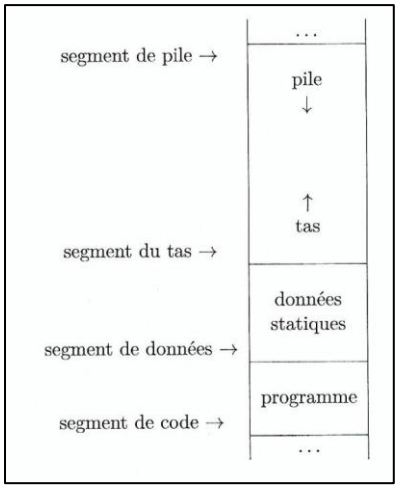
Numérique et Science Informatique

Thème 4 : Architectures matérielles et système d'exploitation

Chapitre 2 : Architecture de von Neumann

1ère partie : gestion de la mémoire



Ce schéma décrit l'organisation de l'espace mémoire d'un programma actif (généralement appelé processus), c'est-à-dire d'un programme en train d'être exécuté par la machine.

Cet espace est découpé en quatre parties (ou segments de mémoire) :

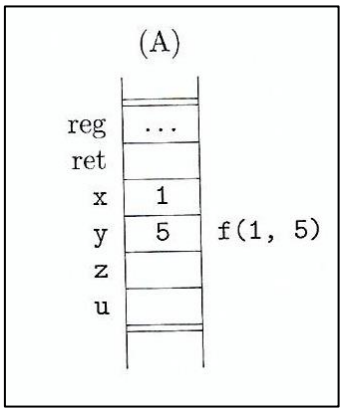
- Le **segment de code** qui contient les instructions du programme
- Le **segment de données** qui contient les données dont l'adresse en mémoire et la valeur sont connues au moment de l'initialisation de l'espace mémoire du programme. On parle de **données statiques** dont la taille est fixe, il n'est donc pas possible d'allouer de nouvelles cases mémoires dans cet espace lors de l'exécution du programme.
- Le **segment de pile.** Ce segment, comme le suivant, contient **l'espace mémoire alloué dynamiquement** par un programme. La pile est utilisée au moment de l'appel de **fonctions** d'un programme pour stocker des paramètres mais également des variables locales des fonctions.
- Le **segment du tas.** Il s'agit de la zone mémoire qui contient toutes les **données allouées dynamiquement** par un programme. Il peut s'agir de celles dont la durée de vie n'est pas liée à l'exécution des fonctions, ou simplement celles dont le type impose qu'elles soient allouées dans cette zone, par exemple parce que leur taille peut évoluer (comme les listes en python)

Pour illustrer le fonctionnement de la mémoire et notamment la gestion des mots contenus dans la pile prenons un exemple.

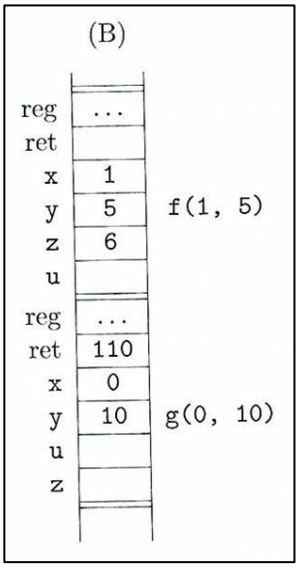
Considérons les deux fonctions suivantes et observons la gestion de la pile lors de l'appel f(1,5)

```
def g(x,y):
    return 100+y
```

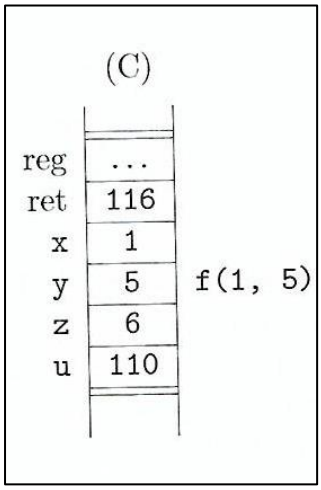
```
def f(x,y):
    z=x+y
    u=g(x-1,y*2)
    return z+u
```



La pile (A) représente l'environnement lors de l'appel à f(1,5). Les premières cases mémoires contiennent respectivement une sauvegarde des registres (reg) du microprocesseur, un espace pour la valeur de retour de la fonction (ret), les valeurs des paramètres x(1) et y(5), ainsi qu'un espace pour la variable locale z (qui pour le moment en contient aucune valeur).



La pile (B) représente l'environnement lors de l'appel à g(x-1,y*2). Pour réaliser cet appel, un nouvel environnement est alloué sur la pile avec une sauvegarde des registres et les valeurs des arguments x(0) et y(10) de g. Cet appel se termine en renvoyant la valeur 100+y (110) qui est stockée dans son espace de retour (ret).



La pile (C) montre enfin l'environnement de f(1,5) après le retour de g(0,10). On voit que l'espace mémoire alloué pour l'appel g a été supprimé de la pile, que la valeur de retour (110) a été récupérée et stockée dans la case mémoire de la variable locale u et que la sonne z+u (116) est stockée dans l'espace de retour.

Exercice 1

Etant données les fonctions python suivantes :

```
def f(x):
        z= x * x
        return z - x
```

```
def g(y):
        x = y + 1
        t=f(x)
        return t + y + x
```

Décrire la configuration de la pile pour l'appel à g(4), au moment (A) où l'appel interne f(x) s'apprête à renvoyer sa valeur et (B) où la fonction g s'apprête à renvoyer son résultat.

Exercice 2

Etant données les fonctions python suivantes :

```
def f1(x):
        t=[x]
        u=[t , t]
        return u
```

```
def f2(x):
        a=f1(x)
        return a[1]
```

Quelles zones mémoires ont été allouées sur le tas pendant l'exécution de f2(10) ? Quelles zones peuvent être libérées ?

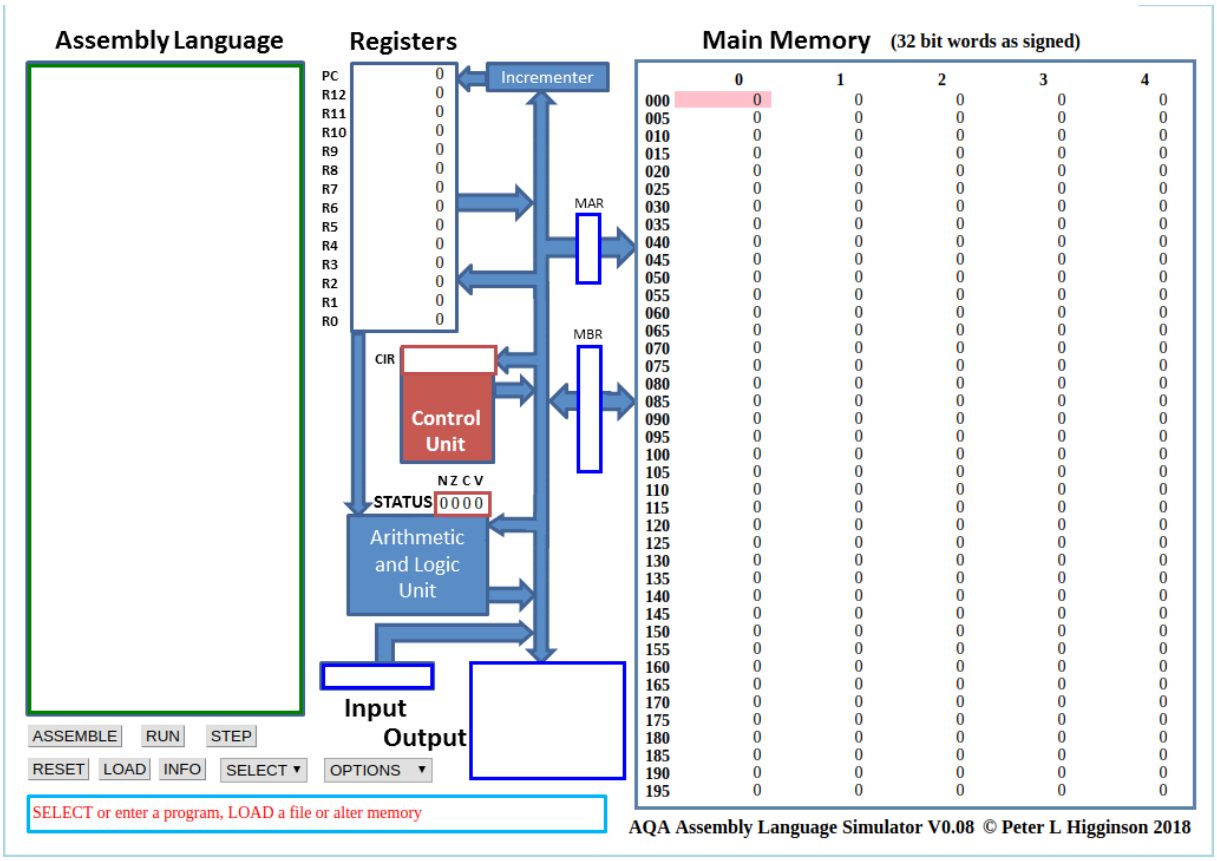
2ème partie : fonctionnement du microprocesseur

Pour mettre en pratique les notions abordés dans le cours sur le modèle d'architecture de von Neumann, nous allons utiliser un simulateur développé par **Peter L Higginson.**

Ce simulateur est basé sur une architecture de von Neumann.
Nous allons trouver dans ce simulateur :

- une RAM
- un CPU

Une version en ligne de ce simulateur est disponible ici : http://www.peterhigginson.co.uk/AQA/



On retrouve les différentes parties étudiées dans la structure de l'ordinateur et du processeur avec :

- A droite : la mémoire vive et ses adresses
- Au centre : le microprocesseur avec :
    - son UAL (Unité Arithmétique et Logique) et ses registres de données(R0 à R12). L'accumulateur est contenu dans l'UAL et n'est donc pas représenté au sens strict.
    - son unité de contrôle avec son registre d'instruction (CIR) et son pointeur d'instruction (ici PC) qui contient l'adresse mémoire de l'instruction en court d'exécution.
- A gauche, une zone dans laquelle nous pouvons saisir un programme en assembleur

Vous remarquerez que 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.

A l'aide du bouton « OPTIONS », passez à un affichage en binaire.
Combien de bits comporte alors chaque cellule de mémoire ? Combien d'octets ?

Vous pouvez repasser à un affichage en base 10 (bouton "OPTION"->"signed")

Avant de se servir de ce simulateur et de voir le fonctionnement du microprocesseur, il faut avoir quelques notions d'assembleur.
La programmation en assembleur consiste à écrire des instructions qui contiennent deux parties comme les instructions en langage binaire : une partie **code d'opération** et une partie **opérandes.**
La 1ère partie indique le type de traitement à réaliser, la seconde la nature des données sur lesquelles l'opération doit être effectuée.
Voici quelques instructions en assembleur :

Instructions	Rôles
LDR R1,78	Place la valeur stockée à l'adresse mémoire 78 dans le registre R1 (LDR R1 correspond à l'opération et 78 à l'opérande)
STR R3,125	Place la valeur stockée dans le registre R3 en mémoire vive à l'adresse 125
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
ADD R0,R1,R2	Additionne la valeur stockée dans le registre R1 et la valeur stockée dans le registre R2, place le résultat dans le registre R0
SUB R1,R0,#128	Soustrait le nombre 128 de la valeur stockée dans le registre R0, place le résultat dans le registre R1
MOV R0,R3	Place la valeur stockée dans le registre R3 dans le registre R0
B 45	Nous avons une structure de rupture de séquence, la prochaine instruction à exécuter se situe en mémoire vive à l'adresse 45
CMP R0,#23	Compare la valeur stockée dans le registre R0 et le nombre 23. Cette instruction CMP doit précéder une instruction de branchement conditionnel BEQ, BNE, BGT, BLT (voir ci-dessous)
CMP R0,#23 BEQ 78	La prochaine instruction à exécuter se situe à l'adresse mémoire 78 si la valeur stockée dans le registre R0 est **égale** à 23
CMP R0,#23 BNE 78	La prochaine instruction à exécuter se situe à l'adresse mémoire 78 si la valeur stockée dans le registre R0 **n'est pas égale** à 23
CMP R0,#23 BGT 78	La prochaine instruction à exécuter se situe à l'adresse mémoire 78 si la valeur stockée dans le registre R0 **est plus grand** que 23
CMP R0,#23 BLT 78	La prochaine instruction à exécuter se situe à l'adresse mémoire 78 si la valeur stockée dans le registre R0 **est plus petit** que 23
HALT	Arrêt de l'exécution du programme

Exercice : que signifie les expressions suivantes écrites en assembleur :

ADD R0,R1,#42	
LDR R5,98	
CMP R4,#18 **BGT 77**	
STR R0,15	
B 100	

Exercice : traduisez les expressions suivantes en langage assembleur :

Additionne la valeur stockée dans le registre R0 et la valeur stockée dans le registre R1, le résultat est stocké dans le registre R5	
Place la valeur stockée à l'adresse mémoire 878 dans le registre R0	
Place le contenu du registre R0 en mémoire vive à l'adresse 124	
la prochaine instruction à exécuter se situe en mémoire vive à l'adresse 478	
Si la valeur stockée dans le registre R0 est égale 42 alors la prochaine instruction à exécuter se situe à l'adresse mémoire 85	

Dans la partie « Assembly Language », saisissez les lignes de codes suivantes :

```
MOV R0,#78
STR R0,150
MOV R1,#80
ADD R2,R1,R0
HALT
```

Rappelez la signification de chacune de ces lignes.

Cliquez sur le bouton « **submit** ».
Vous devriez voir apparaitre des nombres "étranges" dans les cellules mémoires d'adresses 000, 001,002,003 et 004 :

Main Memory (32 bit words as signed)

	0	1	2	3	4
000	-476053426	-443612596	-476049328	-528408576	-285212672
005	0	0	0	0	0
010	0	0	0	0	0

L'assembleur a fait son travail, il a converti les 5 lignes de notre programme en instructions machines,

- la première instruction machine est stockée à l'adresse mémoire 000 (elle correspond à "MOV R0,#78" en assembleur),
- la deuxième à l'adresse 001 (elle correspond à "STR R0,150" en assembleur) etc…
- la dernière à l'adresse 004 (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 :

Main Memory (32 bit words as binary)

	0	1	2	3	4
000	11100011 10100000 00000000 01001110	11100101 10001111 00000010 01001100	11100011 10100000 00010000 01010000	11100000 10000001 00100000 00000000	11101111 00000000 00000000 00000000
005	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000	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	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000000

Nous pouvons donc maintenant affirmer que :

- l'instruction machine "11100011 10100000 00000000 01001110" correspond au code assembleur "MOV R0,#78"
- 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 01001110", vous pouvez remarquer que l'octet le plus à droite, $(01001110)_2$, est bien égale à $(78)_{10}$!

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

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 ">>".**
SI vous êtes sur la version en ligne, ralentissez l'animation au minimum pour pouvoir observer les différentes étapes qu'effectue le processeur pour réaliser le programme.

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

Décrivez les étapes qui se déroule dans le processeur pour réaliser la 1ère instruction
Expliquer alors comment fait le processeur pour lire les instructions suivantes du programme (stockées dans les mémoires 1,2,3 et 4)
Décrivez les événements supplémentaires qui ont lieu dans la réalisation des lignes 1 et 3 du programme.
Quels types d'informations passent par la MAR et la MBR ?

Une fois la simulation terminée, vous pouvez constater que le registre R2 du processeur contient bien le nombre 158 (en base 10). Vous pouvez aussi constater que la case mémoire 150 a bien stocké le nombre 78.

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.

Exercice 1 :
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.

Exercice 2 :
Ecrivez un programme qui récupère une valeur à l'adresse mémoire 78 et qui l'additionnera à la valeur 123. Cette valeur sera affichée dans le registre R2.
RQ. Vous pouvez écrire directement une valeur en mémoire en cliquant sur la case correspondante

Exercice 3

```
  MOV R0, #4
  STR R0,30
  MOV R0, #8
  STR R0,75
  LDR R0,30
  CMP R0, #10
  BNE else
  MOV R0, #9
  STR R0,75
  B endif
else:
  LDR R0,30
  ADD R0, R0, #1
  STR R0,30
endif:
  MOV R0, #6
  STR R0,23
  HALT
```

Après avoir analysé très attentivement le programme en assembleur ci-dessus :
À quoi sert la ligne "B endif" ?
À quoi correspondent les adresses mémoires 23, 75 et 30 à la fin du programme ?
Vous essaierez d'établir une correspondance entre ce programme codé en assembleur et le même écrit en Python.
RQ. Pour vous aider, vous pouvez recopier ce programme dans le simulateur et observer attentivement ce qu'il se passe.

Exercice 4 :
Voici un programme en python :

```
x=0
while x<3 :
        x=x+1
```

Ecrivez et testez un programme en assembleur équivalent au programme ci-dessus.
La valeur finale de x sera écrite dans le registre R0