

1. Structure des ordinateurs

Un ordinateur est une machine programmable, automatique et universelle :

- **programmable** : la séquence d'opérations exécutée par un ordinateur peut être entièrement spécifiée dans le texte d'un programme,
- **automatique** : un ordinateur peut exécuter un programme sans intervention extérieure (câblage, ...),
- **universelle** : un ordinateur peut exécuter tout programme calculable (selon la théorie de Turing) avec le jeu d'instructions câblé dans son processeur.

En 1945, John Von Neumann, mathématicien hongrois exilé aux États-Unis, publie un rapport sur la réalisation du calculateur EDVAC où il propose une architecture permettant d'implémenter une machine universelle, décrite par Alan Turing dans son article fondateur de 1936 sur le problème de l'indécidabilité.

L'architecture de Von Neumann va servir de modèle pour la plupart des ordinateurs de 1945 jusqu'à nos jours, elle se compose de quatre parties distinctes :

x L'**Unité Centrale de Traitement** (Central Processing Unit en anglais) ou Processeur est constituée de deux sous-unités :

• L'**Unité de Commande** (Control Unit) :

- charge la prochaine instruction dont l'adresse mémoire se trouve dans un registre appelé **Compteur de Programme** ou Compteur ordinal (Program Counter ou PC en anglais),
- la stocke dans le **Registre d'Instruction** (Instruction register),
- la décode avec le décodeur et commande l'exécution par l'**ALU** avec le séquenceur OU effectuer une opération de branchement (un saut dans le programme), en modifiant le Compteur de Programme, qui par défaut est incrémenté de 1 lors de chaque instruction.

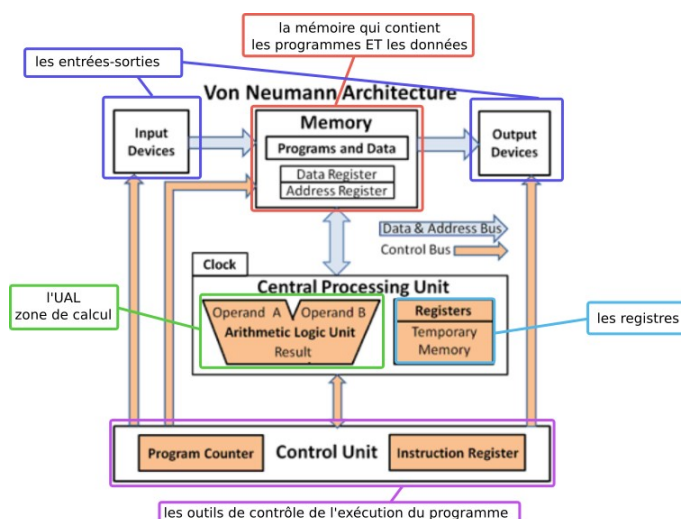
• L'**Unité Arithmétique et Logique** (ALU en anglais) :

- qui réalise des opérations arithmétiques (addition, multiplication, ...), logiques (et, ou, ...), de comparaisons ou de déplacement de mémoire (copie de ou vers la mémoire).
- L'ALU stocke les données dans des mémoires d'accès très rapide appelées **registres**.
- Les opérations sont réalisées par des circuits logiques constituant le **jeu d'instructions du processeur**.

x La **mémoire** où sont stockés les **données** et les **programmes** (Von Neumann).

x Des **bus** qui sont des fils reliant le **CPU** et la **mémoire** et permettant les échanges de données et d'adresses.

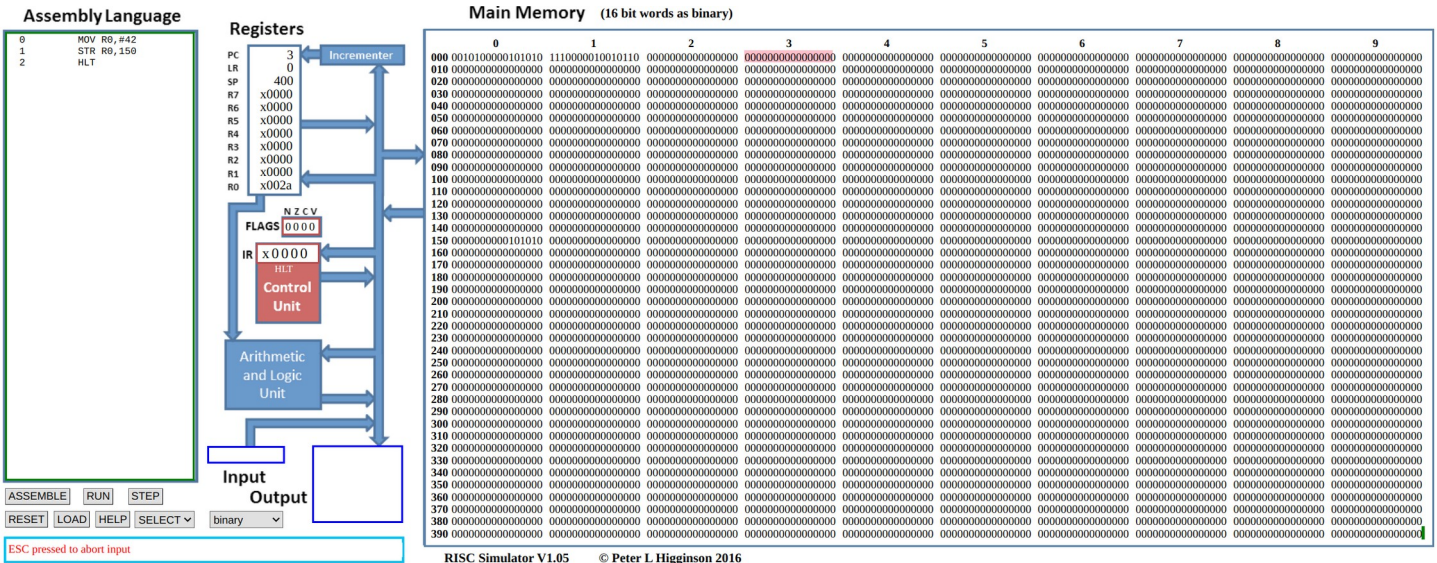
x Des dispositifs d'**entrées/sorties** permettant d'échanger avec l'extérieur (lecture ou écriture de données).



2. Simulateur CPU

Nous utiliserons un simulateur en ligne développé par Peter L Higginson, basé sur une architecture de von Neumann. Il est accessible sur le lien suivant <https://www.peterhigginson.co.uk/RISC/>. La documentation du jeu d'instruction du CPU simulé est accessible au lien http://www.peterhigginson.co.uk/RISC/instruction_set.pdf. Vous dispose également d'un bouton «HELP».

L'utilisation de Chrome est préférable pour accéder à la ressource .



2.1. **Double cliquer** dans la zone « Assembly Language » et **saisir** le code assembleur ci-dessous, **valider** la saisie avec les boutons « submit » puis « Assemble ». **Choisir** le mode « hexadécimal » dans le choix « option ». Que se passe-t-il dans la mémoire aux adresses 000, 001 et 002 ?

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

.....

.....

.....

2.2. **Exécuter** pas à pas à l'aide du bouton « step ». Que fait ce programme ?

.....

.....

.....

2.3. **Conclure** sur le type d'architecture (Von Neumann ou Harvard)

.....

.....

.....

2.4. La documentation du jeu d'instructions du processeur RISC précise le codage de l'instruction « MOV Rd, #immediate » et « STR Rs, direct » en binaire :

	MOV					Rd			#immediate							
MOV R0,#42	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0

	STR				Rd			direct								
STR R0,150	1	1	1	0	0	0	0	0	1	0	0	1	0	1	1	0

Compléter le tableau ci-dessous afin de charger la valeur 62 dans la dernière case mémoire (adresse 399) à l'aide du registre R7. **Modifier** directement la zone mémoire afin de mettre à jour votre programme. **Tester**.

	MOV					Rd			#immediate							
MOV R7,#62	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	0

	STR				Rd			direct								
STR R7,399	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1

2.5. À l'aide de la liste de choix « select », **charger** le programme « add » et **l'exécuter**. **Préciser** ce que fait ce programme.

.....

2.6. Boucle « pour » : Le programme assembleur est proposé ci-dessous. Le **saisir** et **l'exécuter** afin de **proposer** une version python équivalente.

Version assembleur

```
INP R1,2
MOV R0,#0
FOR_LOOP: OUT R0,4
ADD R0,#1
CMP R0,R1
BLT FOR_LOOP
FIN_POUR: HLT
```

Version python

2.7. Boucle « pour » : **Modifier** le programme assembleur de la question 2.5 afin qu'il demande et affecte la borne inférieure de la boucle « pour » dans le registre R2. On cherche à imiter l'instruction python « for R0 in range (R2,R1) ».

Variante : Un raffinement serait de vérifier que R1 > R2 et de n'exécuter la boucle que dans ce cas !!!

2.8. Boucle « pour » : **Modifier** le programme assembleur de la question 2.5 afin qu'il calcule la somme des premiers entiers dans le registre R3, l'affiche en fin de programme et le stocke dans la mémoire à l'adresse 0x100.

2.9. Boucle «inconnu» : Le programme assembleur est proposé ci-dessous. Le **saisir** et l'**exécuter** afin de **proposer** une version python équivalente. Afin de valider la bonne structure algorithmique réalisée, vous pouvez modifier la première instruction MOV R0,#5 par MOV R0,#0

Version assembleur

```
MOV R0,#5

Loop_Begin: CMP R0,#0

            BLS Loop_End

            OUT R0,4

            SUB R0,#1

            BRA Loop_Begin

Loop_End: HLT
```

Version python

2.10. Richard Pawson, co-auteur de l'émulateur en ligne, a développé le jeu « snake » pour celui-ci. Il s'appuie sur l'émulateur simplifié disponible sur <https://www.peterhigginson.co.uk/AQA/> et le code assembleur commenté est disponible sur son github, (et dans les fichiers de TP)

<https://raw.githubusercontent.com/richardpawson/education/master/Snake/Snake/snake.txt>

Assembly Language

```
0 defineRegisters:
1  mov r1,#0x008844
//Snake colour (green)
2  mov r2,#0xfffff
//Background colour (white)
3  mov r3,#271 //Tail
position, initialised
4  mov r4,#272 //Head
position, initialised
5  mov r5,#0 //Apple
position
6  mov r6,#0xff8800
//Apple colour
7  mov r7,#body
//Pointer front of queue,
initialised to first data loc
8  add r8,r7,#1
//Pointer to head address in
body data (1 after tail)
9  mov r9,#0 //ASCII
value of last key pressed
10 mov r10,#767
//Constant representing the
size of screen memory
11 mov r11,#1023
//Constant
12 mov r12,#68
//Current direction of
movement, initialised to
'right'
13 InitialisePointers:
14 str r3,[r7] //r4
points to the tail address
15 str r4,[r8] //r3
points to the head address
16 drawSnake:
17 str r1,[r3+256]
//Tail
18 str r1,[r4+256]
//Head
```

Registers

PC	x0000001b
R12	x00000041
R11	x0000003ff
R10	x0000002ff
R9	x00000041
R8	x00000076
R7	x00000075
R6	x00ff8800
R5	x000002c1
R4	x0000024d
R3	x0000010f
R2	x00ffffff
R1	x00008844
R0	x00ffffff

Main Memory (32 bit words as hexadecimal)

	0	1	2	3	4
000	x1c008844	x2cffffff	x3c00010f	xe3a04e11	xe3a05000
005	x6cff8800	xe3a07058	xe2878001	xe3a09000	xac0002ff
010	xbcc0003ff	xe3a0c044	xe5873000	xe5884000	xe5831400
015	xe5841400	xef015008	xe005500b	xe155000a	xbaffffff
020	xe1550004	x0affffff9	xe5856400	xef019004	xe3590057
025	x0a000014	xe3590041	xe3590041	xe3590053	x0a00000a
030	xe3590044	x0a000001	xela0900c	xeaffffff5	xe35c0041
035	x0affffffb	xe2844001	xe204001f	xe3500000	x0a00002c
040	xea000012	xe35c0057	x0affffff4	xe2844020	xe154000a
045	xb0000026	xea00000c	xe35c0053	x0affffffe	xe2444020
050	xe3540000	xca000020	xea000006	xe35c0044	x0affffffe8
055	xe2444001	xe204001f	xe350001f	x0a000019	xeaffffff
060	xe1a0c009	xe1540005	x0a000005	xe5970000	xe5802400
065	xe2877001	xe35700c8	xca000000	xe3a07058	xe2888001
070	xe35800c8	xca000000	xe3a08058	xe5884000	xe5940400
075	xe1500001	x0a000007	xe1570008	x0a000003	xe5841400
080	xe1540005	x0affffffc2	xeaffffffc2	xe3a00056	xef020008
085	xef000000	x20756f59	x006e6977	x0000010f	x00000110
090	x00000111	x00000112	x00000113	x00000114	x00000115
095	x00000116	x00000117	x00000118	x00000138	x00000158
100	x00000178	x00000198	x000001b8	x000001d8	x000001f8
105	x00000218	x00000238	x00000258	x00000278	x00000298
110	x00000255	x00000254	x00000253	x00000252	x00000251
115	x00000250	x0000024f	x0000024e	x0000024d	x00000000
120	x00000000	x00000000	x00000000	x00000000	x00000000
125	x00000000	x00000000	x00000000	x00000000	x00000000
130	x00000000	x00000000	x00000000	x00000000	x00000000
135	x00000000	x00000000	x00000000	x00000000	x00000000
140	x00000000	x00000000	x00000000	x00000000	x00000000
145	x00000000	x00000000	x00000000	x00000000	x00000000
150	x00000000	x00000000	x00000000	x00000000	x00000000
155	x00000000	x00000000	x00000000	x00000000	x00000000
160	x00000000	x00000000	x00000000	x00000000	x00000000
165	x00000000	x00000000	x00000000	x00000000	x00000000
170	x00000000	x00000000	x00000000	x00000000	x00000000
175	x00000000	x00000000	x00000000	x00000000	x00000000
180	x00000000	x00000000	x00000000	x00000000	x00000000
185	x00000000	x00000000	x00000000	x00000000	x00000000
190	x00000000	x00000000	x00000000	x00000000	x00000000
195	x00000000	x00000000	x00000000	x00000000	x00000000

ASSEMBLE STOP << >> SPEED 23.6

RESET LOAD INFO SELECT hex

Input Output

Done instruction CMP Rn,imm

AQA Assembly Language Simulator V0.08 © Peter L Higginson 2018

Lancer le simulateur simplifié AQA à partir du lien ci-dessus, **charger** le fichier assembleur snake.txt à l'aide du bouton « LOAD », **lancer** la simulation et **régler** la vitesse au maximum à l'aide du bouton « >> ». Les déplacements sont obtenus à l'aide des touches w (Monter), s (Descendre), a (Gauche) et d (Droite). Faites-vous plaisir ...