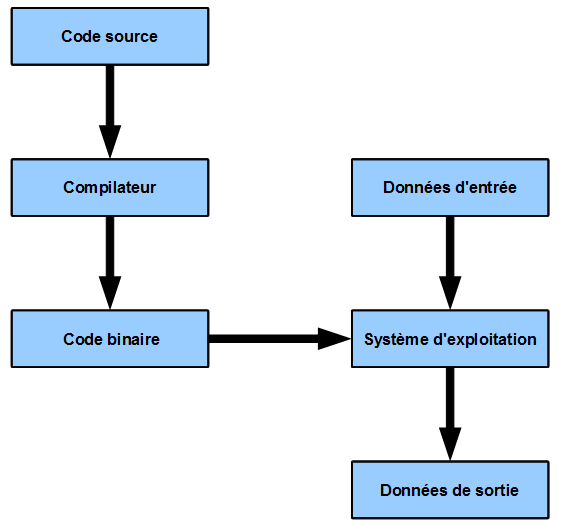
Langages de haut niveau **interprétés** : Python, PHP, JavaScript ...

Langages de haut niveau **compilés** : C, C++, Pascal ...

NB : Java est un langage compilé et interprété à la fois.

Avec les langages **interprétés**, le **code source** (celui que vous écrivez) est interprété par un logiciel qu'on appelle interpréteur (Python3).

L'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite.

Avec les langages **compilés**, le **code source** est tout d'abord compilé en un **code binaire** puis il est exécuté par la machine.

Avec un **langage interprété**, le même code source pourra fonctionner directement sur toutes les machines ayant le bon interpréteur.

Avec un **langage compilé**, il est nécessaire de compiler le programme pour chaque type de machine (il faut avoir le bon compilateur).

Un programme en langage compilé est directement exécuté sur la machine, donc il sera plus rapide qu’un programme en langage interprété.

**Vidéo (4min05) pour comprendre la différence entre l'interpréteur et le compilateur :**

https://www.youtube.com/watch?v=4lXp\_89c3RU

**Programmation en assembleur**

Nous allons travailler avec un assembleur **inventé** (il ressemble beaucoup aux premiers assembleurs).

Aide sur le jeu d'instructions : <http://www.peterhigginson.co.uk/RISC/instruction_set.pdf>

Le programme **assembleur suivant** est composé de 5 instructions :

|  |  |
| --- | --- |
| **INP R0,2** | L'utilisateur saisit un chiffre qui sera stocké dans le registre R0 |
| **INP R1,2** | L'utilisateur saisit un chiffre qui sera stocké dans le registre R1 |
| **ADD R2,R1,R0** | Valeur registre R2 = valeur registre R0 + valeur registre R1 |
| **OUT R2,4** | Affichage du contenu du registre R2 |
| **HLT** | Fin du programme |

Chaque instruction est codée sur **16 bits** en langage machine.

La première instruction :

|  |  |  |
| --- | --- | --- |
| **Opération :** | **Registre :** | **Opérande :** |
| **INP** | **R0** | **2** |
| 0111 0001 0 | 000 | 0010 |

On peut remarquer qu'on a :

* 9 bits pour indiquer que l'on souhaite réaliser une instruction INP (INPut : saisie clavier par l'utilisateur).
* 3 bits pour préciser le numéro du registre R0 (il y a 8 registres généraux).
* 4 bits pour donner l'adresse d'un périphérique (2 : adresse du clavier).
* En consultant l'aide sur les instructions, complétez les tableaux suivants :

|  |  |  |
| --- | --- | --- |
| **INP** | **R1** | **2** |
| 0111 0001 0 | 001 | 0010 |

|  |  |  |  |
| --- | --- | --- | --- |
| **ADD** | **R2** | **R1** | **R0** |
| 0110 000 | 010 | 001 | 000 |

|  |  |  |
| --- | --- | --- |
| **OUT** | **R2** | **4** |
| 0111 0001 1 | 010 | 0100 |

|  |  |
| --- | --- |
| **HLT** | **Complétez avec des 0** |
| 0000 0 | 000 0000 0000 |

* Complétez le tableau suivant :

|  |  |  |
| --- | --- | --- |
| **Langage assembleur** | **Langage Machine** (en binaire) | **Langage Machine** (en hexadécimal) |
| **INP R0,2** | 0111 0001 0000 0010 | 7102 |
| **INP R1,2** | 0111 0001 0001 0010 | 7112 |
| **ADD R2,R1,R0** | 0110 0000 1000 1000 | 6088 |
| **OUT R2,4** | 0111 0001 1010 0100 | 71A4 |
| **HLT** | 0000 0 000 0000 0000 | 0000 |

Allez sur le simulateur Risc 16 bits de Peter Higginson : <http://peterhigginson.co.uk/RISC/>

*Le simulateur RISC travaille sur des entiers codés sur 16 bits (nombres signés entre -32768 et 32767 ou non-signés entre 0 et 65535).*

*Il ne permet pas de manipuler des nombres à virgule flottante (réels).*

Dans la liste déroulante **OPTIONS** en bas de page, sélectionnez :

* **hex** pour d'afficher toutes les valeurs en hexadécimal.
* **def slow** pour limiter la vitesse de déplacement des informations.

Dans la liste déroulante **SELECT**, sélectionnez le programme **add**.

Le **langage machine** correspondant aux lignes du code assembleur du programme **add** sont stockées en mémoire (**Main Memory**), vérifiez que les codes machine correspondent à ceux que vous avez trouvé.

* Simulez le programme (**RUN**) avec **Input=3** puis **Input=5** et complétez le tableau suivant décrivant le déroulement des opérations (Input représente le clavier du PC et Output son écran) :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PC** | **R0** | **R1** | **R2** | **Input** | **Output** |
| 0 | 3 | 0 | 0 | 3 | 0 |
| 1 | 3 | 5 | 0 | 5 | 0 |
| 2 | 3 | 5 | 8 | 0 | 0 |
| 3 | 3 | 5 | 8 | 0 | 8 |
| 4 | 3 | 5 | 8 | 0 | 0 |

**Modification du programme assembleur**

Nous souhaitons modifier le programme **add** pour réaliser une **soustraction** à la place de d'addition.

* Modifiez le programme et testez-le.

Nous souhaitons maintenant réaliser une **addition** entre trois nombres rentrés par l'utilisateur.

Dans la liste déroulante **OPTIONS**, sélectionnez **def fast** pour avoir une vitesse rapide de simulation.

* Réalisez une première version de ce programme puis testez-le dans le simulateur.
* Réalisez une seconde version en utilisant seulement 2 registres (si ce n’était pas le cas dans votre première version) puis testez-le dans le simulateur.

**Faire une condition en assembleur**

Exemple de condition (Python) :

**R0=int(input())**

**R1=int(input())**

**R2=R0+R1**

**if (R2==8):**

**print(99)**

**else:**

**print(11)**

* Traduisez ce code Python en assembleur puis testez-le dans le simulateur.

Instructions assembleur utiles :

|  |  |
| --- | --- |
| **MOV R0,#0** | R0 ← 0 |
| **CMP R0,#8**  **BEQ SAUT**  **SAUT ...** | Comparaison de valeur registre R0 avec 8  **BEQ** : si R0=8 alors sauter à l'étiquette SAUT  **BEQ** est un branchement conditionnel |
| **BRA SAUT**  **SAUT ...** | Sauter à l'étiquette SAUT  **BRA** est un branchement inconditionnel |

**Faire une boucle en assembleur**

Exemple de boucle (Python) exécutée 5 fois :

**n=0**

**while n<5:**

**n=n+1**

* Traduisez ce code Python en assembleur puis testez-le dans le simulateur.

Instructions assembleur utiles :

|  |  |
| --- | --- |
| **CMP R0,#10**  **BLT SAUT**  **SAUT ...** | Comparaison de la valeur registre R0 avec 10  **BLT** : Si R0<10 alors sauter à l'étiquette SAUT  **BLT** est un branchement conditionnel |

**Pour les élèves en avance : Calcul d'une moyenne en assembleur**

Exemple avec l’algorithme (pseudo-code) qui calcule la moyenne d’une suite de nombres positifs donnés par l’utilisateur. La lecture des nombres se termine lorsque l’utilisateur rentre un nombre négatif.

**s  0**

**n  0**

**Lire x**

**Tant que x  0 faire**

**s  s + x**

**n  n + 1**

**Lire x**

**Fin tant que**

**m  s/n**

**Afficher m**

* Traduisez cet algorithme en assembleur puis testez-le dans le simulateur.

Instruction assembleur utile :

|  |  |
| --- | --- |
| **DIV R1,R0** | R1 ← R1/R0 |

|  |  |
| --- | --- |
| **Programme en C arduino d'une addition :** | **Résultat de la compilation :** |
|  |  |