Pour interagir avec des ordinateurs, il faut une méthode pour communiquer avec la machine. Comment faire passer des instructions complexes à un ordinateur qui ne comprend que « 0 » et « 1 » ?

On rappelle que le processeur (CPU) est constitué de 3 parties :

- L'**unité de commande** qui analyse les instructions à exécuter.
- L'**unité arithmétique et logique (UAL)** permettant de réaliser des opérations arithmétiques et logiques.
- Des **registres internes**, petite quantité de mémoire permettant de stocker de façon très temporaire des opérandes ou résultats intermédiaires de calcul.

Partie I : Découverte de l'assembleur

Programmer en langage machine est extrêmement difficile (très longue suite de 0 et de 1), pour pallier cette difficulté, les informaticiens ont remplacé les codes binaires par des symboles (plus facile à retenir qu'une suite de "1" et de "0") : c'est le langage assembleur.
Le processeur est uniquement capable d'interpréter le langage machine, un programme appelé "assembleur" assure donc le passage du langage assembleur au langage binaire.

Les instructions assembleur

Les différentes instructions de l'assembleur font référence aux cases mémoires (ROM, RAM) en utilisant leur adresse. Par souci de simplification, nous utiliserons les adresse mémoires codées en base 10 plutôt qu'en binaire.

Dans cette description, nous utiliserons *n* pour indiquer une adresse mémoire, et *R* pour indiquer un registre.
On peut utiliser des valeurs immédiates (entiers non enregistrés en mémoire) à la place d'un registre pour les opérations arithmétiques. Elles sont alors indiquée par un #.

Instructions pour échange de données entre le processeur et la mémoire

- *LDR R0, n* Transférer le contenu d'une case mémoire d'adresse *n* dans le registre *R0*.
  Exemple : *LDR R0, 7* Transfer le contenu de la case mémoire 7 dans *R0*
- *STR R0, n* Transférer le contenu du registre *R0* dans une case mémoire d'adresse *n*
  Exemple : *STR R0, 7* Transfer le contenu de *R0* dans la case mémoire 7
- *MOV R0, R1* Transférer le contenu du registre *R1* dans le registre *R0*

Instruction pour les opérations arithmétiques et logiques exécutées par le processeur à partir des registres

- *ADD R2, R0, R1* Additionner les contenus des registres *R0* et *R1*, et stocker le résultat dans *R2*
  Exemple : *ADD R2, R1, #12* Additionne le nombre 12 au contenu de *R1* et stock le résultat dans *R2*
- *SUB R2, R0, R1* Soustraire les contenus des registres *R0* et *R1*, et stocker le résultat dans *R2*
- *CMP R0, R1* Comparer les contenus des registres *R0* et *R1* et renvoie 0 (R0 = R1), 1 (R0 > R1) ou -1 (R0 < R1)

Instruction pour les déplacements vers un autre registre

- *JMP n* Passer directement à la case d'adresse *n*.
  Si cette instruction suit une instruction CMP, on peut indiquer un déplacement sous conditions avec :
  JMPZ: Déplacement si R0 = R1 (si CMP renvoie 0)
  JMPP: Déplacement si R0 > R1 (si CMP renvoie 1)
  JMPN: Déplacement si R0 < R1 (si CMP renvoie -1)

Instruction de fin de programme

- *HALT*

A la place des registres ou adresses mémoire, on peut aussi utiliser des **labels** dans les instructions. Un label correspond à une adresse en mémoire vive, qui peut contenir des instructions particulière (un peu comme une fonction en python).

Exemple :		
	CMP R4, #18	Compare la valeur contenue dans *R4* au nombre 18
	JMPP MonLabel	Si le contenu de *R4* > 18, va au label MonLabel
	MOV R0, R4	Sinon, met le contenu de *R4* dans *R0*
	HALT	Fin du programme
	Monlabel :	Définition du label
	MOV R0, #18	Met le nombre 18 dans *R0*
	HALT	Fin du programme

I.1. On considère les cases mémoires et registres ci-dessous, à l'état initial, ainsi qu'une séquence d'instructions :

Adresse	107	108	109	110
Valeur	42	68	47	33

R0	R1
0	0

```
LDR R0, 107
LDR R1, 108
ADD R0, R0, R1
STR R0, 110
HALT
```

Compléter les tableaux ci-dessous afin de déterminer ce que l'exécution des opérations fera et donc quel sera l'état final des cases mémoire.

- Après exécution de : *LDR R0, 107*
  LDR R1, 108

Adresse	107	108	109	110
Valeur				

R0	R1

- Après exécution de : *ADD R0, R0, R1*

Adresse	107	108	109	110
Valeur				

R0	R1

- Après exécution de : *STR R0, 110*

Adresse	107	108	109	110
Valeur				

R0	R1

I.2. La case mémoire 50 contient une valeur inconnue, *x*. On considère la séquence d'instructions suivante :

```
LDR R0, 50
CMP R0, #2
JMPZ Verif
MOV R1, #1
HALT
Verif:
    MOV R1, #0
    HALT
```

Que fait ce programme assembleur ? A quoi sert alors la valeur stockée dans le registre *R1* ?

I.3. On considère le code python suivant, où *N* et *M* sont des entiers quelconques :

```
1  x = N
2  y = M
3  if x > 10:
4      y = x
5  else :
6      x = x-y
```

Pour assigner une valeur à une variable en assembleur, il faut mettre cette valeur dans un registre puis l'enregistrer dans une case mémoire.
Donner la séquence d'instructions assembleur correspondant à ce code. On utilisera les cases mémoire d'adresse 30 et 40 et aussi peu de registres que possible.

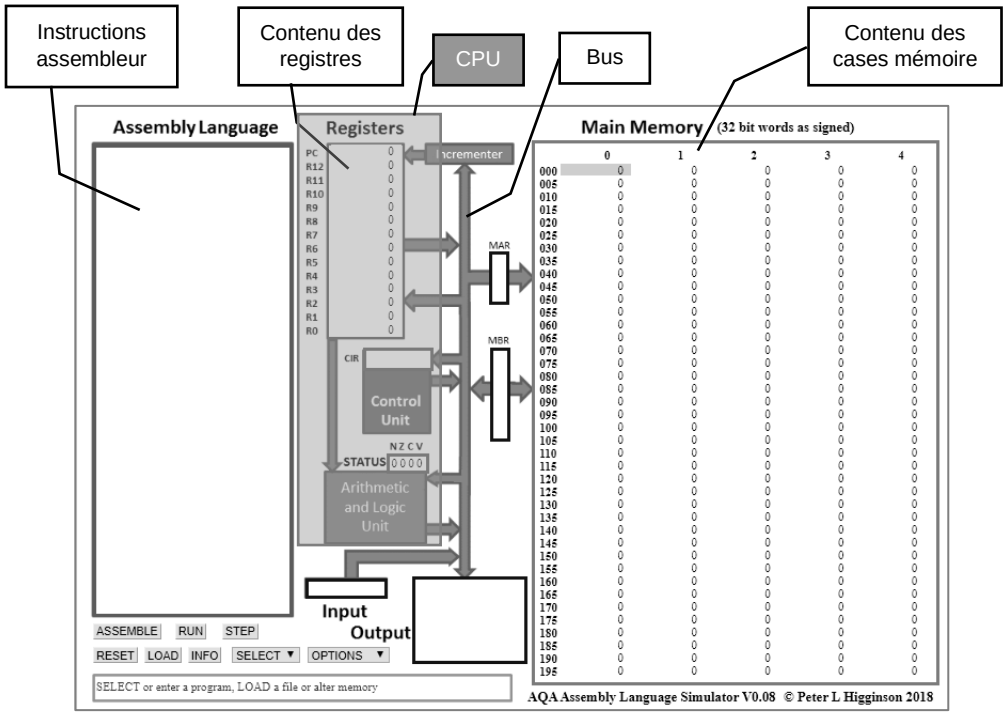
Pour aller un peu plus loin…
Proposer une séquence d'instructions qui multiplie par 5 le nombre contenu dans la case mémoire d'adresse 80 et stocke le résultat dans la case d'adresse 81.

Partie II : Simulation

Comme nous avons pu le voir précédemment, les données sont stockées dans la mémoire centrale et transportées jusqu'au registres du processeur afin d'être utilisées.

Afin de mieux visualiser ce procédé, nous allons utiliser un simulateur qui, à partir d'une séquence d'instructions assembleur, présente les différentes étapes du traitement des instructions.
Le simulateur se trouve sur internet, à l'adresse : *http://www.peterhigginson.co.uk/AQA/*

Les instructions assembleur sont codées (en binaire, mais représenté en hexadécimal) dans les premières cases mémoires, à partir de l'adresse 0. Les lignes d'instructions sont elles aussi numérotée, à partir de 0.
Dans ce simulateur, les valeurs voyageant dans des ronds rouges correspondent aux adresses et celles voyageant dans des ronds bleus correspondent aux valeurs.



Avant de commencer, dans l'onglet OPTION, choisir def slow (pour voir le parcours des valeurs).

A. Séquence d'instruction 1

Dans la partie Assembly Langage, entrer l'instruction suivante :

MOV R0, #4
HALT

Cliquer sur Submit, puis RUN pour suivre le parcours des valeurs.

II-A.1. A quoi correspondent les valeurs qui apparaissent en mémoire lorsque vous cliquez sur Submit ? A quel langage cela correspond-il ?

II-A.2. A quoi correspond la première valeur voyageant entre le CPU et la mémoire ? Vers où la valeur correspondante est-elle envoyée ?

II-A.3. Quel est le rôle de l'unité de contrôle ?

B. Séquence d'instruction 2

Dans la partie Assembly Langage, entrer la séquence d'instruction suivante :

MOV R0, #4
MOV R1, #8
ADD R2, R0, R1
HALT

Cliquer sur Submit, puis RUN. Vous pouvez augmenter la vitesse avec les flèches >> qui apparaissent.

II.B.1. A quoi correspond le registre PC – INCEMENTER ?

Cliquer sur RESET, puis utiliser STEP pour suivre le parcours des valeurs étape par étape (la simulation s'arrête à la fin de chaque étape, il faut cliquer de nouveau sur STEP pour voir l'étape suivante).
Passer rapidement les 2 premières étapes puis étudier l'étape 3.

II.B.2. Décrire ce qui se passe dans l'étape 3 : Quelles valeurs/adresses voyagent ? De où à où ? Quelles actions se font dans les différentes parties du CPU ? (On peut ignorer INCREMENTER)

Savoir faire

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