1 Assembleur AQA

Présentation du simulateur `https://www.peterhigginson.co.uk/AQA/`.
Voir le document annexe pour connaître le jeu d'instructions.

2 Exercices

2.1 Addition (23 + 31)

1. Étudier avec le professeur ce 1er exemple commenté (voir version prof).
2. Écrire et exécuter le programme.
3. Charger le programme `somme_labels.txt`. Découvrir l'utilité des labels.
4. Charger le programme `somme_out.txt`. Découvrir la fonctionnalité d'affichage.

langage haut niveau

```
a = 23
b = 31
s = a + b
```

langage machine	**langage assembleur**
11100101100111110000000000001100	0. LDR R0, 5
11100101100111110001000000001100	1. LDR R1, 6
11100000100000000010000000000001	2. ADD R2, R0, R1
11100101100011110010000000001000	3. STR R2, 7
11101111000000000000000000000000	4. HALT
00000000000000000000000000010111	5. 23
00000000000000000000000000011111	6. 31

Les variables a et b sont mémorisées aux adresses 5 et 6. Le résultat est écrit à l'adresse 7.

2.2 Branchement

Les instructions de branchement permettent de modifier la valeur du registre `PC`, qui contient l'adresse de la prochaine instruction à réaliser, en fonction du contenu du registre `STATUS` (drapeaux `NZCV`).
Les adresses de branchement sont identifiées aisément à l'aide des labels.

langage haut niveau

```
a = 1
b = 5
if a > b:
    print(a)
else:
    print(b)
```

langage machine	**langage assembleur**
11100011101000000000000000000001	0. MOV R0, #1
11100011101000000001000000000101	1. MOV R1, #5
11100001010100000000000000000001	2. CMP R0, R1
10111010000000000000000000000001	3. BGT superieur
11101111000000100001000000000100	inferieur:
11101111000000000000000000000000	4. OUT R1, 4
11101111000001000000000000000100	5. HALT
11101111000000000000000000000000	superieur:
	6. OUT R0, 4
	7. HALT

Questions :

1. Charger et exécuter le programme `branchement.txt`.
2. Quelle instruction modifie le registre `STATUS` ?
3. Modifier le programme en donnant la valeur 8 à la variable a.

2.3 Boucle for

langage haut niveau

```
a = 100
n = 5
for i in range(n):
    a = a + 4
print(a)
```

langage machine

```
11100011101000001010000001100100
11100011101000000001000000000101
11100011101000000000000000000000
11100010100000000000000000000001
11100010100010101010000000000100
11100001010100000000000000000001
11001010111111111111111111111011
11100101100011111010000000000100
11101111000000101010000000000100
11101111000000000000000000000000
```

langage assembleur

```
0. MOV R10, #100
1. MOV R1, #5
2. MOV R0, #0
boucle:
3. ADD R0, R0, #1
4. ADD R10, R10, #4
5. CMP R0, R1
6. BLT boucle
7. STR R10, resultat
8. OUT R10, 4
9. HALT
resultat:
10. 0
```

Questions :

1. Charger, exécuter et analyser le programme `boucle_for.txt`.
2. Quel registre mémorise la valeur de `a` ? Quel registre mémorise la valeur de `n` ?
3. Quel registre mémorise la valeur de `i` ?
4. À quelle adresse mémoire est enregistré le contenu de `a` en fin de boucle ?
5. Modifier le programme pour réaliser l'instruction suivante écrite en Python :
   `for i in range(3, 12, 5)`

2.4 Boucle while

On propose un programme mystère en assembleur et 3 programmes en Python :

```
0. MOV R10, #1
1. MOV R1, #1000
boucle:
2.   LSL R10, R10, #3
3.   CMP R10, R1
4.   BLT boucle
5. OUT R10, 4
6. HALT
```

Prog Python 1

```
a = 1
n = 1000
while a < n:
    a = a * 8
print(a)
```

Prog Python 2

```
a = 1
n = 1000
while a <= n:
    a = a * 3
print(a)
```

Prog Python 3

```
a = 1
n = 1000
while a > n:
    a = a // 3
print(a)
```

Questions :

1. Charger, exécuter et analyser le programme `boucle_while.txt`.
2. Expliquer ce que fait l'instruction `LSL R10, R10, #3`
3. Quel programme Python correspond au programme mystère ?

2.5 Dépassement de capacité

Questions :

1. Charger et exécuter les programmes `soustraction.txt` et `depassement.txt`.
2. Expliquer pourquoi $23 - 31 = 4294967288$ en mode « non signé ».
3. Expliquer pourquoi $2147483647 + 1 = -2147483648$ en mode « signé ».

3 Un programme réel : division euclidienne (pour les plus rapides)

Pour réaliser une division euclidienne ($a = q \times b + r$), on propose l'algorithme suivant :

```
Fonction div_euclide(a, b)
|   r = a
|   q = 0
|   Tant que r >= b:
|   |    r = r - b
|   |    q = q + 1
|   Fin Tant que
|   Renvoyer q, r
```

Compléter le programme suivant pour réaliser une implémentation en langage assembleur de l'algorithme précédent.

```
// division euclidienne : calcule q, r = a//b, a%b
// a et b sont lus en mémoire (labélisés)
// q et r sont enregistrés en mémoire (labélisés)
// r : défini en R9
// q : défini en R10
// a : chargé initialement en R9
// b : chargé en R1
      LDR R9, a
      LDR R1, b
      CMP R9, R1
      BLT fin
      MOV R10, #0
boucle:
      SUB ...
      ADD ...
      CMP ...
      BEQ ...
      BGT ...
fin:
      STR R10, q
      STR R9, r
      OUT R10, 4
      OUT R9, 4
      HALT
a:    27
b:    6
q:    0
r:    0
```

Charger, compléter et exécuter le programme `div_euclide_trou.txt`.

4 Dichotomie (très facultatif)

Ce programme donné ici de façon anticipée sera étudié de manière plus approfondie après avoir vu le chap. sur le tri et la recherche dichotomique dans un tableau trié.
Il permet d'illustrer la pénibilité d'écrire en langage machine !

langage haut niveau

```
code_OK, code_erreur = 200, 404
tab = [3, 6, 7, 15, 22, 24, 31,
 50, 79, 94]
cible = int(input())
print(cible)
while cible != 0:
    debut = 0
    fin = len(tab) - 1
    while debut <= fin:
        milieu = (debut+fin)//2
        if tab[milieu] == cible:
            print(code_OK)
            break
        elif tab[milieu] > cible
            fin = milieu - 1
        else:
            debut = milieu + 1
    if debut > fin:
        print(code_erreur)
    cible = int(input())
    print(cible)
```

langage machine

```
11100011101000000101000011001000
11100011101000000110111101100101
11100011101000000001000000011011
11100011101000000011000000100100
11101111000000010000000000000010
11101111000000010000000000000100
11100011010100000000000000000000
00001010000000000000000000011100
11100000100000010010000000000011
11100001101000000010000010100010
11100101100100100100000000000000
11100001010100000000000000000100
00001010000000000000000000001011
11001010000000000000000000000010
11100001101000000001000000000010
11100010100000010001000000000001
11101010000000000000000000000010
11100001101000000011000000000010
11100010010000110011000000000001
11101010111111q1111111111111111
11100001010100010000000000000011
10111010000000000000000000000000
11101010111111111111111111110000
11101111000000100110000000000100
11101010111111111111111111101000
11101111000000100101000000000100
11101010111111111111111111100110
00000000000000000000000000000011
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001111
00000000000000000000000000010110
00000000000000000000000000011000
00000000000000000000000000011111
00000000000000000000000000110010
00000000000000000000000001001111
00000000000000000000000001011110
11101111000000000000000000000000
```

langage assembleur

```
MOV R5, #200
MOV R6, #404
initialisation:
  MOV R1, #premier
  MOV R3, #dernier
  INP R0, 2
  OUT R0, 4
  CMP R0, #0
  BEQ fin
boucle:
  ADD R2, R1, R3
  LSR R2, R2, #1
  LDR R4, [R2]
  CMP R0, R4
  BEQ trouve
  BLT chercherAGauche
chercherADroite:
  MOV R1, R2
  ADD R1, R1, #1
  B verifierIntervalleNul
chercherAGauche:
  MOV R3, R2
  SUB R3, R3, #1
  B verifierIntervalleNul
verifierIntervalleNul:
  CMP R1, R3
  BGT pasTrouve
  B boucle
pasTrouve:
  OUT R6, 4
  B initialisation
trouve:
  OUT R5, 4
  B initialisation
premier:  3
          6
          7
         15
         22
         24
         31
         50
         79
dernier: 94
fin: HALT
```