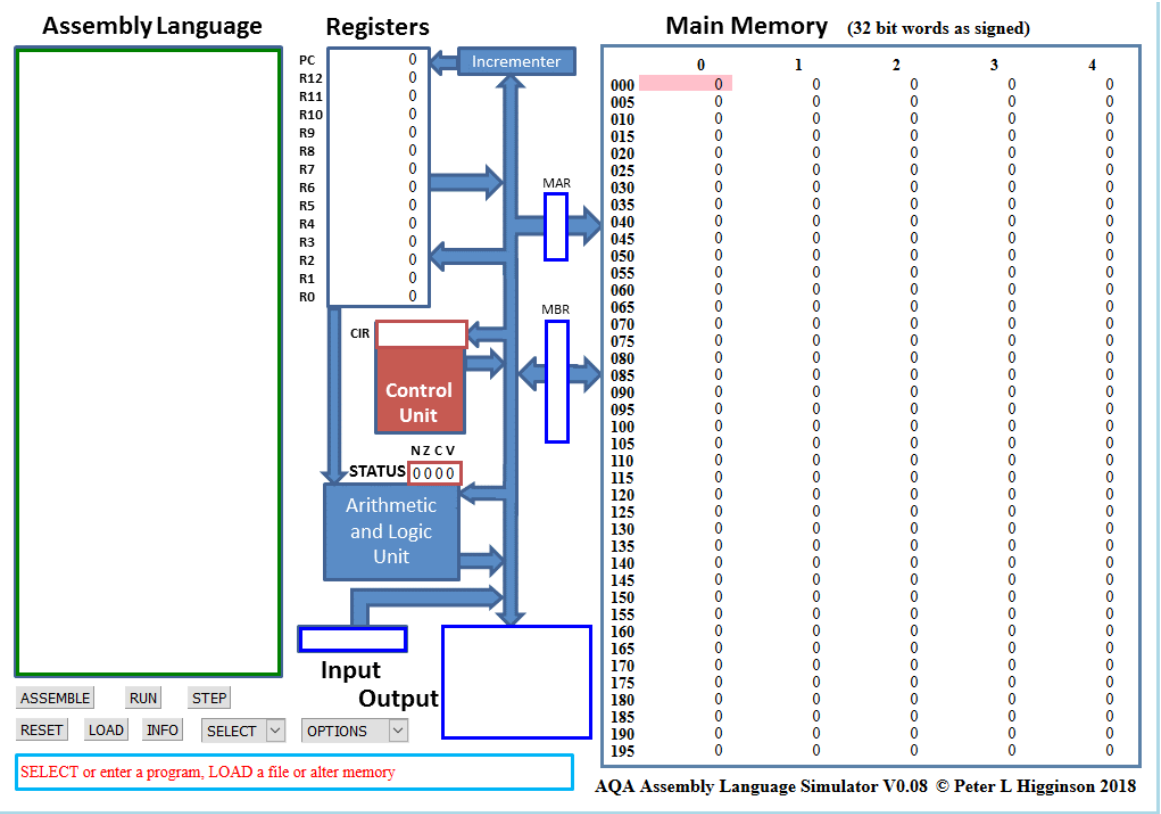
Rendez vous à cette adresse : http://www.peterhigginson.co.uk/AQA/

Vous devriez voir :



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), 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.

À faire :

À l'aide du bouton "OPTIONS", passez à un affichage en binaire.

Comme vous pouvez le constater, chaque cellule de la mémoire comporte 32 bits (nous avons vu que classiquement une cellule de RAM comporte 8 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"->"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")

Programmer en assembleur

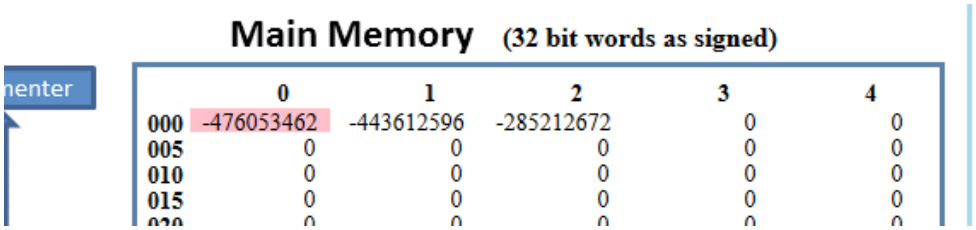
À faire :

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

```
MOV R0,#42
STR R0,150
HALT
```

→ Place le nombre 42 dans le registre R0
→ Stocke le contenu de R0 dans la mémoire 150
→ Arrête le processus

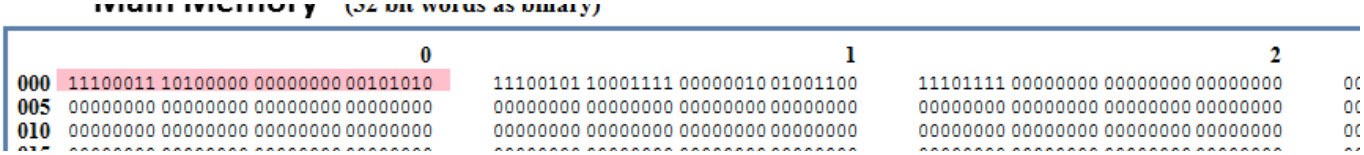
Une fois la saisie terminée, cliquez sur le bouton "submit". Vous devriez voir apparaitre des nombres "étranges" dans les cellules mémoires 000, 001 et 002 :



Main Memory (32 bit words as signed)

	0	1	2	3	4
000	-476053462	-443612596	-285212672	0	0
005	0	0	0	0	0
010	0	0	0	0	0
015	0	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 :



	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
010	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.

À 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 "<<" et ">>". 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 "clr memory". Si vous remettez votre mémoire à 0, il faudra cliquer sur le bouton "ASSEMBLE" avant de pouvoir exécuter de nouveau votre programme.

À faire :

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.

À faire :

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)

À faire :

Pour récupérer une valeur stockée précédemment dans la mémoire, on doit la placer dans un registre.
L'instruction est : `LDR R0,50` (place dans R0 la valeur de la mémoire 50)

On souhaite additionner 3 nombres placés dans des mémoires, en n'utilisant que deux registres…

À faire :

On souhaite calculer 2^5 en n'utilisant qu'un seul registre.
Il n'y a pas d'instruction "multiplier", il va falloir se débrouiller avec des additions…

Comment réaliser une boucle (while)

Voici un programme qui calcule 2^5 avec une boucle :

```
    MOV R0,#2
    MOV R1,#1
    B maboucle
maboucle:
    ADD R0,R0,R0
    ADD R1,R1,#1
    CMP R1,#5
    BNE maboucle
    STR R0,100
    HALT
```

1. On place le nombre 2 dans le registre R0
2. On place le nombre 1 dans le registre R1 (ce sera le compteur)

3. `B` signifie "Branch" et maboucle est un "label"
   `B maboucle` renvoie l'exécution à maboucle:

4. On stocke dans R0 le résultat de l'addition de R0 avec R0
5. On incrémente le compteur de 1
6. On compare R1 avec 5 (CMP – comparaison)
   S'ils ne sont pas égaux (NE – not equal), on relance maboucle

7. On stocke le résultat dans la mémoire 100
8. On stoppe le programme

À faire: Modifiez ce programme pour calculer 2^{10}

À faire :

Écrire en assembleur un programme qui calcule la somme des n premiers entiers
(1 + 2+ 3 + 4 +…+ n)

Pour les plus courageux…

Voici la liste des instructions disponibles :

Instruction set

`LDR Rd, <memory ref>`	Load the value stored in the memory location specified by `<memory ref>` into register `d`.
`STR Rd, <memory ref>`	Store the value that is in register `d` into the memory location specified by `<memory ref>`.
`ADD Rd, Rn, <operand2>`	Add the value specified in `<operand2>` to the value in register `n` and store the result in register `d`.
`SUB Rd, Rn, <operand2>`	Subtract the value specified by `<operand2>` from the value in register `n` and store the result in register `d`.
`MOV Rd, <operand2>`	Copy the value specified by `<operand2>` into register `d`.
`CMP Rn, <operand2>`	Compare the value stored in register `n` with the value specified by `<operand2>`.
`B <label>`	Always branch to the instruction at position `<label>` in the program.
`B<condition> <label>`	Branch to the instruction at position `<label>` if the last comparison met the criterion specified by `<condition>`. Possible values for `<condition>` and their meanings are: `EQ`: equal to `NE`: not equal to `GT`: greater than `LT`: less than
`AND Rd, Rn, <operand2>`	Perform a bitwise logical AND operation between the value in register `n` and the value specified by `<operand2>` and store the result in register `d`.
`ORR Rd, Rn, <operand2>`	Perform a bitwise logical OR operation between the value in register `n` and the value specified by `<operand2>` and store the result in register `d`.

`EOR Rd, Rn, <operand2>`	Perform a bitwise logical XOR (exclusive or) operation between the value in register `n` and the value specified by `<operand2>` and store the result in register `d`.
`MVN Rd, <operand2>`	Perform a bitwise logical NOT operation on the value specified by `<operand2>` and store the result in register `d`.
`LSL Rd, Rn, <operand2>`	Logically shift left the value stored in register `n` by the number of bits specified by `<operand2>` and store the result in register `d`.
`LSR Rd, Rn, <operand2>`	Logically shift right the value stored in register `n` by the number of bits specified by `<operand2>` and store the result in register `d`.
`HALT`	Stops the execution of the program.

Labels

A label is placed in the code by writing an identifier followed by a colon (:). To refer to a label, the identifier of the label is placed after the branch instruction.

Interpretation of **<operand2>**

`<operand2>` can be interpreted in two different ways, depending on whether the first character is a `#` or an `R`:

- `#` – Use the decimal value specified after the `#`, eg `#25` means use the decimal value 25.
- `Rm` – Use the value stored in register `m`, eg `R6` means use the value stored in register 6.

The available general purpose registers that the programmer can use are numbered 0 to 12.

Imaginez et réalisez un programme en assembleur…

Par exemples :

- **Un programme qui stocke le plus grand de 3 nombres…**
- **Traduire en assembleur le code Python**

```
x = 4
y = 8
if(x == y):
    y = x-4
else:
    y = x+y
```