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^{ième} 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^{ième} 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. 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, comme le *problème de l'arrêt* : on ne saurait construire un programme général capable de prouver la terminaison de n'importe quel programme

1.3 Machines à programmes externes

Histoire 3

La seconde guerre mondiale accélère la réalisation de machines à calculs pour calculer des trajectoires balistiques ou déchiffrer des codes secrets. L
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.

1.5 Miniaturisation et démocratisation

Histoire 5

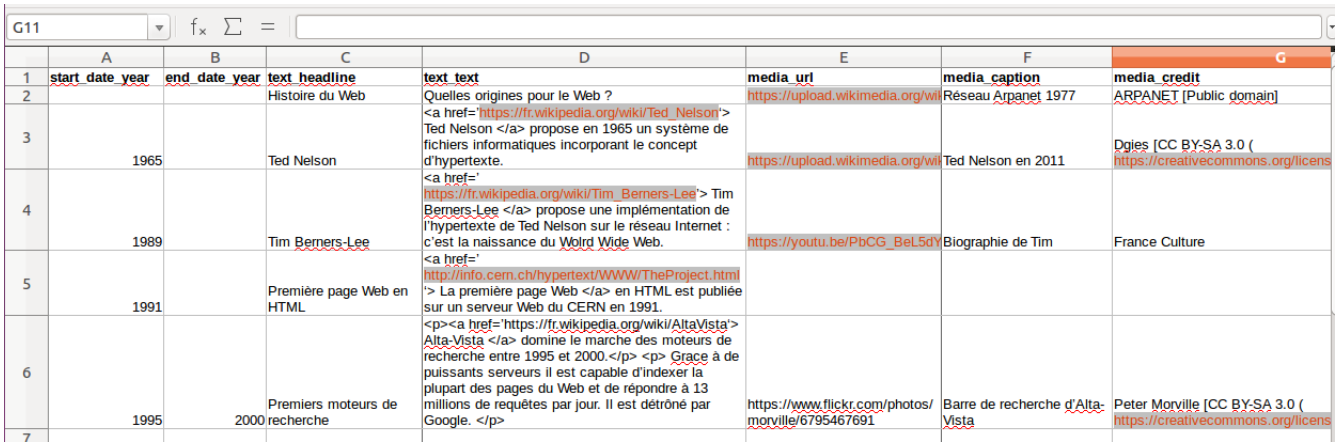
Dans les années 1950, les firmes DEC, BULL et surtout IBM développent les premiers ordinateurs commerciaux et les progrès technologiques s'enchaînent.
Le transistor, inventé en 1947, remplace progressivement les tubes à vide. À partir de la fin des années 1960, la densité de transistors par unité de surface des plaques de silicium constituant les 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 du fondateur d'Intel. La miniaturisation accompagne la progression des capacités de calcul et la démocratisation des ordinateurs. En 1971, l'apparition du microprocesseur Intel 4004, marque les débuts de la micro-informatique. 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 . . .

1.6 Mini projet

Projet 1 ***Frise chronologique***

Ce petit projet a pour but de générer une frise chronologique en HTML/CSS/Javascript sur l'une des cinq périodes décrites précédemment.
Pour le contenu de la frise, il suffit de remplir avec Libreoffice le fichier source `frise_data.ods` avec des informations pertinentes en respectant le format suivant :

- La première ligne est particulière car elle renseigne le titre. Pour chaque ligne, à l'exception de celle du titre, seule la colonne `start_date_year` doit être nécessairement remplie.
- `start_date_year` : année de début de l'événement/période de la diapositive
- `end_date_year` : année de fin de l'événement/période de la diapositive
- `text_headline` : titre de l'événement/période de la diapositive
- `text_text` : texte décrivant l'événement/période de la diapositive
- `media_url` : URL d'une ressource média (image, vidéo) illustrant l'événement/période de la diapositive
- `media_caption` :légende de la ressource média illustrant l'événement/période de la diapositive
- `media_credit` :citation des sources de la ressource média illustrant l'événement/période de la diapositive. Il est indispensable de vérifier les droits d'utilisation d'une image ou d'une vidéo avant son utilisation dans une page Web.



	A	B	C	D	E	F	G
1	start_date_year	end_date_year	text_headline	text_text	media_url	media_caption	media_credit
2			Histoire du Web	Quelles origines pour le Web ?	https://upload.wikimedia.org/wil	Réseau Arpanet 1977	ARPANET [Public domain]
3	1965		Ted Nelson	 Ted Nelson propose en 1965 un système de fichiers informatiques incorporant le concept d'hypertexte.	https://upload.wikimedia.org/wil	Ted Nelson en 2011	Dgies [CC BY-SA 3.0 (https://creativecommons.org/licens
4	1989		Tim Berners-Lee	 Tim Berners-Lee propose une implémentation de l'hypertexte de Ted Nelson sur le réseau Internet : c'est la naissance du Wolrd Wide Web.	https://youtu.be/PbCG_BeL5dY	Biographie de Tim	France Culture
5	1991		Première page Web en HTML	 La première page Web en HTML est publiée sur un serveur Web du CERN en 1991.			
6	1995	2000	Premiers moteurs de recherche	 Alta-Vista domine le marché des moteurs de recherche entre 1995 et 2000. Grace à de puissants serveurs il est capable d'indexer la plupart des pages du Web et de répondre à 13 millions de requêtes par jour. Il est détrôné par Google.	https://www.flickr.com/photos/morville/6795467691	Barre de recherche d'Alta-Vista	Peter Morville [CC BY-SA 3.0 (https://creativecommons.org/licens

En pratique, après avoir téléchargé et déballé l'archive zip depuis https://frederic-junier.org/SNT/2019/Theme1_Web/ressources/frise.zip, il faut compléter le fichier `frise_data.ods` depuis Libreoffice, avant de l'exporter au format `csv` avec un encodage en `UTF-8`. Un programme Javascript extrait les données du `csv` avec la bibliothèque Papaparse puis la frise est générée avec la bibliothèque Timeline.
Pour que le programme Javascript fonctionne, il faut utiliser Firefox et changer le paramétrage de sécurité par défaut. Après avoir saisi `about:config` dans la barre d'adresses, il faut accepter le risque dans la page d'avertissement, saisir la préférence `privacy.file_unique_origin` dans la barre de recherche et cliquer sur la valeur booléenne pour la passer à `false`. Pour en savoir plus, sur ces mécanismes de sécurité voir https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

Voici le cahier des charges :

1. présenter sous forme de frise l'une des cinq périodes décrites précédemment, avec un contenu plus riche et développé ;
2. structurer le contenu avec certaines balises HTML : paragraphes (`<p>`), listes (`<ul>`, `<ol>`), liens hypertextes (`<a href="..." alt="..">`), mise en forme de texte (`<strong>`, `<em>` ...) ... ;
3. associer une feuille de style CSS avec des styles personnalisés ;
4. compléter le fichier Javascript `frise_javascript.js` avec quelques fonctions Javascripts personnalisées ;
5. respecter les droits de réutilisation pour les images intégrées à la frise.

2 Architecture de Von Neumann

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

Cours-Essentiel 1

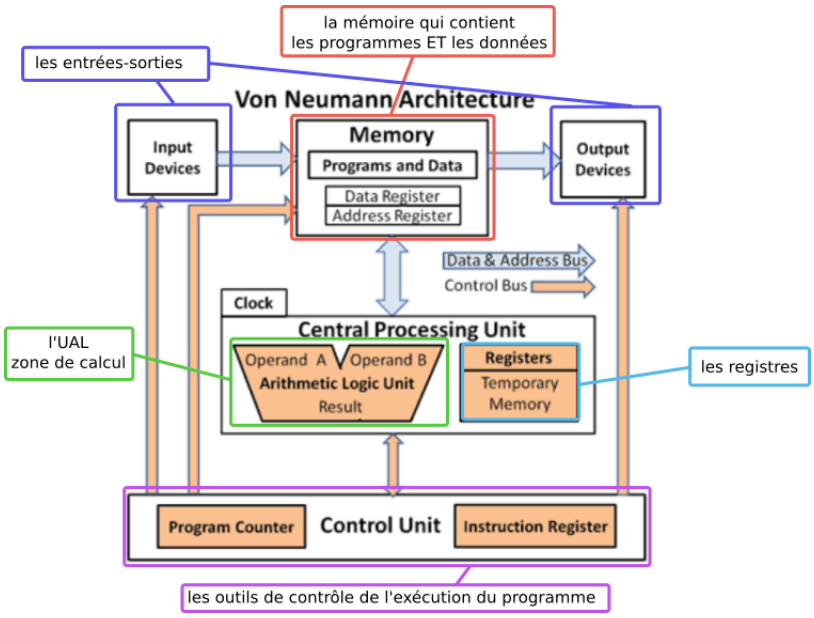
Un ordinateur est une machine **programmable**, **automatique** et **universelle** :

- **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 ans 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'indécidabilité.
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 :

- Le **CPU** (*Central Processing Unit*), ou **processeur** constitué de :
  - 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.
  - L'**Unité de Contrôle** charge la prochaine instruction dont l'adresse mémoire se trouve dans un registre appelé **Compteur de Programme** (PC en anglais), la décode et commande l'exécution par l'ALU. L'instruction en cours d'exécution est chargée dans le **Registre d'Instruction**. L'**Unité de Contrôle** 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.
- 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 echanges de données et d'adresses.
- 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

L'**Unité de Contrôle** réalise en boucle un cycle d'exécution rythmé par une horloge à quartz :

- **Chargement** de l'instruction pointée par le **Compteur de Programme** dans le **Registre d'Instruction**.
- **Décodage** de l'instruction stockée dans le **Registre d'Instruction** et chargement des opérandes.
- **Exécution** de l'instruction par **l'ALU** s'il s'agit d'une opération arithmétique ou logique ou par **l'Unité de Contrôle**

s'il s'agit d'un branchement avec modification du **Compteur de Programme**.

Par exemple si l'horloge a une fréquence de 2,5 gigahertz, 2,5 milliards de cycles d'exécutions sont effectués par seconde. Jusqu'en 2 004, la fréquence des processeurs a augmenté linéairement, mais depuis elle stagne à cause des problèmes de dissipation thermique. Les performances des ordinateurs sont améliorées par d'autres techniques (pipe-line,exécution en parallèle sur plusieurs coeurs de processeurs . . .)
Le **pipe-line d'instructions** est une évolution du modèle de Von Neumann, qui découpe chaque instruction en étapes élémentaires confiées à des circuits indépendants. Par exemple, dans un pipe-line à cinq étages, une instruction se décompose en cinq étapes : FETCH (lecture), DEC (écodage), EXEC (exécution) ; MEM (accès mémoire) et RES (livraison du résultat). Le processeur peut exécuter en parallèle jusqu'à cinq instructions lorsque les circuits sont libérés. Ainsi l'exécution de cinq instructions ne nécessite plus que 9 cycles au lieu de 25.

i_1	FETCH	DEC	EXEC	MEM	RES				
i_2		FETCH	DEC	EXEC	MEM	RES			
i_3			FETCH	DEC	EXEC	MEM	RES		
i_4				FETCH	DEC	EXEC	MEM	RES	
i_5					FETCH	DEC	EXEC	MEM	RES

source : Laurent Bloch

2.2 Hiérarchie des mémoires

Cours-Essentiel 2

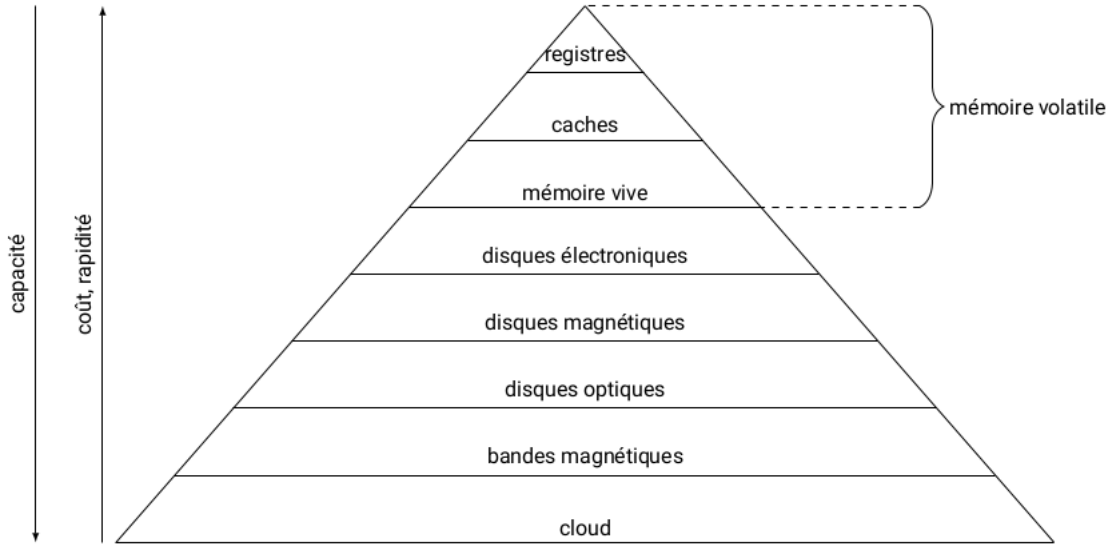
On distingue plusieurs types de mémoires suivant leur **persistance**, leur **capacité**, leur **rapidité d'accès**. Tous les types de mémoire utilisent une unité élémentaire d'information qui peut prendre deux états 0 ou 1 qu'on appelle **binary digit** ou **bit**.

- 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 rapide car elle est située directement dans le processeur.
  - La **mémoire centrale**, ou **Random Access Memory** est une mémoire de grande capacité.
    Les bits disponibles sont regroupés par blocs en **mots mémoires** de même taille. Celle-ci correspond à la largeur d'un registre : 32 bits ou 64 bits actuellement. C'est aussi la taille de l'adresse repérant un mot mémoire. En pratique, on peut adresser des blocs de 8 bits appelés **octets**. Un processeur avec des registres de 32 bits ne peut adresser plus de 2^{32} octets soit environ 4 gigaoctets. Chaque mot mémoire est adressable directement à partir de son adresse, on parle de **mémoire à accès direct**.
    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 et le **goulot d'étranglement de l'architecture de Von Neumann**.
  - 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.
- 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 proportionnel. Le système d'exploitation peut étendre virtuellement la mémoire centrale en utilisant les capacités des mémoires persistantes.
- 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 1

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.

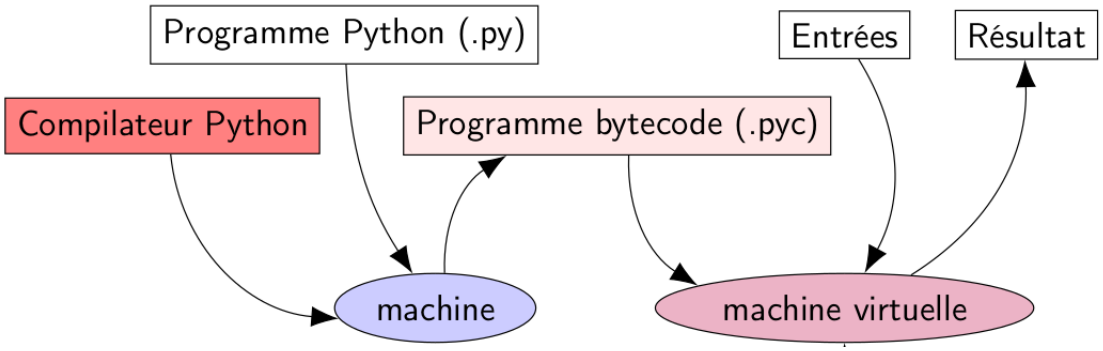
1. Quelle taille de mémoire centrale en octets peut-on adresser avec un processeur d'architecture 64 bits ?
2. Un disque dur a une capacité de 500 Go. Exprimer cette capacité en Gio.
3. Sachant qu'une minute de musique au format mp3 occupe un espace de 1 Mo, combien d'heures de musique peut-on stocker 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 produite 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 2

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

```
#include "stdlib.h"

int main()
{
int a = 5;
a =  a + 3;
return 0;
}
```

1. **a.** Récupérer le fichier source du programme `exemple.c`.
   b. Compiler le programme pour produite 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
```

c. Si on édite le fichier `exemple` avec un éditeur de textes, obtient-on un affichage lisible ?

d. Afficher le contenu du fichier binaire `exemple` avec un éditeur hexadécimal comme `okteta`.

e. 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
```

f. 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 `%rbp` 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
```

2. Le module `dis` de la bibliothèque standard permet de désassembler le `bytecode` produit par un code source `Python` pour obtenir les instructions du compilateur `Python` ayant permis de lé générer.

a. Créer un nouveau fichier ou notebook Python puis saisir le programme ci-dessous :

```
import dis

code_source = """
a = 5
a = a + 3
"""

dis.dis(code_source)
```

À partir de la documentation https://docs.python.org/3/library/dis.html, expliquer la traduction du programme contenu dans code_source en assembleur Python. Comparer avec la traduction d'un programme similaire obtenue précédemment en assembleur x86.

```
2           0 LOAD_CONST               0 (5)
            3 STORE_NAME               0 (a)


3           6 LOAD_NAME                0 (a)
            9 LOAD_CONST               1 (3)
           12 BINARY_ADD
           13 STORE_NAME               0 (a)
           16 LOAD_CONST               2 (None)
           19 RETURN_VALUE
```

b. Afficher les codes numériques des instructions du programme contenu dans code_source en assembleur Python avec le programme ci-dessous.

```
print(list(dis.get_instructions(code_source)))

code = compile(code_source, '<string>','exec')
for octet in code.co_code:
    print(octet, end = ' ')
```

3.2 Premiers programmes en assembleur avec le simulateur Aqua

Méthode

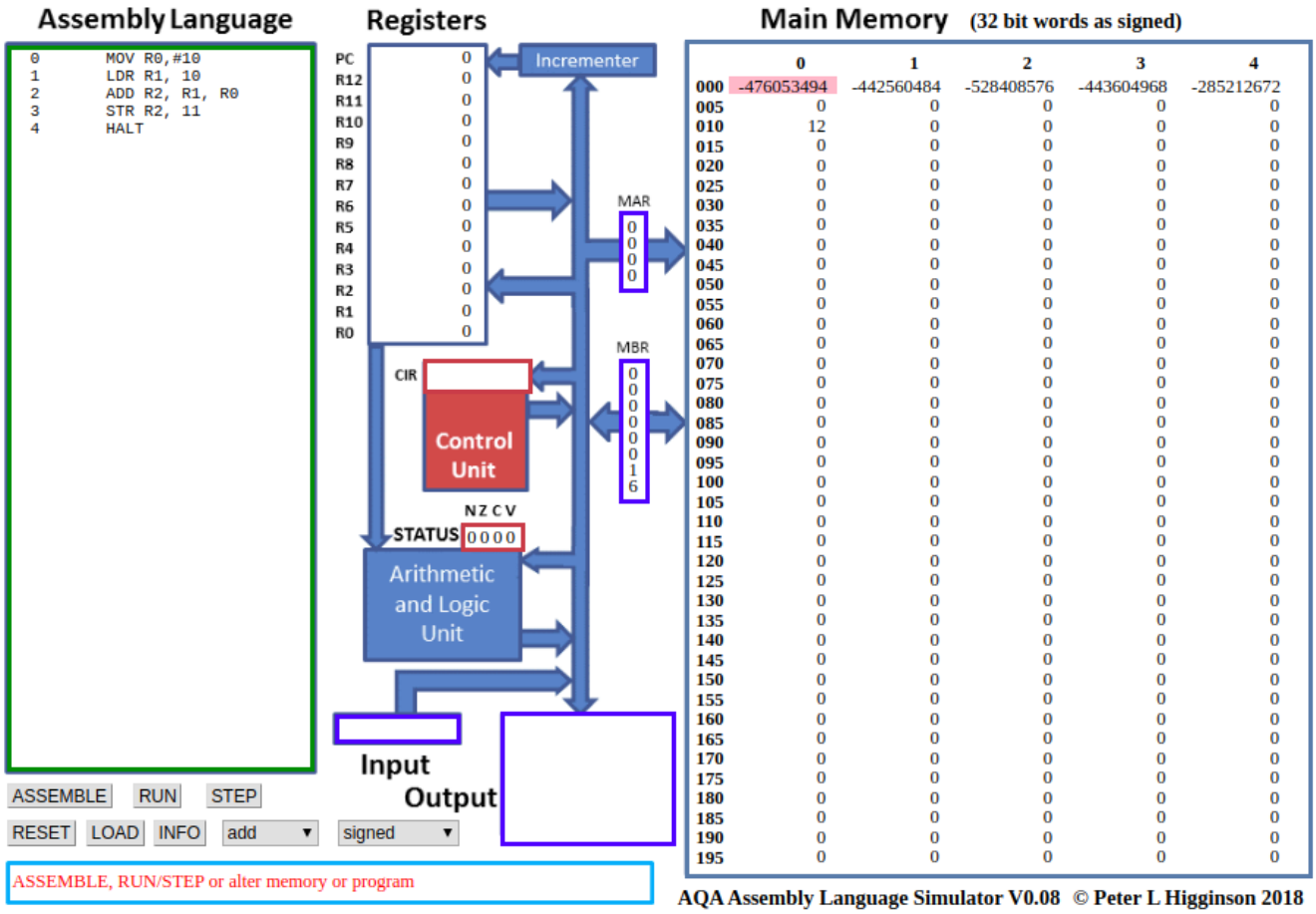
Nous allons utiliser un simulateur d'architecture de Von Neumann, réalisé par Peter Higginson pour préparer des étudiants anglais à leur examen de Computer Science. Il se nomme AQUA et on peut l'exécuter en ligne sur http://www.peterhigginson.co.uk/AQA/.
Quelques principes de base :

- On ne peut pas définir de variables. Les données manipulées sont soient 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 R3,123`	Stocke le contenu du registre R1 à l'adresse 123 en mémoire
`STR R3,123`	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 3

1. Ouvrir le simulateur AQUA depuis le lien sur le bureau ou le Web : http://www.peterhigginson.co.uk/AQA/.

2. 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. 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 instruction, l'**ALU** est-elle sollicitée ?

Exercice 4

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 *l'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 5

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 R0,2` 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 6

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 7

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 4, 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
```