14

Modèle d'architecture séquentielle

Extrait du programme

Thème : Architectures matérielles et systèmes d'exploitation

Contenus :

Modèle d'architecture séquentielle (von Neumann)

Capacités attendus :

Distinguer les rôles et les caractéristiques des différents constituants d'une machine.

Dérouler l'exécution d'une séquence d'instructions simples du type langage machine.

Commentaires :

La présentation se limite aux concepts généraux.

On distingue les architectures monoprocesseur et les architectures multiprocesseur.

Des activités débranchées sont proposées.

Les circuits combinatoires réalisent des fonctions booléennes.

Ce cours est très fortement inspiré du cours de David Roche :
`https://pixees.fr/informatiquelycee/n_site/nsi_prem_von_neu.html`
`https://pixees.fr/informatiquelycee/n_site/nsi_prem_sim_cpu.html`

I. Les exposés en lien avec ce cours

John von Neumann (28 décembre 1903 - 8 février 1957)

Grace Hopper (9 décembre 1906 - 1er janvier 1992)

Sources Photos Wikipédia.
Une référence intéressante :
`https://clg-lesfontaines-bouillargues.ac-montpellier.fr/sites/clg-lesfontaines-bouillargues/files/atelier_info/levolution_de_lordinateur.pdf`

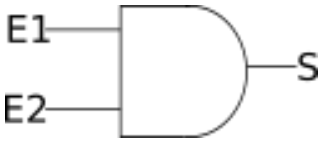
II. Introduction

Nous allons voir ici comment fonctionne une « machine » informatique. Pour bien poser les choses, il faut prendre conscience que ce que nous appellerons machine peut correspondre à une grande gamme d'appareils informatiques : un ordinateur personnel bien sûr, mais aussi smartphone, ordinateur de bord de voiture, robot ou objet connecté.
En simplifiant, tous ces appareils ont plusieurs points communs :

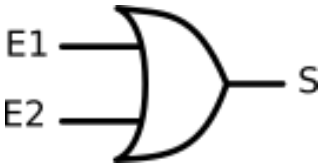
- ils reçoivent des signaux en entrée : clavier, souris, écran tactile, ondes radio, signaux électriques transmis par câble, signaux provenant de capteurs (vitesse, température, etc.)
- ils disposent d'un processeur qui effectue un traitement sur les signaux reçus en entrée
- ils produisent des signaux en sortie : sur un écran, sur une sortie audio, sur une carte réseau, sur une interface dédiée, à des moteurs électriques etc.

Le modèle qui a été fait de ces machines informatiques : le modèle d'architecture séquentielle est un modèle universel. Ce modèle d'architecture a été conçu durant la seconde guerre mondiale à des fins initialement militaires et a été finalisé par John Von Neumann en 1945. Ce modèle est simpliste au regard de la complexité et de la diversité des machines actuelles, néanmoins il permet de mieux appréhender comment fonctionne une machine informatique en la présentant sous une version simplifiée.

L'informatique a réellement pris son essor grâce à l'invention et l'utilisation massive du transistor. Pour simplifier le transistor est un interrupteur commandé, il a donc deux états 0 ou 1. En les combinant entre eux on réalise des opérateurs logiques (ET, OU, NON, XOR) et des mémoires (bascule). En combinant les opérateurs logiques entre eux on réalise des additionneurs, soustracteurs, multiplicateurs... et bien d'autres encore. Ces « boites » élémentaires sont câblées à l'intérieur des processeurs.



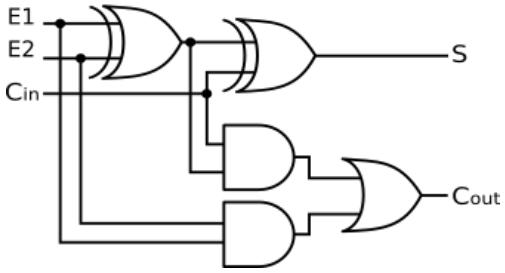
Porte AND



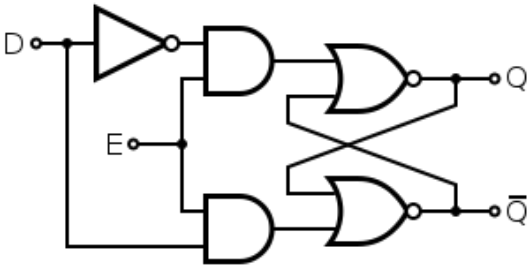
Porte OR

Porte NOT

Porte XOR



Additionneur



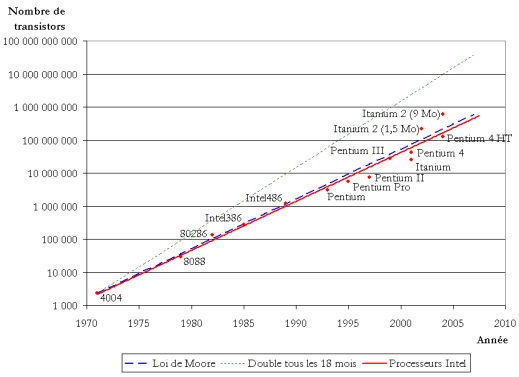
Circuit de type bascule, permet de stocker 1 bit

Au fil du temps les processeurs n'ont cessé de compter de plus en plus de transistors qui leur confèrent de plus en plus de puissance de calcul et des mémoires de capacité toujours plus gigantesque. Le tableau ci-dessous montre cette évolution.

Année	Nom	Nombre de transistors	Fréquence / Capacité
1972	Intel 4004	2 300	740 kHz
1982	Intel 80286	134 000	6 MHz
1993	Intel Pentium	3 100 000	66 MHz
2003	AMD Athlon 64	105 900 000	3,2 GHz
2018	SDRAM Samsung	137 000 000	128GB
2019	AMD Threadripper 2990WX	19 200 000 000	3 GHz
2019	Flash Samsung	2 048 000 000	8 TB

Pour autant, l'évolution des processeurs est confronté A deux problèmes majeurs :

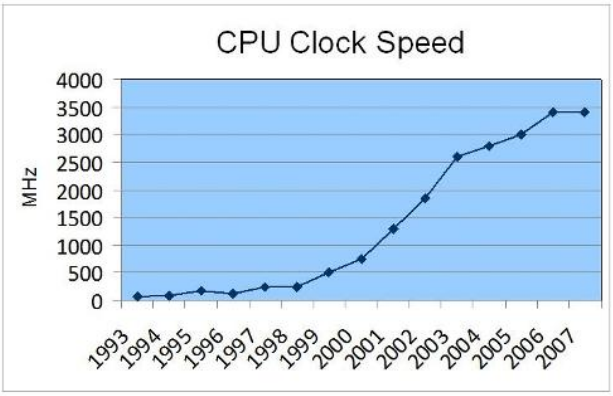
- La loi de Moore.
  La loi de Moore a été exprimée en 1965 dans le magazine Electronics (en) par Gordon E. Moore, ingénieur de Fairchild Semiconductor, un des trois fondateurs d'Intel. Constatant que la « complexité des semi-conducteurs proposés en entrée de gamme » doublait tous les ans à coût constant depuis 1959, date de leur invention, il postulait la poursuite de cette croissance (en 1965, le circuit le plus performant comportait 64 transistors). Cette augmentation exponentielle fut rapidement nommée « loi de Moore » ou, compte tenu de l'ajustement ultérieur, « première loi de Moore ».
  En 1975, Moore réévalua sa prédiction en posant que le nombre de transistors des microprocesseurs (et non plus de simples circuits intégrés moins complexes) sur une puce de silicium double tous les deux ans. Bien qu'il ne s'agisse pas d'une loi physique mais seulement d'une extrapolation empirique, cette prédiction s'est révélée étonnamment exacte. Entre 1971 et 2001, la densité des transistors a doublé chaque 1,96 année. En conséquence, les machines électroniques sont devenues de plus en plus petites et de moins en moins coûteuses tout en devenant de plus en plus rapides et puissantes.
  Source Wikipédia : `https://fr.wikipedia.org/wiki/Loi_de_Moore`.



Loi de Moore : Source Wikipédia

Le seuil de réduction des dimensions toujours nécessaire pour augmenter les performances sera atteint d'ici quelques années (seuil de gravure de 5 nm atteint en 2019, prévu à 3nm en 2022). D'autres approches technologiques que la réduction des dimensions devront donc être envisagées pour continuer à doubler la puissance des processeurs tous les 2 environs.
Article de *developpez.com* datant du 6 Mars 2020 :
`https://www.developpez.com/actu/296076/Sommes-nous-prepares-a-la-fin-de-la-loi-de-Moore-Elle-a-`

- La fréquence.
  Pendant des années, pour augmenter les performances des ordinateurs, les constructeurs augmentaient la fréquence d'horloge des microprocesseurs : la fréquence d'horloge d'un microprocesseur est liée à sa capacité d'exécuter un nombre plus ou moins important d'instructions machines par seconde. Plus la fréquence d'horloge du microprocesseur est élevée, plus il est capable d'exécuter un grand nombre d'instructions machines par seconde (en fait, c'est un peu plus compliqué que cela, mais nous nous contenterons de cette explication).



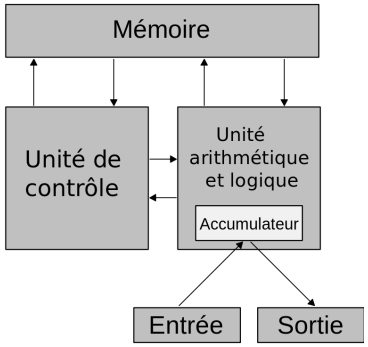
Fréquence du microprocesseur :
Source `https://pixees.fr/informatiquelycee/n_site/nsi_prem_von_neu.html`

A partir de 2006 environ, la fréquence d'horloge a cessé d'augmenter, pourquoi ? A cause d'une contrainte physique : en effet plus on augmente la fréquence d'horloge d'un microprocesseur, plus ce dernier chauffe. Il devenait difficile de refroidir le CPU. Les constructeurs (principalement Intel et AMD) ont décidé d'arrêter la course à l'augmentation de la fréquence d'horloge en adoptant une nouvelle tactique.
Augmentons le nombre de cœurs présent sur un microprocesseur ! La technologie permettant de graver toujours plus de transistors sur une surface donnée, il est donc possible, sur une même puce, d'avoir plusieurs cœurs, alors qu'auparavant on en trouvait qu'un seul. Cette technologie a été implémentée dans les ordinateurs grand public à partir de 2006. Aujourd'hui on trouve sur le marché des CPU possédant jusqu'à 18 cœurs !
Mais qu'est qu'un cœur ? C'est ce que nous allons voir dans la suite de ce cours.

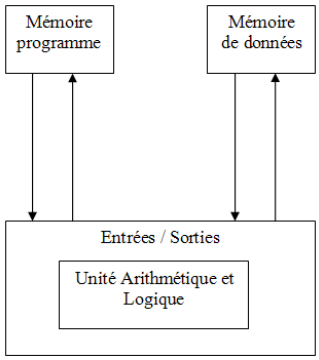
III. L'architecture de Von Neumann

Le fait d'attribuer à **Von Neumann**[1] la paternité de ce modèle fait débat chez les scientifiques dont certains préfèrent le nom d'**architecture de Princeton**.
Ce modèle n'est pas unique. Il en existe par exemple un dit **de Harvard**[2] dont la mémoire est séparée en deux parties distinctes l'une pour les données l'autre pour le programmes. Cette architecture peut se montrer plus efficace à technologie identique mais au prix d'une complexité accrue de la structure. Elle est, par exemple, utilisée pour les applications de traitement du signal (processeur DSP pour Digital Signal Processor).

1. Architecture de Von Neumann : `https://fr.wikipedia.org/wiki/Architecture_de_von_Neumann`.
2. Architecture de Harvard : `https://fr.wikipedia.org/wiki/Architecture_de_type_Harvard`.



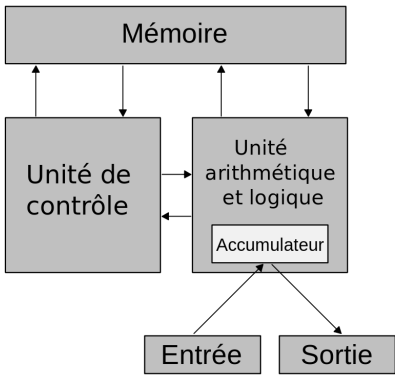
Architecture de Von Neumann



Architecture de Harvard

a) Présentation du modèle séquentiel

Le programme informatique à exécuter est situé dans la mémoire. Auparavant il fallait câbler physiquement chaque programme. Avec un programme stocké en mémoire on gagne en souplesse, en productivité etc. En conséquence directe les instructions (telles que mémoriser, additionner, comparer ...) sont représentées exactement comme les autres données (valeurs ou adresses), typiquement par des octets (ou des mots de 16, 32 ou 64 bits).
Le mécanisme, cadencé par une horloge, qui s'occupe de lire les instructions dans la mémoire les unes après les autres de façon séquentielle est séparé de la partie de la machine qui exécute ces instructions.
Ce mécanisme qui gère le déroulement du programme doit être capable si besoin de sauter une ou plusieurs instructions, voire de revenir en arrière dans la séquence d'instructions. En effet lorsqu'on fait un `if ... then ... else ...` , on ne prend pas en compte certaines instructions. De façon similaire les boucles requièrent de revenir en arrière dans le programme. Sans rentrer dans le détail, la partie d'une machine informatique qui s'occupe de la lecture séquentielle des instructions s'appelle « **Unité de contrôle** » alors que la partie qui s'occupe d'exécuter ces instructions s'appelle « **Unité arithmétique et Logique** ».
Nous pouvons maintenant présenter le modèle d'architecture séquentielle de Von Neumann donnée par la figure ci-dessous :



Architecture de Von Neumann

On peut distinguer trois constituants pour un processeur.

- L'**unité arithmétique et logique** (UAL ou ALU en anglais).
  Elle est chargée des calculs.
  On y retrouve par exemple des circuits du type de l'additionneur.
- L'**unité de contrôle**.
  Elle est chargée de lire et décoder les instructions en mémoire.
- Les **registres** ou la mémoire.
  Ils sont chargés de mémoriser de l'information (instructions ou données).

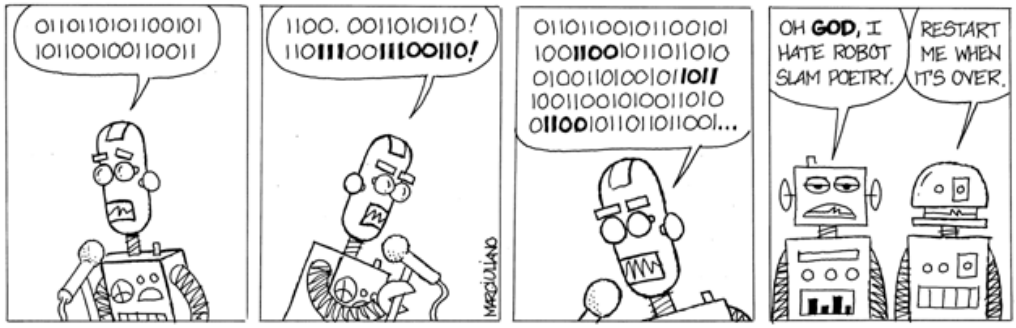
Les mémoires se décomposent en deux catégories suivant l'usage que l'on veut en faire :

- Les **ROM** pour **Read Only Memory**.
  On y stocke les informations nécessaires au démarrage d'un ordinateur, elles contiennent le BIOS, les instructions de démarrage, ... Elles travaillent uniquement en lecture et l'information et conservée une fois le circuit hors énergie. On retrouve différentes technologies PROM, EEPROM, UVPROM.
- Les **RAM** pour **Random Access Memory**.
  On y stocke les programmes et les données, elles travaillent en lecture écriture R/W. Celles des trois premières couches (CPU, cache et mémoire physique) sont volatiles ce qui veut dire que l'information est perdue à la coupure de l'énergie. Les deux couches suivantes sont des mémoires de masse qui conservent programmes et données sur le long termes.

Donc dans un microprocesseur, un cœur est donc principalement composé d'une UAL, de registres (R0, R1...) et d'une unité de commande. Un cœur est donc capable d'exécuter des programmes de façon autonome. La technologie permettant de graver toujours plus de transistors sur une surface donnée, il est donc possible, sur une même puce, d'avoir plusieurs cœurs, alors qu'auparavant on en trouvait qu'un seul. Cette technologie a été implémentée dans les ordinateurs grand public à partir de 2006. Aujourd'hui on trouve sur le marché des CPU possédant jusqu'A 18 cœurs !
On pourrait se dire que l'augmentation du nombre de cœurs entraîne obligatoirement une augmentation des performances du processeur, en faite, c'est plus complexe que cela : pour une application qui n'aura pas été conçue pour fonctionner avec un microprocesseur multicœur, le gain de performance sera très faible, voir même nul. En effet, la conception d'applications capables de tirer profit d'un CPU multicœur demande la mise en place de certaines techniques de programmation (techniques de programmation qui ne seront pas abordées ici). Il faut aussi avoir conscience que les différents cœurs d'un CPU doivent se partager l'accès à la mémoire vive : quand un cœur travaille sur une certaine zone de la RAM, cette même zone n'est pas accessible aux autres cœurs, ce qui, bien évidemment va brider les performances.

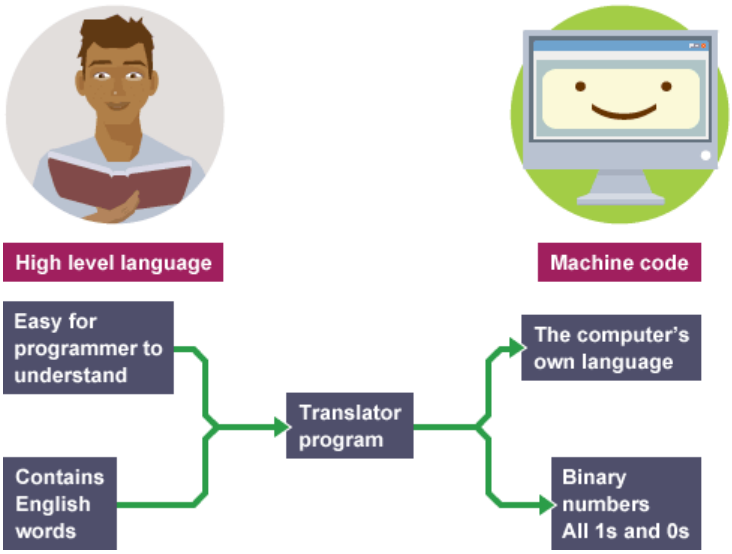
b) Langage Machine, Assembleur, Compilateur

- Le **langage machine** est le langage compris et exécutable par le processeur. C'est une suite de mots binaires (donc composés de 0 et/ou de 1) écrits en mémoire. Chaque type de processeur a son propre langage machine.



- Le **langage assembleur** est le langage de **bas niveau** (langage proche du langage machine) permettant une correspondance (1 pour 1) entre une instruction compréhensible par un humain et la même en langage machine.
  Chaque processeur a son propre langage assembleur.
- L'humain peut écrire en assembleur mais ce langage est assez difficile et fastidieux à écrire (même si certains aiment ça !).
  Dés 1950, Grace Hopper [3] défend l'idée qu'un programme devrait pouvoir être écrit dans un langage proche de l'anglais plutôt que d'être calqué sur le langage machine, comme l'assembleur. Elle crée alors en 1950 le premier **compilateur** [4], c'est-à-dire un programme qui transforme un programme écrit dans un langage non machine en langage machine. Dans le même ordre d'idée, elle crée le langage COBOL [5] en 1959.

3. https://fr.wikipedia.org/wiki/Grace_Hopper
4. A-0 System est le premier compilateur développé pour un ordinateur. Il a été écrit par Grace Hopper en 1951 et 1952 pour l'UNIVAC I. https://fr.wikipedia.org/wiki/A-0_System
5. COBOL est un langage toujours utilisé actuellement, surtout dans les secteurs de la banque, des assurances, des grandes administrations. https://fr.wikipedia.org/wiki/Cobol.
De plus, on recherche énormément de développeurs COBOL : https://www.numerama.com/tech/618709-au-coeur-de-toutes-les-banques-le-lang
html.



- Plus un langage est proche du langage machine, plus il sera de **bas niveau** : l'assembleur est un langage de bas niveau.
  Plus un langage est proche de l'humain, plus il sera de **haut niveau** : Python est un langage de haut niveau.
- En conséquence, pour un même algorithme, plus le langage utilisée est de bas niveau, plus son exécution sera rapide.
  En effet .....
  .....
  Aussi certains reprochent au langage Python sa lenteur et préfèrent des langages influencés par Python comme Cython [6].

6. `https://fr.wikipedia.org/wiki/Cython` ou `https://cython.org/`

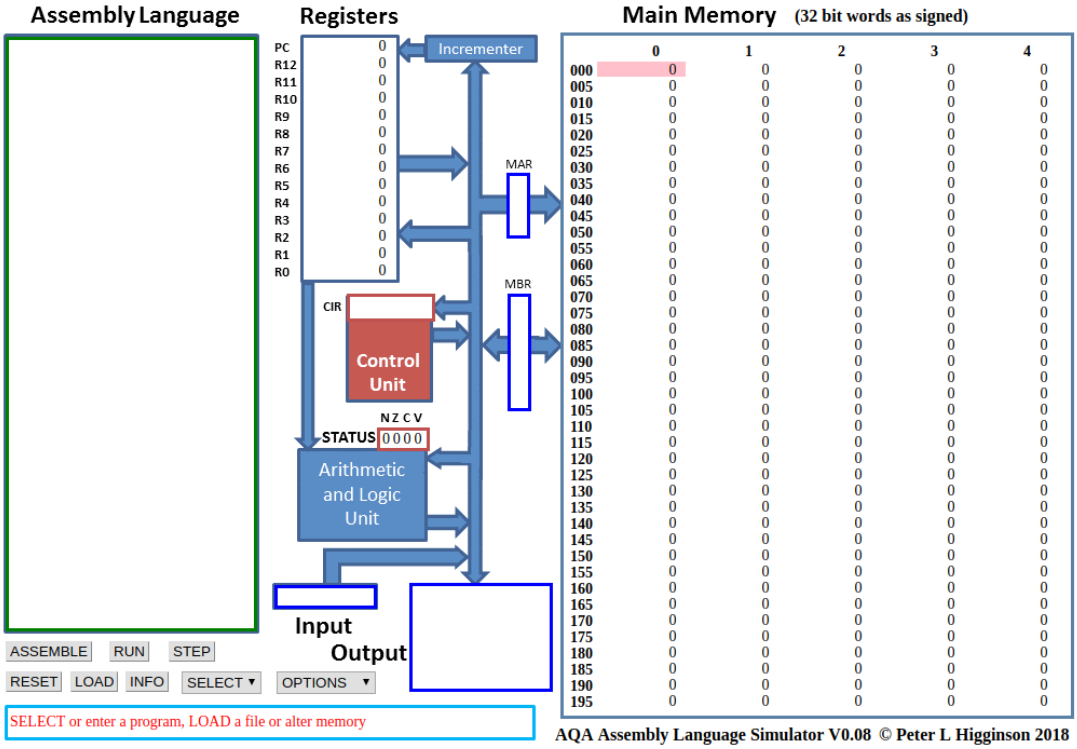
c) Utilisation d'un simulateur

Afin de mettre en pratique ce que nous venons de voir, nous allons utiliser un simulateur développé par Peter L. Higginson. Ce simulateur, basé sur une architecture de von Neumann, simule un processeur de type RISC [7].

Nous allons trouver dans ce simulateur :

- une RAM
- un CPU (pour *Central Processing Unit* en anglais, soit *Unité Centrale de Traitement*, UCT, en français. Autrement dit un processeur)

Une version en ligne de ce simulateur est disponible ici : `http://www.peterhigginson.co.uk/AQA/`
Voici ce que vous devriez obtenir en vous rendant à l'adresse indiquée ci-dessus :



Il est relativement facile de distinguer les différentes parties du simulateur :

- à droite, on trouve la mémoire vive (*main memory*)
- au centre, on trouve le microprocesseur
- à gauche on trouve la zone d'édition (*Assembly Language*), c'est dans cette zone que nous allons saisir nos programmes en Assembleur

Revenons sur les parties RAM et CPU :

- La RAM
  Par défaut le contenu des différentes cellules de la mémoire est en base 10 entier signé, mais d'autres options sont possibles : base 10 entier non-signé (`unsigned`), base 16 (`hex`), base 2 (`binary`). On accède à ces options à l'aide du bouton `OPTIONS` situé en bas dans la partie gauche du simulateur.

7. RISC pour *Reduced Instruction Set Computer* ou *processeur à jeu d'instructions réduit* est un type particulier d'architecture matérielle de processeurs qui se caractérise par un nombre d'instructions de base aisées à décoder, uniquement composé d'instructions simples.
`https://fr.wikipedia.org/wiki/Reduced_instruction_set_computing`
Cette architecture est notamment utilisée de nos jours dans les processeurs ARM conçus par Acorn Computer.
`https://fr.wikipedia.org/wiki/Architecture_ARM`
Ces processeurs sont utilisés notamment sur les cartes micro :bit `https://fr.wikipedia.org/wiki/Micro:bit`
et les nano-ordinateurs Raspberry Pi `https://fr.wikipedia.org/wiki/Raspberry_Pi`.

Exercice 1
A l'aide du bouton `OPTIONS`, passez à un affichage en binaire.
Comme vous pouvez le constater, chaque cellule de la mémoire comporte 32 bits. Chaque cellule de la mémoire possède une adresse (de 000 à 199), ces adresses sont codées en base 10.
Vous pouvez repasser à un affichage en base 10 (bouton `OPTION` puis `signed`).

- Le CPU
  Dans la partie centrale du simulateur, nous allons trouver en allant du haut vers le bas :
  — le bloc **registre** (*Registers*) : nous avons 13 registres (R0 à R12) + 1 registre (PC) qui contient l'adresse mémoire de l'instruction en court d'exécution
  — le bloc **unité de commande** (*Control Unit*) qui contient l'instruction machine en cours d'exécution (au format hexadécimal)
  — le bloc **unité arithmétique et logique** (*Arithmetic and Logic Unit*)
  Nous ne nous intéresserons pas aux autres composants de la partie CPU.

Exercice 2 (Un premier programme en Assembleur)
Dans la partie éditeur (*Assembly Language*) saisissez les lignes de codes suivantes :

```
1 MOV R0,#42
2 STR R0,150
3 HALT
```

language=Assembler
Une fois la saisie terminée, cliquez sur le bouton `submit`. Vous devriez voir apparaître des nombres étranges dans les cellules mémoires d'adresses 000, 001 et 002 :

Main Memory (32 bit words as signed)

	0	1	2	3
000	-476053462	-443612596	-285212672	0
005	0	0	0	0

L'assembleur a fait son travail, il a converti les 3 lignes de notre programme en instructions machines, la première instruction machine est stockée à l'adresse mémoire 000 (elle correspond à `MOV R0,#42` en assembleur), la deuxième à l'adresse 001 (elle correspond à `STR R0,150` en assembleur) et la troisième à l'adresse 002 (elle correspond à `HALT` en assembleur). Pour avoir une idée des véritables instructions machines, vous devez repasser à un affichage en binaire (bouton `OPTION->binary`)). Vous devriez obtenir ceci :

Main Memory (32 bit words as binary)

	0	1	2
000	11100011 10100000 00000000 00101010	11100101 10001111 00000010 01001100	11101111 00000000 00000000 00000000
005	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000

Nous pouvons donc maintenant affirmer que :
- l'instruction machine `11100011 10100000 00000000 00101010` correspond au code assembleur `MOV R0,#42`
- l'instruction machine `11100101 10001111 00000010 01001100` correspond au code assembleur `STR R0,150`
- l'instruction machine `11101111 00000000 00000000 00000000` correspond au code assembleur `HALT`

Au passage, pour l'instruction machine `11100011 10100000 00000000 00101010`, vous pouvez remarquer que l'octet le plus à droite, $(00101010)_2$, est bien égale à $(42)_{10}$!
Repassez à un affichage en base 10 afin de faciliter la lecture des données présentes en mémoire.

Exercice 3

Pour exécuter notre programme, il suffit maintenant de cliquer sur le bouton `RUN`. Vous allez voir le CPU travailler en direct grâce à une petites animations. Si cela va trop vite (ou trop doucement), vous pouvez régler la vitesse de simulation à l'aide des boutons « ou ». Un appui sur le bouton `STOP` met en pause la simulation, si vous réappuyez une deuxième fois sur ce même bouton `STOP`, la simulation reprend là où elle s'était arrêtée.
Une fois la simulation terminée, vous pouvez constater que la cellule mémoire d'adresse 150, contient bien le nombre 42 (en base 10). Vous pouvez aussi constater que le registre R0 a bien stocké le nombre 42.

ATTENTION : pour relancer la simulation, il est nécessaire d'appuyer sur le bouton `RESET` afin de remettre les registres R0 à R12 à 0, ainsi que le registre PC (il faut que l'unité de commande pointe de nouveau sur l'instruction située à l'adresse mémoire 000). La mémoire n'est pas modifiée par un appui sur le bouton `RESET`, pour remettre la mémoire à 0, il faut cliquer sur le bouton `OPTIONS` et choisir `clr memory`. Si vous remettez votre mémoire à 0, il faudra cliquer sur le bouton `ASSEMBLE` avant de pouvoir exécuter de nouveau votre programme.

Quelques explications sur le code :

- `MOV R0,#42` : place le nombre 42 dans le registre R0
- `STR R0,150` : place la valeur stockée dans le registre R0 en mémoire vive à l'adresse 150
- `HALT` : arrête l'exécution du programme

Exercice 4

Modifiez le programme précédent pour qu'à la fin de l'exécution on trouve le nombre 54 à l'adresse mémoire 50. On utilisera le registre R1 à la place du registre R0. Testez vos modifications en exécutant la simulation.

Exercice 5 (Comprendre un programme écrit en assembleur)

Voici quelques autres commandes de base :

- une valeur immédiate est identifiée grâce au symbole #
- `MOV R0,#42` : place le nombre 42 dans le registre R0
- `STR R0,150` : place la valeur stockée dans le registre R0 en mémoire vive à l'adresse 150
- `HALT` : arrête l'exécution du programme
- `LDR R1,78` : place la valeur stockée à l'adresse mémoire 78 dans le registre R1
- `CMP R0, #23` : compare la valeur stockée dans le registre R0 et le nombre 23. Cette instruction CMP doit précéder une instruction de branchement conditionnel BEQ, BNE, BGT, BLT
- `CMP R0, #23`
  `BNE 78` : la prochaine instruction à exécuter se situe à l'adresse mémoire 78 si la valeur stockée dans le registre R0 n'est pas égale à 23
- `B 45` : nous avons une structure de rupture de séquence, la prochaine instruction à exécuter se situe en mémoire vive à l'adresse 45
- `ADD R1,R0,#128` : additionne le nombre 128 et la valeur stockée dans le registre R0, place le résultat dans le registre R1
- les instructions assembleur `B`, `BEQ`, `BNE`, `BGT` et `BLT` n'utilisent pas directement l'adresse mémoire de la prochaine instruction à exécuter, mais des *labels*. Un label correspond à une adresse en mémoire vive (c'est l'assembleur qui fera la traduction *label*→*adresse mémoire*).
  L'utilisation d'un label évite donc d'avoir à manipuler des adresses mémoires en binaire ou en hexadécimale.

Expliquez ce que doit effectuer le programme suivant :

```
CMP R4,#18
    BGT monLabel
    MOV R0,#14
    HALT
monLabel:
    MOV R0,#18
    HALT
```

Exercice 6 (Correspondance Python - Assembleur)
Voici un programme simple, écrit en Python :

```
x = 4
y = 8
if x == 10:
    y = 9
else :
    x=x+1
z=6
```

et voici sa traduction en Assembleur :

```
MOV R0, #4
    STR R0,30
    MOV R0, #8
    STR R0,75
    LDR R0,30
    CMP R0, #10
    BNE else
    MOV R0, #9
    STR R0,75
    B endif
else:
    LDR R0,30
    ADD R0, R0, #1
    STR R0,30
endif:
    MOV R0, #6
    STR R0,23
    HALT
```

Après avoir analysé très attentivement le programme en assembleur ci-dessus, vous essaierez d'établir une correspondance entre les lignes du programme en Python et les lignes du programme en assembleur.
A quoi sert la ligne `B endif` ?
A quoi correspondent les adresses mémoires 23, 75 et 30 ?

Exercice 7
Voici un programme Python :

```
x=0
while x<3:
    x=x+1
```

Écrivez et testez un programme en assembleur équivalent au programme ci-dessus.