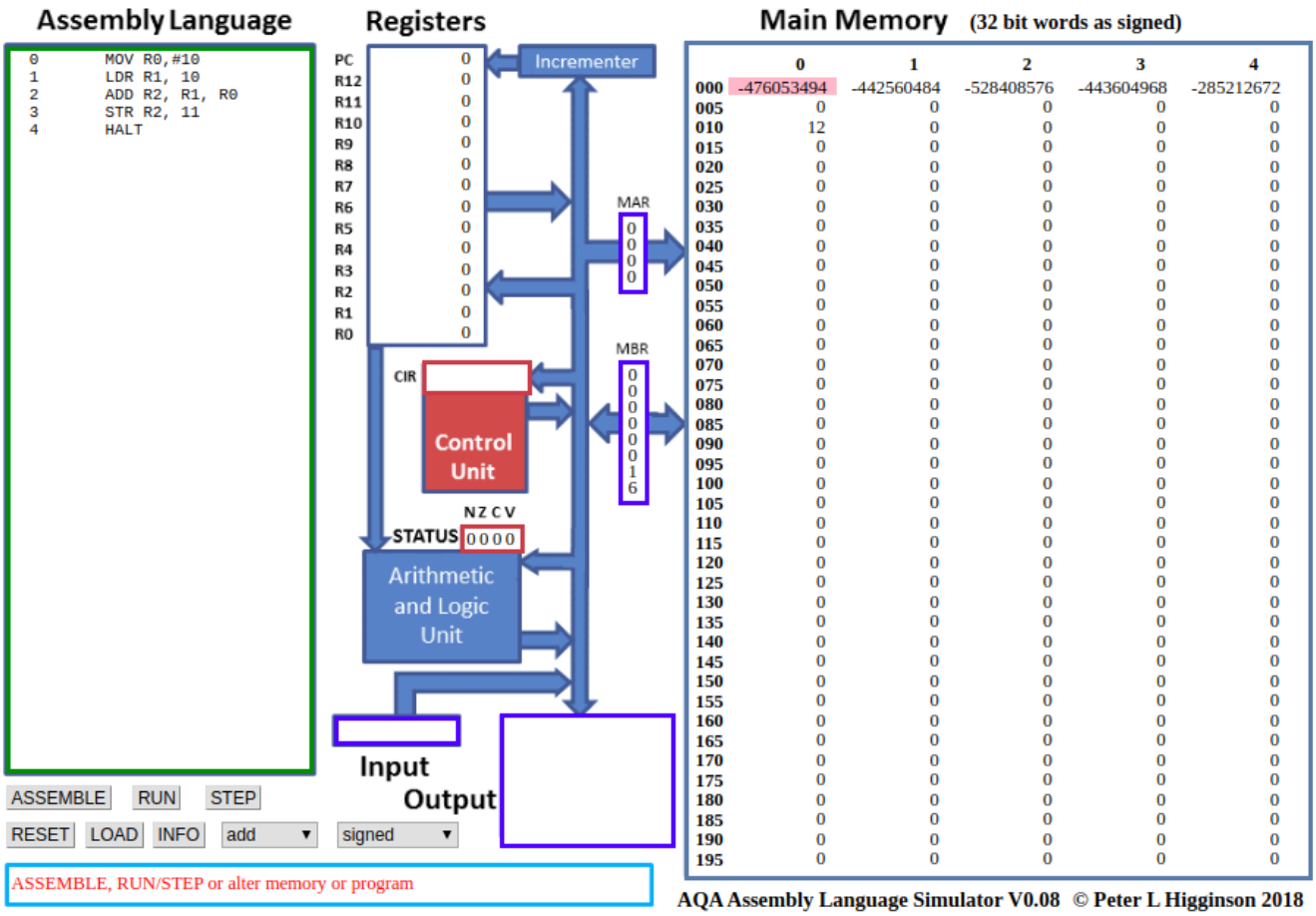
Thème : Architectures matérielles et systèmes d'exploitation.

1 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 https://www.peterhigginson.co.uk/AQA/.
Quelques principes de base :

- On ne peut pas définir de variables. Les données manipulées sont soit 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 https://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
`LDR R1,[R2]`	Charge dans le registre R1 la valeur stockée en mémoire à l'adresse contenue dans le registre R2
`STR R1,123`	Stocke le contenu du registre R1 à l'adresse 123 en mémoire
`STR R1,[R0]`	Stocke le contenu du registre R1 à l'adresse contenue dans le registre R0
`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 1

1. Ouvrir le simulateur AQUA depuis le lien sur le bureau ou le Web : https://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. Sélectionner l'affichage `Unsigned`. 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(s) instruction(s), l'**ALU** est-elle sollicitée?

Exercice 2

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 3

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.

2 Programmer une instruction conditionnelle en assembleur

Méthode

Dans le programme assembleur ci-dessous, on introduit de nouveaux symboles :

- `INP R1,2` 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
```

```
//étiquette else
else:
      MOV R2, #0
      SUB R1, R2, R0
//étiquette fin
fin:
//affichage du registre R1 dans Output
      OUT R1, 4
      HALT
```

Exercice 4

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 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`.

```
      //initialise le registre R2 avec le nombre 32
      MOV R2,#32
LOOP:
      //affiche dans Output le caractère dont le code ascii est contenu dans R2
      OUT R2,7
      //incrémente R2
      ADD R2,R2,#1
      //compare R2 avec 127
      CMP R2,#127
      //si R2 < 127 branchement conditionnel sur l'étiquette loop
      BLT LOOP
      //sinon le programme se poursuit
      MOV R2,#10
      //affichage du caractère de code ascii 10 qui est un saut de ligne
      OUT R2,7
      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 5

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 2, 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
```