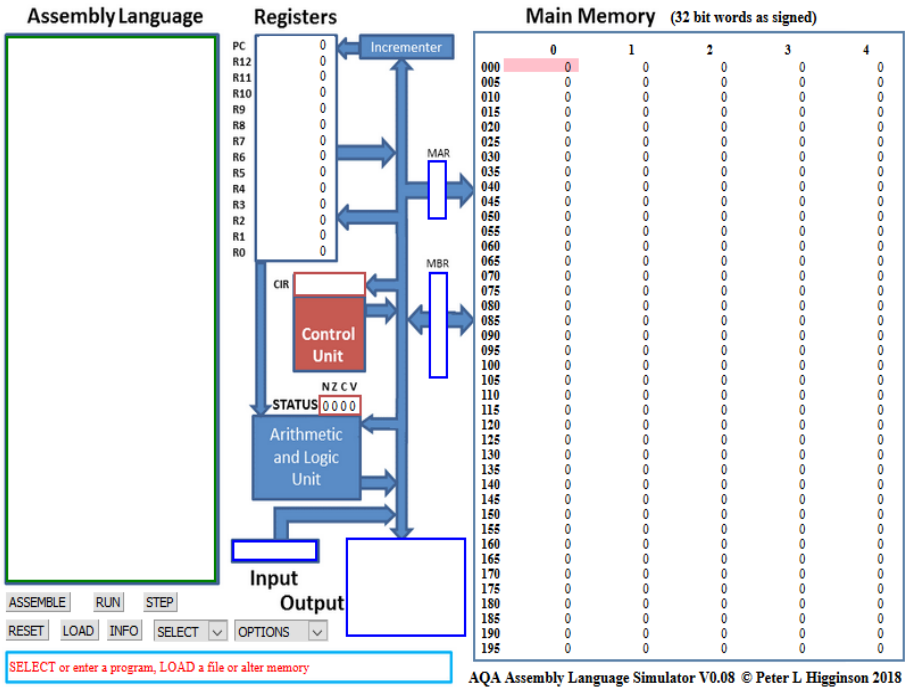
Il existe plusieurs logiciels de simulation permettant de programmer en assembleur.
Dans ce TP nous allons utiliser le simulateur de Peter L. Higginson.

**A faire :** Allez à l'adresse : **http://www.peterhigginson.co.uk/AQA/**. Vous devez obtenir ceci :




## Identification des 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.
-

**La RAM :** Par défaut le contenu des différentes cellules de la mémoire est en base 10 (entier signé : **signed**), 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 

situé en bas dans la partie gauche du simulateur.

**A faire :** À l'aide du bouton 

passez à un affichage en binaire.

Comme vous pouvez le constater, chaque cellule de la mémoire comporte 32 bits et possède une adresse (de 000 à 199), ces adresses sont codées en base 10.

**A faire :** Repassez à un affichage en base 10 : 


**Le CPU :** Dans la partie centrale du simulateur, nous allons trouver en allant du haut vers le bas :

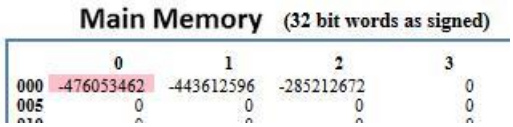
- le bloc registre (Registers) : nous avons 13 registres (R0 à R12) + 1 registre (PC) qui contient l'adresse mémoire de l'instruction en cours d'exécution
- le bloc unité de commande (Control Unit) qui contient l'instruction machine en cours d'exécution (au format hexadécimal),
- le bloc unité arithmétique et logique (Arithmetic and Logic Unit).

## Programmer en assembleur :

**Programme 1 :** Dans la partie gauche, saisir ces 3 lignes de code en assembleur :


puis, cliquez sur le bouton Submit.

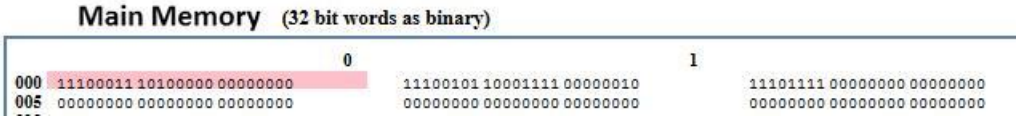
Vous devriez voir apparaitre des nombres étranges dans les cellules mémoires 000, 001 et 002 :




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.

Vous devriez obtenir ceci :




Repassez à un affichage en **base 10** afin de faciliter la lecture des données présentes en mémoire.

**A faire :** Pour exécuter notre programme, il suffit maintenant de cliquer sur le bouton RUN.

Vous allez voir le CPU travailler en direct grâce à de petites animations.

Si cela va trop vite (ou trop doucement), vous pouvez régler la vitesse de simulation à l'aide des boutons << >>.

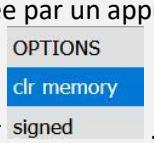
Un appui sur le bouton STOP met en pause la simulation, si vous appuyez une deuxième fois sur ce même bouton STOP, la simulation reprend là où elle s'était arrêtée.

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




.

Si vous remettez votre mémoire à 0, il faudra cliquer sur le bouton ASSEMBLE avant de pouvoir exécuter de nouveau votre programme.

**Programme 2 :**

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.

| **Votre programme** |
| --- |
| |

**Programme 3 :**

Faites en sorte que dans la mémoire 100, il y ait le résultat de la somme de 42 et 54.
L'instruction pour effectuer une addition est : ADD R2,R0,R1 (place dans R2 le résultat de R0+R1)

| **Votre programme** |
| --- |
| |

**Programme 4 :**

Additionner les 3 nombres 50, 42 et 54, en n'utilisant que les deux registres R0 et R1.

| **Votre programme** |
| --- |
| |

**Programme 5 :**

On souhaite calculer $\mathbf{2^5}$. Il n'y a pas d'instruction « multiplier », il faut se débrouiller avec des additions.
Voici un programme, qui à l'aide d'une boucle, calcule $\mathbf{2^5}$. (Et aussi grâce au fait que $2 + 2 = 4 = 2^2$, $4 + 4 = 8 = 2^3$, $8 + 8 = 16 = 2^4$, etc...)

| n° | Instruction | Commentaire |
| --- | --- | --- |
| 0 | `MOV R0,#2` | On place le nombre 2 dans le registre R0 |
| 1 | `MOV R1,#1` | On place le nombre 1 dans le registre R1 (ce sera le compteur de boucle) |
| 2 | `B maboucle` | B signifie « Branch » et `maboucle` est un label ou étiquette |
| | `maboucle:` | `B maboucle` renvoie l'exécution à maboucle : |
| 3 | `ADD R0,R0,R0` | On stocke dans R0 le résultat de l'addition de R0 avec R0 |
| 4 | `ADD R1,R1,#1` | On stocke dans R1 le résultat de l'addition de R1 avec 1 : on incrémente le compteur de 1 |
| 5 | `CMP R1,#5` | On compare R1 avec 5 (CMP – comparaison) |
| 6 | `BNE maboucle` | S'ils ne sont pas égaux (NE – not equal), on relance maboucle |
| 7 | `STR R0,100` | On stocke le résultat dans la mémoire 100 |
| 8 | `HALT` | On stoppe le programme |

Après avoir saisi puis exécuté le programme précèdent, modifiez le pour qu'il calcule $\mathbf{2^{10}}$.

**Programme 6 :**

Soit **Sn**, la somme des **n** premiers entiers : **Sn = 1 + 2 + 3 + 4 + . . . + n**.
Écrire un programme qui calcule **S10**.
Utilisez une boucle. Stockez le résultat dans la mémoire à l'adresse 100.

**Pour les plus courageux** : Voici la liste des instructions :

| Mnémonique | Signification |
|---|---|
| `LDR Rd, <Memory ref>` | Charge dans le registre `d`, la valeur stockée dans la mémoire à l'adresse `<Memory ref>`. |
| `STR Rd, <Memory ref>` | Range dans la mémoire à l'adresse `<Memory ref>`, la valeur stockée dans le registre `d`. |
| `ADD Rd, Rn, <operand2>` | Ajoute la valeur spécifiée par `<operand2>` de la valeur contenue dans le registre `n` et range le résultat dans le registre `d`. |
| `SUB Rd, Rn, <operand2>` | Soustrait la valeur spécifiée par `<operand2>` de la valeur contenue dans le registre `n` et range le résultat dans le registre `d`. |
| `MOV Rd, <operand2>` | Copie la valeur spécifiée par `<operand2>` dans le registre `d`. |
| `CMP Rd, <operand2>` | Compare la valeur contenue dans le registre `d` avec la valeur spécifiée par `<operand2>`. |
| `B <label>` | Saute à l'instruction à la position du `<label>` dans le programme. |
| `B<condition> <label>` | Saute à l'instruction à la position du `<label>` dans le programme, si le résultat de la dernière comparaison correspond au critère spécifié par `<condition>`. Ce critère peut-être : `EQ` : égal à `NE` : non égal à `GT` : plus grand que `LT` : plus petit que |
| `AND Rd, Rn, <operand2>` | Opère un ET logique entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`. |
| `ORR Rd, Rn, <operand2>` | Opère un OU logique entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`. |
| `EOR Rd, Rn, <operand2>` | Opère un XOR logique (ou exclusif) entre les valeurs contenues dans le registre `n` et `<operand2>` et range le résultat dans le registre `d`. |
| `MVN Rd, <operand2>` | Opère un NON logique de la valeur spécifiée par `<operand2>` et range le résultat dans le registre `d`. |
| `LSL Rd, Rn, <operand2>` | Fait subir à la valeur contenue dans le registre `n`, un décalage vers la gauche d'un nombre de bits spécifié par `<operand2>` et range le résultat dans le registre `d`. |
| `LSR Rd, Rn, <operand2>` | Fait subir à la valeur contenue dans le registre `n`, un décalage vers la droite d'un nombre de bits spécifié par `<operand2>` et range le résultat dans le registre `d`. |
| `HALT` | Arrête l'exécution du programme. |

**Les labels :** Un label est placé dans le code en écrivant un nom d'étiquette suivi de deux points « : ».
Pour sauter au label, placer l'étiquette du label après instruction de saut B.

**Signification de <operand2> :** <operand2> a 2 significations selon que le 1er caractère est R ou # :

- **Rm** signifie que l'on utilise la valeur contenue dans le registre **m**. Par exemple **R6** signifie que l'on utilise la valeur contenue dans le registre 6.
- **#d** signifie que l'on utilise la valeur décimale **d**. Par exemple **#12** signifie que l'on utilise la valeur décimale 12.

**Programmes 7 et 8 :**

**a.** Écrire un programme en assembleur qui détermine puis stocke dans la mémoire à l'adresse 100 le plus grand des 3 nombres situés dans les registres **R0**, **R1** et **R2**.

**b.** Traduire en assembleur le programme Python ci-dessous :

Code

```
x=4
y=6
if x==y:
    y=x - 4
else :
    y=x + y
```